Verilog Language Reference

Verilog Modeling Style Guide (CFE), Product Version 3.1

Table of contents:

1.	Overview	2
2.	Lexical Conventions	2
3.	Data Types	4
4.	Expressions	7
5.	Assignments	13
6.	Gate and Switch Level Modeling	14
7.	User-Defined Primitives (UDPs)	15
8.	Behavioral Modeling	21
9.	Tasks and Functions	25
10.	Hierarchical Structures	27
11.	System Tasks	31
12.	Compiler Directives	32
13.	Pragmas	33

Készült a program eredeti Online-Help-je alapján az Elektronikus Eszközök Tanszék *ASIC és FPGA labor gyakorlat* hallgatói számára

Budapest, 2002 Szeptember 11

Gärtner Péter

1. Overview

This brochure describes the common Verilog language syntax supported by the Cadence tools that accept models written at the Register Transfer Level (RTL) of abstraction.
Note: The tools may have other mechanisms to support other Verilog constructs. Please refer the respective documentation for details.

2. Lexical Conventions

Verilog language source files are a stream of lexical tokens. A lexical token consists of one or more characters. The layout of tokens in a source file is free format--that is, spaces and newlines are not syntactically significant.
This brochure uses a syntax formalism based on the Backus-Naur Form (BNF) to define the Verilog language syntax.

White Space and Comments

White space can contain the characters for blanks, tabs, newlines, and formfeeds. These characters are ignored except when they serve to separate other tokens. However, blanks and tabs are significant in strings.
There are two forms to introduce comments. A one-line comment starts with the two characters // and ends with a newline. A block comment starts with /* and ends with */. Block comments cannot be nested, but a one-line comment can be nested within a block comment.

Operators

Operators are single-, double-, or triple-character sequences and are used in expressions. "Expressions" discusses the use of operators in expressions. Unary operators appear to the left of their operand. Binary operators appear between their operands. A ternary operator has two operator characters that separate three operands.

Numbers

You can specify constant numbers in decimal, hexadecimal, octal, or binary format. Negative numbers are represented in 2's complement form. When used in a number, the question mark (?) character is the Verilog alternative for the z character. The underscore character (_) is legal anywhere in a number except as the first character, where it is ignored.
Note: Real numbers are rounded off to the nearest integer.

Syntax

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

decimal_number ::=
      [ sign ] unsigned_number
   | [ size ] decimal_base unsigned_number

binary_number ::= [ size ] binary_base binary_digit { _ | binary_digit }

octal_number ::= [ size ] octal_base octal_digit { _ | octal_digit }

hex_number ::= [ size ] hex_base hex_digit { _ | hex_digit }

real_number ::=
      [sign] unsigned_number.unsigned_number
   | [sign]unsigned_number[.unsigned_number]e[sign]unsigned_number
   | [sign]unsigned_number[.unsigned_number]E[sign]unsigned_number
```

sign ::= + | -
size ::= unsigned_number
unsigned_number ::= decimal_digit { _ | decimal_digit }
decimal_base ::= 'd | 'D
binary_base ::= 'b | 'B
octal_base ::= 'o | 'O
hex_base ::= 'h | 'H
decimal_digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
binary_digit ::= x | X | z | Z | 0 | 1
octal_digit ::= x | X | z | Z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
hex_digit ::= x | X | z | Z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7| 8 | 9 | a | b | c | d | e | f | A | B | C | D | E | F

Strings

A string is a sequence of characters enclosed by double quotes and all contained on a single line. Strings used as operands in expressions and assignments are treated as a sequence of eight-bit ASCII values, with one eight-bit ASCII value representing one character. To declare a variable to store a string, declare a register large enough to hold the maximum number of characters the variable will hold. Note that no extra bits are required to hold a termination character; Verilog does not store a string termination character. Strings can be manipulated using the standard operators.
Note: When a variable is larger than required to hold a value being assigned, Verilog pads the contents on the left with zeros after the assignment. This is consistent with the padding that occurs during assignment of non-string values.
Certain characters can be used in strings only when preceded by an introductory character called an escape character. The following table lists these characters in the right-hand column with the escape sequence that represents the character in the left-hand column.

Specifying Special Characters in Strings

Characters Produced by Escape String

\n	New line character
\t	Tab character
\\	Backslash (\) character
\"	Double quote (") character
\ddd	A character specified in 1-3 octal digits (0 <= d <= 7)
%%	Percent (%) character

Identifiers, Keywords, and System Names

An identifier is used to give an object, such as a register or a module, a name so that it can be referenced from other places in a description. An identifier is any sequence of letters, digits, dollar signs ($), and the underscore (_) symbol. The first character must not be a digit or $; it can be a letter or an underscore. Upper- and lower-case letters are considered to be different. Identifiers can be up to 1024 characters long.

Escaped Identifiers

Escaped identifiers provide a means of including any of the printable ASCII characters in an identifier (the decimal values 33 through 126, or 21 through 7E in hexadecimal). An escaped identifier starts with the backslash character (\) and ends with white space (blank, tab, newline). When using bit- or part- selects on the escaped identifier, the bit- or part- select operator must be preceded by a space. Neither the leading backslash character nor the terminating white space is considered to be part of the identifier.
Note: Remember to terminate escaped identifiers with white space, otherwise characters that should follow the identifier are considered as part of it.

Keywords

Keywords are predefined non-escaped identifiers that are used to define the language constructs. A Verilog HDL keyword preceded by an escape character is not interpreted as a keyword. All keywords are defined in lowercase. Therefore, you must be type them in lowercase in source files.

3. Data Types

Data types represent the data storage and transmission elements of digital hardware.

Value Set

The value set consists of four basic values:

The value	Represents
0	A logic zero, or false condition
1	A logic one, or true condition
x	An unknown logic value
z	A high-impedance state

The values 0 and 1 are logical complements of one another. When the z value is present at the input of a gate, or when it is encountered in an expression, the effect is usually the same as an x value. All of the data types except event store all four basic values. All bits of vectors can be independently set to one of the four basic values.

Nets

The net data types represent physical connections between structural entities, such as gates. A net does not store a value. Instead, it must be driven by a driver, such as a gate or a continuous assignment.

Syntax

net_declaration ::= net_type [vectored | scalared] [range] list_of_net_identifiers ;
| net_type [vectored | scalared] [range] list_of_net_decl_assignments ;

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

range ::= [msb_constant_expression : lsb_constant_expression]

list_of_net_decl_assignments ::= net_decl_assignment {, net_decl-assignment }

Net Types

wire and tri nets

The wire and tri nets connect elements. The net types wire and tri are identical in their syntax and functions; two names are provided so that the name of a net can indicate the purpose of the net in that model. A wire net is typically used for nets that are driven by a single gate or continuous assignment. The tri net type is typically used where multiple drivers drive a net.

Wired Nets

Wired nets are of type wor, wand, trior, and triand, and are used to model wired logic configurations. Wired nets resolve the conflicts that result when multiple drivers drive the same net. The wor and trior nets create wired or configurations, such that when any of the drivers is 1, the net is 1. The wand and triand nets create wired and configurations, such that if any driver is 0, the net is 0.
The net types wor and trior are identical in their syntax and functionality--as are the wand and

triand.

tri0 and tri1 Nets

The tri0 and tri1 nets are treated as wire.

Supply Nets

The supply0 and supply1 nets model the power supplies in a circuit. The supply0 nets are used to model Vss (ground) and supply1 nets are used to model Vdd or Vcc (power). These nets should never be connected to the output of a gate or continuous assignment, because the strength they possess will override the driver.

Multiple Driver Resolution

When a net is being driven by more than one drivers, the net value is resolved as follows:

For Nets	The value is
wire, tri0, tri1, wor and	Logical OR of driver values
wand and triand	Logical AND of driver values

Registers

A register is an abstraction of a data storage element. The keyword for the register data type is reg. A register stores a value from one assignment to the next. An assignment statement in a procedure acts as a trigger that changes the value in the register.

Syntax

reg_declaration ::= reg [range] list_of_register_identifiers ;

list_of_register_identifiers ::= register_name {, register_name }

register_name ::=
  register_identifier
  | memory_identifier [upper_limit_constant_expression : lower_limit_constant_expression]

If a register is unconditionally assigned in an always statement, then it is reduced to be a wire. This is called the combinational reduction of a register.

Vectors

A net or reg declaration without a range specification is one bit wide; that is, it is scalar. Multiple bit net and reg data types are declared by specifying a range, and are known as vectors. The range specification gives addresses to the individual bits in a multi-bit net or register. The most significant bit (MSB) is the left-hand value in the range and the least significant bit (LSB) is the right-hand value in the range. The range is specified as follows:

range ::= [msb_constant_expression : lsb_constant_expression]

Both <msb_constant_expression> and <lsb_constant_expression> are non-negative constant expressions. There are no restrictions on the values of the indices. The MSB and LSB expressions can be any value, and <lsb_constant_expression> can be a greater value than <msb_constant_expression>, if desired.
Vector nets and registers obey laws of arithmetic modulo 2 to the power n, where n is the number of bits in the vector. Vector nets and registers are treated as unsigned quantities. A vector net can be used as a single entity or as a group of n scalars, where n is the number of bits in the vector net. The keyword vectored allows you to specify that a vector net can be modified only as an indivisible entity. The keyword scalared explicitly allows access to bit and parts.

Only when a net is not specified as vectored can bit selects and part selects be driven by outputs of gates, primitives, and modules--or be on the left-hand side of continuous assignments. The following are examples of vector net declarations:

```
tri1 scalared [63:0] bus64;      //a bus that will be expanded
tri vectored [31:0] data;                  //a bus that will not be expanded
```

Note: The keywords scalared and vectored apply only to vector nets and do not apply to vector registers.

Implicit Declarations

In the absence of an explicit declaration of a variable, statements for gate, user-defined primitive, and module instantiations assume an implicit variable declaration. This happens if you specify a variable that has not been explicitly declared previously in one of the declaration statements of the instantiating module in the terminal list of an instance of a gate, a user-defined primitive, or a module. These implicitly declared variables are scalar nets of type wire.

Memories

An array of registers can be used to model memories. Each register is known as a word and is addressed by a single array index. Memories are declared in register declaration statements by specifying the element address range after the declared identifier.

Syntax

```
memory_declaration ::= reg [ range ] list_of_memory_identifiers ;

list_of_memory_identifiers ::= register_name {, register_name }

register_name ::=
   memory_identifier [ upper_limit_constant_expression : lower_limit_constant_expression ]
```

For example, the following declaration declares a memory mema of 256 8-bit registers:

```
reg [7:0] memb[0:255]
```

Depending upon the assignment, a memory register is interpreted as a wire, a latch or a flip-flop.

Integers

In addition to modeling hardware, there are other uses for variables in an HDL model. Although you can use the reg variables for general purposes such as counting the number of times a particular net changes value, the integer register data type is provided for convenience and to make the description more self-documenting.

Syntax

```
integer_declaration ::= integer list_of_register_identifiers ;
list_of_register_identifiers ::= register_name {, register_name }

register_name ::=
   register_identifier
   | memory_identifier [ upper_limit_constant_expression : lower_limit_constant_expression ]
```

You may use an integer variable as a general-purpose variable for manipulating quantities that are not regarded as hardware registers. The size of an integer variable is 32 bits.
You can use arrays of integer variables. The integer arrays are declared in the same manner as arrays of reg variables, as in the following example:

```
integer a[1:64];                // an array of 64 integers
```

You can assign values to the integer variable the same manner as reg variables and use procedural assignments to trigger their value changes.
The integer variables are signed quantities. Arithmetic operations performed on integer variables produce 2's complement results.

Parameters

Parameters represent constants that need to be used many places in the design. Parameters do not belong to either the register or the net group. Parameters are not variables.

Syntax

parameter_declaration ::= parameter list_of_parameter_assignments ;

list_of_parameter_assignments ::= param_assignment {, param_assignment }

param_assignment ::= parameter_identifier = constant_expression

Even though they represent constants, you can modify parameters at compilation time to have values that are different from those specified in the declaration assignment. This allows you to customize module instances. You can modify the parameter with the defparam statement, or you can modify the parameter in the module instance statement. Typical uses of parameters are to specify the width of variables. You can access bits and parts of parameters declared in this way and use them in assignments and logic operations.
See "Overriding Module Parameter Values" for more details on parameter value assignment.

4. Expressions

An expression is a construct that combines operands with operators to produce a result that is a function of the values of the operands and the semantic meaning of the operator. Alternatively, an expression is any legal operand -- for example, a net bit-select. Wherever a value is needed in a statement, an expression can be given. However, several statement constructs limit an expression to a constant expression. A constant expression consists of constant numbers and predefined parameter names only, but can use any of the operators defined in "Operators".
For its use in expressions, data type integer shares the same traits as the data type reg. Descriptions of register usage apply to integers as well.

An operand can be one of the following:

- number
- net
- register, integer
- net bit-select
- register bit-select
- net part-select
- register part-select
- memory element
- a call to a user-defined function that returns any of the above

Operators

The following table summarizes the supported operators:

Operator	Type
{ }	concatenation
+ - * /	arithmetic
%	modulus

> >= < <=	relational
!	logical negation
&&	logical and
||	logical or
==	logical equality
!=	logical inequality
===	case equality
!==	case inequality
~	bit-wise negation
&	bit-wise and
|	bit-wise inclusive or
^	bit-wise exclusive or
^~ or ~^	bit-wise equivalence
&	reduction and
~&	reduction nand
|	reduction or
~|	reduction nor
^	reduction xor
~^ or ^~	reduction xnor
<<	shift left
>>	shift right
<<<	arithmetic shift left
>>>	arithmetic shift right
? :	conditional

The following table lists the Verilog's binary and ternary operators from highest precedence to the lowest; operators with the same precedence are shown in the same row.

! ~	highest precedence
* / %	
+ -	
<< >> <<< >>>	
< <= > >=	
== != === !==	
& ^~ | && ||	
?: (ternary operator)	lowest precedence

All operators associate left to right. Associativity refers to the order in which a language evaluates operators having the same precedence. Thus, in the following example, B is added to A and then C is subtracted from the result of A+B: A + B - C
When operators differ in precedence, the operators with higher precedence apply first. In the following example, B is divided by C (division has higher precedence than addition) and then the result is added to A: A + B / C
Parentheses can change the operator precedence: (A + B) / C // not the same as A + B / C

Numeric Conventions in Expressions

Operands can be expressed as based and sized numbers--with the following restriction: The Verilog language interprets a number of the form sss 'fnnn, when used directly in an expression, as the unsigned number represented by the two's complement of nnn. The following example shows two ways to write the expression "minus 12 divided by 3." Note that -12 and -d12 both evaluate to the same bit pattern, but in an expressionn -d12 loses its identity as signed, negative number.

```
integer IntA;
IntA = -12 / 3;     // The result is -4.
IntA = -'d 12 / 3;  // The result is 1431655761
```

Arithmetic Operators

The unary operators are the plus (+) and minus (-) signs.
The binary arithmetic operators are the following:

+ - * / % (the modulus operator)

The unary arithmetic operators take precedence over the binary operators. For the arithmetic operators, if any operand bit value is the unknown value x, then the entire result value is x. Integer division truncates any fractional part.

Arithmetic Expressions with Registers and Integers

An arithmetic operation on a register data type behaves differently than an arithmetic operation on an integer data type. A register data type is seen as an unsigned value and an integer data type is seen as a signed value. As a result, when you assign a value of the form

- size base_format number

to a register and then use that register as an expression operand, you are actually using a positive number that is the two's complement of number. In contrast, when you assign a value of the form

- size base_format number

to an integer and then use that integer as an expression operand, the expression evaluates using signed arithmetic. The following example shows various ways to divide minus twelve by three using integer and register data types in expressions.

```
integer intA;
reg [15:0] regA;
intA = -4'd12;
regA = intA / 3;            // Result is 65532, which is the bit pattern
                            // for 16 bit -4 assigned to an unsigned register
regA = -4'd12;              // Result is 65524 because regA is unsigned
intA = regA / 3;
intA = -4'd12 / 3;          // Result is 1431655761 because it is evaluated to 32 bits
regA = -12 / 3;             // Result is 65532 because regA is unsigned.
```

Relational Operators

The relational operators are:

<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

All the relational operators yield the scalar value 0 if the specified relation is false, or the value 1 if the specified relation is true. If, due to unknown bits in the operands, the relation is ambiguous, then the result is the unknown value (x).

All the relational operators have the same precedence.

Equality Operators

The equality operators are:

==	===	Equality operator	a == b or a === b mean a equal to b, result may be unknown

!=	!==	Inequality operator	a != b or a !== b mean a not equal to b, result may be unknown

These four operators compare operands bit for bit, with zero filling if the two operands are of unequal bit-length. As with the relational operators, the result is 0 if false, 1 if true.
For the == and != operators, if either operand contains an x or a z, then the result is the unknown value (x). The === and !== operators are treated just like == and != respectively.
All four equality operators have the same precedence.

Logical Operators

The logical operators are:

&&	Logical AND
||	Logical OR
!	Logical negation

The operators logical AND (&&) and logical OR (||) are logical connectives. The result of the evaluation of a logical comparison is one (defined as true),
zero (defined as false), or, the unknown value (x) if either operand is x or z.
The precedence of && is greater than that of ||, and both are lower than relational and equality operators.
The third logical operator is the unary logical negation operator !. The negation operator converts a non-zero or true operand into 0 and a zero or false operand into 1. An ambiguous truth value (x or z) remains as x.

Bit-Wise Operators

The bit-wise operators are:

&	Bit-wise binary AND operator
|	Bit-wise binary OR operator
^	Bit-wise binary Exclusive OR operator
^~ ~^	Bit-wise binary Exclusive NOR operator

The bit operators perform bit-wise manipulations on the operands--that is, the operator compares a bit in one operand to its equivalent bit in the other operand to calculate one bit for the result. When the operands are of unequal bit length, the shorter operand is zero-filled in the most significant bit positions.

Reduction Operators

The reduction unary operators are:

&	Reduction unary AND operator
|	Reduction unary OR operator
^	Reduction unary Exclusive OR operator

The unary reduction operators perform a bit-wise operation on a single operand to produce a single bit result.

Shift Operators

The shift operators, << and >>, perform left and right shifts of their left operand by the number of bit positions given by the right operand. Both shift operators fill the vacated bit positions with zeroes.

Arithmetic Shift Operators for Signed Objects

The arithmetic shift operators (<<< and >>>) work the same as regular shift operators on unsigned

objects. However, when used on signed objects, the following rules apply:

Arithmetic shift left ignores the signed bit and shifts bit values to the left (like a regular shift left operator), filling the open bits with zeroes.

Arithmetic shift right propagates all bits, including the signed bit, to the right while maintaining the signed bit value.

Conditional Operator

The conditional operator has three operands separated by two operators in the following format:

<cond_expr> ? <true_expr> : <false_expr>

If <cond_expr> evaluates to false, then <false_expr> is evaluated and used as the result. If <cond_expr> evaluates to true, then <true_expr> is evaluated and used as the result.

Concatenations

A concatenation is the joining together of bits resulting from two or more expressions. The concatenation is expressed using the brace characters { and }, with commas separating the expressions within.

The syntax of concatenation expressions is as follows:

concatenation ::= { expression {, expresion } }
multiple_concatenation ::= { expression { expression {, expresion } } }

Unsized constant numbers are not allowed in concatenations because the size of each operand in the concatenation is needed to calculate the complete size of the concatenation. Concatenations can be expressed using a repetition multiplier as shown in the next example:

```
{4{w}}                  // This is equivalent to {w, w, w, w}
```

The next example illustrates nested concatenations.

```
{b, {3{a, b}}}          // This is equivalent to {b, a, b, a, b, a, b}
```

The repetition multiplier must be a constant expression.

Operands

There are several types of operands that can be specified in expressions. The simplest type is a reference to a net or register in its complete form -- that is, just the name of the net or register is given. In this case, all of the bits making up the net or register value are used as the operand. If a single bit of a vector net or register is required, then a bit-select operand is used. A part-select operand is used to reference a group of adjacent bits in a vector net or register.
A memory element can be referenced as an operand.
A concatenation of other operands (including nested concatenations) can be specified as an operand.
A function call is an operand.

Net and Register Bit Addressing

Bit-selects extract a particular bit from a vector net or register. The bit can be addressed using an expression. The next example specifies the single bit of acc that is addressed by the operand index.

```
acc[index]
```

The actual bit that is accessed by an address is, in part, determined by the declaration of acc. For instance, each of the declarations of acc shown in the next example causes a particular value of

index to access a different bit:

reg [15:0] acc;
reg [1:16] acc;

If the bit select is out of the address bounds or is x, then the value returned by the reference is x.

Several contiguous bits in a vector register or net can be addressed, and are known as part-selects. A part-select of a vector register or net is given with the following syntax:

vect[ms_expr:ls_expr]

Both expressions must be constant expressions. The first expression must address a more significant bit than the second expression.

The next example and the bullet items that follow it illustrate the principles of bit addressing. The code declares an 8-bit register called vect and initializes it to a value of 4. The bullet items describe how the separate bits of that vector can be addressed.

reg [7:0] vect;
vect = 4;

If the value of addr is 2, then vect[addr] returns 1.
If the value of addr is out of bounds, then vect[addr] returns x.
If addr is 0, 1, or 3 through 7, vect[addr] returns 0.
If any bit of addr is x/z, then the value of addr is x.
vect[3:0] returns the bits 0100.
vect[5:1] returns the bits 00010.
vect[<expression that returns x>] returns x.
vect[<expression that returns z>] returns x.

Memory Addressing

The syntax for a memory address consists of the name of the memory and an expression for the address, specified with the following format:

mem_name[addr_expr]

The addr_expr can be any expression; therefore, memory indirections can be specified in a single expression. The next example illustrates memory
indirection:

mem_name[mem_name[3]]

In the above example, mem_name[3]addresses word three of the memory called mem_name. The value at word three is the index into mem_name that is used by the memory address mem_name [mem_name[3]]. As with bit-selects, the address bounds given in the declaration of the memory determine the effect of the address expression. If the index is out of the address bounds or is x, then the value of the reference is x.
There is no mechanism to express bit-selects or part-selects of memory elements directly. If this is required, then the memory element has to be first transferred to an appropriately sized temporary register.

Strings

String operands are treated as constant numbers consisting of a sequence of 8-bit ASCII codes, one per character, with no special termination character. The syntax of strings is as follows:

string ::= "{ Any_ASCII_Characters_except_newline }"

Any Verilog HDL operator can manipulate string operands. The operator behaves as though the entire string were a single numeric value.
The common string operations copy, concatenate, and compare are supported by Verilog operators. Copy is provided by simple assignment. Concatenation is provided by the concatenation operator. Comparison is provided by the equality operators.
When manipulating string values in vector variables, at least 8*n bits are required in the vector, where n is the number of characters in the string.
When strings are assigned to variables, the values stored are padded on the left with zeros. Padding can affect the results of comparison and concatenation operations. The comparison and concatenation operators do not distinguish between zeros resulting from padding and the original string characters. The null string ("") is equivalent to the value zero (0).

5. Assignments

The assignment is the basic mechanism for getting values into nets and registers. An assignment consists of two parts, a left-hand side and a right-hand side, separated by the equal sign (=). The right-hand side can be any expression that evaluates to a value. The left-hand side indicates the variable that the right-hand side is to be assigned to. The left-hand side can take one of the following forms, depending on whether the assignment is a continuous assignment or a procedural assignment:
Legal left-hand Side Forms in Assignment Statements:

Statement	Left-hand Side
Continuous Assignment	Net (vector or scalar)
	Constant bit-select of a vector net
	Constant part-select of a vector net Constant word select of memory
	Concatenation of any of the above
Procedural Assignment	Register (vector or scalar)
	Bit-select of a vector register
	Constant part-select of a vector register
	Memory element
	Concatenation of any of the above four items

Continuous Assignments

Continuous assignments drive values onto nets, both vector and scalar. The word continuous is used to describe this kind of assignment because the assignment is always active. Whenever simulation causes the value of the right-hand side to change, the assignment is re-evaluated and the output is propagated. Continuous assignments provide a way to model combinational logic without specifying an interconnection of gates. Instead, the model specifies the logical expression that drives the net. The expression on the right-hand side of the continuous assignment is not restricted in any way, and can even contain a reference to a function. Thus, the result of a case statement, if statement, or other procedural construct can drive a net.

Syntax

net_declaration ::=
  net_type [vectored | scalared] [range] list_of_net _identifiers ;
 | net_type [vectored | scalared] [range] list_of_net _assignments ;

continuous_assignment ::= assign list_of_net _assignments ;
range ::= [<constant_expression> : <constant_expression>]
list_of_net_assignments> ::= net_assignment { , net_assignment }
net_assignment ::= net_lvalue = expression

Continuous Assignments vs. Procedural Assignments

Continuous assignments drive nets in a manner similar to the way gates drive nets. The expression on the right-hand side can be thought of as a combinational circuit that drives the net continuously. Continuous assignments cannot be disabled.
In contrast, procedural assignments can only assign values to registers or memory elements, and the assignment (that is, the loading of the value into the register or memory) is done only when control is transferred to the procedural assignment statement.
Procedural assignments occur only within procedures, such as always and initial statements and in functions and tasks. They can be thought of as triggered assignments. The trigger occurs when the flow of execution reaches an assignment within a procedure. Reaching the assignment can be controlled by conditional statements such as if statements, case statements, and looping statements.

6. Gate and Switch Level Modeling

A gate or switch declaration names a gate or switch type and specifies how the gate or switch connects to other components in the model.

Syntax

gate_instantiation ::= gatetype gate_instance {, gate_instance };

gatetype ::= n_input_gatetype | n_output_gatetype | enable_gatetype

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

n_output_gatetype ::= buf | not

enable_gatetype ::= bufif0 | bufif1 | notif0 | notif1

gate_instance ::= n_input_gate_instance | n_output_gate_instance | enable_gate_instance
n_input_gate_instance ::=
[name_of_gate_instance] (output_terminal, input_terminal {, input_terminal })

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

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

name_of_gate_instance ::= gate_instance_identifier [range]

input_terminal ::= scalar_expression

enable_terminal ::= scalar_expression

output_terminal ::= terminal_identifier | terminal_identifier [constant_expression]

7. User-Defined Primitives (UDPs)

You can create your own gate primitives called user-defined primitives (UDPs) and use them like the standard gate primitives.

A combinational UDP uses the value of its inputs to determine the next value of its output. Each UDP has exactly one output that can be in one of three states 0, 1, or x.

A sequential UDP uses the value of its inputs and the current value of its output to determine the next value of its output.

UDP Definition

Syntax

```
upd_declaration ::=
   primitive
         udp_identifier (udp_port_list);
         udp_port_declaration { udp_port_declaration }
         udp_body
   endprimitive

udp_port_list ::=  output_port_identifier, input_port_identifier {, input_port_identifier }

udp_port_declaration ::=  output_declaration  | input_declaration  | reg_declaration

udp_body ::=  combinational_body | sequential_body

combinational_body ::=
   table
         combinational_entry { combinational_entry }
   endtable

combinational_entry ::= level_input_list : output_symbol ;

sequential_body ::=
   table
         sequential_entry { sequential_entry }
   endtable

sequential_entry ::= seq_input_list : current_state : next_state;

seq_input_list ::= level_input_list | edge_input_list

level_input_list ::= level_symbol { level_symbol }

edge_input_list ::= { level_symbol } edge_indicator { level_symbol }

edge_indicator ::= (level_symbol level_symbol ) | edge_symbol

current_state ::= level_symbol

next_state ::= output_symbol | -

output_symbol ::= 0 | 1 | x | X

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

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

UDP definitions are independent of modules; they are at the same level as module definitions in the syntax hierarchy. They can appear anywhere in the source text, either before or after they are used inside a module. They cannot appear between the keywords module and endmodule.

UDP Terminals

UDPs can have multiple input terminals, but only one output terminal. They cannot have bidirectional inout terminals. All UDP terminals are scalar. No vector terminals are allowed.

Only logic values of 0, 1, or x are allowed on input and output. The tri-state value z is not supported.

The output terminal must be the first terminal in the terminal list.

The output terminal of a sequential UDP requires an additional declaration as type reg. It is illegal to declare a reg for the output terminal of a combinational UDP.

UDP Declarations

UDPs must contain input and output terminal declarations. The output terminal declaration begins with the keyword output, followed by one output terminal name. The input terminal declaration begins with the keyword input, followed by one or more input terminal names.
Sequential UDPs must contain a reg declaration for the output terminal. Combinational UDPs cannot contain a reg declaration. You can specify the initial value of the output terminal reg in an initial statement in a sequential UDP.

UDP State Table

The state table that defines the behavior of a UDP begins with the keyword table and ends with the keyword endtable. Each row of the table is created using a variety of characters that indicate input and output states. Three states--0, 1, and x--are supported. The z state is not supported. There are a number of special characters you can use to represent certain combinations of state possibilities. These are listed in "Summary of UDP Symbols".
Combinational UDPs have one field per input and one field for the output. Use a colon to separate the input fields from the output field. Each row of the table is terminated by a semicolon. For example, the following state table entry specifies that when the three inputs are all 0, the output is 0.

```
table
  0 0 0 : 0;
  ...
endtable
```

Sequential UDPs have an additional field inserted between the input fields and the output field. This additional field represents the current state of the UDP and is considered equivalent to the current output value. It is delimited by colons. For example:

```
table
  0 0 0 : 0 : 0;
...
endtable
```

The order of the inputs in the state table description must correspond to the order of the inputs in the port list in the UDP definition header. It is not related to the order of the input declarations. Each row in the table defines the output for a particular combination of input states. If all inputs are specified as x, then the output must be specified as x. All combinations that are not explicitly specified result in a default output state of x.
Consider the following entry from a Sequential UDP state table:

```
0 1 : ? : 1 ;
```

In this entry, the ? represents a don't-care condition. This symbol indicates iterative substitution of 1, 0, and x. The table entry specifies that when the inputs are 0 and 1, the output is 1 irrespective of the value of the current state. You do not have to explicitly specify every possible input combination. All combinations that are not explicitly specified result in a default output state of x. It is illegal to have the same combination of inputs, including edges, specified for different outputs.

Summary of UDP Symbols

Like the ? symbol described in the preceding section, there are several symbols that you can use in UDP definitions to make the description more readable. The following table summarizes the

meaning of all the value symbols that are valid in the table part of a UDP definition.

Table 2-1 UDP Table Symbols

Symbol	Interpretation	Notes
0	Logic 0	
1	Logic 1	
x or X	Unknown	
?	Iteration of 0, 1, and x	Cannot be used in output field
b or B	Iteration of 0 and 1	Like ?, except x is excluded Cannot be used in output field
-	No change	Can only be used in output field of a sequential UDP
(vw)	Value change from v to w	v and w can be any one of: 0, 1, x, ?, or b
*	Same as ??	Any value change on input
r or R	Same as 01	Rising edge on input
f or F	Same as 10	Falling edge on input
p or P	Iteration of (01), (0x), and (x1)	Positive edge on input
n or N	Iteration of (10), (1x), and (x0)	Negative edge on input

Combinational UDPs

In combinational UDPs, the output state is determined solely as a function of the current input states. Whenever an input changes state, the UDP is evaluated and one of the state table rows is matched. The output state is set to the value indicated by that row. All combinations of the inputs that are not explicitly specified drive the output to the unknown value x.

Level-Sensitive Sequential UDPs

Level-sensitive sequential UDPs model latches. Level-sensitive sequential UDP behavior is represented the same way as combinational UDP behavior, except that the output is declared to be of type reg, and there is an additional field in each table entry. This new field represents the current state of the UDP. The output field in a sequential UDP represents the next state.
The description of a level-sensitive sequential UDP differs from a combinational UDP in two ways:

- The output identifier q has an additional reg declaration to indicate that there is an internal state q. The output value of the UDP is always the same as the internal state.
- A field for the current state has been added. This field is separated by colons from the inputs and the output.

Edge-Sensitive UDPs

Edge-sensitive sequential UDPs model latches and flip-flops. In level-sensitive UDP behavior, the values of the inputs and the current state are sufficient to determine the output value. Edge-sensitive behavior differs in that changes in the output are triggered by specific transitions of the inputs. This makes the state table a transition table.
The previous example has terms like (01) in the input fields. These terms represent transitions of the input values. Specifically, (01) represents a transition from 0 to 1. The first line in the table can be interpreted as follows: when clock changes value from 0 to 1 and data equals 0, the output goes to 0 no matter what the current state is.
Note: Each table entry can have a transition specification on only one input. Entries such as the one shown below are illegal:

```
(01)(01)0 : 0 : 1;
```

As in the combinational and the level-sensitive entries, a ? implies iteration of the entry over the values 0, 1, and x. A dash (-) in the output column indicates no value change.
If the UDP is sensitive to edges of any input, the desired output state must be specified for all edges of all inputs.

For synthesis semantics analysis, UDPs must be completely specified for all edges of all inputs. For unspecified combinations, the output is assumed to hold previous output value.

Mixing Level-Sensitive and Edge-Sensitive Descriptions

UDP definitions allow a mixing of level-sensitive and edge-sensitive constructs in the same description. The following example, which shows an edge-triggered JK flip-flop with asynchronous preset and clear, illustrates this concept:
Example of Sequential UDP for Level-Sensitive and Edge-Sensitive Behavior

```
primitive jk_edge_ff(q, clock, j, k, preset, clear);
        output q; reg q;
        input clock, j, k, preset, clear;
        table
    //clock  jk pc  state  output/next state

        ?   ?? 01 : ? :   1  ;       //preset logic
        ?   ?? *1 : 1 :   1  ;
        ?   ?? 10 : ? :   0  ;       //clear logic
        ?   ?? 1* : 0 :   0  ;
        r   00 00 : 0 :   1  ;       //normal clocking cases
        r   00 11 : ? :   -  ;
        r   01 11 : ? :   0  ;
        r   10 11 : ? :   1  ;
        r   11 11 : 0 :   1  ;
        r   11 11 : 1 :   0  ;
        f   ?? ?? : ? :   -  ;
        b   *? ?? : ? :   -  ;       //j and k transition cases
        b   ?* ?? : ? :   -  ;
        endtable
endprimitive
```

In this example, the preset and clear logic is level-sensitive. Whenever the preset and clear combination is 01, the output has value 1. Similarly, whenever the preset and clear combination has value 10, the output has value 0.
The remaining logic is sensitive to edges of the clock. In the normal clocking cases, the flip-flop is sensitive to the rising clock edge as indicated by an r in the clock field in those entries. The insensitivity to the falling edge of clock is indicated by a hyphen (-) in the output field (see UDP Table Symbols table) for the entry with an f as the value of clock. Remember, you must specify the desired output for this input transition to avoid unwanted x values at the output. The last two entries show that the transitions in j and k inputs do not change the output on a steady low or high clock.

Level-Sensitive Dominance

In some cases, an edge-sensitive and a level-sensitive table entry may conflict with each other. In these cases, the general rule is that when the input and current state conditions of a level-sensitive table entry and an edge-sensitive table entry specify conflicting next states, the level-sensitive entry dominates the edge-sensitive entry.
The following table shows a level-sensitive table entry and an edge-sensitive entry from the Example of Sequential UDP for Level-Sensitive and Edge-Sensitive Behavior figure. The column on the right shows a case that is included by the table entry.

Conflicting level-sensitive and edge-sensitive entries

Behavior	Table entry	Included case
Level-sensitive	? ?? 01 : ? : 1 ;	0 00 01 : 0 : 1 ;
Edge-sensitive	f ?? ?? : ? : - ;	f 00 01 : 0 : 0 ;

The included cases specify opposite next state values for the same input and current state

combination. The level-sensitive case specifies that when the inputs clock, jk and pc are 0 00 01, and the current state is 0, the output changes to 1. The edge-sensitive case specifies that when clock falls from 1 to 0, and the other inputs jk and pc are 00 01, and the current state is 0, the output changes to 0. In this example, the level-sensitive entry dominates, and the output changes to 1.

Reducing Pessimism

Three-valued logic tends to make pessimistic estimates of the output when one or more inputs are unknown. You can use UDPs to reduce this pessimism. Consider the following model of a latch:

```
primitive latch(q, clock, data);
        output q;
        reg q ;
        input clock, data;
        table
    // clock  data state output/next state
        0     1  : ? :   1   ;
        0     0  : ? :   0   ;
        1     ?  : ? :   -   ;                  // - = no change
        x     0  : 0 :   -   ;                  // ignore x on clock when data equals state
        x     1  : 1 :   -   ;
        endtable
endprimitive
```

The last two entries specify what happens when the clock input has value x. If these are omitted, the output q will go to x whenever the clock is x. This is a pessimistic model, as the latch should not change its output if it is already 0 and the data input is 0. This is also true when the data input is 1 and the current output is 1.
Consider the JK flip-flop with preset and clear in the following example. This example has additional entries for the positive clock (p) edges, the negative clock edges (?0 and 1x), and with the clock value x. In all of these situations, the output remains unchanged rather than going to x. Thus, this model is less pessimistic than the latch example given earlier.

Example: UDP for a JK flip-flop with preset and clear

```
primitive jk_edge_ff(q, clock, j, k, preset, clear);
        output q; reg q;
        input clock, j, k, preset, clear;
        table
   // clock jk pc state output/next state
        ?   ?? 01 : ? :   1 ;        // preset logic
        ?   ?? *1 : 1 :   1 ;
        ?   ?? 10 : ? :   0 ;        // clear logic
        ?   ?? 1* : 0 :   0 ;
        r   00 00 : 0 :   1 ;        // normal clocking cases
        r   00 11 : ? :   - ;
        r   01 11 : ? :   0 ;
        r   10 11 : ? :   1 ;
        r   11 11 : 0 :   1 ;
        r   11 11 : 1 :   0 ;
        f   ?? ?? : ? :   - ;
        b   *? ?? : ? :   - ;        // j and k cases
        b   ?* ?? : ? :   - ;
        p   00 11 : ? :   - ;        // cases reducing pessimism
        p   0? 1? : 0 :   - ;
        p   ?0 ?1 : 1 :   - ;
        (?0)?? ?? : ? :   - ;
        (1x)00 11 : ? :   - ;
        (1x)0? 1? : 0 :   - ;
```

```
        (1x)?0 ?1 : 1 :   - ;
        x   *0 ?1 : 1 :   - ;
        x   0* 1? : 0 :   - ;
        endtable
endprimitive
```

UDP Instances

You specify instances of user-defined primitives inside modules in the same manner as gate and switch primitives. The instance name is optional except when the instance is declared as an array.

Syntax

udp_instantiation ::= udp_identifier udp_instance {, udp_instance };

udp_instance ::=
[name_of_udp_instance]
(output_port_connection,
input_port_connection [, input_port_connection})

The port order is as specified in this syntax definition. Only two delays (rising and falling) can be specified, because z is not supported for UDPs. An optional
range may be specified for an array of UDP instances. The port connection rules are the same as outlined in "Port Connection Rules".

8. Behavioral Modeling

All procedures in Verilog are specified within one of the following four statements:

always statement
initial statement
task
function

Tasks and functions are procedures that are enabled from one or more places in other procedures. Tasks and functions are covered in "Tasks and Functions".
The initial and always statements are enabled at the beginning of simulation. The initial statement executes only once and its activity dies when the statement has finished. The always statement executes repeatedly. Its activity dies only when the simulation is terminated. There is no limit to the number of initial and always blocks that can be defined in a module.

always Statement

Each always statement repeats continuously throughout the whole simulation run.

Syntax

always_construct ::= always statement

The always statement, because of its looping nature, is useful only when used in conjunction with some form of step control.
If the sensitivity list of an always statement is not complete, then a latch is inferred.

initial Statement

An initial statement is similar to an always statement, except that it is executed only once.

Syntax

initial_construct ::= initial statement

If the initial statement is initializing a combinational register, the initial value is ignored. If the initial statement is initializing a sequential register, the initial value is initialized at time 0.
The initial statement supports only blocking and non-blocking assignments.

Procedural Assignments

As described in "Assignments", procedural assignments are for updating reg, integer, and memory variables. There is a significant difference between procedural assignments and continuous assignments, as described below:

- Continuous assignments drive net variables and are evaluated and updated whenever an input operand changes value.
- Procedural assignments update the value of register variables under the control of the procedural flow constructs that surround them.

The right-hand side of a procedural assignment can be any expression that evaluates to a value. However, part-selects on the right-hand side must have constant indexes. The left-hand side of the procedural assignment indicates the variable that receives the assignment from the right-hand side.
The left-hand side of a procedural assignment can take one of the following forms:

- Register or integer variable; that is, an assignment to the name reference of one of these data types.
- Bit-select of a register or integer variable; that is, an assignment to a single bit that leaves the other bits untouched.
- Part-select of a register or integer variable; that is, a part-select of two or more contiguous bits that leaves the rest of the bits untouched. For the part-select form, only constant expressions are legal.
- Memory element; that is, a single word of a memory. Note: Bit-selects and part-selects are illegal on memory element references.
- concatenation of any of the above
- A concatenation of any of the previous four forms, which effectively partitions the result of the right-hand side expression and assigns the partition parts to the various parts of the concatenation.

Note: Assignment to a register or time variable does not sign-extend. Assignment to a register differs from assignment to an integer variable when the right-hand side evaluates to fewer bits than the left-hand side. Registers are unsigned; if you assign a register to an integer, the variable does not sign-extend.

Blocking and Non-blocking Procedural Assignments

A blocking procedural assignment statement is executed before the execution of the statements that follow it in a sequential block. The non-blocking procedural assignment allows assignment scheduling without blocking the procedural flow.

Syntax

blocking_assignment ::= reg_lvalue = [event_control] expression
non_blocking_assignment ::= reg_lvalue <= [event_control] expression

Procedural Continuous Assignments

The assign and deassign procedural assignment statements allow continuous assignments to be placed onto registers for controlled periods of time. The assign and deassign procedural statements allow, for example, modeling of asynchronous clear/preset on a D-type edge-triggered flip-flop, where the clock is inhibited when the clear or preset is active.

Syntax

procedural_continuous_assignments ::= assign reg_assignment; | deassign reg_lvalue;

reg_assignment::= reg_lvalue = expression

The left-hand side of the assignment in the assign statement can be a register reference or a concatenation of registers. The assign statement overrides all procedural assignments to a register. The deassign statement ends a procedural continuous assignment to a register. The value of the register remains the same until the register is assigned a new value through a procedural assignment or a procedural continuous assignment.
If the keyword assign is applied to a register for which there is already a procedural continuous assignment, then this new procedural continuous assignment deassigns the register before making the new procedural continuous assignment.

Conditional Statements

Conditional statements are used to conditionally execute one of the many sets of statements based on specified conditions.

Syntax

if_else_statement ::= if (expression) statement_or_null [else statement_or_null]

if_else_if_statement ::= if (expression) statement_or_null
{ else if (expression) statement_or_null }
else statement

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

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

statement_or_null ::= statement |;

Each conditional statement must conform to one or both of the following rules:
- The conditional statement must be in a sequential (begin-end) procedural block.
- The conditional statement must include an else statement.

The expressions are evaluated in order. If any expression is true, the statement set associated with it is executed, and the whole conditional chain is terminated.
The case statement is a special multi-way decision statement that tests whether an expression matches one of several other expressions, and branches accordingly. The default statement is optional. Using multiple default statements in one case statement is illegal syntax.

Your model may be inferred as a latch in the following conditions:
- Incomplete case statement. In the following example, all branches of the case statement have not been defined. Hence a latch is inferred for cond.

```
module L1(in1,in2,cond,out1);
        input [1:0] in1,in2,cond;
        output [1:0] out1;
        reg [1:0] out1;
        always @(in1 or in2 or cond)
                begin
                        case(cond)
                                2'b00 : out1 = in1 & in2;
                                2'b11 : out1 = in1 | in2;
                                2'b00 : out1 = in1 ^ in2;
                        endcase
                end
endmodule
```

Incomplete assignment of signals in a case statement. Consider the following example:

```
module L2(in1,in2,cond,out1,out2);
        input [1:0] in1,in2,cond;
        output [1:0] out1,out2;
        reg [1:0] out1,out2;
        always @(cond or in1 or in2)
                begin
                        case(cond)
                                2'b00 : out1 = in1 & in2;
                                2'b01 : out2 = in1 & in2;
                                2'b11 : out1 = in1 | in2;
                                2'b10 : out2 = in1 ^ in2;
                        endcase
                end
endmodule
```

While all possible branches of the case statement have been enumerated, signals out1 and out2 are not assigned in all branches. Hence a latch is inferred for out1 and out2.

Incomplete If construct. In the following example, the if construct is not completely defined. Hence a latch is inferred for cond.

```
module L3(cond,in1,out1);
        input cond;
        input [1:0] in1;
        output [1:0] out1;
        reg [1:0] out1;
        always @(in1 or cond)
                if(cond)
                        out1 = in1;
endmodule
```

Looping Statements

There are three types of looping statements that provide a means of controlling the execution of a statement, either zero, one, or more times.

Repeat executes a statement a fixed number of times.

While executes a statement until an expression becomes false. If the expression starts out false, the statement is not executed at all.

for controls execution of its associated statement(s) by a three-step process, as follows:

a. Executes an assignment, normally used to initialize a variable, that controls the number of loops executed.

b. Evaluates an expression--if the result is zero, the for loop exits. If it is not zero, the for loop executes its associated statement(s) and then performs step c.

c. Executes an assignment, normally used to modify the value of the loop-control variable, then repeats step b.

Syntax

```
loop_statement ::=
     repeat ( expression ) statement
   | while (expression ) statement
   | for ( reg_assignment ; expression ; reg_assignment ) statement
```

Note: Data dependent loops are not supported. Hence, the repeat statement in the following example is illegal:

```
module m1(in1, ...)
input in1;
```

```
...
repeat (in1)
        ...
...
```

Procedural Event Control

The execution of a procedural statement can be synchronized with a value change on a net or register or with the occurrence of a declared event.

Syntax

```
event_control ::= @ (event_expression)

event_expression ::= expression
                   | posedge expression
                   | negedge expression
                   | event_expression or event_expression
```

Sequential Blocks

The sequential block statements group two or more statements together so that they act syntactically like a single statement. In sequential blocks, the statements are executed one after another.
A sequential block has the following characteristics:

- Statements execute in sequence, one after another.
- Control passes out of the block after the last statement executes.

Blocks can be named by adding block_identifier after the keyword begin. Thus, you can declare local variables for the block.
In the Verilog language, all variables are static--that is, a unique location exists for all variables and leaving or entering blocks does not affect the values stored in them. Thus, block names give a means of uniquely identifying all variables at any simulation time.

Syntax

```
seq_block ::=
   begin
   [ : block_identifier { block_item_declaraction } ] { statement }
   end

block_item_declaration ::= parameter_declaration | reg_declaration | integer_declaration
```

9. Tasks and Functions

Tasks and functions provide the ability to execute common procedures at several different places in a description. They also provide a means of breaking up large procedures into smaller ones to make the code easier to read and to debug the source descriptions. The input, output, and inout argument values can be passed into and out of both tasks and functions.

Distinctions Between Functions and Tasks

The following rules distinguish functions from tasks:

- A function cannot enable a task; a task can enable other tasks and functions.
- A function must have at least one input argument; a task can have zero or more arguments of any type.
- A function returns a single value; a task does not return a value.

The purpose of a function is to respond to an input value by returning a single value. A task can support multiple goals and can calculate multiple result values. However, only the output or inout

arguments can pass result values back from the invocation of a task. A model uses a function as an operand in an expression; the value of that operand is the value returned by the function.

Tasks

A task is enabled by the statement that defines the argument values to be passed to the task, and by the variables that will receive the results. Control is passed back to the enabling process after the task is complete. A task can enable other tasks, which in turn can enable still other tasks--with no limit on the number of tasks enabled. Regardless of how many tasks have been enabled, control does not return until all enabled tasks are complete.

Syntax

Task Declaration

```
task_declaration ::=
        task task_identifier;
           { task_item_declaration }
           statement_or_null
        endtask

task_item_declaration ::=
   block_item_declaration
   | input_declaration
   | output_declaration
   | inout_declaration

block_item_declaration ::= parameter_declaration | reg_declaration | integer_declaration
statement_or_null ::= statement |;
```

Task Enabling

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

If an argument in the task is declared as input, then the corresponding expression can be any expression. If the argument is declared as an output or inout, then the expression must be valid on the left-hand side of a procedural assignment.

Functions

A function returns a value that is to be used in an expression. Functions have at least one input argument and no output arguments. They return a single value. Functions are more limited than tasks. The following rules govern their usage:

- A function cannot enable tasks.
- A function definition must contain at least one input argument.
- A function definition must include an assignment of the function result value to the internal variable that has the same name as the function.
- A function definition cannot contain an inout declaration or an output declaration.

Syntax

Function Declaration

```
function_declaration ::=
        function [ range_or_type ] function_identifier;
           function_item_declaration { function_item_declaration }
           statement
        endfunction

range_or_type ::= range | integer
```

function_item_declaration ::= input_declaration | block_item_declaration
block_item_declaration ::= parameter_declaration | reg_declaration | integer_declaration

Function Calling

function_call ::= function_identifier [(expression {, expression })];

The function definition implicitly declares a register, internal to the function, with the same name as the function. This register either defaults to one bit or is the type that range_or_type specifies. The range_or_type can specify that the function's return value is an integer, or a value with a range of [n:m] bits. The function assigns its return value to the internal variable bearing the function's name.
A function call is an operand within an expression.

10. Hierarchical Structures

The Verilog HDL supports a hierarchical hardware description structure by allowing modules to be embedded within other modules. Higher-level modules create instances of lower-level modules and communicate with them through input, output, and bidirectional ports. These module ports can be scalar or vector.
To describe a hierarchy of modules, the design must have textual definitions of the various modules. Each module definition stands alone; the definitions are not nested. Statements within the module definitions create instances of other modules, thus describing the hierarchy.

Module Declaration

module_declaration ::=
        module | macromodule module_identifier [list_of_ports] ;
            { module_item }
        endmodule

list_of_ports ::= (port {, port })
port ::= [port_expression] | .port_identifier ([port_expression])
port_expression ::= port_reference | { port_reference {, port_reference } }

module_item ::=
     module_item_declaration
   | parameter_override
   | continuous_assign
   | gate_instantiation
   | udp_instantiation
   | initial_construct
   | always_construct

module_item_declaration ::=
     parameter_declaration
   | input_declaration
   | output_declaration
   | inout_declaration
   | net_declaration
   | reg_declaration
   | integer_declaration
   | task_declaration
   | function_declaration

Module Instantiation

module_instantiation ::=
   module_identifier [parameter_value_assignment]

module_instance {, module_instance };

parameter_value_assignment ::= #(expression {, expresion })

module_instance ::= name_of_instance ([list_of_module_connections])
name_of_instance ::= module_instance_identifier [range]

list_of_module_connections ::=
ordered_port_connection {, ordered_port_connection }
| named_port_connection {, named_port_connection }

ordered_port_connection ::= [expression]

Overriding Module Parameter Values

When one module instantiates another module, it can alter the values of any parameters declared within the instantiated module in the following two ways:
1. Through the defparam statement that allows assignment to the parameters using their hierarchical names
2. Through module instance parameter value assignment that allows values to be assigned inline during module instantiation.

defparam statement

The defparam statement changes the value of any parameter in any module instance throughout the design. Specify the value of the parameter by using the hierarchical name of the parameter.

parameter_override ::= defparam list_of_param_assignments;
list_of_param_assignments ::= param_assignment {, param_assignment }
param_assignment ::= parameter_identifier = constant_expression

Module Instance Parameter Value Assignment

The alternative method of assigning values to parameters is by supplying values to all or some parameters of particular module instances. The order of values must be the same as the order of module parameters. It is not necessary to supply values for all parameters. However, all parameters must be specified.
Note: If a parameter has been defined with an expression containing another parameter, overriding the value of the second parameter also overrides the value of the first parameter.

Macromodules

A macromodule serves the same purpose as a standard module but has certain limitations. Hence, it uses significantly lower memory during compilation. Macromodules are normally used for simple modules that are frequently used in a design. A macromodule can contain only the following constructs:
- gate and switch instances
- user-defined primitive instances
- nets

The following restrictions apply to the constructs used in macromodules:
- The terminal lists in gate instances and the port lists in UDP instances cannot contain expressions with variable operands (such as those used in dynamic bit selects).
- Procedural statements and register declarations are illegal in macro modules. If these are present, then the macromodule is treated as a normal module.
- If there are part-selects or concatenations in the port connections, then the macromodule instance is treated as a normal module.

You can specify macromodules by using the keyword macromodule in place of the keyword module in the module definition. Instances of macromodules are specified in exactly the same way as instances of normal modules.

Ports

Ports interconnect the hardware description consisting of modules, primitives, and macromodules. For example, module A can instantiate module B, using port connections appropriate to module A. These port names can differ from the names of internal nets and registers specified in the definition of module B.

Syntax

port ::= [port_expression] | .port_identifier ([port_expression])
port_expression ::= port_reference | { port_reference {, port_reference } }

port_reference ::=
    port_identifier
  | port_identifier [constant_expression]
  | port_identifier [msb_constant_expression : lsb_constant_expression]

The port_expression syntax item in the port definition can be one of the following:
- a simple identifier
- a bit-select of a vector declared within the module
- a part-select of a vector declared within the module a concatenation of any of the above

Bit-selects and part-selects result in the automatic expansion of the vector nets they reference. Note that the port_expression is optional because ports can be defined that do not connect to anything internal to the module.

Port Declarations

Each port listed in the module definition's list_of_ports must be declared in the body of the module as an input, output, or bidirectional inout. This is in addition to any other declaration for a particular port-- for example, a net, a reg, or a wire.

input_declaration ::= input [range] list_of_port_identifiers;
output_declaration ::= output [range] list_of_port_identifiers;
inout_declaration ::= inout [range] list_of_port_identifiers;
list_of_port_identifiers ::= port_identifier {, port_identifier }

Connecting Module Ports

To connect the ports listed in a module instantiation and the ports defined by the instantiated module, use any of the following methods:
- Connecting module ports by the ordered list
- The ports listed for the module instance are in the same order as the ports listed in the module definition.
- Connecting module ports by name
- Explicitly link the two names for each side of the connection--the name used in the module definition, followed by the terminal used in the instantiating module.

Note: The two types of module port connections cannot be mixed; connections to the ports of a particular module instance must be made either all by position or all by name.
Note: A port that is declared as input but is used as an output, and vice versa, is coerced to inout.

Port Connection Rules

The following rules govern the ways module ports are declared and the ways they are interconnected:

1. An input or inout port must be declared as a net type.
2. Each port connection is a continuous assignment of source to sink--that is, where one connected item is a signal source and the other is a signal sink.
   - Only nets are permitted to be the sinks in an assignment.

- Both scalar and vector nets are permitted. The output ports of a module are by definition connected to signal source items internal to the module.

3. The following external items cannot be connected to the output or inout ports of modules:
   - Registers
   - Expressions other than a scalar net, a vector net, a constant bit-select of a vector net, a part-select of a vector net and a concatenation of the expressions listed above

Net Types Resulting from Dissimilar Port Connections

When the two nets connected by a port are of different net types, the resulting single net is assigned one of the following:
- The dominating net type, if one of the two nets is "dominating"
- The net type external to the module

The following table shows the resultant net type as a result of collapsing a module port that connects two nets.

external net:	wire & tri	wand & triand	wor & trior	tri0	tri1	supply0	supply1
Internal net:							
wire & tri	ext	ext	ext	ext	ext	ext	ext
wand & triand	int	ext	warn	warn	warn	ext	ext
wor & trior	int	warn	ext	warn	warn	ext	ext
tri0	int	warn	warn	ext	warn	ext	ext
tri1	int	warn	warn	warn	ext	ext	ext
supply0	int	int	int	int	int	ext	warn
supply1	int	int	int	int	int	warn	ext

ext indicates the external net is used for merging, int indicates that the internal net is used for merging, and warn indicates that a warning is issued and the external net type is used for merging.

Hierarchical Names

Every identifier in a Verilog description has a unique hierarchical path name. The hierarchy of modules and the definition of items such as tasks and named blocks within modules define these path names. The hierarchy of names can be viewed as a tree structure, in which each module instance, task, function, or named begin-end block defines a new hierarchical level, or scope. At the top of the scope are the names of modules for which no instances have been created. The top of the scope is the root of the hierarchy. Inside any module, each module instance, task definition, function definition, and named begin-end block defines a new branch of the hierarchy. Named blocks within named blocks and within tasks and functions also create new branches.

Each node in the hierarchical name tree is a separate scope with respect to identifiers. A particular identifier can be declared, at most, once in any scope.

You can reference any named Verilog object uniquely in its full form by concatenating the names of the modules, tasks, functions, or blocks that contain it. Use the period character to separate each of the names in the hierarchy. For example, wave.a.bmod.keep.hold shows five levels. The complete path name to any object starts at a top-level module. You can use this path name from any level in the description. The first node name in this path name (wave, in the example) can also be the top of a hierarchy that starts at the level in which the path is being used.

Upwards Name Referencing

The name of a module is sufficient to identify the module and its location in the hierarchy. A lower-level module can reference items in a module above it in the hierarchy if the name of the higher-level module is known.

Syntax

<name_of_module>.<name_of_item>

There can be no spaces within the reference.

11. System Tasks

System tasks perform often repeated routine operations. They are available only in the simulation context.

Syntax

System Task Definition

```
system_task ::= $system_task_identifier [ ( list_of_arguments ) ]
list_of_arguments ::= argument {, [ argument ] }
argument ::= expression
```

System Task Enabling

```
system_task_enable ::=
   system_task_name [ ( system_task_expression {, system_task_expression } ) ] ;

system_task_name ::=$identifier
```

Load Memory System Tasks

The $readmemb and $readmemh system tasks read a specified text file and load its contents into a specified memory.

Syntax

```
load_memory_tasks ::=
   $readmemb | $readmemh
   ("file_name", memory_name [, start_addr [, finish_addr]]);
```

The text file shall contain only these types of data:
- White space
- Comments
- Binary ($readmemb) or hexadecimal ($readmemh) numbers

You can use the unknown value (x or X), high impedance value (z or Z) and the underscore character (_) to specify a number. The numbers shall be separated by white space or comments.

12. Compiler Directives

Compiler directives control the compilation of HDL source.

'define Compiler Directive

The 'define compiler directive allows text substitution at compilation time. You can use this directive both inside and outside of a module definition construct.

Syntax

Text Macro Definition

text_macro_definition ::= 'define text_macro_name macro_text
text_macro_name ::= text_macro_identifier [(list_of_formal_arguments)]
list_of_formal_arguments ::= formal_argument_identifier { , formal_argument_identifier }

Text Macro Usage

text_macro_usage ::= 'text_macro_identifier [(list_of actual_arguments)]
list_of_actual_arguments ::= actual_argument { , actual_argument }
actual_argument ::= expression

'undef Compiler Directive

The 'undef compiler directive undefines a previously defined text macro.

Syntax

undefine_compiler_directive ::= 'undef text_macro_name

'ifdef, 'else and 'endif Compiler Directives

These compiler directives allow you to include lines of code that are optionally compiled. The 'ifdef compiler directive checks for the definition of a variable name. If the variable name is defined, the lines following the 'ifdef directive are included. If the variable name is not defined and an 'else directive exists, the line following the 'else directive are included. The 'endif directive closes the conditional compilation definition.

Syntax

```
conditional_compilation_directive ::=
   'ifdef text_macro_name
   first_group_of_lines
   [ 'else
   second_group_of_lines
   'ifdef ]
```

13. Pragmas

Pragmas are directives that control the cycle interpretation of the HDL source. They are inserted as meta-comments in the HDL source.

The Translation Control Pragmas

The translation control pragmas include or exclude a part of the HDL source from processing. These pragmas are always used in pairs and you can use them anywhere in the HDL description. You can use these pragmas on complete statements only. You can not specify these pragmas as a part of an expression.
Note: The code between the pragma pair is still checked for syntax correctness.

synthesis_off and synthesis_on Pragmas

The synthesis_off and synthesis_on pragmas instruct the tool to stop and start processing the code respectively. These pragmas are normally used to hide simulation-only constructs from the synthesis. In the following example, the status monitoring code has been hidden from synthesis using the synthesis_off and synthesis_on pragmas:

```
module DFF (data, q, clock, set, reset)
        input data, clock, set, reset;
        output q;
        reg q;
        always@(...)
        ...
        //<synthesis_off_directive>
        always@(reset or set)
                if (reset ='1' and set='1') then
                        $write("Error: Both set and reset are 1.")
        //<synthesis_on_directive>
endmodule
```

Note: The translate_off and translate_on pragmas also work as the synthesis_off and synthesis_on pragmas.

verification_on and verification_off Pragmas

The verification_on and verification_off pragmas have similar functions as the synthesis_off and synthesis_on pragmas but are always nested within a pair of the synthesis_off and synthesis_on pragmas. They also instruct the tool to start and stop processing the code respectively. The verification_on and verification_off pragmas are normally used to segment out methodology-specific information in the HDL source. In the following example, the code between the synthesis_off and synthesis_on pragmas is not synthesized. The code between the verification_on and verification_off pragmas is used by the tools for methodology verification.

```
...
//<synthesis_off_directive>
//simulation-specific code
//<verification_on_directive>
cycle C1(cl, reset, num, max);
fc_edge_detector clock (.fc_signal(ck),.fc_risingedge(ck_rising),
        .fc_fallingedge(ck_falling));
...
//<verification_off_directive>
//<synthesis_on_directive>
...
```

The Case Decoding Pragmas

The case decoding pragmas control the synthesis of case statements. You can insert the pragma directives in a case statement wherever a meta-comment can be legally inserted. You can insert multiple pragma directives in one case statement.

full_case Pragma

The full_case pragma forces the tool to assume that the associated case statement is complete even when all branches may not be covered. By using of this pragma, you need not write the default logic and avoid latch inference as a latch may be inferred if you do not assign values of a variable for all possible conditions. For example,

```
reg [1:0] in, out
```

```
case (in) // <full_case_pragma_directive>
        0: out= 1;
        1: out= 0;
        3: out= 1;
endcase
```

Here the out value for in == 2 is not defined. Also the default value is not defined. Normally this construct is inferred as a latch. When you use the full_case pragma directive, this construct is assumed to be complete that is, all possible values are assumed to have been enumerated. However, the full_case pragma directive cannot avoid latch inference in some cases. For example:

```
reg a, b;
reg [1:0] c;
case (c)   // <full_case_pragma_directive>
        0: begin a = 1; b = 0; end
        1: begin a = 0; b = 0; end
        2: begin a = 1; b = 1; end
        3: b = 1;               // a is not assigned here
endcase
```

In this example, all possible branches of the case statement have been enumerated. But a is not assigned when c==3. Hence, the full_case pragma directive prevents a latch for b but not for a.

parallel_case Pragma

The parallel_case pragma forces the tool to assume that the branches of the associated case statement are mutually exclusive. For example, the generated logic is very complex for the following case statement:

```
reg [3:0] current_state, next_state;
parameter state1 = 4'b0001,
                state2 = 4'b0010,
                state3 = 4'b0100,
                state4 = 4'b1000;
case (1)
        current_state[0] : next_state = state2;
        current_state[1] : next_state = state3;
        current_state[2] : next_state = state4;
        current_state[3] : next_state = state1;
endcase
```

Instead, you may use the parallel_case pragma directive as follows to simplify the generated logic:

```
case (1) // <parallel_case_directive>
        current_state[0] : next_state = state2;
        current_state[1] : next_state = state3;
        current_state[2] : next_state = state4;
        current_state[3] : next_state = state1;
endcase
```