Verilog® HDL

Quick Reference Guide

based on the Verilog-2001 standard
(IEEE Std 1364-2001)

by
Stuart Sutherland

SUTHERLAND HDL

www.sutherland-hdl.com

Verilog® HDL
Quick Reference Guide

based on the Verilog-2001 standard
(IEEE Std 1364-2001)

by
Stuart Sutherland

published by

SUTHERLAND
HDL

Sutherland HDL, Inc.
22805 SW 92nd Place
Tualatin, OR 97062
(503) 692-0898

www.sutherland-hdl.com

Sutherland HDL, Inc.
22805 SW 92nd Place
Tualatin, OR 97062-7225

Phone: (503) 692-0898
URL: www.sutherland-hdl.com

ISBN: 1-930368-03-8

Table of Contents

1.0 New Features In Verilog-2001

Verilog-2001, officially the “IEEE 1364-2001 Verilog Hardware Description Language”, adds several significant enhancements to the Verilog-1995 standard.

2.0 Reserved Keywords

```
always
and
assign
automatic†
begin
buf
bufif0
bufif1
case
casex
casez
cell†
cmos
config†
deassign
default
defparam
design†
disable
edge
else
end
endcase
endconfig†
endfunction
endgenerate†
endmodule
endprimitive
endspecify
endtable
endtask
event
for
force
forever
fork
function
generate†
genvar†
highz0
highz1
if
ifnone
initial
instance†
inout
input
integer
join
large
liblist†
localparam†
macromodule
medium
module
nand
negedge
nmos
nor
not
noshowcancelled†
notif0
notif1
or
output
parameter
pmos
posedge
primitive
pull0
pull1
pulldown
pullup
pulsestyle_onevent†
pulsestyle_ondetect†
rcmos
real
realtime
reg
release
repeat
rnmos
rpmos
rtran
rtranif0
rtranif1
scalared
signed
showcancelled†
small
specify
specparam
strength
strong0
strong1
supply0
supply1
table
task
time
tran
tranif0
tranif1
tri
tri0
tri1
triand
trior
trireg
unsigned
use†
vectored
wait
wand
weak0
weak1
while
wire
wor
xnor
xor
```

3.0 Concurrency

The following Verilog HDL constructs are independent processes that are evaluated concurrently in simulation time:

- module instances
- primitive instances
- continuous assignments
- procedural blocks

4.0 Lexical Conventions

4.1 Case Sensitivity

Verilog is case sensitive.

4.2 White Space Characters

blanks, *tabs*, *newlines* (carriage return), and *formfeeds*.

4.3 Comments

`//` begins a single line comment, terminated by a *newline*.

`/*` begins a multi-line block comment, terminated by a `*/`.

4.4 Attributes

`(*` begins an attribute, terminated by a `*)`.

- An attribute specifies special properties of a Verilog object or statement, for use by specific software tools, such as synthesis. Attributes were added in Verilog-2001.
- An attribute can appear as a prefix to a declaration, module items, statements, or port connections.
- An attribute can appear as a suffix to an operator or a call to a function.
- An attribute may be assigned a value. If no value is specified, the default value is 1.
- Multiple attributes can be specified as a comma-separated list.
- There are no standard attributes in the Verilog-2001 standard; Software tools or other standards will define attributes as needed.

Attribute Example

`(* full_case, parallel_case *) case (state)`
`  ...`
`endcase`

`assign sum = a + (* CLA=1 *) b;`

† indicates new reserved words that were added in the Verilog-2001 standard.

4.5 Identifiers (names)

- Must begin with alphabetic or underscore characters **a-z A-Z** _
- May contain the characters **a-z A-Z 0-9** _ and **$**
- May use any character by escaping with a backslash (\ ) at the beginning of the identifier, and terminating with a white space.
- Identifiers created by an array of instances or a generate block may also contain the characters **[** and **]**.

Examples	Notes
`adder`	legal identifier name
`XOR`	uppercase identifier is unique from **xor** keyword
`\reset-`	an escaped identifier (must be followed by a white space)

4.6 Hierarchical Path Names

A net, variable, task or function can be referenced anywhere in the design hierarchy using either a full or relative hierarchy path.

- A full path consists of the top-level module, followed by any number of module instance names down to the object being reference. A period is used to separate each name in the hierarchy path.
- A relative path consists of a module instance name in the current module, followed by any number of module instance names down to the object being referenced. A period is used to separate each name in the hierarchy path.

4.7 Hierarchy Scopes and Name Spaces

There are four primary types of name spaces.

- Global names are visible in all names spaces:
  - Module, primitive and configuration definition names
  - Text macro names (created by `` `define ``). Macro names are only visible from the point of definition on; source code compiled prior to the definition cannot see the macro names.
- Scope names create a new level of hierarchy:
  - module definitions
  - function definitions
  - task definitions
  - named blocks (**begin**—**end** or **fork**—**join**)
- Other name spaces:
  - specify blocks
  - attributes

An identifier name defined within a name space is unique to that space and cannot be defined again within the same space. In general, references to an identifier name within a scope will search first in the local scope, and then search upward through the scope hierarchy up to a module boundary.

4.8 Logic Values

Verilog uses a 4 value logic system for modeling. There are two additional unknown logic values that may occur internal to the simulation, but which cannot be used for modeling.

Logic Value	Description
0	zero, low, or false
1	one, high, or true
z or **Z**	high impedance (tri-stated or floating)
x or **X**	unknown or uninitialized
L	partially unknown; either 0 or Z, but not 1 (internal simulation value only)
H	partially unknown; either 1 or Z, but not 0 (internal simulation value only)

4.9 Logic Strengths

Logic values can have 8 strength levels: 4 driving, 3 capacitive, and high impedance (no strength). A net with multiple drivers can have a combination of strengths, represented as a pair of octal numbers, plus the value (e.g. 65X).

Strength Level	Strength Name	Specification Keyword	Display Mnemonic
7	supply drive	**supply0 supply1**	**Su0 Su1**
6	strong drive	**strong0 strong1**	**St0 St1**
5	pull drive	**pull0 pull1**	**Pu0 Pu1**
4	large capacitive	**large**	**La0 La1**
3	weak drive	**weak0 weak1**	**We0 We1**
2	medium capacitive	**medium**	**Me0 Me1**
1	small capacitive	**small**	**Sm0 Sm1**
0	high impedance	**highz0 highz1**	**HiZ0 HiZ1**

4.10 Literal Real Numbers

value.*value*	decimal notation
base **e** *exponent* *base* **E** *exponent*	scientific notation; there should be no space before and after the **e** or **E** token

- Real numbers are represented in double-precision floating point form.
- There must be a value on either side of the decimal point.
- The value may only contain the characters **0**-**9** and underscore.

Examples	Notes
`0.5`	must have value on both sides of decimal point
`3e4`	3 times 10^4 (30000)
`5.8E-3`	5.8 times 10^{-3} (0.0058)

4.11 Literal Integer Numbers

value	unsized decimal integer
size'base value	sized integer in a specific radix (base)

- *size* (optional) is the number of bits in the number. Unsized integers default to at least 32-bits.
- *'base* represents the radix and sign property of the value. The base and sign characters are *not* case sensitive (e.g. 'b and 'B are equivalent).

Radix	Symbol	Legal Values
unsigned binary	**'b**	**0**, **1**, **x**, **X**, **z**, **Z**, **?**, _
unsigned octal	**'o**	**0**–**7**, **x**, **X**, **z**, **Z**, **?**, _
unsigned decimal	**'d**	**0**–**9**, _
unsigned hexadecimal	**'h**	**0**–**9**, **a**–**f**, **A**–**F**, **x**, **X**, **z**, **Z**, **?**, _
signed binary	**'sb**	**0**, **1**, **x**, **X**, **z**, **Z**, **?**, _
signed octal	**'so**	**0**–**7**, **x**, **X**, **z**, **Z**, **?**, _
signed decimal	**'sd**	**0**–**9**, _
signed hexadecimal	**'sh**	**0**–**9**, **a**–**f**, **A**–**F**, **x**, **X**, **z**, **Z**, **?**, _

- The **?** is another way of representing the **Z** logic value.
- An underscore is ignored (used to enhance readability). The underscore cannot be used as the first character of the value.
- Values are expanded from right to left (lsb to msb).
  - When size is fewer bits than value, the upper bits are truncated.
  - When size is more bits than value, and the left-most bit of value is **0** or **1**, zeros are left-extended to fill the size.
  - When size is more bits than value, and the left-most bit of value is **Z** or **X**, the **Z** or **X** is left-extended to fill the size.
- Signed numbers are interpreted as 2's complement values.
- Specifying a literal number as signed affects operations on the number; it does not affect expanding a value to the specified size of the number.

Examples	Size	Sign	Radix	Binary Equivalent
`10`	unsized	signed	decimal	`0...01010` (32-bits)
`'o7`	unsized	unsigned	octal	`0...00111` (32-bits)
`1'b1`	1 bit	unsigned	binary	`1`
`8'sHc5`	8 bits	signed	hex	`11000101`
`6'hF0`	6 bits	unsigned	hex	`110000` (truncated)
`6'hA`	6 bits	unsigned	hex	`001010` (zero filled)
`6'shA`	6 bits	signed	hex	`001010` (zero filled)
`6'bZ`	6 bits	unsigned	binary	`ZZZZZZ` (Z filled)

5.0 Module Definitions

Verilog HDL models are represented as modules.

ANSI-C Style Port List *(added in Verilog-2001)*

```
module module_name
   #(parameter_declaration, parameter_declaration,... )
   (port_declaration port_name, port_name,...,
    port_declaration port_name, port_name,...);
   module items
endmodule
```

Old Style Port List

```
module module_name (port_name, port_name, ... );
   port_declaration port_name, port_name,...;
   port_declaration port_name, port_name,...;
   module items
endmodule
```

(refer to the next page for the syntax of port declarations)

port_name can be either:

- A simple name, which implicitly connects the port to an internal signal with the same name.
- A name with an explicit internal connection, in the form of **.***port_name***(***signal***)**, which connects the port to an internal signal with a different name, or a bit select, part select, or concatenation of internal signals.

  Note: it is the internal signal name that is given a direction, not the port name.

The keyword **macromodule** is a synonym for **module**.

5.1 Module Items

A module may contain any of the following items:

data_type_declarations	(see section 6.0)
parameter_declarations	(see section 6.3)
module_instances	(see section 7.0)
primitive_instances	(see section 8.0)
generate_blocks	(see section 9.0)
procedural_blocks	(see section 10.0)
continuous_assignments	(see section 11.0)
task_definitions	(see section 13.0)
function_definitions	(see section 14.0)
specify_blocks	(see section 15.0)

- Module items may appear in any order, but port, data_type or parameter declarations must come before the declared name is referenced.
- Module functionality may be represented as:
  - *Behavioral* or *RTL* — modeled with procedural blocks or continuous assignment statements.
  - *Structural* — a netlist of module instances or primitive instances.
  - A mix of behavioral and structural.

5.2 Port Declarations

Combined Declarations *(added in Verilog-2001)*

port_direction data_type **signed** *range port_name*, *port_name*, ... ;

Old Style Declarations

port_direction **signed** *range port_name*, *port_name*, ... ;
data_type_declarations (see section 6.0)

- *port_direction* is declared as:
  - **input** for scalar or vector input ports.
  - **output** for scalar or vector output ports.
  - **inout** for scalar or vector bidirectional ports.
- *data_type* (optional) is any of the types listed in section 6.0, except real. Combined port/data type declarations were added in Verilog-2001.
- **signed** (optional) indicates that values passed through the port are interpreted as 2's complement signed values. If either the port or the data type of the internal signal connected to the port are declared as signed, then both are signed. Signed ports were added in Verilog-2001.
- *range* (optional) is a range from **[***msb* **:***lsb***]** (most-significant-bit to least-significant-bit).
  - If no range is specified, ports are 1-bit wide.
  - The *msb* and *lsb* must be a literal number, a constant, an expression, or a call to a constant function.
  - Either little-endian convention (the *lsb* is the smallest bit number) or big-endian convention (the *lsb* is the largest bit number) may be used.
  - The maximum port size may be limited, but will be at least 256 bits. Most software tools have a limit of 1 million bits.
- Port/data type connection rules:

	input ports	**output ports**	**inout ports**
Module Instance (outside the module)	expression, net or variable types (except real)	net types only	net types only
Module Definition (inside the module)	net types only	net or variable types (except real)	net types only

  - A real variable cannot be directly connected to a port. Real numbers can first be converted to or from 64-bit vectors using the $realtobits and $bitstoreal system tasks.
  - The port range and data type range must be the same (if different, some software tools will use the data type size instead of reporting an error).
  - The port direction must be declared before the data type is declared.

Port Declaration Examples	Notes
`input a,b,sel;`	three scalar (1-bit) ports
`input signed [15:0] a, b;`	two 16-bit ports that pass 2's complement values, little endian convention
`output signed [31:0] result;`	32-bit port; values passed through the port are in 2's complement form
`output reg signed [32:1] sum;`	32-bit port; the internal signal connected to the port is a signed reg data type
`inout [0:15] data_bus;`	big endian convention
`input [15:12] addr;`	msb:lsb may be any integer
`parameter WORD = 32;` `input [WORD-1:0] addr;`	constant expressions may be used in the declaration
`parameter SIZE = 4096;` `input [log2(SIZE)-1:0] addr;`	constant functions may be called in the declaration

6.0 Data Type Declarations

Verilog has two major data type classes:

- Net data types are used to make connections between parts of a design.
  - Nets reflect the value and strength level of the drivers of the net or the capacitance of the net, and do not have a value of their own.
  - Nets have a resolution function, which resolves a final value when there are multiple drivers on the net.
- Variable data types are used as temporary storage of programming data.
  - Variables can only be assigned a value from within an initial procedure, an always procedure, a task or a function.
  - Variables can only store logic values; they cannot store logic strength.
  - Variables are un-initialized at the start of simulation, and will contain a logic X until a value is assigned.

General Rules For Choosing The Correct Data Type Class	
when a signal is driven by a module output, a primitive output, or a continuous assignment	use a ***net*** type
when a signal is assigned a value in a Verilog procedure	use a ***variable*** type

6.1 Net Data Types

net_type **signed** [*range*] #(*delay*) *net_name* [*array*], ... ;
net_type (*drive_strength*) **signed** [*range*] #(*delay*) *net_name* = *continuous_assignment*;
trireg (*capacitance_strength*) **signed** [*range*] #(*delay, decay_time*) *net_name* [*array*], ... ;

Nets are used connect structural components together.

- A net data type must be used when a signal is:
  - Driven by the output of a module instance or primitive instance.
  - Connected to an input or inout port of the module in which it is declared.
  - On the left-hand side of a continuous assignment.
- *net_type* is one of the following keywords:

wire	interconnecting wire; CMOS resolution
wor	wired outputs OR together; ECL resolution
wand	wired outputs AND together; open-collector resolution
supply0	constant logic 0 (supply strength)
supply1	constant logic 1 (supply strength)
tri0	pulls down when tri-stated
tri1	pulls up when tri-stated
tri	same as **wire**
trior	same as **wor**
triand	same as **wand**
trireg	holds last value when tri-stated (capacitance strength)

- **signed** (optional) indicates that values are interpreted as 2's complement signed values. If either a port or the net connected to the port is declared as signed, then both are signed. Signed nets were added in Verilog-2001.
- **[***range***]** (optional) is a range from **[***msb* **:***lsb***]** (most-significant-bit to least-significant-bit).
  - If no range is specified, the nets are 1-bit wide.
  - The *msb* and *lsb* must be a literal number, a constant, an expression, or a call to a constant function.
  - Either little-endian convention (the *lsb* is the smallest bit number) or big-endian convention (the *lsb* is the largest bit number) may be used.
  - The maximum net size may be limited, but will be at least 65,536 bits (2^{16}) bits. Most software tools have a limit of 1 million bits.
- *delay* (optional) may only be specified on net data types. The syntax is the same as primitive delays (refer to section 8.0).
- **[***array***]** is **[***first_address* **:** *last_address***][***first_address* **:** *last_address***]**...
  - Any number of array dimensions may be declared. Arrays of nets were added in Verilog-2001.
  - *first_address* and *last_address* must be a literal number, a constant, an expression, or a call to a constant function.
  - Either ascending or descending address order may be used.
  - The maximum array size for each dimension may be limited, but is at least 16,777,216 (2^{24}). Most software tools have unlimited array sizes.
- **(***strength***)** (optional) is specified as **(***strength1***,** *strength0***)** or **(***strength0***,** *strength1***)**. See section 4.9 for the strength keywords.
- *decay_time* (optional) specifies the amount of time a **trireg** net will store a charge after all drivers turn-off, before decaying to logic X. The syntax is *(rise_delay, fall_delay, decay_time)*. The default decay is infinite.
- The keywords **vectored** or **scalared** may be used immediately following the *net_type* keyword. Software tools and/or the Verilog PLI may restrict access to individual bits within a vector that is declared as vectored.

Net Declaration Examples	Notes
`wire a, b, c;`	3 scalar (1-bit) nets
`tri1 [7:0] data_bus;`	8-bit net, pulls-up when tri-stated
`wire signed [1:8] result;`	an 8-bit signed net
`wire [7:0] Q [0:15][0:256];`	a 2-dimensional array of 8-bit wires
`wire #(2.4,1.8) carry;`	a net with rise, fall delays
`wire [0:15] (strong1,pull0) sum = a + b;`	a 16-bit net with drive strength and a continuous assignment
`trireg (small) #(0,0,35) ram_bit;`	net with small capacitance and 35 time unit decay time

6.2 Variable Data Types

variable_type **signed** **[***range***]** *variable_name*, *variable_name*, ... ;
variable_type **signed** **[***range***]** *variable_name* = *initial_value*, ... ;
variable_type **signed** **[***range***]** *variable_name* **[***array***]**, ... ;

Variable data types are used for programming storage in procedural blocks.

- Variables store logic values only, they do not store logic strength.
- A variable data type must be used when the signal is on the left-hand side of a procedural assignment.
- Variables were called “registers” in older versions of the Verilog standard.

- *variable_type* is one of the following:

reg	a variable of any bit size; unsigned unless explicitly declared as signed
integer	a signed 32-bit variable
time	an unsigned 64-bit variable
real	a double-precision floating point variable
realtime	same as real

- **signed** (optional) may only be used with **reg** variables, and indicates that values are interpreted as 2’s complement signed values. If either a port or the reg connected to the port is declared as signed, then both are signed. Signed reg variables were added in Verilog-2001.
- **[***range***]** (optional) may only be used with **reg** variables, and is a range from **[***msb* **:***lsb***]** (most-significant-bit to least-significant-bit).
  - If no range is specified, then **reg** variables are 1-bit wide.
  - The *msb* and *lsb* must be a literal number, a constant, an expression, or a call to a constant function.
  - Either little-endian convention (the *lsb* is the smallest bit number) or big-endian convention (the *lsb* is the largest bit number) may be used.
  - The maximum **reg** size may be limited, but will be at least 65,536 (2^{16}) bits. Most software tools have a limit of 1 million bits.
- **[***array***]** is **[***first_address* **:** *last_address***][***first_address* **:** *last_address***]**...
  - Any number of array dimensions may be declared. Variable arrays of more than one dimension were added in Verilog-2001.
  - *first_address* and *last_address* must be a literal number, a constant, an expression, or a call to a constant function.
  - Either ascending or descending address order may be used.
  - The maximum array size for each dimension may be limited, but is at least 16,777,216 (2^{24}). Most software tools have unlimited array sizes.
  - A one-dimensional array of **reg** variables with is referred to as a ***memory***.
- *initial_value* (optional) sets the initial value of the variable.
  - The value is set in simulation time 0, the same as if the variable had been assigned a value in an initial procedure.
  - If not initialized, the default value for reg, integer and time variables is `X`, and the initial value for real and realtime variables is `0.0`.
  - Specifying the initial value as part of the variable declaration was added in Verilog-2000

- The keywords **`vectored`** or **`scalared`** may be used immediately following the **`reg`** keyword. Software tools and/or the Verilog PLI may restrict access to individual bits within a vector that is declared as vectored.

Variable Declaration Examples	Notes
`reg a, b, c;`	three scalar (1-bit) variables
`reg signed [7:0] d1, d2;`	two 8-bit signed variables
`reg [7:0] Q [0:3][0:15];`	a 2-dimensional array of 8-bit variables
`integer i, j;`	two signed integer variables
`real r1, r2;`	two double-precision variables
`reg clock = 0, reset = 1;`	two reg variables with initial values

6.3 Other Data Types

parameter	a run-time constant for storing integers, real numbers, time, delays, or ASCII strings; may be redefined for each instance of a module (see section 7.0).
localparam	a local constant for storing integers, real numbers, time, delays, or ASCII strings; may not be directly redefined, but may be indirectly redefined by assigning the localparam the value of a parameter..
specparam	a specify block constant for storing integers, real numbers, time, delays or ASCII strings; may be declared in the module scope or the specify block scope; may be redefined through SDF files or the PLI.
genvar	a temporary variable used only within a generate loop; cannot be used anywhere else, and cannot be read during simulation.
event	a momentary flag with no logic value or data storage; can be used for synchronizing concurrent activities within a module.

Declaration syntax:

parameter signed [*range***]** *constant_name* **=** *value*, ... ;
parameter *constant_type constant_name* **=** *value*, ... ;
localparam signed [*range***]** *constant_name* **=** *value*,...;
localparam *constant_type constant_name* **=** *value*, ... ;
specparam *constant_name* **=** *value*, ... ;
event *event_name*, ... ;

- **signed** (optional) indicates that values are interpreted as 2's complement signed values. Signed constants were added in Verilog-2001.
- **[***range***]** (optional) is a range from **[***msb* **:***lsb***]** (most-significant-bit to least-significant-bit).
  - If no range is specified, the constant will default to the size of the last value initially assigned to it after any parameter redefinitions.
  - The *msb* and *lsb* must be a literal number, a constant, an expression, or a call to a constant function.
  - Either little-endian convention (the *lsb* is the smallest bit number) or big-endian convention (the *lsb* is the largest bit number) may be used.
  - The maximum range may be limited, but will be at least 65,536 (2^{16}) bits. Most software tools have a limit of 1 million bits.
- *constant_type* (optional) can be **integer**, **time**, **real** or **realtime**. A constant declared with a type will have the same properties as a variable of that type. If no type is specified, the constant will default to the data type of the last value assigned to it, after any parameter redefinitions.

Data Type Examples	Notes

`parameter [2:0] s1 = 3'b001,`
`s2 = 3'b010,`
`s3 = 3'b100;`
	three 3-bit constants

`parameter integer period = 10;`	an integer constant
`localparam signed offset = -5;`	unsized signed constant defaults to width of initial value
`event data_ready, data_sent;`	two event data types

6.4 Vector Bit Selects and Part Selects

Bit Select
vector_name[*bit_number*]
Constant Part Select
vector_name[*bit_number* : *bit_number*]
Variable Part Select *(added in Verilog-2001)*
vector_name[*starting_bit_number* +: *part_select_width*]
vector_name[*starting_bit_number* -: *part_select_width*]

- A bit select can be an integer, a constant, a net, a variable or an expression.
- A constant part select is a group of bits from within the vector
  - The part select must be contiguous bits.
  - The bit numbers must be a literal number or a constant.
  - The order of the part select must be consistent with the declaration of the vector (e.g. if the lsb is the the lowest bit number in the declaration, then the lsb of the part select must also be the lowest bit number).
- Variable part selects can vary the starting point of the part select, but the width of the part select must be a literal number, a constant or a call to a constant function. Variable part selects were added in Verilog-2001.
  - **+:** indicates the part select increases from the starting point.
  - **-:** indicates the part select decreases from the starting point.

6.5 Array Selects

array_name[*index*][*index*]...
array_name[*index*][*index*]...[*bit_number*]
array_name[*index*][*index*]...[*part_select*]

- An array select can be an integer, a net, a variable, or an expression.
- Multiple indices, bit selects and part selects from an array were added in Verilog-2001.

6.6 Reading and Writing Arrays

- Only one element at a time within an array can be read from or written to.
- A memory array (a one-dimensional array of **reg** variables) can be loaded using the **$readmemb**, **$readmemh**, **$sreadmemb**, or **$sreadmemh** system tasks.

7.0 Module Instances

Port Order Connections
module_name instance_name instance_array_range **(***signal***,** *signal***,** *...***);**
Port Name Connections
module_name instance_name instance_array_range **(** **.***port_name***(***signal***),** **.***port_name***(***signal***),** *...* **);**
Explicit Parameter Redefinition
defparam *heirarchy_path***.***parameter_name* **=** *value***;**
In-line Implicit Parameter Redefinition
module_name **#(***value***,***value***,** *...***)** *instance_name* **(***signal***,** *...* **);**
In-line Explicit Parameter Redefinition *(added in Verilog-2001)*
module_name **#(.***parameter_name***(***value***),** **.***parameter_name***(***value***),** *...***)** *instance_name* **(***signal***,** *...* **);**

- *Port order* connections list the signals in the same order as the port list in the module definition. Unconnected ports are designated by two commas with no signal listed.
- *Port name* connections list both the port name and signal connected to it, in any order.
- *instance_name* (required) is used to make multiple instances of the same module unique from one another.
- *instance_array_range* (optional) instantiates multiple modules, each instance is connected to different bits of a vector.
  - The range is specified as **[***left_hand_index* **:** *right_hand_index***]**.
  - If the bit width of a module port in the array is the same as the width of the signal connected to it, the full signal is connected to each instance of the module.
  - If the bit width of a module port is different than the width of the signal connected to it, each module port instance is connected to a part select of the signal, with the right-most instance index connected to the right-most part of the vector, and progressing towards the left.
  - There must be the correct number of bits in each signal to connect to all instances (the signal size and port size must be multiples).
  - Instance arrays were added in Verilog-1995, but many software tools did not support them until Verilog-2001.
  - Multiple instances of a module can also be created using a generate block (see section 9.0).
- parameter values within a module may be redefined for each instance of the module. Only **parameter** declarations may be redefined; **localparam** and **specparam** constants cannot be redefined.
  - Explicit redefinition uses a **defparam** statement with the parameter's hierarchical name.
  - In-line implicit redefinition uses the **#** token as part of the module instantiation. Parameter values are redefined in the same order in which they are declared within the module.
  - In-line explicit redefinition uses the **#** token as part of the module instantiation. Parameter values may be redefined in any order. In-line explicit parameter redefinition was added in Verilog-2001.

Module Instance Examples

```
module reg4 (output wire [3:0] q,
             input  wire [3:0] d,
             input  wire       clk);
  //port order connection, no connection to 2nd port position
  dff u1 (q[0], , d[0], clk);
  //port name connection, qb not connected
  dff u2 (.clk(clk),.q(q[1]),.data(d[1]));
  //explicit parameter redefinition
  dff u3 (q[2], ,d[2], clk);
  defparam u3.delay = 3.2;
  //in-line implicit parameter redefinition
  dff #(2) u4 (q[3], , d[3], clk);
  //in-line explicit parameter redefinition
  dff #(.delay(3)) u5 (q[3], , d[3], clk);
endmodule
```

```
module dff (output q,
            output qb,
            input  data,
            input  clk);
  parameter delay = 1; //default delay parameter
  dff_udp #(delay) (q, data, clk);
  not (qb, q);
endmodule
```

Array of Instances Example

```
module tribuf64bit (output wire [63:0] out,
                    input  wire [63:0] in,
                    input  wire        enable);
  //array of 8 8-bit tri-state buffers; each instance is connected
  //to 8-bit part selects of the 64-bit vectors;  The scalar enable line
  //is connected to all instances
  tribuf8bit  i[7:0] (out, in, enable);
endmodule
```

```
module tribuf8bit (output wire [7:0] y,
                   input  wire [7:0] a,
                   input  wire       en);
  //array of 8 Verilog tri-state primitives; each bit of the
  //vectors is connected to a different primitive instance
  bufif1  u[7:0] (y, a, en);
endmodule
```

8.0 Primitive Instances

```
gate_type (drive_strength) #(delay) instance_name
  [instance_array_range] (terminal, terminal, ... );
switch_type #(delay) instance_name
  [instance_array_range] (terminal, terminal, ... );
```

Gate Primitives		Terminal Order and Quantity
and **or** **xor**	**nand** **nor** **xnor**	(1–output, 1-or-more–inputs)
buf	**not**	(1-or-more–outputs, 1–input)
bufif0 **bufif1**	**notif0** **notif1**	(1–output, 1–input, 1–control)
pullup	**pulldown**	(1–output)
User Defined Primitive		(1–output, 1-or-more–inputs)

Switch Primitives		Terminal Order and Quantity
pmos **rpmos**	**nmos** **rnmos**	(1–output, 1–input, 1–control)
cmos	**rcmos**	(1–output, 1–input, n-control, p-control)
tran	**rtran**	(2–bidirectional-inouts)
tranif0 **rtranif0**	**tranif1** **rtranif1**	(2–bidirectional-inouts, 1–control)

- *delay* (optional) represents the propagation delay through a primitive. The default delay is zero. Integers or real numbers may be used.
- Separate delays for 1, 2 or 3 transitions may be specified.
  - Each transition may have a single delay or a min:typ:max delay range.

Delays	Transitions represented (in order)
1	all output transitions
2	rise, fall output transitions
3	rise, fall, turn-off output transitions (turn-off delay is the time for a tri-state primitive to transition to Z)

- *strength* (optional) is specified as **(***strength1***,** *strength0***)** or **(***strength0***,** *strength1***)**. The default is (strong1, strong0). Refer to section 4.9 for strength keywords.
  - Only gate primitives may have the output drive strength specified. Switch primitives pass the input strength level to the output. Resistive switches reduce the strength level as it passes through.
- *instance_name* (optional) may used to reference specific primitives in configurations, debugging tools, schematic diagrams, etc.

- *instance_array_range* (optional) instantiates multiple primitives, each instance is connected to different bits of a vector.
  - The range is specified as **[***left-hand-index* **:** *right-hand-index***]**.
  - Primitive instances are connected with the right-most instance index connected to the right-most bit of each vector, and progressing to the left.
  - Vector signals must be the same size as the array.
  - Scalar signals are connected to all instances in the array.
  - Instance arrays were added in Verilog-1995, but many software tools did not support them until Verilog-2001.
  - Multiple instances of a primitive can also be created using a generate block (see section 9.0).

Primitive Instance Examples	Notes
`and i1 (out,in1,in2);`	zero delay gate primitive
`and #5 (o,i1,i2,i3,i4);`	same delay for all transitions
`not #(2,3) u7 (out,in);`	separate rise & fall delays
`buf (pull0,strong1)(y,a);`	output drive strengths
`wire [31:0] y, a;` `buf #2.7 i[31:0] (y,a);`	array of 32 buffers

9.0 Generate Blocks

```
genvar genvar_name, ... ;
generate
   genvar genvar_name, ... ;
   generate_items
endgenerate
```

Generate blocks provide control over the creation of many types of module items. A generate block must be defined within a module, and is used to generate code within that module. Generate blocks were added in Verilog-2001.

- **genvar** is an integer variable which must be a positive value. They may only be used within a generate block. Genvar variables only have a value during elaboration, and do not exist during simulation. Genvar variables must be declared within the module where the genvar is used. They may be declared either inside or outside of a generate block.
- *generate_items* are:

  genvar_name **=** *constant_expression***;**
  net_declaration
  variable_declaration
  module_instance
  primitive_instance
  continuous_assignment
  procedural_block
  task_definition
  function_definition

  if (*constant_expression***)**
  generate_item or *generate_item_group*

  if (*constant_expression***)**
  generate_item or *generate_item_group*
  else
  generate_item or *generate_item_group*

  case (*constant_expression***)**
  genvar_value **:** *generate_item* or *generate_item_group*
  genvar_value **:** *generate_item* or *generate_item_group*
  ...
  default: *generate_item* or *generate_item_group*
  endcase

  for (*genvar_name* **=** *constant_expression***;** *constant_expression***;**
  genvar_name **=** *constant_expression***)**
  generate_item or *generate_item_group*

- *generate_item_group* is:

  begin: *generate_block_name*
  generate_item
  generate_item
  ...
  end

- A generate **`for`** loop permits one or more generate items to be instantiated multiple times. The index loop variable must be a genvar.
- A generate **`if`**—**`else`** or **`case`** permits generate items to be conditionally instantiated based on an expression that is deterministic at the time the design is elaborated.
- *generate_block_name* (optional) is used to create a unique instance name for each generated item.
- Task and function definitions are permitted within the generate scope, but not in a generate for-loop. That is, only one definition of the task or function can be generated.

Generate Block Examples

```
/* If the input bus widths are 8-bits or less, generate an instance of a carry-
look-ahead multiplier. If the input bus widths are greater than 8-bits,
generate an instance of a wallace-tree multiplier */

module multiplier (a, b, product);
  parameter a_width = 8, b_width = 8;
  localparam product_width = a_width + b_width;
  input   [a_width-1:0]       a;
  input   [b_width-1:0]       b;
  output  [product_width-1:0] prod;

  generate
   if ((a_width < 8) || (b_width < 8))
     CLA_mult #(a_width, b_width) m (a, b, prod);
   else
     WALLACE_mult #(a_width, b_width) m (a, b, prod);
  endgenerate
endmodule
```

```
/* A parameterized gray-code to binary-code converter using a loop to
generate a continuous assignment for each bit of the converter */

module gray2bin1 (bin, gray);
  parameter SIZE = 8;
  output [SIZE-1:0] bin;
  input  [SIZE-1:0] gray;

  genvar i;

  generate
    for (i=0; i<SIZE; i=i+1)
      begin: bit
        assign bin[i] = ^gray[SIZE-1:i];
      end
  endgenerate
endmodule
```

10.0 Procedural Blocks

```
type_of_block  @(sensitivity_list)
    statement_group :group_name
      local_variable_declarations
      time_control  procedural_statements
    end_of_statement_group
```

- *type_of_block* is either **`initial`** or **`always`**
  - **`initial`** blocks process statements one time.
  - **`always`** blocks are an infinite loop which process statements repeatedly.
- *sensitivity_list* (optional) is an event time control that controls when all statements in the procedural block will be evaluated (refer to section 10.2).
- *statement_group* — *end_of_statement_group* controls the execution order of two or more procedural statements. A statement group is not required if there is only one procedural statement.
  - **`begin`—`end`** groups two or more statements together sequentially, so that statements are evaluated in the order they are listed. Each time control in the group is relative to previous time controls.
  - **`fork`—`join`** groups two or more statements together in parallel, so that all statements are evaluated concurrently. Each time control in the group is absolute to the time the group started.
- *group_name* (optional) creates a local hierarchy scope. Named groups may have local variables, and may be aborted with a **`disable`** statement.
- *local_variable_declarations* (optional) must be a variable data type (may only be declared in named statement groups).
- *time_control* is used to control when the next statement in a procedural block is executed (refer to section 10.1).
- *procedural_statement* is either an assignment statement or a programming statement (refer to sections 10.3 and 10.4).

Procedural Block Examples	Notes

`initial`
`  begin: test_loop`
`    integer i;`
`    for (i=0; i<=15; i=i+1;`
`      #5 test_in = i;`
`  end`
	`initial` procedure executes statements one time; the named group allows a local variable to be declared.

`initial`
`  fork`
`    bus = 16'h0000;`
`    #10 bus = 16'hC5A5;`
`    #20 bus = 16'hFFAA;`
`  join`
	`initial` procedure executes statements one time; the **`fork`—`join`** group places statements in parallel (the delays before each statement are in absolute times).

`always @(a or b or ci) begin`
`  sum = a + b + ci;`
`end`
	`always` procedure executes statements repeatedly, controlled by the sensitivity list.

`always @(posedge clk)` `  q = data;`	a statement group is not required when there is only one statement.

10.1 Procedural Time Controls

#delay

Delays execution of the next statement for a specific amount of time. The delay may be a literal number, a variable, or an expression.

@(*edge signal* **or** *edge signal* **or** *...* **)**
@(*edge signal***,** *edge signal***,** *...* **)**
@(*)

Delays execution of the next statement until there is a transition on a signal.

- *edge* (optional) maybe either **posedge** or **negedge**. If no edge is specified, then any logic transition is used.
- Either a comma or the keyword **or** may be used to specify events on any of several signals. The use of commas was added in Verilog-2001.
- *signal* may be a net type or variable type, and may be any vector size.
- An asterisk in place of the list of signals indicates sensitivity to any edge of all signals that are read in the statement or statement group that follows. **@*** was added in Verilog-2001.
- Parenthesis are not required when there is only one signal in the list and no edge is specified.

wait (*expression***)**

Delays execution of the next statement until the expression evaluates as true.

10.2 Sensitivity Lists

The sensitivity list is used at the beginning of an always procedure to infer combinational logic or sequential logic behavior in simulation.

- **always @(***signal***,** *signal***,** *...* **)** infers combinational logic if the list of signals contains all signals read within the procedure.
- **always @*** infers combinational logic. Simulation and synthesis will automatically be sensitive to all signals read within the procedure. **@*** was added in Verilog-2001.
- **always @(posedge** *signal***, negedge** *signal***,** *...* **)** infers sequential logic. Either the positive or negative edge can be specified for each signal in the list. A specific edge should be specified for each signal in the list.

  NOTE: The Verilog language does not have a true “sensitivity list”. Instead, the @ time control at the beginning of a procedure delays the execution of all statements within the procedure until an edge occurs on the signals listed. Thus, if the @ control is the first thing in the procedure, the entire procedure appears to be sensitive to changes in the signals listed. The @ token is a time control, however, and not a true sensitivity list. An edge-sensitive time control is only sensitive to changes when the procedure is suspended at that control. If the procedure is suspended at another time control inside the procedure, it will not be sensitive to changes at the time control in the pseudo sensitivity list.

10.3 Procedural Assignment Statements

variable **=** *expression*;

Blocking procedural assignment. Expression is evaluated and assigned when the statement is encountered. In a *begin—end* sequential statement group, execution of the next statement is blocked until the assignment is complete. In the sequence `begin m=n; n=m; end`, the first assignment changes `m` before the second assignment reads `m`.

variable **<=** *expression*;

Non-blocking procedural assignment. Expression is evaluated when the statement is encountered, and assignment is postponed until the end of the simulation time-step. In a *begin—end* sequential statement group, execution of the next statement is not blocked; and will be evaluated before the assignment is complete. In the sequence `begin m<=n; n<=m; end`, both assignments will be evaluated before `m` or `n` changes.

> *MODELING TIP*: To avoid potential simulation race conditions in zero-delay models:
> - Use blocking assignments (=) to model combinational logic.
> - Use non-blocking assignments (<=) to model sequential logic.

timing_control variable **=** *expression*;

timing_control variable **<=** *expression*;

Delayed procedural assignments. Evaluation of the expression on the right-hand side is delayed by the timing control.

variable **=** *timing_control expression*;

Blocking intra-assignment delay. Expression is evaluated in the time-step in which the statement is encountered, and assigned in the time-step specified by the timing control. In a *begin—end* sequence, execution of the next statement in the sequence is blocked until the assignment is completed (which is when the delay time has elapsed).

variable **<=** *timing_control expression*;

Non-blocking intra-assignment delay. Expression is evaluated in the time-step in which the statement is encountered, and assigned at the end of the time-step specified by the timing control. In a *begin—end* sequence, execution of the next statement(s) in the sequence are not blocked, and can execute before the delay has elapsed. Models transport delay.

assign *variable* **=** *expression*;

Procedural continuous assignment. Overrides any other procedural assignments to a variable.

deassign *variable*;

De-activates a procedural continuous assignment.

force *net_or_variable* **=** *expression*;

Forces any data type to a value, overriding all other logic.

release *net_or_variable*;

Removes the effect of a force.

10.4 Procedural Programming Statements

`if (` *expression* **`)`** *statement or statement_group*

Executes the next statement or statement group if the expression evaluates as true.

`if (` *expression* **`)`** *statement or statement_group*
`else` *statement or statement_group*

Executes the first statement or statement group if the expression evaluates as true. Executes the second statement or statement group if the expression evaluates as false or unknown.

`case (` *expression* **`)`**

case_item **`:`**	*statement or statement_group*
case_item **`,`** *case_item* **`:`**	*statement or statement_group*
`default:`	*statement or statement_group*

`endcase`

Compares the value of the expression to each case item and executes the statement or statement group associated with the first matching case. Executes the default if none of the cases match (the default case is optional).

`casez (` *expression* **`)`**

Special version of the case statement which uses a **`Z`** logic value to represent don't-care bits in either the case expression or a case item. (the **`Z`** can also be represented as a **`?`**).

`casex (` *expression* **`)`**

Special version of the case statement which uses **`Z`** or **`X`** logic values to represent don't-care bits in either the case expression or a case item. (the **`Z`** can also be represented as a **`?`**).

`for (` *initial_assignment* **`;`** *expression* **`;`** *step_assignment* **`)`**
statement or statement group

- Executes *initial_assignment* once, when the loop starts.
- Executes the statement or statement group as long as *expression* evaluates as true.
- Executes *step_assignment* at the end of each pass through the loop.

`while (` *expression* **`)`** *statement or statement group*

A loop that executes a statement or statement group as long as an expression evaluates as true. The expression is evaluated at the start of each pass of the loop.

`repeat (` *number* **`)`** *statement or statement_group*

A loop that executes the statement or statement group a set number of times. The number may be an expression (the expression is only evaluated when the loop is first entered).

`forever` *statement or statement_group*

An infinite loop that continuously executes the statement or statement group.

`disable` *group_name* **`;`**

Discontinues execution of a named group of statements. Simulation of that group jumps to the end of the group without executing any scheduled events.

Procedural Statement Examples

```
initial   // A 50 ns clock oscillator that starts after 1000 time units
  begin
    clk = 0;
    #1000 forever #25 clk = ~clk;
  end
```

```
// In this example, the sensitivity list infers sequential logic
always @(posedge clk)
  begin   // non-blocking assignments prevent race conditions in byte swap
    word[15:8]<= word[7:0];
    word[7:0] <= word[15:8];
  end
```

```
// In this example, the sensitivity list infers combinational logic
always @(a, b, ci)
  sum = a + b + ci;
```

```
// In this example, the sensitivity list infers combinational logic,
// (the @* token infers sensitivity to any signal read in the statement or
// statement group which follows it, which are sel, a and b)
always @*
  begin
    if (sel==0) y = a + b;
    else        y = a * b;
  end
```

```
// This example using illustrates several programming statements
always @(posedge clk) begin
  casez (opcode)   //casez makes Z a don't care
    3'b1??: alu_out = accum;   // ? in literal integer is same as Z
    3'b000: while (bloc_xfer) // loop until false
              repeat (5) @(posedge clk) // loop 5 clock cycles
                 begin
                   RAM[address] = data_bus;
                   address = address + 1;
                 end
    3'b011: begin : load        // named group
               integer i;       // local variable
               for (i=0; i<=255; i=i+1)
                 @(negedge clk)
                   data_bus = RAM[i];
             end
    default:$display("illegal opcode in module %m");
  endcase
end
```

11.0 Continuous Assignments

Explicit Continuous Assignment
net_type **[***size***]** *net_name***;** **assign #(***delay***)** *net_name* **=** *expression***;**
Implicit Continuous Assignment
net_type **(***strength***) [***size***] #(***delay***)** *net_name* **=** *expression***;**

Continuous assignments drive net types with the result of an expression. The result is automatically updated anytime a value on the right-hand side changes.

- Explicit continuous assignments use the **assign** keyword to continuously assign a value to a net.
  - The net can be explicitly declared in a separate statement (see section 6.1).
  - A net will be inferred if an undeclared name appears on the left side of the assignment, and the name is declared as a port of the module containing the continuous assignment. The net vector size will be the size of the port.
  - New in Verilog-2001: A 1-bit net will be inferred if an undeclared name appears on the left side of the assignment, and the name is *not* a port of the module containing the continuous assignment.
- Implicit continuous assignments combine the net declaration and continuous assignment into one statement, omitting the **assign** keyword.
- *net_type* may be any of the net data types except **trireg**.
- *strength* (optional) may only be specified when the continuous assignment is combined with a net declaration. The default strength is *(strong1, strong0)*.
- *delay* (optional) follows the same syntax as primitive delays (refer to section 8.0). The default delay is zero.
- *expression* may include any data type, any operator, and calls to functions.
- Continuous assignments model combinational logic. Each time a signal changes on the right-hand side, the right-hand side is re-evaluated, and the result is assigned to the net on the left-hand side.
- Continuous assignments are declared outside of procedural blocks. They automatically become active at time zero, and are evaluated concurrently with procedural blocks, module instances, and primitive instances.

Continuous Assignment Examples

```
// Explicit continuous assignment
wire [31:0] mux_out;
assign mux_out = sel? a : b;
```

```
// Implicit continuous assignment;  the net declaration
// and the continuous assignment are combined
tri [0:15] #2.8 buf_out = en? in: 16'bz;
```

```
// Implicit continuous assignment with strengths
wire [63:0] (strong1,pull0) alu_out =
  alu_function(opcode,a,b);
```

12.0 Operators

- For most operations, the operands may be nets, variables, constants or function calls. Some operations are not legal on real (floating-point) values.
- Operators which return a true/false result will return a 1-bit value where **1** represents true, **0** represents false, and **X** represents indeterminate.

Bitwise Operators		
~	~m	invert each bit of m
&	m & n	AND each bit of m with each bit of n
|	m | n	OR each bit of m with each bit of n
^	m ^ n	exclusive-OR each bit of m with n
~^ *or* ^~	m ~^ n	exclusive-NOR each bit of m with n
<<	m << n	shift m left n-times and fill with zeros
>>	m >> n	shift m right n-times and fill with zeros
Unary Reduction Operators		
&	&m	AND all bits in m together (1-bit result)
~&	~&m	NAND all bits in m together (1-bit result)
|	|m	OR all bits in m together (1-bit result)
~|	~|m	NOR all bits in m together (1-bit result)
^	^m	exclusive-OR all bits in m (1-bit result)
~^ *or* ^~	~^m	exclusive-NOR all bits in m (1-bit result)
Logical Operators		
!	!m	is m not true? (1-bit True/False result)
&&	m && n	are both m and n true? (1-bit True/False result)
||	m || n	are either m or n true? (1-bit True/False result)
Equality and Relational Operators (return X if an operand has X or Z)		
==	m == n	is m equal to n? (1-bit True/False result)
!=	m != n	is m not equal to n? (1-bit True/False result)
<	m < n	is m less than n? (1-bit True/False result)
>	m > n	is m greater than n? (1-bit True/False result)
<=	m <= n	is m less than or equal to n? (1-bit True/False result)
>=	m >= n	is m greater than or equal to n? (1-bit True/False result)
Identity Operators (compare logic values 0, 1, X, and Z)		
===	m === n	is m identical to n? (1-bit True/False results)
!==	m !== n	is m not identical to n? (1-bit True/False result)
Miscellaneous Operators		
?:	sel?m:n	conditional operator; if sel is true, return m: else return n
{}	{m,n}	concatenate m to n, creating a larger vector
{{}}	{n{ }}	replicate inner concatenation n-times
–>	-> m	trigger an event on an event data type

Arithmetic Operators		
+	m + n	add n to m
–	m - n	subtract n from m
–	-m	negate m (2's complement)
*	m * n	multiply m by n
/	m / n	divide m by n
%	m % n	modulus of m / n
**	m ** n	m to the power n (new in Verilog-2001)
<<<	m <<< n	shift m left n-times, filling with 0 (new in Verilog-2001)
>>>	m >>> n	shift m right n-times; fill with value of sign bit if expression is signed, otherwise fill with 0 (Verilog-2001)

12.1 Operator Expansion Rules

As a general rule, all operands in an expression are first expanded to the size of the largest vector in the statement (including both sides of an assignment statement). Concatenate and replicate operations are evaluated before the expansion, and represent a new vector size.

- Unsigned operands are expanded by left-extending with zero.
- Signed operands are expanded by left-extending with the value of the most-significant bit (the sign bit).

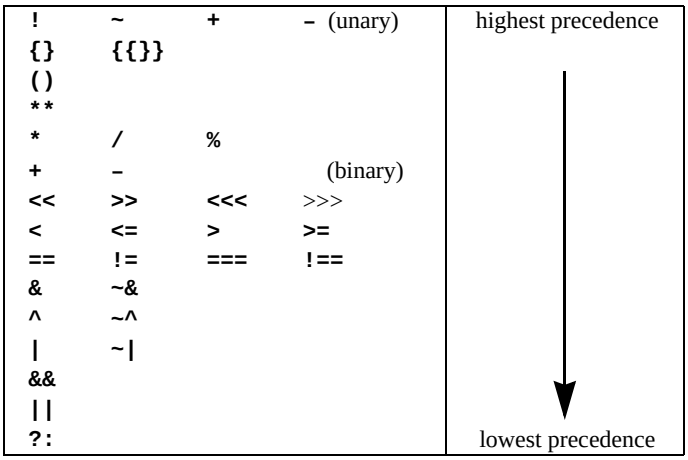
12.2 Arithmetic Operation Rules

For most operators (there are exceptions) all operands in the expression are used to determine how the operation is performed:

- If any operand is real, then floating-point arithmetic will be performed.
- If any operand is unsigned, then unsigned arithmetic will be performed.
- If all operands are signed, then signed arithmetic will be performed.
- An operand can be "cast" to signed or unsigned using the **`$signed`** and **`$unsigned`** system functions (added in Verilog-2001).

12.3 Operator Precedence

Compound expressions are evaluated in the order of operator precedence. Operators within parenthesis have a higher precedence and are evaluated first.



13.0 Task Definitions

ANSI-C Style Task Declaration *(added in Verilog-2001)*

```
task automatic task_name (
   port_declaration port_name, port_name, ... ,
   port_declaration port_name, port_name, ... );
   local variable declarations
   procedural_statement or statement_group
endtask
```

Old Style Task Declaration

(port declarations determine the order signals are passed in/out of the task)

```
task automatic task_name;
   port_declaration port_name, port_name, ...;
   port_declaration port_name, port_name, ...;
   local variable declarations
   procedural_statement or statement_group
endtask
```

Tasks are analogous to subroutines in other languages.

- Must be declared within a module, and are local to that module.
- Must be called from an initial procedure, an always procedure or another task.
- May have any number of input, output or inout ports, including none.
- Tasks may contain time controls (**#**, **@**, or **wait**).

automatic (optional) allocates storage space each time the task is called, allowing the task to be re-entrant (the task can be called while previous calls to the task are still executing). Automatic tasks were added in Verilog-2001.

port_declaration can be:

- *port_direction* **signed** *range*
- *port_direction* **reg signed** *range*
- *port_direction port_type*

port_direction can be **input**, **output** or **inout**.

range (optional) is a range from **[***msb* **:***lsb***]** (most-significant-bit to least-significant-bit). *msb* and *lsb* must be a literal number, a constant, an expression, or a call to a constant function. If no range is specified, the ports are 1-bit wide.

port_type can be **integer**, **time**, **real** or **realtime**.

signed (optional) indicates that values are interpreted as 2's complement signed values.

Example of a Task

```
//TASK DEFINITION (must be declared within a module)
task read_mem (input  [15:0] address,
               output [31:0] data );
  begin
    read_request = 1;
    wait (read_grant) addr_bus = address;
    data = data_bus;
    #5 addr_bus = 16'bz; read_request = 0;
  end
endtask
```

```
//TASK CALL
always @(posedge clock)
  read_mem(PC, IR);
```

14.0 Function Definitions

ANSI-C Style Function Declaration *(added in Verilog-2001)*

```
function  automatic range_or_type  function_name (
   input range_or_type  port_name, port_name, ... ,
   input range_or_type  port_name, port_name, ... );
   local variable declarations
   procedural_statement or statement_group
endfunction
```

Old Style Function Declaration

(port declarations determine the order signals are passed into the function)

```
function automatic [range_or_type] function_name;
   input range_or_type  port_name, port_name, ... ;
   input range_or_type  port_name, port_name, ... ;
   local variable declarations
   procedural_statement or statement_group
endfunction
```

Functions:

- Must be declared within a module, and are local to that module.
- Return the value assigned to the function name.
- May be called any place an expression value can be used.
- Must have at least one input; may not have outputs or inouts.
- May not contain time controls or non-blocking assignments.

automatic (optional) allocates storage space for each function call, allowing recursive function calls. Automatic functions were added in Verilog-2001.

range_or_type (optional) is the function return type or input type. The default is a 1-bit reg. *range_or_type* can be:

- **signed [***msb* **:***lsb***]**
- **reg signed [***msb* **:***lsb***]**
- **integer**, **time**, **real** or **realtime**

signed (optional) indicates that the return value or input values are interpreted as 2's complement signed values. Signed functions were added in Verilog-2001.

Example of a Function

```
function automatic [63:0] factorial (input reg [31:0] n);
  if (n<=1) factorial = 1;
  else      factorial = n * factorial(n-1); //recursive call
endfunction
```

```
if ( factorial(data) <= LIMIT )   //function call
```

14.1 Constant Functions

Constant functions (new in Verilog-2001) are functions with restrictions so that the function can be evaluated at elaboration (before simulation starts running).

- Only locally declared variables can be referenced. Net types cannot be used.
- Parameter values used within the function must be defined before the function is called. Parameter values should not be redefined using `defparam`.
- Can call other constant functions, but not in a context where a constant expression is required (such as to declare port sizes).
- System function calls are illegal. System task calls are ignored.
- Hierarchical references are illegal.

15.0 Specify Blocks

```
specify
    specparam_declarations  (see 6.3)
    simple_pin-to-pin_path_delay
    edge-sensitive_pin-to-pin_path_delay
    state-dependent_pin-to-pin_path_delay
    timing_constraint_checks
endspecify
```

15.1 Pin-to-pin Path Delays

- Simple path delay statement:
  (*input_port polarity***:***path_token output_port* **) = (***delay***);**
- Edge-sensitive path delay:
  (*edge input_port path_token (output_port polarity***:***source***)) = (***delay***);**
  - *edge* (optional) may be either **posedge** or **negedge**. If not specified, all input transitions are used.
  - *source* (optional) is the input port or value the output will receive. The source is ignored by most logic simulators, but may be used by timing analyzers.
- State-dependent path delay:
  if (*first_condition***)** *simple_or_edge-sensitive_path_delay*
  if (*next_condition***)** *simple_or_edge-sensitive_path_delay*
  ifnone *simple_path_delay*
  - Different delays for the same path can be specified.
  - *condition* may only be based on input ports.
  - Most operators may be used, but should resolve to true/false (**X** or **Z** is considered true; if the condition resolves to a vector, only the *lsb* is used).
  - Each delay for the same path must have a different condition or a different edge-sensitive edge.
  - The **ifnone** condition (optional) may only be a simple path delay, and serves as a default if no other condition evaluates as true.
- *polarity* (optional) is either **+** or **–**. A **–** indicates the input will be inverted. Polarity is ignored by most simulators, but may be used by timing analyzers.
- *path_token* is either ***>** for full connection or **=>** for parallel connection.
  - Parallel connection indicates each input bit of a vector is connected to its corresponding output bit (bit 0 to bit 0, bit 1 to bit 1, ...)
  - Full connection indicates an input bit may propagate to any output bit.
- Separate delay sets for 1, 2, 3, 6 or 12 transitions may be specified.
  - Each delay set may have a single delay or a min:typ:max delay range.

Delays	Transitions represented (in order)
1	all output transitions
2	rise, fall output transitions
3	rise, fall, turn-off output transitions
6	rise, fall, **0–>Z**, **Z–>1**, **1–>Z**, **Z–>0**
12	rise, fall, **0->Z**, **Z->1**, **1->Z**, **Z->0**, **0->X**, **X->1**, **1->X**, **X->0**, **X->Z**, **Z->X**

Path Delay Examples	Notes
`(a => b) = 1.8;`	parallel connection path; one delay for all output transitions
`(a -*> b) = 2:3:4;`	full connection path; one min:typ:max delay range for all output transitions; *b* receives the inverted value of *a*

`specparam t1 = 3:4:6,`
`  t2 = 2:3:4;`
`(a => y) = (t1,t2);`
	different path delays for rise, fall transitions

`(posedge clk => (qb -: d))` `  = (2.6, 1.8);`	edge-sensitive path delay; timing path is positive edge of clock to *qb*; *qb* receives the inverted value of data
`if (rst && pst)` `(posedge clk=>(q +: d))=2;`	state-dependent edge sensitive path delay

`if (opcode = 3'b000)`
`  (a,b *> o) = 15;`
`if (opcode = 3'b001)`
`  (a,b *> o) = 25;`
`ifnone (a,b *> o) = 10;`
	state-dependent path delays; an ALU with different delays for certain operations (default delay has no condition)

15.2 Path Pulse (Glitch) Detection

A pulse is a glitch on the inputs of a module path that is less than the delay of the path. A special specparam constant can be used to control whether the pulse will propagate to the output (transport delay), not propagate to the output (inertial delay), or result in a logic X on the output.

specparam PATHPULSE$***input*$*output* = (*reject_limit*, *error_limit*);**
specparam PATHPULSE$ = (*reject_limit*, *error_limit*);**

- *reject_limit* is a delay value, or min:typ:max delay set, that is less than or equal to the delay of a module path. Any pulse on the input that is less than or equal to the reject limit will be cancelled (not propagate to the output).
- *error_limit* is a delay value, or min:typ:max delay set, that is greater than or equal to the *reject_limit* and less than or equal to the delay of a module path. Any pulse on the input greater than the *error_limit* will propagate to the output. Any pulse less than or equal to the *error_limit* and greater than the *reject_limit* will be propagated as a logic X to the output.
- A PATHPULSE$ specparam with no input to output path applies to all module paths that do not have a specific PATHPULSE$ specparam.
- A single limit can be specified, in which case the reject and error limits will be the same. The parenthesis can be omitted when there is a single value.

Verilog-2001 adds the following reserved words which can be used within a specify block for greater pulse propagation accuracy.

pulsestyle_onevent *list_of_path_outputs*;
Indicates that a pulse propagates to a path output as an X, with the same delay as if a valid input change had propagated to the output. This is the default behavior, and matches Verilog-1995.

pulsestyle_ondetect *list_of_path_outputs*;
Indicates that as soon as a pulse is detected, a logic X is propagated to a path output, without the path delay.

(continued on next page)

showcancelled *list_of_path_outputs*;
Indicates that a negative pulse, where the trailing edge of the pulse occurs before the leading edge, will not propagate to the output. This is the default behavior, and matches Verilog-1995.

noshowcancelled *list_of_path_outputs*;
Indicates that negative pulses propagate to the output as a logic X.

15.3 Timing Constraint Checks

Timing constraint checks are special tasks that model restrictions on input changes, such as setup times and hold times.

$setup(*data_event, reference_event, limit, notifier***);**
$hold(*reference_event, data_event, limit, notifier***);**
$setuphold(*reference_event, data_event, setup_limit, hold_limit, notifier, stamptime_condition, checktime_condition, delayed_ref, delayed_data***);**
$recovery(*reference_event, data_event, limit, notifier***);**
$removal(*reference_event, data_event, limit, notifier***);**
$recrem(*reference_event, data_event, recovery_limit, removal_limit, notifier, stamptime_cond, checktime_cond, delayed_ref, delayed_data***);**
$skew(*reference_event, data_event, limit, notifier***);**
$timeskew(*reference_event, data_event, limit, notifier, event_based_flag, remain_active_flag***);**
$fullskew(*reference_event, data_event, data_skew_limit, ref_skew_limit, notifier, event_based_flag, remain_active_flag***);**
$period(*reference_event, limit, notifier***);**
$width(*reference_event, limit, width_threshold, notifier***);**
$nochange(*reference_event, data_event, start_edge_offset, end_edge_offset, notifier***);**

Timing checks measure the delta between a *reference_event* and a *data_event*.

- *data_event* and *reference_event* signals must be module input ports.
- *limit* is a constant expression that represents the amount of time that must be met for the constraint. The expression can be a min:typ:max delay set.
- *notifier* (optional) is a 1-bit reg variable that is automatically toggled whenever the timing check detects a violation.
- *stamptime_condition* (optional) and *checktime_condition* (optional) are conditions for enabling or disabling negative timing checks. These arguments were added in Verilog-2001.
- *delayed_ref* (optional) and *delayed_data* (optional) are delayed signals for negative timing checks. These arguments were added in Verilog-2001.
- *event_based_flag* (optional) when set, causes the timing check to be event based instead of timer based. This argument was added in Verilog-2001.
- *remain_active_flag* (optional) wen set, causes the timing check to not become inactive after the first violation is reported. This argument was added in Verilog-2001.
- *start_edge_offset* and *end_edge_offset* are delay values (either positive or negative) which expand or reduce the time in which no change can occur.
- $recrem, $timeskew and $fullskew were added in Verilog-2001.

16.0 User Defined Primitives (UDPs

User Defined Primitives define new primitives, which are used exactly the same as built-in primitives.

ANSI-C Style Port List *(added in Verilog-2001)*

```
primitive primitive_name
   ( output reg = logic_value  terminal_declaration,
     input terminal_declarations  );
   table
     table_entry;
     table_entry;
   endtable
endprimitive
```

Old Style Port List

```
primitive primitive_name (output, input, input, ... );
   output terminal_declaration;
   input terminal_declarations;
   reg output_terminal;
   initial output_terminal = logic_value;
   table
     table_entry;
     table_entry;
   endtable
endprimitive
```

- All terminals must be scalar (1-bit).
- Only one output is allowed, which must be the first terminal.
- The maximum number of inputs is at least 9 inputs for a sequential UDP and 10 inputs for a combinational UDP.
- Logic levels of 0, 1, X and transitions between those values may be represented in the table. The logic value Z is not supported with UDPs. A Z value on a UDP input is treated as an X value.
- **reg** declaration (optional) defines a sequential UDP by creating internal storage. Only the output may be declared a **reg**.
- **initial** (optional) is used to define the initial (power-up) state for sequential UDP's. Only the logic values 0, 1, and X may be used. The default state is X. In Verilog-2001, the initial value can be assigned in the declaration.

16.1 UDP Table Entries

input_logic_values **:** *output_logic_value* **;**

- Combinational logic table entry. Only logic level values may be specified (0, 1, X and don't cares).

input_logic_values **:** *previous_state* **:** *output_logic_value* **;**

- Sequential logic table entry. May only be used when the output is also declared as a **reg** data type. Both input logic level and input logic transition values may be specified.

- A white space must separate each input value in the table.
- The input values in the table must be listed in the same order as the terminal list in the primitive statement.
- Any combination of input values not specified in the table will result in a logic X (unknown) value on the output.

- Only one signal may have an edge transition specified for each table entry.
- If an edge transition is specified for one input, the UDP becomes sensitive to transitions on all inputs. Therefore, all other inputs must have table entries to cover transitions, or when the transition occurs the UDP will output an X.
- Level sensitive entries have precedence over edge sensitive table entries.

16.2 UDP Table Symbols

Truth Table Symbol	Definition
0	logic 0 on input or output
1	logic 1 on input or output
x or **X**	unknown on input or output
–	no change on output (sequential UDPs only)
?	don't care if an input is 0, 1, or X
b or **B**	don't care if and input is 0 or 1
(*vw***)**	input transition from logic *v* to logic *w* e.g.: **(01)** represents a transition from 0 to 1
r or **R**	rising input transition: same as **(01)**
f or **F**	falling input transition: same as **(10)**
p or **P**	positive input transition: **(01)**, **(0X)** or **(X1)**
n or **N**	negative input transition: **(10)**, **(1X)** or **(X0)**
*****	Any possible input transition: same as **(??)**

UDP Examples

```
primitive mux (y, a, b, sel);  //COMBINATIONAL UDP
  output y;
  input  sel, a, b;
  table           //Table order for inputs
  // a b sel : y  //matches primitive statement
     0 ?  0  : 0; //select a; don't care on b
     1 ?  0  : 1; //select a; don't care on b
     ? 0  1  : 0; //select b; don't care on a
     ? 1  1  : 1; //select b; don't care on a
  endtable
endprimitive
```

```
primitive dff   //SEQUENTIAL UDP
  (output reg q = 0,
   input  clk, rst, d );
  table
  // d clk rst:state:q
     ?  ?   0 :  ?  :0; //low true reset
     0  R   1 :  ?  :0; //clock in a 0
     1  R   1 :  ?  :1: //clock in a 1
     ?  N   1 :  ?  :-; //ignore negedge of clk
     *  ?   1 :  ?  :-; //ignore all edges on d
     ?  ?   P :  ?  :-; //ignore posedge of rst
     0 (0X) 1 :  0  :-; //reduce pessimism
     1 (0X) 1 :  1  :-; //reduce pessimism
  endtable
endprimitive
```

17.0 Common System Tasks and Functions

- System tasks and functions begin with a **$** (dollar sign).
- The IEEE 1364 Verilog standard defines a number of standard system task and system functions.
- Software tool vendors may define additional proprietary system tasks and functions specific to their tool, such as for waveform displays.
- Simulator users may define additional system tasks and functions using the Verilog Programming Language Interface (PLI).

17.1 Text Output System Tasks

$display(*"text_with_format_specifiers", list_of_arguments***);**
$displayb(*"text_with_format_specifiers", list_of_arguments***);**
$displayo(*"text_with_format_specifiers", list_of_arguments***);**
$displayh(*"text_with_format_specifiers", list_of_arguments***);**
Prints the formatted message when the statement is executed. A newline is automatically added to the message. If no format is specified, the routines default to decimal, binary, octal and hexadecimal formats, respectively.

$write(*"text_with_format_specifiers", list_of_arguments***);**
$writeb(*"text_with_format_specifiers", list_of_arguments***);**
$writeo(*"text_with_format_specifiers", list_of_arguments***);**
$writeh(*"text_with_format_specifiers", list_of_arguments***);**
Like $display statement, except that no newline is added.

$strobe(*"text_with_format_specifiers", list_of_arguments***);**
$strobeb(*"text_with_format_specifiers", list_of_arguments***);**
$strobeo(*"text_with_format_specifiers", list_of_arguments***);**
$strobeh(*"text_with_format_specifiers", list_of_arguments***);**
Like the $display statement, except that the printing of the text is delayed until all simulation events in the current simulation time have executed.

$monitor(*"text_with_format_specifiers", list_of_arguments***);**
$monitorb(*"text_with_format_specifiers", list_of_arguments***);**
$monitoro(*"text_with_format_specifiers", list_of_arguments***);**
$monitorh(*"text_with_format_specifiers", list_of_arguments***);**
Invokes a background process that continuously monitors the arguments listed, and prints the formatted message whenever one of the arguments changes. A newline is automatically added to the message.

Text Formatting Codes

%b	binary values	**%m**	hierarchical name of scope
%o	octal values	**%l**	configuration library binding
%d	decimal values	**\t**	print a tab
%h	hex values	**\n**	print a newline
%e	real values–exponential	**\"**	print a quote
%f	real values–decimal	****	print a backslash
%t	formatted time values	**%%**	print a percent sign
%s	character strings		

%0b, **%0o**, **%0d** and **%0h** truncates any leading zeros in the value.
%e and **%f** may specify field widths (e.g. **%5.2f**).
%m and **%l** do not take an argument; they have an implied argument value.
The format letters are not case sensitive (i.e. **%b** and **%B** are equivalent).

17.2 File I/O System Tasks and Functions

mcd = **$fopen(***"file_name"***);**
fd = **$fopen(***"file_name", type***);**

A function that opens a disk file for writing, and returns an integer value.

- *mcd* is a multi-channel-descriptor with a single bit set. Multiple mcd's can be or'ed together to write to multiple files at the same time. An mcd file is always opened as a new file for writing only. Bit 0 is reserved and represents the simulator's output window. Bit 31 is reserved and represents that the channel is an fd, not an mcd.
- *fd* is a single-channel descriptor which has multiple bits set; bit 31 and at least one other bit will be set. An fd file can be opened for either reading or writing, and can be opened in append mode. Only one file can be read or written to at a time using an fd. The fd was added in Verilog-2001.
- *type* is one of the following character strings:

"r" or "rb"	open for reading
"w" or "wb"	truncate to zero length or create for writing
"a" or "ab"	append; open for writing at end of file
"r+", "r+b", or "rb+"	open for update (reading and writing)
"w+", "w+b", or "wb+"	truncate or create for update
"a+", "a+b", or "ab+"	append; open or create for update at end-of-file

$fclose(*mcd_or_fd***);**

Closes a disk file that was opened by **$fopen**.

$fmonitor(*mcd_or_fd, "text with format specifiers", signal, signal,...***);**
$fdisplay(*mcd_or_fd, "text with format specifiers", signal, signal,...***);**
$fwrite(*mcd_or_fd, "text with format specifiers", signal, signal,...***);**
$fstrobe(*mcd_or_fd, "text with format specifiers", signal, signal,...***);**

Variations of the text display tasks that write to files.

Verilog-2001 adds several system functions similar to C file I/O functions:

c = **$fgetc(***fd***);**
code = **$ungetc(***c, fd***);**
code = **$fgets(***str, fd***);**
code = **$fscanf(***fd, format, arguments***);**
code = **$fread(***reg_variable, fd***);**
code = **$fread(***memory_array, fd, start, count***);**
position = **$ftell(***fd***);**
code = **$fseek(***fd, offset, operation***);**
code = **$rewind(***fd***);**
errno = **$ferror(***fd, str***);**
$fflush(*mcd_or_fd***);**

17.3 Other Common System Tasks and Functions

$finish(*n***);**

Finishes a simulation and exits the simulation process. *n* (optional) is 0, 1 or 2, and may cause extra information about the simulation to be displayed.

$stop(*n***);**

Halts a simulation and enters an interactive debug mode.

$time
$stime
$realtime
Returns the current simulation time as a 64-bit vector, a 32-bit integer or a real number, respectively.

$timeformat(*unit, precision, "suffix", min_field_width***);**
Controls the format used by the **%t** text format specifier.

- *unit* is the base that time is to be displayed in, where:

0 = 1sec	-4 = 100us	-7 = 100ns	-10 = 100ps	-13 = 100fs
-1 = 100ms	-5 = 10us	-8 = 10ns	-11 = 10ps	-14 = 10fs
-2 = 10ms	-6 = 1us	-9 = 1ns	-12 = 1ps	-15 = 1fs
-3 = 1ms				

- *precision* is the number of decimal points to display.
- *suffix* is a string appended to the time, such as "*ns*".
- *min_field_width* is the minimum number of characters to display.

Example: `$timeformat (-9, 2, "ns", 10);`

$printtimescale(*module_hierarchical_name***);**
Prints the time scale of the specified module, or the scope from which it is called if no module is specified.

signed_value = **$signed(***unsigned_value***)**
unsigned_value = **$unsigned(***signed_value***)**
Converts a value to or from a signed value; affects math operations and sign extension. These system functions were added in Verilog-2001.

$swrite(*reg_variable, format, arguments, format, arguments,...***);**
$swriteb(*reg_variable, format, arguments, format, arguments,...***);**
$swriteo(*reg_variable, format, arguments, format, arguments,...***);**
$swrited(*reg_variable, format, arguments, format, arguments,...***);**
$sformat(*reg_variable, format, arguments***);**
Similar to $write, except that the string is written to the reg variable instead of to a file. These system tasks and functions were added in Verilog-2001.

code = **$sscanf(***str, format, arguments***);**
Similar to `$fscanf`, but reads values from a string. Added in Verilog-2001.

$readmemb(*"file_name", variable_array, start_address, end_address***);**
$readmemh(*"file_name", variable_array, start_address, end_address***);**
Loads the contents of a file into a memory array. The file must be an ASCII file with values represented in binary (**$readmemb**) or hex (**$readmemh**). Start and end address are optional.

64-bit_reg_variable = **$realtobits(***real_variable***);**
real_variable = **$bitstoreal(***64-bit_reg_variable***);**
Converts double-precision real variables to and from 64 bit reg vectors, so that the real value can be passed through 64-bit ports.

integer = **$test$plusargs("***invocation_option***")**
integer = **$value$plusargs("***invocation_option=format***",** *variable***)**
Tests the invocation command line for the invocation option. The option must begin with a + on the command line, but the + is not included in the string. If found, the routines return a non-zero value. $value$plusargs converts any text following the string up to a white space to the format specified, and puts the value into the second argument. Allowable formats are %b, %o, %d, %h, %e, %f, %g and %s.

18.0 Common Compiler Directives

Compiler directives provide a method for software tool vendors to control how their tool will interpret Verilog HDL models.

- Compiler directives begin with the grave accent character (`).
- Compiler directives are not Verilog HDL statements; there is no semi-colon at the end of compiler directives.
- Compiler directives are not bound by modules or by files. When a tool encounters a compiler directive, the directive remains in effect until another compiler directive either modifies it or turns it off.

`resetall
Resets all compiler directives that have a default back to their default. Directives that have no default are not affected.

`timescale *time_unit base / precision base*
Specifies the time units and precision for delays:

- *time_unit* is the amount of time a delay of 1 represents. The time unit must be 1, 10, or 100
- *base* is the time base for each unit, ranging from seconds to femtoseconds, and must be: **s ms us ns ps** or **fs**
- *precision* and *base* represent how many decimal points of precision to use relative to the time units.

Example: `timescale 1 ns / 10 ps
Indicates delays are in 1 nanosecond units with
2 decimal points of precision (10 ps is .01 ns).

Note: There is no default timescale in Verilog; delays are simply relative numbers until a timescale directive declares the units and base the numbers represent.

`define *macro_name text_string*
`define *macro_name (arguments) text_string (arguments)*
Text substitution macro. Allows a text string to be defined as a macro name.

- *text_string* will be substituted in place of the *macro_name* where ever the macro name is used.
- *text_string* is terminated by a carriage return—the string must be on one line.
- *arguments* are evaluated before text is substituted.
- The *macro_name* must also be preceded by the grave accent mark (`) each time the macro name is used.
- Comments may be used—they are not substituted into the place of the macro name.

Examples:

```
`define cycle 20  //clock period
always #(`cycle/2) clk = ~clk;
```

```
`define NAND(dval) nand #(dval)
`NAND(3)     i1 (y,a,b);
`NAND(3:4:5) i2 (o,c,d);
```

`undef *macro_name*
Removes the definition of a macro name.

`ifdef *macro_name*
`ifndef *macro_name*
    verilog_source_code
`else
    verilog_source_code
`elsif
    verilog_source_code
`endif

Conditional compilation. Allows Verilog source code to be optionally included, based on whether or not *macro_name* has been defined using the **`define** compiler directive or the **+define+** invocation option. The **'ifndef** and **'elsif** directives were added in Verilog-2001.

Example:

```
`ifdef RTL
    wire  y = a & b;
`else
    and #1 (y,a,b);
`endif
```

`include *"file_name"*

File inclusion. The contents of another Verilog HDL source file is inserted where the **`include** directive appears.

`celldefine
`endcelldefine

Flags the Verilog source code between the two directives as a cell. Some tools, such as a delay calculator for an ASIC, need to distinguish between a module that represents an ASIC cell and other modules in the design.

`default_nettype *net_data_type*
`default_nettype *none*

Changes the net data type to be used for implicit net declarations. Any of the net data types may be specified. By default, the implicit net data type is wire. If *none* is specified, then implicit net declarations are disabled, and all nets must be explicitly declared (specifying *none* was added in Verilog-2001).

`unconnected_drive pull1
`unconnected_drive pull0
`nounconnected_drive

Determines what logic value will be applied to unconnected module inputs. The default is **`nounconnected_drive**, meaning unconnected inputs and nets float at high impedance.

`uselib file=<*file*> **dir=**<*directory*> **libext=**<*extension*>

Specifies the Verilog source library file or directory in which the compiler should search for the definitions of modules or UDPs instantiated in a design. A **`uselib** directive with no arguments removes any preceding library search directives. *Note*: This directive is not part of the IEEE 1364 Verilog standard, but is implemented in most Verilog simulators.

Example:

```
`uselib file=/models/rtl_lib
   ALU i1 (y1,a,b,op); //RTL model
`uselib dir=/models/gate_lib libext=.v
   ALU i2 (y2,a,b,op); //Gate model
`uselib   //turn off `uselib searching
```

19.0 Configurations

Configurations (added in Verilog-2001) are a set of rules to specify the exact source description to be used for each module or primitive instance in a design. The configuration block is Verilog source code; it can be compiled along with the Verilog model source code.

- Verilog designs are modeled the same as in Verilog-1995.
- Configuration blocks are specified outside of module boundaries. The blocks can be in the same files as the Verilog source code, or they can be in separate files.
- A *cell* is the name of a module, primitive or another configuration.
- *Symbolic library names* are used within the configuration block. A symbolic library is a logical collection of cells. The cell name must be the same as the name of the module, primitive or configuration.
- *library map files* are used to map the symbolic library names to physical file locations.
- The library binding information for module instances can be displayed during simulation using the format specifier %l, which will print the *library_name.cell_name* of the module containing the print statement.

19.1 Configuration Blocks

```
config config_name;
   design lib_name.cell_name;
   default liblist list_of_library_names;
   cell lib_name.cell_name liblist list_of_library_names;
   cell lib_name.cell_name use lib_name.cell_name:config_name;
   instance hierarchy_name liblist list_of_library_names;
   instance hierarchy_name use lib_name.cell_name:config_name;
endconfig
```

The **config**—**endconfig** is a design element, similar to a module, and exists in the same Verilog name space as module and primitive names. The configuration block contains a set of rules for searching for the Verilog source description to bind to a particular instance of the design.

- **design** specifies the library and cell of the top-level module or modules in the design hierarchy. There can only be one design statement, but multiple top-level modules can be listed. The design statement must the first statement in the configuration.
  - *lib_name*. (optional) specifies which symbolic library contains the cell. If the library name is omitted, then the library which contains the config is used to search for the cell.
  - *cell_name* (required) is the name of the module that is the top of the design hierarchy represented by the configuration.
- **default liblist** specifies in which libraries to search for all instances which do not match a more specific selection clause. The libraries are searched in the order listed. For many designs, the default liblist may be all that is needed to specify the configuration.
  - *list_of_library_names* is a comma-separated list of symbolic library names.

- **cell** specifies a specific set of libraries in which to search for the source code for that module or primitive name, instead of the libraries and order specified in the default statement.
  - *lib_name*. (optional) specifies which symbolic library contains the cell.
  - *cell_name* (required) is the name of a module or primitive.
- **instance** specifies a specific set of libraries in which to search for the source code for that specific module or primitive instance, instead of the libraries and/or order specified in the default statement.
  - *hierarchy_name* is the full hierarchy path name of an instance of a module or primitive. The hierarchy path must start with the name specified in the design statement.
- **use** (optional) specifies the location for a specific cell or instance of a cell, instead of searching a for the cell in the default libraries.
  - *lib_name*. (optional) specifies which symbolic library contains the cell.
  - :*config_name* (optional) specifies that a different configuration block should be used for the specified instance or cell. The design statement in that configuration specifies the actual binding information.

19.2 Library Map Files

```
library lib_name list_of_file_paths, -incdir list_of_file_paths;
include library_map_file_path;
```

A separate file is used to map symbolic libraries to the physical file locations.

- The library map file contains library statements, include statements and Verilog-style comments.
- The map file is not Verilog source code.
- If the source files are moved, only the map file needs to be modified; The Verilog source code and configuration blocks do not need to be changed.

- *lib_name* defines the symbolic library name which will be reference in configuration blocks.
- *list_of_file_paths* is a comma-separated list of operating system paths to one or more directories or specific files.
  - A path which ends in / includes all files in the specified directory (identical to a path which ends with /*).
  - A path which does not begin with / is relative to the directory in which the current library map file is located.
  - Special symbols can be used in the path:

?	single character wild card (matches any single character)
*	multiple character wild card (matches any number of characters)
...	hierarchical wild card (matches any number of hierarchical directories)
..	specifies the parent directory
.	specifies the directory containing the lib.map

- **-incdir** specifies where to search for files referenced by **`include** directives in the Verilog source code.
- **include** *library_map_file_path* allows one library map file to reference another library map file.

20.0 Synthesis Supported Constructs

Following is a list of Verilog HDL constructs supported by most synthesis tools. The list is based on a preliminary draft of the IEEE 1364.1 *"Verilog Register Transfer Level Synthesis"* standard (this standard was not complete at the time this reference guide was written). The list is not specific to any one tool—each synthesis tool supports a unique subset of the Verilog language.

Verilog HDL Constructs	Notes
module declarations	fully supported
port declarations **input output** **inout**	fully supported; any vector size supported
net data types **wire wand wor** **supply0 supply1**	scalars and vectors fully supported
variable data types **reg integer**	• may be scalar or vector or variable array • may be restricted to only making assignments to a variable from just one procedure • integers default to 32 bits
parameter constants	limited to integers; parameter redefinition may not be supported
literal integer numbers	fully supported; all sizes and bases
module instances	fully supported; both port order and port name instantiation supported
primitive instances **and nand** **or nor** **xor buf not** **bufif1 bufif0** **notif1 notif0**	fully supported
assign continuous assignment	fully supported; both explicit and implicit forms are supported
assign procedural continuous assignment	fully supported, but the **deassign** keyword may not be supported
function definitions	may only use supported constructs
task definitions	may only use supported constructs
always procedural block	must have a sensitivity list
begin—**end** statement groups	fully supported; both named and unnamed blocks are supported; **fork**—**join** statement groups are not supported
= blocking procedural assignment **<=** non-blocking procedural assignment	fully supported; may be restricted to using only one type of assignment for all assignments to the same variable

Verilog HDL Constructs	Notes
`if  if–else` **`case  casex  casez`** decision statements	logic X and Z only supported as “don’t care” bits
`for` loops	the step assignment must be an increment or decrement (**`+ -`**)
`while` loops **`forever`** loops	loop must take one clock cycle for each loop cycle (i.e.: an **`@(posedge clk)`** or **`@(negedge clk)`** must be within the loop)
`disable` statement group	must be used within the same named block that is being disabled
operators **`&  ~&  |  ~|`** **`^  ^~  ~^`** **`==  !=  <  >`** **`<=  =>`** **`!  &&  ||`** **`<<  >>`** **`{}  {{}}  ?:`** **`+  -  *  /`**	operands may be: • scalar or vector • constant or variable • the **`===`** and **`!==`** operators are not supported
vector bit selects vector part selects	fully supported on the right-hand side of an assignment; restricted to constant bit or part selects on the left-hand side of an assignment

New constructs in the Verilog-2001 standard that are expected to be supported by synthesis:

- Comma-separated sensitivity lists
- **`@*`** combinational logic sensitivity
- Combined port/data type declaration
- ANSI C style port declarations
- Implicit nets with continuous assignments
- Multi-dimensional arrays
- Array bit and part selects
- Signed data types
- Signed literal numbers
- **`<<<`**, **`>>>`** arithmetic shifts
- **`**`** power operator (may have restrictions)
- Recursive functions (the number of recursions must be able to be determined at elaboration time)
- Sized parameters
- Explicit in-line parameter passing
- **`‘ifndef`** and **`‘elsif`** compiler directives

Notes:

Symbols

A

B

C

D

suggested retail: $14.95

Verilog® HDL

Quick Reference Guide

based on the Verilog-2001 standard

(IEEE Std 1364-2001)

A complete reference on the Verilog Hardware Description Language, covering the syntax and semantics of the Verilog HDL. Many examples illustrate how to use Verilog. Includes synthesis supported constructs, common system tasks and a list of what is new in the Verilog-2001 standard. Fully indexed for easy reference.

published by

SUTHERLAND HDL

Sutherland HDL, Incorporated
22805 SW 92nd Place
Tualatin, OR 97062

(503) 692-0898

www.sutherland-hdl.com

Sutherland HDL, Inc. provides expert
Verilog and SystemVerilog training workshops

Sutherland HDL also sells the **Verilog PLI Quick Reference Guide**, covering the Verilog Programming Language Interface.

ISBN 1-930368-03-8

9 781930 368033