The Verilog 2001 Reference Guide

V3.1

Confidentiality Notice

Third Edition, Version 1 August 2007

Content and implementation by:-

Brian Dickinson

email: info@esperan.com
www.esperan.com

About this guide

The Esperan Verilog Reference Guide is intended as a concise reference for Verilog users writing RTL design and verification models.

The keywords section of this guide contains entries explaining each major construct or feature in the language, arranged in alphabetical order. Each entry includes a description of the construct, syntax and examples and rules for common use.

This guide also contains language definitions and short articles on Verilog

Note: As a concise reference, this guide does **not** cover the full language and some explanations have been deliberately simplified.

About Esperan

Esperan's mission is to provide the highest quality and broadest range of leading edge, independent methodology training to the electronic design community. Based upon the excellent reputation we have built up in VHDL and Verilog training worldwide, we are the only company to have recognized the need for education covering the widest possible range of electronic design methodologies.

Esperan's approach to training is unique in the electronics industry. Education is our sole mission, and this focus and commitment has allowed us to develop a style of business which gives you the best possible return on your investment with us. See our web site (`www.esperan.com`) for the attributes which set us apart from the training available from software tool vendors and small scale design consultancies.

Esperan Training Course Overview

VHDL Application Workshop	A comprehensive exploration of VHDL and its application to ASIC and FPGA design. Duration 5 days, which can be taken in two parts.
VHDL for Verilog Engineers	An intensive introduction to VHDL for engineers familiar with Verilog. Duration 2 days, or 3 days including code constructs for testbench design.
Verilog Application Workshop	A comprehensive exploration of Verilog and its application to ASIC and FPGA design. Duration 5 days, which can be taken in two parts.
Verilog for VHDL Engineers	An intensive introduction to Verilog for engineers familiar with VHDL. Duration 2 days, or 3 days including testbench design constructs.
SystemVerilog Application Workshop	A comprehensive exploration of Verilog and the advantages it offers for hardware design and verification. Duration 5 days.
Verification With VHDL	Methodology, language constructs and design techniques for effective testbench design, including PSL, TLM, SystemC and formal verification. 4 days.
Verification with PSL	A in-depth introduction to PSL, together with guidelines and methodologies for creating effective assertions for complex design properties. 2 days
Verification with SystemVerilog Assertions	Writing effective SystemVerilog Assertions (SVA), covering language, coding guidelines and methodologies. Duration 2 days.
SystemC Workshop	Verification and system modelling using SystemC, an open-source design language based on C++. An essential course for C testbench designers. 5 days
SystemC Verification	The SCV extensions to the SystemC language to apply advanced verification techniques.
Designing with Altera	Making the most of Altera technology resources and Quartus features from RTL-to-implementation.
Designing with Xilinx	Making the most of Xilinx technology resources and ISE features from RTL-to-implementation.
High Speed PCB Design	A comprehensive introduction to the fundamental principles of high-speed PCB design and EMI minimization. Duration 5 days
Tcl Scripting for EDA	An intensive introduction to the EDA standard scripting language Tcl. Duration 2 days.
Introduction to Perl	An introduction to Perl for designers seeking to use this powerful language for automating design flow tasks and data manipulation. Duration 3 days.

Contacts

General Contact Details
info@esperan.com
www.esperan.com

Esperan North America - USA, Canada

Phone: 1-800 220 8148 (sales)
Fax: 1-888 641 6431
Email: us-sales@esperan.com

Esperan Northern Europe - UK, Scandinavia

Phone: +44 (0) 1344 865436
Fax: +44 (0) 1344 865347
Email: info@esperan.com

Esperan Southern Europe - France, Spain, Belgium, The Netherlands, Italy

Phone: +33 1 34 88 53 48/54/83
Fax: +33 1 34 88 53 01
Email: training_france@esperan.com

Esperan Central Europe - Germany, Austria, Switzerland, East Europe:

Phone: +49 89 4563 1960
Fax: +49 89 4563 1919
Email: training_germany@esperan.com

Israel

Phone: +972 9 9511799
Fax: +972 9 9511796
Email: training_israel@esperan.com

Verilog Reference Contents

Verilog Event Scheduling 8
Tutorial A: RTL Coding Guidelines 10

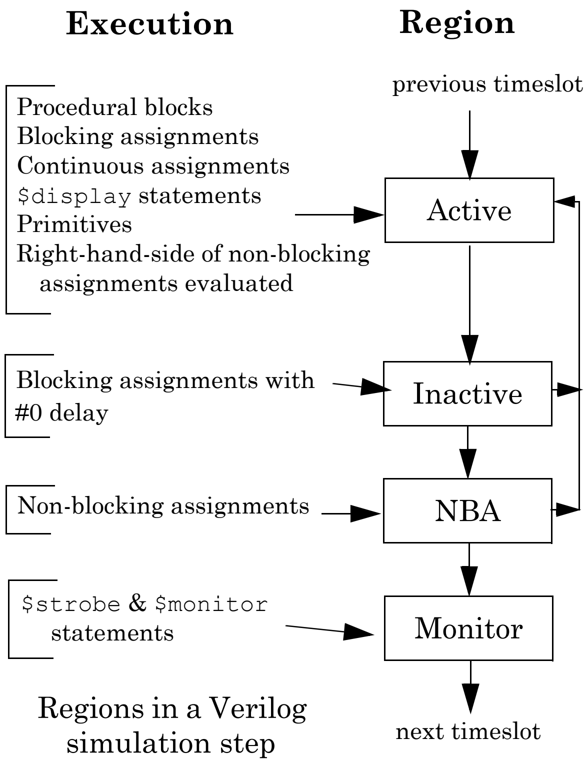
Language Reference

Verilog Keywords 15
'Compiler Directives 16
`'default_nettype` 17
`'define, 'undef` 18
`'ifdef,'ifndef,'else,'elsif` 19
`'include` 20
`'timescale` 21
\ Special Characters 22
$System Control 23
$System Control: Display Tasks 24
$System Control: Display Tasks 25
$System Control: File Open/Close 26
$System Control: File Input 28
$System Control: File Control 30
$System Control: File Output 31
$System Control: Formatted Read 32
$System Control: Loading Memory 33
$System Control: Plusargs 34
$System Control: Randomization 35
$System Control: SDF Annotation 36
$System Control: Simulation Control 37
$System Control: String Format 38
$System Control: Time Display 39
$System Control: Type Conversion 41
% Format Specifier 42
always 44
assign (continuous) 46
assign (procedural) 48
automatic 49
begin...end 51
Blocking Assignment = ... 52
case, casex, casez 54
cell 57
config 58
deassign 60
default (case) 61
default (configuration) 62
defparam 63
Delay # 64
design 66
disable 67
event 68
Event Timing Control @ 69

for 71
force.... 72
forever 73
fork...join.... 74
function 76
generate 78
genvar 81
if ... else if ... else 82
initial.... 84
inout 86
input.... 87
instance.... 89
liblist 90
library 91
Literals.... 92
localparam 93
module.... 94
negedge 97
Net Data Types.... 98
Non-Blocking Assignment <=.... 100
Operators 102
output.... 109
parameter 111
posedge.... 113
Primitive - built-in 114
Primitive - User-Defined (UDP) 115
Register Data Types.... 119
release 121
repeat 122
signed.... 123
specify 124
specparam.... 127
Strength Specification.... 128
String 129
Switch Devices.... 130
task.... 131
Timing Checks.... 135
use 138
wait 139
while.... 140
.... 96
Non-Blocking Assignment <=.... 98
Operators 100
output.... 107
parameter 109
posedge.... 111
Primitive - built-in 112
Primitive - User-Defined (UDP) 113

Verilog Event Scheduling

A *simplified* form of the Verilog event scheduler is as follows:-



Regions in a Verilog simulation step (simplified)

The Verilog event scheduler is evaluated as follows:-.

- In the **Active** region, `always` and `initial` procedural blocks are executed. Blocking assignments are updated in the Active region, but for Non-blocking assignments, only the right-hand-side is evaluated. `$display` statements are output when executed. Continuous `assign` statements are also executed and the inputs and outputs of primitives are updated.
- If a blocking assignment changes the value of a variable in procedural block event control, the block is triggered. If a variable in a continuous `assign` is changed, the `assign` is re-evaluated and if the variable is an input to a primitive, the primitive outputs are recalculated.

When there is not more activity in the Active region, the scheduler moves to the Inactive region:-

- In the **Inactive** region, blocking assignments made with zero delay are updated. Such assignments are NOT recommended as they usually indicate a race condition which is better fixed with other means.
- In the **NBA** region, Non-Blocking Assignments, evaluated in the active region, are updated.
- If a non-blocking assignment changes the value of a variable in procedural block event control, the block is triggered. If a variable in a continuous `assign` is changed, the `assign` is re-evaluated and if the variable is an input to a primitive, the primitive outputs are recalculated.

Once the design has reached a steady state, and there is no further activity at this point of simulation time, the event scheduler moves to the monitor region:-

- In the **monitor** region, `$strobe` and `$monitor` statements are output to capture steady-state values of variables before simulation time advances.

Simulation time then advances to the next scheduled event. Future events are scheduled using delayed assignments.

Note that this is a deliberately simplified description of the Verilog event scheduler. The Procedural Language Interface (PLI), which allows joint simulation of Verilog and C-code, has many dedicated event regions in the scheduler which are not described here.

Tutorial A: RTL Coding Guidelines

RTL code for synthesis is written using two forms of Verilog procedures, one for pure combinational logic, and the other for the inference of clocked registers. These procedures must conform to specific templates and coding styles to be synthesizable.

The guidelines described here are compatible with virtually all synthesis tools. Some tools may support enhancements to these templates, but such enhancements may not be portable across all tools.

Combinational Procedures

Rule 1: A RTL combinational procedure must have a **complete sensitivity list**, where all of the signals read in the process are included in the sensitivity list.

```
always @(a or b or sel)
  if (sel)
    op = a;
  else
    op = b;
```

In Verilog2001, the * character can be used to infer an automatic event list:-

```
always @(*)  // Verilog2001
...
```

Rule 2: A RTL combinational procedure must use **blocking assignment**.

```
always @(a or b or sel)
  if (sel)
    op = a;
  else
    op = b;
```

Rule 3: A RTL combinational procedure must have a **complete assignment** to every output variable. If there is an execution path through the procedure where an output variable is not updated, then that variable must retain its previous value. To implement this in hardware, a synthesis tool will use a transparent latch.

```
always @(a or b or sel)
  // latch inferred on op
  if (sel)
    op = a;
```

Incomplete assignment can be avoided by use of unconditional assignment branches in all conditional statements (`else` clauses for `if` statements and `default` branches for `case` statements) or by using default assignments to output signals

```
always @(a or b or sel)
begin
  op = b;
  if (sel)
    op = a;
end
```

Rule4: Verilog **case statements**.

Case statements with overlapping branch conditions are implemented using priority logic.

A case with no overlapping conditions is known as a **parallel case** and can be implemented with more efficient non-prioritized logic. Most synthesis tools can recognize parallel cases, but others may need to be told with a **synthesis directive**, an embedded comment which is recognized by synthesis tools.

```
case(ip)
  // rtl_synthesis parallel_case
  0       : y = a;
  1       : y = b;
  default : y = c;
endcase
```

Case branches do not need to cover all values for the case expression, but can imply a latch for synthesis.

A **full case** (for synthesis) has a branch for all binary values of the case expression (ignoring x and z). Most synthesis tools can recognize full cases, but others may need to be told with a synthesis directive.

```
case(ip)
  // rtl_synthesis full_case
  0       : y = a;
  1       : y = b;
  default : y = c;
endcase
```

Warning—if a full case directive is applied to a non-full case, missing case values can be implemented as "don't care". The full case issue is avoided if `default` case branches are always used.

Clocked Procedures

Rule 1: A clocked procedure uses **edge-triggered events** in its sensitivity list. `posedge` for rising-edge triggered and `negedge` for falling-edge triggered. The variable associated with the `posedge`/`negedge` becomes the clock of the inferred register(s).

```
always @(posedge clk)
  // rising edge triggered
  q <= d;
```

```
always @(negedge clk)
  // falling edge triggered
  q <= d;
```

Rule 2: **Non-blocking assignment** must always be used in clocked procedures to infer registers and avoid race conditions. A register will be inferred for every non-blocking variable assignment in a correctly constructed clocked procedure.

```
always @(posedge clk)
  q <= d;
```

Blocked assignment can only be used in clocked procedures for temporary variables, i.e. variables which are written first and then read. Temporary variables should not infer registers in synthesis.

```
always @(posedge clk)
begin
temp = a + b; // temporary
q <= temp + c;
end
```

Rule 3: **Resets** are inferred using top-level `if` statements in the clocked procedure.

```
always @(posedge clk)
  // active high synchronous reset
  if (rst)
    q <= 0;
  else
    q <= d;
```

Note that no statements can be placed before or after the `if` statement which checks the reset.

For asynchronous resets, the correct edge of the rest variable must be added to the procedure event list.

```
always @(posedge clk or negedge rst)
  // active low asynchronous reset
  if (!rst)
    q <= 0;
  else
    q <= d;
```

A clocked procedure is **only** sensitive to an edge of a clock and an edge of an asynchronous reset (if required).

The rules for clocked procedures are different and contrary to those for a combinational procedure:-

1 Only the clock and asynchronous reset (if used) should be in the sensitivity list of a clocked procedure, whereas a combinational procedure must have a complete sensitivity list.

2 Clocked procedures are triggered off a specific edge of a clock/asynchronous reset using posedge or negedge in the event list, whereas combinational procedures trigger off any change in value of input variables.

3 Clocked procedures must use non-blocking assignment to infer registers, whereas combinational procedures must use blocking assignment.

4 Incomplete assignment in clocked procedures infers synchronous feedback and so is useful for describing clock enables. For example:

```
always @(posedge clock)
  if (enable)
    q <= d;
```

Verilog Keywords

Verilog keywords in the IEEE 1364-2001 Language Reference Manual.

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

Compiler Directives

Standard IEEE 1364-2001 compiler directives

`celldefine	`define
`endcelldefine	`undef
`default_nettype	`ifdef
`include	`ifndef
`resetall	`else
`timescale	`elsif
`unconnected_drive	`endif
`nounconnected_drive	

System Tasks and Functions

Selected IEEE 1364-2001 system tasks and functions:-

$bitstoreal	$random
$display(b/h/o)	$readmem(b/h)
$fclose	$realtime
$fdisplay(b/h/o)	$realtobits
$ferror	$rewind
$fflush	$rtoi
$fgetc	$sdf_annotate
$fgets	$sformat
$finish	$signed
$fmonitor(b/h/o)	$sscanf
$fopen	$stime
$fread	$stop
$fscanf	$strobe(b/h/o)
$fseek	$swrite(b/h/o)
$fstrobe(b/h/o)	$test$plusargs
$ftell	$time
$fwrite(b/h/o)	$timeformat
$itor	$ungetc
$monitor(b/h/o)	$unsigned
$monitoroff	$value$plusargs
$monitoron	$write(b/h/o)
$printtimescale	

The following timing check constructs are not considered as system tasks or functions:-

```
$setup $hold $setuphold $recovery $removal
$recrem $skew $timeskew $fullskew $width
$period $nochange
```

See `Timing Checks` on page 135 for more information on these constructs.

`Compiler Directives

Embedded commands in Verilog code which provide information for the compiler. Directives are file-order dependent, i.e. their scope extends from the point of compilation, across all following files, until either superseded by another compiler directive or the compilation ends.

Directives are preceded by the accent grave character (`) and are **not** terminated by a semicolon.

The more complex compiler directives are described in detail on the following pages, while the simpler directives are summarised below.

Many simulator vendors supply additional compiler directives, but this list is confined to those declared in the Verilog2001 Language Reference Manual.

Simple Directives

`celldefine, `endcelldefine

Used to identify modules as cell modules. Cells are used in certain PLI routines, e.g. delay calculators.

`resetall

Sets all compiler directives to default values. This is useful for ensuring that only required directives are active when compiling a particular source file.

`unconnected_drive, `nounconnected_drive

Defines that all unconnected module inputs between the directives are automatically pulled-up or pulled-down depending on the argument supplied to the `unconnected_drive` directive. These directives are placed outside a module declaration:-

```
`unconnected_drive pull0
// unconnected inputs pulled down
module mux4 (input  wire [3:0] a,
             input  wire [3:0] b,
             input  wire sel,
             output reg [3:0] op);
...
endmodule
`nounconnected_drive
```

`default_nettype

Defines the type of implicit net declarations, i.e. where identifiers are used without being fully declared.

Syntax

```
`default_nettype nettype
```

Where *nettype* is one of `wire, tri, tri0, wand, triand, wor, trior, trireg` or `none`.

This compiler directive must be placed outside a module declaration.

Rules and Examples

Undeclared variables default to a single bit wire in Verilog.

```
module test;
  assign undec = 1'b0;  // wire
...
```

The default data type can be over-ridden with the compiler directive `default_nettype`

```
`default_nettype tri
module test;
  assign undec = 1'b0;  // tri
...
```

The directive affects all modules that follow the directive, across multiple files. Multiple directives are allowed as later directives supersede those previous.

Verilog2001 default_nettype none

Verilog2001 adds the net type option `none`. With this option, any undeclared variables will generate a compiler error. This prevents connectivity problems due to a mis-type in a variable name.

`none` is **not** a reserved word

```
`default_nettype none
module test;
  //error - undefined object undec
  assign undec = 1'b0;
...
```

Defines a text macro which is processed by the compiler.

Syntax

```
'define name text
'undef name
```

The compiler replaces every occurrance of `'name` with `text`. `'undef` removes the definition of a macro.

Rules and Examples

The macro text can be any arbitrary text and terminates at a line break or comment specifier. Text can break over several lines by preceding a new-line with a backslash (\). Defines can also use other text macros in the text definition.

Common uses for `` `define `` are:

- Make code more readable.

```
'define vfile "/d1/library/vectors"
initial $readmemb('vfile, mema);
```

- Define global design parameters, like vector widths, in a single place. Then to alter the width, only the define needs to be changed.

```
'define width 7
reg ['width:0] vec8;
reg ['width:0] mema [1:1024];
```

- Define shorthand strings for Verilog commands.

```
'define e {b, c, a}
always @ (posedge clk)
  f <= 'e;       // f <= {b, c, a};
```

Defines can also contain arguments in brackets after the macro name:-

```
'define max(a,b) ((a) > (b) ? (a) : (b))
n = 'max(v1, v2);
// expands to
// n = ((v1) > (v2) ? (v1) : (v2));
```

Defines can also be overridden from the command line (check your simulator command line options).

Used with `` `define `` text macros to enable conditional compilation, where a section of code is only compiled if a specific test macro is defined.

Syntax

```
`ifdef name1
  name1_code
`elsif name2
  name2_code
`else
  else_code
`endif
```

If the text macro `name1` is defined, then `name1_code` is compiled, otherwise if macro `name2` is defined, `name2_code` is compiled, otherwise the `else_code` is compiled. The `` `elsif `` and `` `else `` branches are optional.

Rules and Examples

```
`define init2zero  // no value
reg [31:0] memory [0:255];
integer i;

initial
  `ifdef init2zero
     for (i=0; i < 255; i=i + 1)
       memory[i] = 0;
  `else
     $readmemb("rdata.dat",memory);
  `endif
...
```

In the above example, a ROM can be initialized either to all zeros or to values read from a file, depending on whether the macro `init2zero` is defined.

Note the macro does **not** need a text value defined for the purposes of an `ifdef`.

`ifdef` directives can be nested and the directives can appear anywhere in the code. There is also a `'ifndef` directive which checks if a macro has **not** been defined.

`define` can be overridden from the command line, allowing easier control over compilation.

`include

Inserts the contents of an entire file at the point where the directive is compiled.

Syntax

```
`include "filename"
```

where `filename` is an absolute or relative pathname to a text file containing Verilog code.

Rules and Examples

If the file `globals.txt` has the following contents:-

```
// globals.txt
localparam PERIOD = 20;
```

and the file is included in the following code:-

```
`include "globals.txt"
always begin
  #(period/2) clk = 0;
  #(period/2) clk = 1;
  end
```

The compiler replaces the `` `include `` with the `localparam` declaration.

There are no restrictions on the placement of include directives. Include files can be nested, i.e. an included file can contain another `` `include `` directive.

Compilers can automatically translate between UNIX and DOS format pathnames.

A common use for `` `include `` is to declare design constants (e.g. `localparams`) or repeated code sections, e.g. function declarations, in one file and include the file throughout the design, where required.

Warning—using directory names in the include file pathname implies a specific directory structure. This can affect the portability of the design and is a common problem with include files.

```
// absolute pathname is bad
`include "~/mydesign/source/globals.txt"
// relative pathname better
`include "../source/globals.txt"
// no directory path is best
`include "globals.txt"
```

Specifies the time unit and time precision for a simulation. The time unit specifies the units of measurement for delays and time. The precision tells the simulator how to round delay values.

Syntax

```
`timescale unit / precision
```

Both `unit` and `precision` are declared as an integer, followed by a string. Valid integers are `1`, `10`, `100`. Valid strings are the time units `fs`, `ps`, `ns`, `us`, `ms`, `s`. The `precision` must be less than or equal to the `unit`. Timescales must be placed **outside** the module.

Rules and Examples

```
`timescale 1ns/100ps
module first;
  ...
  #10; // 10 ns delay
  ...
```

In module `first`, time units are scaled to 1 ns with a precision of 100 ps. There is no rounding on the delay, so it is simply scaled to 10 ns.

```
`timescale 10ns/1ns
module test;
  parameter tdelay = 2.55;
  ...
   #tdelay; // 26 ns delay
   ...
```

In module `test`, time units are scaled to 10 ns with a precision of 1 ns. So `tdelay` is rounded and scaled to 26 ns. The timescale is applied before delays are calculated, so `2*tdelay` is 52, not 51 ns.

It is a compilation error if some modules have a timescale specified and others do not. Some simulators have a default timescale compilation option which can be used for all blocks without a specific timescale.

The `$printtimescale` system task displays the current time unit and precision of a module.

See Also

$time, $realtime, $timeformat, $printtimescale

\ Special Characters

Special characters can be used in string literals to print any ASCII character, including tabs, new-lines and non-printing characters.

Display, file output and string format system tasks use arguments which can include string literals.

Special Characters	
`\xxx`	converts 1-3 octal digits to ASCII
`\\`	single backslash `\`
`%%`	single percentage `%`
`\t`	horizontal tab
`\n`	new-line
`\"`	quote

For example:-

```
$display("var1 is \t", var1);
```

when executed, displays the string "`var1 is `" followed by a horizontal tab, followed by the value of `var1` in decimal.

Note that inserting a percentage `%` is different from other special characters as it is used within strings in many System Tasks and Functions as part of a format specifier.

See Also

$System Control: Display tasks, Strings, % Format Specifiers

$System Control

System Tasks and Functions are useful, built-in subprograms which are part of the Verilog language. Collectively that are also known as System Control. They are particularly useful in verification to read stimulus data from files and write out response data, and in debugging, to show variable values.

- Display tasks (`$display`, `$write`, `$strobe`, `$monitor`, `$monitoron`, `$monitoroff`)
- File open/close (`$fopen`, `$fclose`)
- File input (`$fgetc`, `$ungetc`, `$fgets`, `$fread`)
- File control (`$rewind`, `$ferror`, `$fseek`, `$ftell`, `$fflush`)
- File output (`$fdisplay`, `$fmonitor`, `$fstrobe`, `$fwrite`)
- Formatted Read input (`$fscanf`, `$sscanf`)
- Memory loading (`$readmemh`, `$readmemb`)
- Plusargs - command line arguments (`$test$plusargs`, `$value$plusargs`)
- Simulation control (`$finish`, `$stop`)
- String format (`$swrite`, `$sformat`)
- Time formatting and display (`$realtime`, `$time`, `$timeformat`, `$stime`, `$printtimescale`)
- Type conversion (`$bitstoreal`, `$realtobits`, `$itor`, `$rtoi`, `$signed`, `$unsigned`)
- Reading timing data (`$sdf_annotate`)
- Random number generator (`$random`)

Not all System Tasks and Functions are described in detail here. For those which are not, please refer to chapter 17 of the Verilog Language Reference manual, IEEE1364-2001.

These tasks are typically used to display status messages and show variable values:-.

`$display(<args>)` `$displayb(<args>)` `$displayo(<args>)` `$displayh(<args>)`	Displays arguments in either decimal, binary, octal or hex formats and adds a new-line character to the end
`$write(<args>)` `$writeb(<args>)` `$writeo(<args>)` `$writeh(<args>)`	Displays arguments in either decimal, binary, octal or hex formats, without adding a new-line character
`$strobe(<args>)` `$strobeb(<args>)` `$strobeo(<args>)` `$strobeh(<args>)`	Displays arguments in either decimal, binary, octal or hex formats, at the end of the current simulation time*
`$monitor(<args>)` `$monitorb(<args>)` `$monitoro(<args>)` `$monitorh(<args>)`	Displays arguments in either decimal, binary, octal or hex formats, every time a variable in the argument list changes value. Variable values are sampled at the end of the simulation time*. Only one monitor can be active at a time
`$monitoroff`	Disables monitor.
`$monitoron`	Enables monitor. When a disabled monitor is re-enabled, the arguments are displayed.

where `<args>` is a list of string literals; variable names; format specifiers (see `% Format Specifier` on page 42) and special characters (see `\ Special Characters` on page 22).

*`$monitor` and `$strobe` execute in the monitor region of the Verilog event scheduler, when variables have reached a steady-state value for the current simulation time. See `Verilog Event Scheduling` on page 8.

```
initial begin
  var1 = 8’h63;
  $displayb(var1);   // "01100011"
  $display(var1);    // " 99"
end
```

The output is sized to the variable, i.e. an 8 bit variable is always written in decimal as three characters.

For example this code...

```
initial begin
 v1 = 8'h0F;                    // 15
 v2 = 8'b01010101;              // 85
 v3 = 8'b11001100;              // 204

 $display(v1, v2, v3);

 //extra comma adds space between values
 $displayh(v1,, v2,, v3);

 //use string literals for info & spacing
 $displayh("v2 = ",v2,"; v1 = ",v1);

 //format specifiers control value display
 $write("v1: %h (hex) %o (octal)",v1,v1);
 $write("%d", v1, " (decimal));
end
```

... generates the following:-

```
 15 85204
OF 55 CC
v2 = 55; v1 = CC
v1: 0f (hex) 17 (octal)  15 (decimal)
```

`$display` and `$write` execute immediately using the current variable values. `$strobe` and `$monitor` execute at the end of the current simulation time period using the final variable values:-

```
initial begin
 v1 = 8’h0f;
 $strobe("strobe %h", v1);
 $display("display %h", v1);
 v1 = 8’hf0;
 #10;  //strobe executes here
...
```

Generates the following:-

```
strobe f0
display 0f
```

See Also

$System Control: File Output

`% Format Specifier` on page 42

`\ Special Characters` on page 22

$System Control: File Open/Close

These tasks are used to open and close files. This tasks are slightly different in Verilog95 and Verilog2001:-

Verilog95 File Open/Close

`mcd = $fopen("name")`	Opens the file `name` and returns a Multi-Channel Designator into `mcd`. Returns 0 if open fails.
`$fclose(mcd)`	Closes the file associated with the Multi-Channel Designator `mcd`.

```
integer data_chan;
initial begin
  data_chan = $fopen("data.txt");
  if (!data_chan) $finish;
  $fmonitor (data_chan, var1, var2);
  ...
  $fclose(data_chan);
end
```

Opens file `data.txt` and saves the MCD into `data_chan`, checking for success. The file output task `$fmonitor` writes to the file and the file is later closed. `data_chan` must be declared as type `integer`.

Verilog95 Multi-Channel Designator (MCD)

In Verilog1995, `$fopen` returns a 32-bit unsigned Multi-Channel descriptor (MCD) which is a pointer to the file opened by the task.

The MCD can be considered as a set of 32 bits, where each bit represents a single IO channel. The least significant bit is reserved for the standard output (simulator transcript window). All other bits are available for files to be opened by `$fopen`. Therefore a total of 31 files can be open simultaneously.

As each bit represents a file, a pointer to *multiple* files can be created by logically OR-ing the MCD's.

```
datafile = $fopen("data.txt");
timefile = $fopen("timing.txt");
multi = datafile | timefile;
$fdisplay(multi, "Output to both files");
```

Verilog2001 File Open/Close

`fd = $fopen("name", type)`	Opens the file `name` in mode `type` and returns a file descriptor into `fd`. Returns 0 if open fails.
`$fclose(fd)`	Closes the file associated with the file descriptor `fd`.

Type options for Verilog2001 `$fopen`:-

Option	Description
`"r"` or `"rb"`	read from existing file
`"w"` or `"wb"`	write to file (clear if exists, create file if not)
`"a"` or `"ab"`	append to existing file or create new file
`"r+"` or `"r+b"` or `"rb+"`	read/write (existing file)
`"w+"` or `"w+b"` or `"wb+"`	read/write (new file)
`"a+"` or `"a+b"` or `"ab+"`	append (existing or new file)

```
integer data_chan;
initial begin
  data_chan = $fopen("data.txt", "rb");
  if (!data_chan) $finish;
  $fmonitor (data_chan, var1, var2);
  ...
  $fclose(data_chan);
end
```

Opens file `data.txt` in read mode and saves the file descriptor into `data_chan`. The file output task `$fmonitor` writes to the file and the file is later closed. `data_chan` must be declared as type `integer`.

Verilog2001 File Designator (fd)

In Verilog2001, `$fopen` returns a 32 bit file descriptor (`fd`). Bit 32 of an `fd` is always set to distinguish it from an MCD. Unlike an MCD, each IO channel is assigned an integer value, not a bit. Three `fd`s are pre-defined; `STDIN` (32'h8000_0000), `STDOUT` (32'h8000_0001) and `STDERR` (32'h8000_0002). Unlike MCD's, `fd`s can not be combined to direct output to multiple files.

These tasks extract information from files:-

`c = $fgetc(fd)`	Character read - reads a byte from file `fd` into `c`. `c = -1` if the read is unsuccessful
`i = $ungetc(c,fd)`	Character push - pushes byte `c` into read buffer for file `fd` so the next `$fgetc` will read `c`. `fd` is not changed. `i` = `-1` if push unsuccessful.
`i = $fgets(str,fd)`	Line read - reads line of data from file `fd` into string `str` (including new-line) or fills `str`. i = number of characters read. `i` = 0 on read error
`i = $fread(var, fd, start, count)`	Binary read - reads binary data from file `fd` into `reg` or memory `var`. i = number of characters read. `i` = 0 on read error. If `var` is a memory, an optional `start` address and optional number of locations `count` can be used

Given file `source.txt` with ASCII contents:-

```
ABCD
1234
```

Read File By Character

`source.txt` can be read a character at a time:-

```
reg [7:0] char;
integer fd;
initial begin
 fd = $fopen("source.txt", "rb");
 char = $fgetc(fd);
 ...
```

`$fgetc` loads `8'h41` = ASCII character `A` into `char`. `$fgetc` reads line-feeds (`8'h0a`) *and* End-Of-File EOF (`8'hff`) characters from the file. This allows `$fgetc` to read all characters in a file by reading up to the EOF:-

```
while (char !== 8'hff) begin
  ... // process character
  char = $fgetc(fd);
end
```

Line Read

source.txt can be read a line at a time:-

```
reg [8*6:0] str;
integer fd, c;
initial begin
  fd = $fopen("source.txt", "rb");
  c = $fgets(str,fd);
...
```

$fgets reads 8-bits for every character up to *and* including the first Line-Feed character (8’h0a) or until the string variable is full. End-Of-File characters (8’hff) are *not* read. $fgets loads str with:-

```
00414243440a =   ABCD¬
```

Where ¬ is a new-line character. See also String on page 129.

Binary Read

source.txt can be read as binary data:-

```
reg [8*11:0] vara;
integer fd, c;
initial begin
  fd = $fopen("source.txt", "rb");
  c = $fread(vara,fd);
  ...
```

$fread loads vector vara with 8-bits for every character read up to, but *not* including, the End-Of-File character (8’hff), or until the entire file is read. $fgets loads str with:

```
00414243440a313233340a =  ABCD¬1234¬
```

Where ¬ is a new-line character.

Start and count addresses are handled similar to $readmemb (see $System Control: Loading Memory).

See Also

$System Control: Loading Memory

\ Special Characters, String

These Verilog2001 tasks are used to navigate within files and handle error conditions:-

`$fflush(fd)`	Any remaining data in the write buffer for file `fd` is written to file
`i = $rewind(fd)`	sets next file operation to *start* of file `fd`. `i` = `-1` if the rewind is unsuccessful, `0` otherwise
`i = $ferror(fd,str)`	Error number for the latest I/O error on file `fd` written to `i` and string description of error written to `str` (`str` must be at least 640 bits wide)
`i = $fseek(fd, offset, op)`	sets position of file pointer for file `fd` depending on `op`:- 0 – start of file + `offset` `1` – current position + `offset` `2` – end of file + `offset` `offset` is a signed value. `i` = `-1` if reposition fails (e.g. op = 2 and `of` is positive), 0 if successful.
`$ftell`	returns the number of the **next** byte to be read from the file `fd`.

```
integer fd, ec;
reg [7:0] char;
initial begin
  fd = $fopen ("file.txt", "r");
  $write("byte %d ", $ftell(fd));   // 0
  $display(" is %c", $fgetc(fd));
  ec = $fseek(fd, 8, 1);
  $write("byte %d ", $ftell(fd));   // 9
  $display(" is %c", $fgetc(fd));
  ec = $rewind(fd);
end
```

This writes the first read location (`$ftell` = 0) of `file.txt`; reads the byte there (`$fgetc`) and displays it as a character. Then we jump forward 8 bytes (`$fseek`) from the current location and display the byte at location 9. Finally the file pointer is set to the beginning of the file (`$rewind`). Remember after the first `$fgetc`, `$ftell` is 1.

These tasks are identical to the display tasks, but output their arguments to a file rather than the standard output:-

`$fdisplay(fd,<args>)`
`$fdisplayb(fd,<args>)`
`$fdisplayo(fd,<args>)`
`$fdisplayh(fd,<args>)`
	Writes arguments in either decimal, binary, octal or hex formats to file `fd`, appending a new-line

`$fwrite(fd,<args>)`
`$fwriteb(fd,<args>)`
`$fwriteo(fd,<args>)`
`$fwriteh(fd,<args>)`
	Writes arguments in either decimal, binary, octal or hex formats to file `fd`, without a new-line

`$fstrobe(fd,<args>)`
`$fstrobeb(fd,<args>)`
`$fstrobeo(fd,<args>)`
`$fstrobeh(fd,<args>)`
	Writes arguments in either decimal, binary, octal or hex formats to file `fd` at the end of the current simulation time*

`$fmonitor(fd,<args>)`
`$fmonitorb(fd,<args>)`
`$fmonitoro(fd,<args>)`
`$fmonitorh(fd,<args>)`
	Displays arguments in either decimal, binary, octal or hex formats every time a variable in the argument list changes value. Variable values are sampled at the end of the simulation time*.

where `<args>` is a list of string literals; variable names; format specifiers (see `% Format Specifier` on page 42) and special characters (see `\ Special Characters` on page 22).

*`$fmonitor` and `$fstrobe` execute in the monitor region of the Verilog event scheduler, when variables have reached steady-state values. See `Verilog Event Scheduling` on page 8.

Unlike `$monitor`, any number of `$fmonitor`s can be active at a time. Also there are no `$fmonitoron`/ `$fmonitoroff` tasks. Executing a `$fclose` on the monitor file(s) terminates the `$fmonitor` activity.

See Also

$System Control: Display Tasks

`% Format Specifier` on page 42

`\ Special Characters` on page 22

These tasks read formatted data from string variables or files, interprets the data according to a specified format and loads information which matches the format into multiple variable arguments.

`i = $fscanf(fd, format, args)`	Reads characters from file `fd`, interpreted by string `format`, and stores results in `args`.
`i = $sscanf(var, format, args)`	Reads characters from array variable `var`, interpreted by string `format`, and stores results in `args`

For both functions, i returns the number of matched and assigned arguments. i = 0 if there are more arguments than fields in the format, or -1 for no matches. Excess arguments are ignored.g. `Format` can be a string variable, allowing dynamic formatting.

Assuming a file `data.txt` containing the following:-

```
data = 8'hff
```

Then the identifier and value can be read as follows:-

```
reg [8*4:1] str;   // string variable
reg [7:0] val;
integer c, fd;
initial begin
  fd = $fopen("data.txt, "rb");
  c = $fscanf(fd, "%s = 8'h%h", id, var);
...
```

There are two conversion specifiers in the format argument, `%s` string to match `data` and `%h` hex to match `ff`. After `$fscanf` execution, `c` = 2 indicating 2 successfully matched fields, `id` = `32'h64617461` = ASCII string `data` and `var` = `8'hff`.

Conversion specifiers are very similar to format specifiers, but take the form `%*nF` where the optional `*` indicates the value is not assigned to an argument and the optional integer `n` is the maximum number of characters to be read. `F` is a format specifier value.

```
str = "ffaabb";
c = $sscanf(str, "%2h%*2h%2h", v1, v2);
```

`c` = 2; `v1` = `ff` (first 2 hex digits of string), `v2` = `bb` (last 2 hex digits). The middle 2 hex digits are not assigned as indicated by the `*` option.

These tasks load data from a specific text file into a specific memory (2-dimensional array)

`$readmemb("name", mem, start, end)`	Reads binary data from file `name` into successive locations of `mem`.
`$readmemh("name", mem, start, end)`	Reads binary data from file `name` into successive locations of `mem`.

The text file can only contain the following data:-

- White-space (spaces, tabs, form-feeds or new-lines)
- Comments (line or block comments)
- Binary or hex numbers including under-scores and `X`, `Z` values, but with no length or base specifiers
- `@` address specification

By default, each number in the file, delimited by white-space, is loaded into a memory location starting with the left-most index and loading successive locations towards the right-most index. An optional `start` argument defines the memory start address and if supplied, then an optional end argument can define the final address.

```
reg [7:0] array4 [3:0];
reg [7:0] array7 [6:0];
initial begin
  $readmemh("vect.txt", array4);
  $readmemh("vect.txt", array7, 2, 5);
end
```

Given the `vect.txt` file contents:-

```
0 1 2 3
```

Then `array4[3:0]` has the contents `{3,2,1,0}` and `array7[6:0]` has contents `{X,3,2,1,0,X,X}`.

An address specifier can also be embedded in the file using the format `@<hex_address>`. When an address specifier is encountered, subsequent data is loaded starting at that address.

A simulator may report errors if there are too many or too few data items for the address specifiers or if they refer to out-of-range addresses for the memory.

Tasks to control simulation using command line arguments (plusargs):-

`$test$plusargs (string)`	Tests plusargs against a set string. Returns non-zero if string matches beginning of a plusarg, zero otherwise
`$value$plusargs (string, var)`	Tests plusargs against a user-defined string with format specifiers. Returns non-zero if string matches beginning of a plusarg and stores string in variable var, converting value according to specifiers in string. Returns zero otherwise.

plusargs are command-line simulator arguments added with a "+" character prefix, e.g. xsim +TEST1...

$test$plusargs tests the start of every plusarg for a specified string:-

```
initial begin
  if ($test$plusargs("TEST1"))
    $display("Running test 1")
  ...
```

For a simulation with a +TEST1 plusarg, this code displays the message. Note that a plusarg +TEST10 would *also* display the message. Only the start of the plusarg has to match.

$value$plusargs allows the *value* of a plusarg to be used in the simulation:-

```
reg [31:0] tnum
initial begin
  if ($value$plusargs("+TEST%d", tnum)
   $display("Running test %0d.", tnum);
  ...
```

For a simulation with a +TEST8 plusarg, this code extracts the decimal characters following +TEST which match the %d specifier, copies them into tnum, and displays the message Running test 8. Note that the following format specifiers are allowed for conversion of the string:-

```
%d, %o, %h, %b, %e, %f, %g, %s
```

See % Format Specifier on page 42 for more details.

A function to generate a random number for use in verification.

`i = $random(seed)`	Generates a random 32-bit signed integer for i. `seed` is an optional argument.

Where optional argument `seed` determines the sequence of random numbers. `Seed` must be an `integer`, `reg` or `time` variable.

`$random` can be called directly for a 32-bit signed random number or used with the modulus operator `%` to create signed numbers in a specific range.

```
integer rand;
initial begin
  rand = $random;        // 32-bit
  rand = $random % 100;   // -99 to +99
  rand = {$random} % 100; // 0 to 100
```

By enclosing just the `$random` call in concatenation brackets, the random value is treated as an unsigned number (concatenation is always unsigned) and positive random values can be generated.

Pseudo-Randomization and Seeds

`$random` is pseudo-random, meaning the same sequence of numbers is always generated. This is essential for verification as a sequence of random numbers needs to be repeatable in order to detect, debug and fix errors.To create different sequences, a seed argument can be passed to the function, usually *once only* at the start of simulation. Each different starting seed will create a difference sequence.

```
integer rand;
reg [31:0] seed := 4335;
initial begin
  rand = $random(seed);
...
```

Some simulators have an command-line option to set the seed, or alternatively plus-arguments can be used (see `$System Control: Plusargs` on page 34).

See Also

`signed`

$System Control: SDF Annotation

A task to read Standard Delay Format (SDF) information from a file into a specific region of a design.

```
$sdf_annotate
  ("sdf_file", module_instance,
   "config_file", "log_file",
   "mtm_spec", "scale_factors",
   "scale_type");
```

Where the task arguments are as follows:-

`sdf_file`	Name of SDF file
`module_instance`	Optional name of design scope to be annotated. Defaults to scope containing `sdf_annotate` call
`config_file`	Optional filename containing more detailed annotation control
`log_file`	Optional filename to log every SDF annotation
`mtm_spec`	Optional string to define which delay to use:- `MAXIMUM` - use max delays `TYPICAL` - use typ delays `MINIMUM` - use min delays `TOOL_CONTROL` (default) - use simulation option
`scale_factors`	Optional string defining scaling factors for delays, e.g. `"1.5:1.3:1.0"` scales min delays by 1.5, typ by 1.3 and max by 1.0. Default is `"1.0:1.0:1.0"`
`scale_type`	Optional string defining how scaling factors are applied to delays:- `FROM_MAXIMUM` - scale max `FROM_TYPICAL` - scale typ `FROM_MINIMUM` - scale min `FROM_MTM` (default) -scale min/typ/max values

Tasks to stop the simulation:-

`$stop(n)`	Ends simulation. Optional integer n defines diagnostic message to be printed
`$finish(n)`	Ends simulation and **exits** simulator. Optional integer n defines diagnostic message to be printed

The optional value n defines diagnostic message levels for both `$stop` and `$finish` as follows:-

`$stop(0)`	No message printed Identical to `$stop` or `$finish`
`$stop(1)`	Prints simulation end-time and location
`$stop(2)`	Prints simulation end-time and location, together with statistical information on CPU and memory usage

```
initial begin
  ...
  LOAD_T <= 1'b1;
  # 10
  LOAD_T <= 1'b0;
  # 100
  $display ("Simulation Over");
  $finish;
end
```

`$finish` exits the simulator and so is better for batch-mode or command line simulation runs.

```
always @(warning)
  if(warning) begin
    $display("Expected != actual");
    $stop;
  end
```

`$stop` only stops simulation, and the simulation can be continued from the stop point or restarted. Therefore $stop is better for interactive simulation runs.

These tasks are similar to the `$fwrite` tasks except they write string data to a `reg` variable rather than to a file:-

`$swrite(var,<args>)`
`$swriteb(var,<args>)`
`$swriteo(var,<args>)`
`$swriteh(var,<args>)`
	Writes arguments in either decimal, binary, octal or hex formats to `reg` array `var`

`$sformat(var, format, <args>)`	Writes arguments to `reg` array `var` according to `format` information

where `<args>` and `format` are a list of string literals; variable names; format specifiers (see `% Format Specifier` on page 42) and special characters (see `\ Special Characters` on page 22).

```
reg [8*2:1] str;
reg [1:0] var;
initial begin
  var = 2'b10;
  $swrite(str, var);      // 16'h0032 = 2
  $swriteb(str, var);     // 16'h3130 = 10
  $swrite(str,"%b",var); // 16'h3130 = 10
end
```

`$swrite` loads the variable `str` with the value of `var`, converted to ASCII according to the version used (e.g. `$swriteb` = binary) or format specifier (`%b`). See `String` on page 129 for details on how Verilog interprets variables as string data.

The key difference for `$sformat` is that the `format` argument is allowed to be a *variable* containing formatting information. This allows dynamic formatting of string writes:-

```
reg [8*2:1] str, fmt;
reg [1:0] var;
initial begin
  var = 2'b10;
  fmt = "%b";     // binary format
  $sformat(str,fmt,var); // 16'h0032 = 2
  fmt = "%d";     // decimal format
  $sformat(str,fmt,var); // 16'h3130 = 10
...
```

See Also

$System Control: File Output, String

These functions are used to display, and control the display, of the current simulation time:-

`$realtime`	returns the current simulation time as a **real** number, scaled to time unit of its module
`$stime`	returns the current simulation time as an unsigned, **32 bit integer**, scaled to time unit of its module.
`$time`	returns the current simulation time as an unsigned **64 bit integer**, scaled to time unit of its module.
`$timeformat( units, precision, suffix, field_width)`	used with the **`%t`** time format specifier in display and file output system tasks, to control how `$realtime`, `$stime` and `$time` are displayed
`$printtimescale (pathname)`	prints the timescale of the current module or the module at the pathname (optional)*

$realtime, $stime, $time

Used within display or file output system control tasks to record the current simulation time.

```
initial begin
  #10.4;
  $display ("time: %0d", $time);
  $display ("real time: ", $realtime);
```

This example generates the output:-

```
time: 10
realtime: 10.4
```

Note the use of the `%0d` format specifier to suppress the leading spaces in the `$time` display.

`$stime` generates the same value as `$time` for simulation time less than 32 bits. For time values over 32 bits, `$stime` only displays the lowest 32 bits. `$stime` is commonly used in un-formatted display output to reduce leading spaces.

Formatted display output using `%t` and `$timeformat` gives much greater control over time output.

$timeformat

`$timeformat` arguments are:

`units`	Integer between 0 (`s`) and -15 (`fs`), indicating the time unit
`precision`	Number of decimal digits to display
`suffix`	String to display after time value (include spaces for separation)
`field_width`	Minimum field width used for display including suffix

A procedural statement, `$timeformat` affects all uses of the `%t` specifier in display or file output system tasks for all following modules until over-ridden by another `$timeformat`.

```
initial begin
  data = 1'b1;
  $timeformat(-9, 2, " ns", 8);
  $monitor("%t %t %b",
             $time, $realtime, data);
  repeat (4)
    #8.3 data = ~data;
end
```

Generates the output:-

```
 0.00 ns   0.00 ns 1
 8.00 ns   8.30 ns 0
17.00 ns  16.60 ns 1
25.00 ns  24.90 ns 0
```

$printtimescale

`$printtimescale` is a procedural statement. In the following module it outputs the message-

```
Time scale of (one) is 1ns / 100ps
```

```
`timescale 1 ns / 100 ps
module one;
...
$printtimescale();
```

See Also

$System Control: Display Tasks, $System Control: File Output, `% Format Specifier`, `` `timescale ``

Convert between different data representations:-

`$realtobits`	Converts real numbers to a 64 bit IEEE754 representation
`$bitstoreal`	Converts a 64 bit IEEE754 representation to a real number
`$itor`	Converts integers to real numbers e.g. 12 becomes 12.0
`$rtoi`	Converts real numbers to integers by truncation, e.g. 12.2 becomes 12
`$signed`	Casts the value of an expression to be signed (2's complement)
`$unsigned`	Casts the value of an expression to be signed (binary)

`$realtobits` and `$bitstoreal` are primarily intended for passing real number data between modules across ports connections. Verilog does not allow a non-integral number, such as a real, to be used for a port type.

```
module send (output wire [63:0] rnet);
  real r;
  assign rnet = $realtobits(r);
endmodule

module receive (input wire [63:0] rnet);
  real r;
  assign r = $bitstoreal(rnet);
endmodule
```

`$signed` and `$unsigned` are used to convert between signed data (integers; vector types declared with the `signed` qualifier and literals using the `s` specifier) and unsigned data. Note that the data value does not change, only the interpretation.

```
reg signed [3:0] c;
reg signed [4:0] d;

initial begin
   c = 4'sb1001;      // c = -7
   d = $unsigned(c); // d = +9;
   ...
```

See Also

`signed`, Literals (s specifier).

% Format Specifier

Used to format strings, primarily for display and file IO system tasks.

Rules and Examples

Format specifiers are embedded in strings for display and file IO system tasks to control the output of individual variables.

```
$display ("var1= %h", var1,
          "var2= %b", var2);
```

Here `var1` is displayed in hex and `var2` in binary.

Specifiers can all be grouped at beginning of statement and are matched to variables in order:-

```
$display ("var1= %h var2= %b", var1,var2);
```

Here `var1` is displayed in hex and `var2` in binary.

It is an error if there are more specifiers than variables, but variables without specifiers are displayed using the defaults of the display task.

Specifiers are primarily used in the following system tasks:-

- Display tasks–`$display`, `$write`, `$monitor`, `$strobe` and variants (`$displayb` etc.).
- File IO tasks –`$fdisplay`, `$fwrite`, `$fmonitor`, `$fstrobe` and variants (`$fdisplayb` etc.)

Format Specifiers			
`%b`	binary values	`%c`	ASCII character
`%o`	octal	`%s`	string
`%d`	decimal	`%m`	hierarchical name
`%h`	hexadecimal	`%t`	time format
`%e`	real -exponential (3e5)	`%v`	strength and value
`%f`	real - decimal (3.12)	`%g`	real - general

Notes:

- Format specifiers are case insensitive, i.e. `%b` and `%B` have the same effect.
- `%m` does not require an argument.

- `%g` displays a real using either `%e` or `%f`, whichever is shorter.
- The `$timeformat` system task defines the exact time format for the `%t` specifier.

Hierarchical Specifier %m

`%m` is used like a specifier, but is not matched to a variable. It is replaced by the module hierarchical pathname when executed, e.g. if the following is called in a module `modone` instantiated with the name `m1` in the highest-level module `top`:-

```
$display ("var3 in %m is hex %h", var3);
```

then this produces:-

```
var3 in top.m1 is hex cc
```

Data Length and Leading Zero Suppression

Numerical display data is automatically sized to the length of the variable, e.g. an 8 bit variable is displayed as 8 characters using `%b`, 2 using `%h` and 3 using `%d` (maximum decimal value 256). Leading zeros are displayed for all formats except `%d`, where spaces are used instead of zeros.

Adding a `0` to the specifier suppresses leading zeros or spaces, e.g. **`%0d`** displays decimal values with no leading spaces, **`%0h`** without leading zeros.

Special Characters

Special characters such as tabs can also be embedded into display strings, e.g.:-

```
$display ("vars %h \t %h", var1, var2);
```

where `\t` is the tab character. See `\ Special Characters` on page 22 for more information.

See Also

System Tasks and Functions, Special Characters

An `always` procedural block is a collection of sequential statements. Always procedures execute *repeatedly* throughout simulation, controlled by a event expression.

Syntax

```
always @(<event_expression>)
  begin
    sequential statement(s)
  end
```

The event expression is optional. An `always` block containing a *single* sequential statement can omit the `begin..end` keywords.

Rules and Examples

An `always` block executes concurrently with other procedural blocks and concurrent statements.

For synthesizable combinational logic, the event expression is a list of signals. A change in value on any of these signals causes the `always` to be executed:-

```
always @(sel or a or b)
  if (sel)
    op = a;
  else
    op = b;
```

For synthesizable registered logic, the event expression includes `posedge` or `negedge` keywords to make the `always` sensitive to a specific edge of a signal, here the rising edge of `clk` or the falling edge of `rst`.

```
always @(posedge clk or negedge rst)
  if (rst == 0)
    q <= 0;
  else
    q <= d;
```

Alternatively the event expression may be omitted and `always` execution may be controlled via embedded event expressions or delays. This is not synthesizable.

```
always
  #50 clk = ~clk;
```

Verilog2001 Comma Separated Event List

In Verilog2001, the keyword `or` in the event expression can be replaced by a comma instead:-

```
always @(sel, a, b)
  if (sel)
    op = a;
  else
    op = b;
```

Verilog2001 Automatic Event List

Verilog2001 also adds an automatic event list for combinational logic only:-

```
always @(*)
  if (sel)
    op = a;
  else
    op = b;
```

The wildcard symbol `*` creates an automatic event list containing any signal read within the `always` block. Brackets for the automatic event list are optional.

Note that if the always contains a function, only variables passed as arguments to the function are added to the automatic event list.

Verilog2001 Local Declarations

In Verilog2001, **named** always blocks are allowed to declare local variables which are only visible in the block where they are declared:-

```
always @(avec)
begin : IBLK    // named block
  integer i;    // local variable
  for(i = 0; i<=7; i = i+1)
    if (avec[i])
      count = count + 1;
end
```

Synthesis Issues

Certain forms of `always` block are synthesizable. See RTL coding styles for more information.

See Also

`initial`, `posedge`, `negedge`, event timing control @

assign (continuous)

An assignment for a net data type used inside a module but outside of a an initial or always procedural block.

Syntax

```
assign <strength> <delay> name = expr;
```

where `strength` and `delay` are optional strength and delay specifications. *`name`* is a net data type variable.

Rules and Examples

```
wire selb, nsel, sum, carry;

assign nsel = ~sel;
assign selb = sel & b;
assign {carry, sum} = a + b;
```

The left-hand side of a continuous assign must be a vector or scalar net data type or a concatenation of these types.

Continuous assignments are always active. Every change in value of the left-hand side expression is immediately driven onto the right-hand side target.

```
assign cnet = read,
       dnet = write,
       enet = start;
```

A single `assign` can be used to drive many separate nets. Each assignment is separated by commas.

Implicit Declarations

```
wire msel;
assign nsel = ~sel; // incorrect name
```

If the target of an `assign` has not been explicitly declared, then a `wire` data type is implicitly created. As incorrectly typed identifiers will lead to new, implicitly declared wires, this behavior can be modified. The compiler directive `` `default_nettype `` controls the type of implicit declarations and is set to `wire` by default. Verilog2001 allows the option `none` for the directive, which will disable implicit declaration and cause compiler errors for objects which are not explicitly declared.

Implicit Continuous Assignment

```
wire selb = sel & b;
wire nsela = nsel & a;
```

A `wire` declaration and an `assign` statement can be merged as above.

Delay and Strength Options

Delay and strength options are useful for gate-level modeling but cannot be used in synthesizable code.

```
assign #1 cnet = write;
assign #(1:2:3) fnet = select;
assign (strong0, pull1) bnet = read;
```

Inertial delays can be defined for using # (see `Delay #` on page 64). Three delays can be specified for "to-zero", "to-Z" and "to-one" transitions in that order.

Strength is defined using two values, one for "to-zero" transitions and the other for "to-one". The order of definition does not matter, but separate `0` and `1` strengths must be used. see `Strength Specification` on page 128.

Synthesis Issues

Continuous assignments are fully supported by synthesis tools. They are used as an alternative to combinational `always` procedural blocks for simple logic.

Strength and delay options are not supported in synthesis.

See Also

Compiler directives (`’default_nettype), assign (procedural) deassign (procedural)`

assign (procedural)

Assign can be placed in `always` or `initial` procedural blocks to drive a register data type, although this form is not synthesizable and therefore not common.

Syntax

```
assign variable_assignment;
```

Rules and Examples

```
module dff (input wire d, clear, reset,
            input wire clock,
            output reg q);

always @(clear or reset)
  if (!clear)
    assign q = 0;
  else if (!reset)
    assign q = 1;
  else
    deassign q;

always @(posedge clock)
  q = d;
endmodule
```

A procedural assign statement overrides all procedural assignments to a variable, e.g. in the module `dff`, the `assign` statements of the `clear`/`reset` procedure override the procedural assignment from the `clock` procedure.

The deassign procedural statement removes any existing procedural assign to a variable. The variable maintains it's value until assigned a new one either through a procedural assignment or a procedural assign.

If a procedural assign is applied to a variable for which there is already a procedural assign in effect, then the variable is deassigned before the new procedural assign statement takes effect.

Synthesis Issues

Synthesis tools do not generally support procedural continuous assignments.

See Also

`deassign, force, release`

automatic

Verilog2001 construct which makes tasks and functions dynamic rather than static. This allows multiple concurrent task calls and recursive functions.

Syntax

```
function automatic <range_type> name;
...
endfunction
```

```
task automatic name;
...
endtask
```

Rules and Examples

In Verilog1995 subprograms are static, i.e. only one copy of a subprogram exists. Multiple concurrent task calls are unsafe as the values of internal subprogram variables (e.g. loop variables) and input/output arguments can be over-written by any active call.

```
task neg_clocks;
  input [31:0] count;
begin
  repeat(count) @(negedge clk);
end
endtask

initial begin
  neg_clocks(6);       ☒
  ...
end

always @(posedge trigger)
begin
  neg_clocks(10);      ☒
  ...
end
```

Task `neg_clocks` is static therefore multiple concurrent calls from the `initial` and `always` block will clash. The exact behaviour of the task calls will depend on the execution order of the procedures, which is indeterminate.

In Verilog2001, the keyword automatic can be placed after a task or function keyword to define the subprogram as dynamic rather than static.

In dynamic subprograms, a unique copy of the subprogram is created each time it is called. This allows safe multiple, concurrent task calls and recursive function calls, as each call has its own copy of variables and arguments.

```
task automatic neg_clocks;
 input [31:0] count;
begin
  repeat(count) @(negedge clk);
end
endtask

initial begin
  neg_clocks(6);  // OK
  ...
end

always @(posedge trigger)
begin
  neg_clocks(10); // OK
  ...
end
```

Defining task `neg_clocks` as `automatic` allows safe multiple concurrent calls.

Automatic also allows recursive functions, which call themselves repeatedly to obtain a value, e.g.

```
function automatic [63:0] factorial;
 input [7:0] n;
  if (n == 1) factorial = 1;
  else factorial = n * factorial(n-1);
endfunction
```

Restrictions of Automatic Subprograms

- Automatic subprogram declarations cannot be accessed by hierarchical references.
- Local variables in an automatic subprogram only exist for the duration of the subprogram call, so they cannot be accessed after the call completes, e.g. with delayed assignment or `$monitor` tasks.

See Also

`function`, `task`,

begin...end

Keywords which group multiple statements into a sequential block.

Syntax

```
<construct>
begin
 <statements>
end
```

Rules and Examples

`begin...end` is used to group multiple statements in:-

- `always` and `initial` procedural blocks:-

```
initial begin
  a = 1'b1;
  b = 1'b0;
end
```

- `case` and `if` conditional statement branches:-

```
if (s == 0) begin
  v = a;
  y = b;
end
```

- Loops (while, for, repeat and forever)

```
repeat(12) begin
  @(posedge clk);
    op <= ~op;
end
```

`begin...end` are *not* optional in tasks and functions.

Named Blocks

Any `begin...end` block can be named by specifying a name after the `begin` statement. Naming is useful for documentation and to disable a block (see `disable`).

```
always @(a or b or c)
begin : ADDING
  o = a + b;
  p = a + c;
end
```

See Also

`disable`

Blocking Assignment =

Procedural assignment to a variable which is *immediate*, i.e the variable takes its new value before the next statement in the procedure is executed.

Syntax

```
target = expression;
```

Rules and Examples

Blocking assignment is used in combinational procedures:-

```
always @ (a, b, sl)
  if (sl)
    d = a;
  else
    d = b;
```

Blocking assignment is more efficient than non-blocking assignment for combinational procedures, particularly where logic contains serial behavior or intermediate variables:-

```
always @(a, b)
begin
  m = a;
  n = b;
  p = m + n;
end
```

```
always @(a, b, m, n)
begin
  m <= a;
  n <= b;
  p <= m + n;
end
```

Using blocking assignment gives a result in a single pass through the procedure. Using non-blocking requires two passes through the procedure, one to update `m` and `n`, and another to update `p`. In addition, `m` and `n` must be added to the sensitivity list.

Race Conditions with Blocking Assignment

Blocking assignment can lead to race conditions, e.g. where a variable is written in one procedure and read in another and both procedures have the same event list, e.g. clocked procedures.

```
always @(posedge clock)
bvar = avar + 1'b1;              ☒

always @(posedge clock)
cvar = bvar;                     ☒
```

Both procedures execute on the positive edge of `clock`. The final value of `cvar` depends on which procedure is executed first. However Verilog procedures can execute in any order and therefore the value of `cvar` cannot be determined.

Non-blocking assignment must always be used in clocked procedures for inferring registers.

```
always @(posedge clock)
bvar <= avar + 1'b1;                ✓

always @(posedge clock)
cvar <= bvar;                       ✓
```

Blocked assignment can only be used in clocked procedures for temporary variables, i.e. variables which are written first and then read

```
always @(posedge clk)
begin
temp = a + b; // temporary
q <= temp + c;
end
```

Assignment Guidelines

- Use blocking assignments in combinational procedures
- In clocked procedures, only use blocking assignment for temporary variables (if at all).
- Use non-blocking assignment for all register inference in clocked procedures.

See Also

non-blocking assignment

A multi-way conditional statement which evaluates an expression and selects the first conditional branch which matches the expression value.

Syntax

```
case (case_expr)
  item_expr  : statement(s);
  item_expr  : statement(s);
  ...
  default : statement(s);
endcase
```

The case executes the statements associated with the *first* `item_expr`, checked in order of appearance, which matches the value of `case_expr`. An optional `default` branch is executed if no `item_expr` matches.

Case statements may be used in an `initial` block, an `always` block, a `function` or a `task`.

Rules and Examples

```
case (state)
  2’b00 : if (data_valid)
            state <= 2’b01;
  2’b01 : begin
            state <= 2’b10;
            rdata <= data;
          end
  2’b10 : state <= 2’b11;
  2’b11 : if (!data_valid)
            state <= 2’b00;
endcase
```

Case statement used for a simple state machine. Note multiple branch statements are enclosed in `begin/ end`.

As branch expressions are checked in order of appearance, overlapping values are allowed:-

```
case(ip)
  0       : y = a;
  0,1     : y = b;
  default : y = c;
endcase
```

If `ip=0`, the first branch, `y = a` is always executed.

Parallel Case

Case statements with overlapping <item_expr> conditions are implemented using priority logic.

A case with no overlapping conditions is known as a **parallel case** and can be implemented with more efficient non-prioritized logic. Most synthesis tools can recognize parallel cases, but others may need to be told with a **synthesis directive**, an embedded comment which is recognized by synthesis tools.

```
case(ip)
  // rtl_synthesis parallel_case
  0       : y = a;
  1       : y = b;
  default : y = c;
endcase
```

Full Case

Case branches do not need to cover all values for the case expression, but this implies a latch for synthesis.

A full case (for synthesis) has a branch for all binary values of the case expression (ignoring x and z). Most synthesis tools can recognize full cases, but others may need to be told with a synthesis directive.

```
case(ip)
  // rtl_synthesis full_case
  0       : y = a;
  1       : y = b;
  default : y = c;
endcase
```

Warning—if a full case directive is applied to a non-full case, missing case values can be implemented as “don’t care”. The full case issue is avoided if `default` case branches are used.

Alternate Case Form

```
case (1’b1)
  en_a : op = a;
  en_b : op = b;
  en_c : op = c;
endcase
```

A case item can be an expression, allowing the case to check several variables, prioritized according to order.

casez

`casez` is a variant of the `case` statement which interprets `z` and `?` as "don't cares" in **both** the case expression `case_expr` and the case item expression `item_expr`.

```
casez (pri)
  3'b1??: op = a;
  3'b01?: op = b;
  3'b001: op = c;
endcase
```

The *second* branch, `op = b`, is executed for `pri` values:

```
3'b010, 3'b011, 3'b01z
```

casex

`casex` interprets `z`, `?` and `x` as "don't cares" in **both** case expression `case_expr` and case item expression `item_expr`.

```
casex (pri)
  3'b1xx: op = a;
  3'b01x: op = b;
  3'b001: op = c;
endcase
```

The *second* branch, `op = b`, is executed for `pri` values:

```
3'b010, 3'b011, 3'b01z, 3'b01x
```

As `casex` treats an un-initialized `x` value in both expressions as being meaningful, `casex` can hide initialization problems during simulation. `casez` is a better construct to use than `casex`.

Both `casez` and `casex` can be full and/or parallel, and synthesis directives can be used to enforce this.

Synthesis Issues

Case statements are fully synthesizable. `casez,` in particular, can be convenient for modeling hardware.

Full/parallel case synthesis directives should only be applied to case statements which are full, parallel or both. Indiscriminate use of these directives can lead to synthesis errors.

See Also

`if`,

Verilog2001 construct used in configurations to bind all instantiations of an object to a specific module or library search order.

Syntax

```
cell module  use <lib.>name ;
```

```
cell module  liblist <lib1> <lib2> ;
```

where `module` is a hierarchical module, *name* a library module or config, and *lib*, *lib1*, *lib2* library names.

Rules and Examples

```
config cfg;
 design rtlLib.top;
 default liblist gateLib rtlLib;
 cell adder use gateLib.adder;
 cell cpu liblist rtlLib gateLib;
endconfig
```

Configuration `cfg` identifies module `top` from the library `rtlLib` as the top-level module. It specifies a default library search order of `gateLib -> rtlLib` for all module instances. The two `cell` clauses override the default library list for `adder` and `cpu` module instances.

The first `cell` binds all instances of `adder` in the design to the module `adder` in library `gateLib`.

All instances of `cpu` are bound in the second `cell` by searching `rtlLib` first, then `gateLib`.

See Also

`config, design, default, liblist, use, instance`

Verilog2001 construct which allows design management, specifically the binding of module instantiations to different source models without editing the design.

Syntax

```
config name;
  design <lib.>module;
  <config_rule(s)>;
endconfig
```

Where `name` is the configuration name and <*config_rule*> is a `default`, `instance` or `cell` configuration rule. A `config` must contain a `design` clause, but rules are optional.

Rules and Examples

By default, the model for a module instantiation is found by matching the module name. An instantiation of `adder` is always bound to a module named `adder`.

A Verilog2001 configuration allows a specific binding to be defined, allowing different mapping for a design with editing the hierarchy. Configurations also allow the use of multiple libraries (see `library` on page 91).

Configurations can define the following:-

- The top-most module in a design hierarchy.
- The order in which multiple libraries are searched to find bindings for module instantiations.
- The binding for every instantiation of a specific module, either by a library search order or specific module.
- The binding for a specific instantiation of a module, either by a library search order or specific module.

```
config cfg;
 design rtlLib.top;
 default liblist gateLib rtlLib;
 cell adder use rtlLib.adder;
 instance top.u2 liblist rtlLib gateLib;
endconfig
```

A configuration must contain one and only one `design` statement, and the design statement be the *first* statement in the configuration.

The following constructs can be used inside a `config`, (e.g. referencing `cfg` above). See individual entries for more information.

`design` defines the top-most module in the hierarchy, e.g. module `top` from the library `rtlLib`.

`default` defines a default library search order for module binding, e.g. `gateLib -> rtlLib`. `Default` can only use a `liblist` clause.

`cell` defines binding for all instantiations of a module. e.g. all instances of `adder` are bound to module `adder` from library `rtlLib`. A `cell` rule can use a `liblist` or `use` clause.

`instance` defines binding for one specific instantiation of a module, e.g. module top.u2 will be bound using the library search order `rtlLib -> gateLib`. An `instance` rule can use a `liblist` or `use` clause.

`liblist` defines a library search order.

`use` defines a specific module or configuration to be used for binding.

Nested Configurations

A use clause can reference another configuration as well as an explicit module name. Usually the name of the configuration is sufficient:-

```
instance top.u2 use rtlLib.mycfg;
```

However to explicitly distinguish a configuration or if the `config` has the same name as a module in the library, the optional `config` suffix can be used.

```
instance top.u2 use rtlLib.mycfg:config;
```

See Also

`cell`, `design`, `default`, `instance`, `liblist`, `use`

deassign

A procedural deassign removes the effect of an existing procedural assign statement, but keeps the current value of the variable.

Syntax

```
dessign variable_name;
```

Rules and Examples

```
module dff (input wire d, clear, reset,
            input wire clock,
            output reg q);

always @(clear or reset)
  if (!clear)
    assign q = 0;
  else if (!reset)
    assign q = 1;
  else
    deassign q;

always @(posedge clock)
  q = d;
endmodule
```

The deassign procedural statement removes any existing procedural assign to a variable. The variable maintains it's current value until assigned a new one either through a procedural assignment or a procedural assign.

e.g. in the module `dff`, the `assign` statements of the `clear/reset` procedure are deassigned when both clear and reset are not active, allowing the procedural assignment from the `clock` procedure to take effect.

If a procedural assign is applied to a variable for which there is already a procedural assign in effect, then the variable is automatically deassigned before the new procedural assign statement takes effect.

Synthesis Issues

Synthesis tools do not generally support procedural deassign statements.

See Also

`assign (procedural), force, release`

Defines an optional execution branch for a `case`, `casex` or `casez` statement which is executed when no explicit case branches match the case expression.

Syntax

```
case(x|z) (case_expr)
  item_expr  : statement(s);
  item_expr  : statement(s);
  ...
  default : statement(s);
endcase
```

The case executes the statements associated with the *first* `item_expr` which matches the value of `case_expr`. The `default` branch is executed if no `item_expr` matches.

Rules and Examples

```
case(ip)
  4'b000             : y = a;
  4'b001, 4'b0010 : y = b;
  default            : y = c;
endcase
```

The `default` branch `y = c` is executed for every value of `ip` other than 0, 1, or 2.

Full Case and Default

Case branches do not need to cover all values for the case expression, but this implies a latch for synthesis.

A full case (for synthesis) has a branch for all binary values of the case expression (ignoring x and z). Most synthesis tools can recognize full cases, but others may need to be told with a synthesis directive. The full case issue is avoided if `default` case branches are used.

```
case(ip)
  // rtl_synthesis full_case
  0       : y = a;
  1       : y = b;
  default : y = c;
endcase
```

See Also

`case, casez, casex`

default (configuration)

Verilog2001 construct used in configurations to specify a default library search order for binding modules.

Syntax

```
default liblist <lib1> <lib2> ;
```

where *lib1*, *lib2* are library logical names.

Rules and Examples

`default` can only be used with a `liblist` configuration rule.

```
config cfg;
 design rtlLib.top;
 default liblist gateLib rtlLib;
 cell cpu liblist rtlLib gateLib;
 instance top.u2 liblist rtlLib gateLib;
endconfig
```

Configuration `cfg` identifies module `top` from the library `rtlLib` as the top-level module. It specifies a default library search order of `gateLib -> rtlLib`, i.e. unless covered by an explicit configuration rule, an instance will be bound by searching for a matching module in library `gateLib` first, then in `rtlLib` if a `gateLib` match is not found.

There cannot be more than one default statement in a configuration. In all other respects, default follows the rules for the liblist clause (see `liblist` on page 90).

See Also

`cell, config, design, instance, liblist, use`

defparam

Construct which changes the value of a `parameter` in any module instance, anywhere in the design, by using the hierarchical name of the parameter.

Syntax

```
defparam name = value;
```

where *`name`* is a `parameter` identifier and *`value`* a constant expression of numbers and `parameter` references.

The parameter identifier must include the relative hierarchical name of the `parameter` from the `defparam` statement location.

Rules and Examples

```
module mux (a, b, sel, out);
  parameter WIDTH = 2;
  ...
endmodule
```

```
module test;
  ...
  mux mux4a (anib, bnib, sel, opnib);
  defparam mux4a.WIDTH = 4;
...
```

The parameter `WIDTH` in the instantiation `mux4a` of the module `mux` in the module `test`, is set by a `defparam` statement also placed in the module `test`.

By specifying the appropriate hierarchical name, all defparam statements for a design can be placed in a single location.

If there are multiple defparams for one parameter, the last defparam statement encountered in the source text determines the value of the parameter.

Warning—as a defparam statement can be placed anywhere in a design, and it's hierarchical path is dependent upon a specific design structure, defparams are a common source of design errors. It is recommended to avoid them where possible.

See Also

`parameter, localparam, module`

Delay

Time delays for testbench stimulus and gate-level propagation delays are defined using `#`.

Syntax

```
# expression
#(minimum:typical:maximum)
```

where *expression* is a single delay value or a triplet of minimum, typical, maximum delay values enclosed in brackets.

Rules and Examples

Delays can be used in procedural assignments, including intra-assignment delays, and in the instantiation of primitives for gate-level delays.

Assignment Delay

```
initial begin
  current_time = 4'b0000;
  #10; // wait 10 time units
  current_time = 4'b1001;
  #5; // wait 5 time units
end
```

`#` provides a simple procedural timing delay for testbench stimulus. It delays the execution of the procedural block by the defined number of time units, which are defined by the current `timescale` directive.

```
'timescale 1ns/100ps
module first;
   ...
   #10; // 10 ns delay
   ...
```

In module `first`, time units are scaled to 1 ns with a precision of 100 ps.

Delays are essential to prevent zero-delay feedback from locking the simulator, e.g. in clock stimulus generation. Remember to initialize the clock variable.

```
initial clk = 0;
always
  clk = ~clk;                ☒
always
  #50 clk = ~clk;            ☑
```

Delay

Intra-Assignment Delay

A simple delay pauses the execution of a procedural block and before making an assignment. An intra-assignment delay samples the source of an assignment; waits for the specified delay and then assigns the sampled value to the target.

```
reg [3:0] q2;
always @ (posedge clock)
  q2 <= #5 d;
```

With intra-assignment delay, the source `d` can change during the delay period `#5`, after it has been sampled, without affecting the target `q2`.

Delay Expressions

A delay expression evaluating to `z` or `x` is treated as zero-delay. An expression which evaluates to a negative value is treated as unsigned.

```
parameter CYCLE = 20;
always
  #(CYCLE/2) clk = ~clk;
```

Gate and Net Delays

Propagation delays for primitives, User-Defined Primitives (UDPs) and switch-level models *only* can be included in their instantiation. Separate delays can be defined for three transitions (where applicable):-

```
(-> 1, -> 0, -> z)
```

with minimum, typical, maximum delays for each transition. **Note** - this is not parameter over-riding, primitives cannot have parameters and module instantiations cannot have delays.

```
bufif0 #(5:7:9, 8:10:12, 15:18:21)
  b1 (io1, io2, dir);
```

`specify` blocks can also define gate-level delays.

Synthesis Issues

Delays are ignored in synthesis. They are generally used only in testbenches and gate level models.

See Also

Event Timing Control `@`, `specify`, `'timescale`

design

Verilog2001 construct used in a configuration to define the library and name of the top-most module (or modules) in the design hierarchy configured by the configuration.

Syntax

```
design <lib.>module;
```

where `lib` is an optional library logical name and `module` the name or names of modules within the library.

Rules and Examples

```
config cfg;
 design rtlLib.top;
 default liblist gateLib rtlLib;
 cell cpu liblist rtlLib gateLib;
 instance top.u2 liblist rtlLib gateLib;
endconfig
```

Configuration `cfg` identifies module `top` from the library `rtlLib` as the top-level module.

A configuration must contain one and only one `design` statement, and the design statement be the *first* statement in the configuration.

Multiple top-level modules can be specified in the one design statement. The design statement cannot reference another configuration.

If the library name is omitted, the library which contains the configuration is used as the search path.

See Also

`cell, config, default, design, instance, liblist, use`

Procedural statement which allows tasks or named blocks to be disabled (forced to exit).

Syntax

```
disable identifier;
```

where *identifier* is the name of or hierarchical path to a task or named block (see `begin...end`).

Rules and Examples

```
for (int i=0; i<=7; i=i+1)
begin : BLK1
  data = {data[6:0], data[7]};
  if (data[7])
    disable BLK1;
end
```

When a block or task is disable, execution resumes at the statement following the block or task call. Here `data` is rotated one bit to the left until a logic 1 is found in the most significant bit. `disable` is used to exit the loop block when `data[7]` is `1'b1`.

```
initial begin
  cpu_driver(8'h00);
...
end
always @(posedge interrupt)
  begin
  disable cpu_driver;
  service_interrupt;
end
```

Here if `interrupt` is detected the `cpu_driver` task is forced to exit and the interrupt can then be serviced. Disabling a task terminates all active calls to that task, but does not prevent subsequent calls. Disabling a task also disables all nested task calls within the disabled task. Functions cannot be disabled.

Synthesis Issues

Some synthesis tools support `disable`.

See Also

`begin...end` (named block), `task`

A lightweight data type which can be triggered from a procedural statement and used in an event expression.

Syntax

```
event name;
```

Rules and Examples

```
event disp_c;
always @(a or b)
  begin
  c = a + b;
  -> disp_c;
  end
always @(disp_c)
  $displayb(c);
```

An event is triggered as a procedural statement using the symbol `->` and the event name. The event can be used in an event expression in an `initial`, `always` or `task`.

A triggered event is instantaneous with no time duration. Therefore it will only activate event expressions which are waiting for the event at the time it is triggered. For example, if the `display always` above contains a delay, any `disp_c` events triggered while the block executes the delay will be missed.

```
always
  begin
  @(disp_c);
  $displayb(c);
  #1;  // error
  end
```

Also if the assignment to `c` is non-blocking, the current value of `c` is displayed, not the scheduled value.

Note an event has no value.

Synthesis Issues

Events are **not** synthesizable. They are used in testbenches to synchronize verification procedures.

See Also

`always,` event expressions

Event Timing Control @

Controls the execution of procedural blocks. Event control suspends the execution of an initial or always block until the change of value on a variable or combination of variables, or until the triggering of an event object.

Syntax

```
@(<expression_list>)
```

where *expression_list* is a list of variables, events or variable expressions (including the edge specifiers `posedge` and `negedge`) separated by the `or` keyword, or, in Verilog2001, commas.

Rules and Examples

The edge event control delays execution until an change of value occurs on a variable or expression of variables. A specific edge can be selected using `posedge` or `negedge`.

Synthesizable Code

In synthesizable code, the event control defines the sensitivity list for an `always` procedural block. Combinational logic is level sensitive:-

```
always @(a or b or sel)
  if (sel)
    op = a;
  else
    op = b;
```

In Verilog2001, the `*` character can be used to infer an automatic event list for combinational logic. This automatically includes every variable read within the always block in the event list. Brackets are optional:-

```
always @(*)
  ...
```

```
always @*
  ...
```

Sequential logic is edge sensitive:-

```
always @(posedge clk or negedge rst)
  if (!rst)
    q <= 0;
  else
    q <= d;
```

In Verilog2001, commas can be used instead of `or`

```
always @(a, b, sel)
  ...
```

```
always @(posedge clk, negedge rst)
  ...
```

Verification Code

In non-synthesizable code, event control can be embedded in `always` or `initial` blocks:-

```
initial
  begin
  repeat(12)
    begin
    @(posedge clk);
    waveform_b <= ~waveform_b;
    end
end
```

Event control can also be embedded into assignments (called intra-assignment event control):-

```
a = @(posedge clk) b;
```

This takes the current value of `b` and assigns it to `a` after the next posedge on `clk`. It is equivalent to:-

```
begin
  temp = b;
  @(posedge clk) a = temp;
end
```

An event object can also be used to trigger an event timing control:-

```
event disp_c;
always @(a or b)
  begin
  c = a + b;
  -> disp_c;
  end
always @(disp_c)
  $displayb(c);
```

See Also

`always`, delay, event, `posedge`, negedge, `wait`

A procedural loop statement where the number of iterations is controlled by a loop variable.

Syntax

```
for (init ; condition ; step)
  <for loop statements>
```

where *init* initializes the loop variable, *condition* defines when the loop terminates and *step* is the operation performed on each loop iteration.

Rules and Examples

```
integer i;
...
always @(a)
  begin
  tmp = 0;
  for (i = 0; i <= 3; i = i + 1)
    tmp = tmp ^ a[i];
  end
```

The loop variable `i` is initialized to 0. Then the condition `i <= 3` is checked. If true, the loop statement is executed. Then the step operation to increment i is performed and the condition rechecked. The loop executes as long as the condition is true. Note the loop variable must be separately declared. Multiple loop statements must be enclosed in `begin-end`.

Note if the condition is initially false, the loop is never executed. Also if the loop step and condition are incompatible, an infinite loop may be created.

```
for (i = 0; i < 0; i = i + 1)
 // ERROR-condition initially false !
for (i = 0; i >= 0; i = i + 1)
 // ERROR-condition never false for step
```

The for loop variable can be written as well as read, although this can be dangerous!

Synthesis Issues

For loops are synthesizable if the number of iterations are fixed, i.e. known at compilation.

See Also

`forever`, `repeat`, `while`

force

Force can be placed in `always` or `initial` procedural blocks to drive both register and net data types.

Syntax

```
force assignment;
```

Rules and Examples

Force is similar to the procedural assign, except it can drive net types as well as register types.

```
assign d = a & b & c;

initial begin
  #10;
  force d = (a | b | c);
  #10;
  release d;
  #10;
   ...
```

A procedural force statement overrides all procedural assignments to a register type or all drivers on a net variable, including module outputs.

In the `initial` block, the `force` statement makes `d` behave as the logical `or` of its inputs for the simulation time 10 to 20.

The release procedural statement removes any existing force driver to a variable. If a procedural assign statement is active on the variable, the assignment immediately takes effect when the force is released. Otherwise the variable maintains it’s current value until assigned by a procedural assignment.

Synthesis Issues

`force` is not synthesizable.

See Also

`assign (procedural), deassign, release`

A procedural loop statement which loops continuously.

Syntax

```
forever
  <forever loop statements>
```

where the loop statements must include event control or delays to avoid an infinite simulation loop.

Rules and Examples

```
initial begin // delayed
  clk = 0;
  forever
    #(period/2) clk = ~clk;
end
```

Forever statements loop forever, therefore they must contain an event expression to prevent an infinite simulation loop. Multiple loop statements must be enclosed in `begin-end`.

Forever loops can be terminated by using the `disable` statement (see `disable` on page 67).

```
forever begin : BLK1
  @(posedge clk);
  ...
  if (complete)
    disable BLK1;
end
```

Synthesis Issues

Forever loops are not synthesizable.

See Also

`for, repeat, while`

fork...join

A procedural statement which spawns multiple concurrent statements or blocks. The `fork join` remains active until the last spawned statement completes. `Fork join` is used in verification.

Syntax

```
fork
  <declarations>;
  blocka;
  blockb;
  ...
join
```

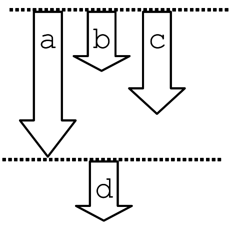
Rules and Examples

A fork join is used within an initial or always block to spawn multiple statements or blocks which execute concurrently. This means they are no longer sequential statements, executed in the order they appear, but execute according to delays or event control associated with each statement. Inside the fork join block, all delays are relative to the time when the block was entered.

```
initial
  fork
    #15 addr = 8'hf0;      // 1
    #10 data_bus = 8'h45;  // 2
    data_bus = 8'h00;      // 3
    #30 data_bus = 8'h0f;  // 4
  join
```

Here the order of execution, defined by the delays, is statements 3, 2, 1 and 4. The fork join does not complete until the last timed or event controlled statement completes. For example, if `taska`, `taskb` and `taskc` are time-consuming task calls, the `fork` does not complete, and `taskd` begin, until the last embedded task (`taska`) completes.

```
initial begin
  fork
    taska;
    taskb;
    taskc;
  join
  taskd;
end
```



Concurrent blocks can contain any procedural statement, including tasks, loops etc. Multiple statements in each concurrent branch can be enclosed in `begin end`.

```
fork
  begin : req1blk
    @(posedge req1);
    $display("req1");
    if (!busy)
      {busy, gnt1} = 2'b11;
  end
  begin : req2blk
    @(posedge req2);
    $display("req2");
    if (!busy)
      {busy, gnt2} = 2'b11;
  end
join
```

The above waits for *both* `req1` and `req2` and acknowledges the *first* request and sets `busy`. The first block which receives a request could cancel the other block by executing a disable statement on the other block name (see `disable` on page 67).

A fork block may never complete, e.g. one statement is a `forever` loop or a request is never received.

If multiple block statements are triggered at the same time point, then the order of execution executed is non-deterministic, which can lead to race conditions.

Synthesis Issues

fork join is not synthesizable.

See Also

`begin end, disable`

A function is a subprogram, returning a single value from one or more input arguments. A function is used as in an expression, usually on the right-hand side of an assignment.

Syntax

```
function <automatic> return_type name;
> (input_argument_list);
   <declarations>;
begin
   procedural_statement(s)
end
endfunction
```

where <*declarations*> are optional local variable declarations used only within the function. Note the function statements **must** be enclosed in `begin`-`end`.

Rules and Examples

A function is a group of procedural statements which is declared once, and used many times (with different variables if required) via a function call.

```
function integer zero_count;
  input [7:0] in_bus;
  integer i;
begin
  zero_count = 0;
  for (i = 0; i < 8; i = i + 1)
    if (!in_bus[i])
      zero_count = zero_count + 1;
end
endfunction
```

The function named `zero_count` has a formal input argument named `in_bus` and returns an `integer` value.

A local register variable with the same name as the function is implicitly created when the function is declared. The function value is created in this implicit variable and substituted for the function call when the function ends.

```
assign bcount = zero_count(b_bus);
```

The output of a function is not returned until it ends.

A function must contain at least one input. It cannot contain output or inout arguments. Although functions only return a single value, a return vector value can be assigned directly to a concatenation of signals, effectively giving them more than one output.

```
{o1,o2,o3} = zero_cnt_vec (a,b,c,d);
```

A functions cannot contain any event control or delays, i.e. they must execute in zero time.

Functions can contain calls to other functions, but a cannot contain task calls. However functions are by default static and so cannot call themselves (recursive).

Automatic Functions (Verilog2001)

By default, function calls are not duplicated like conventional subprograms. Only one copy of the function exists. Therefore recursive functions (where a function calls itself) are not allowed.

In Verilog2001 functions can be defined as `automatic`. A unique copy of an automatic function is created each time it is called. This allows recursive function calls as each call now has its own local copy of variables and arguments.

```
function automatic [63:0] factorial;
 input [7:0] n;
begin
  if (n == 1)
    factorial = 1;
  else
    factorial = n * factorial(n-1);
end
endfunction
```

As multiple copies of a specific function may exist, hierarchical references to declarations within automatic functions are not permitted.

Synthesis Issues

Functions are synthesizable as long as they do not contain non-synthesizable statements or types (e.g. real).

See Also

`automatic, task`

Verilog2001 construct used to create multiple or conditional instances of objects. Generates are placed inside a module scope, outside of a procedural block.

The following objects are allowed within a generate statement:- modules, primitives, variables, nets, task calls, function calls, continuous assignments, initial blocks and always blocks.

The following declarations are **not** permitted within a generate:- parameters, local parameters, in ports, out ports, inout ports and specify blocks.

There are 3 forms of generate - `for`, `if` and `case`.

For (Loop) Generate

Syntax

```
genvar identifier;
generate
  for ( init ; condition ; step )
    begin <: label>
     generate_statement(s)
    end
endgenerate
```

where the loop variable used in the `init`, `condition` and `step` clauses is declared as a `genvar`. A named loop begin-end block is compulsory for a for generate.

Rules and Examples

```
genvar i;
...
generate
  for (i=0; i<3; i=i+1)
  begin : gl  //required label
    reg r1 (din(i), clk, rst, dout(i));
  end
endgenerate
```

This creates 3 instantiations of the module `reg`:-

```
reg gl[0].r1 (din(0), clk, rst, dout(0));
reg gl[1].r1 (din(1), clk, rst, dout(1));
reg gl[2].r1 (din(2), clk, rst, dout(2));
```

Each instantiation has a unique name formed from the block label and the genvar value.

If (Conditional) Generate

Syntax

```
generate
  if (condition)
    generate_statement(s)
  else if (<condition>)
    generate_statement(s)
  else
    generate_statement(s)
endgenerate
```

where the if condition must be known at elaboration (i.e. when the design hierarchy is created). A label is not required for a conditional generate as only one branch will be active.

Rules and Examples

```
generate //
  if ( (SIZEA < 8) || (SIZEB < 8) )
    cla #(SIZEA, SIZEB) u1 (a, b, op);
  else
    wallace #(SIZEA, SIZEB) u1 (a, b, op);
endgenerate
```

Here a condition on the parameters `sizea` and `sizeb` determines whether an instance of module `cla` or module `wallace` is instantiated.

Case (Conditional) Generate

Syntax

```
generate
  case (case_expr)
    item_expr  : generate_statement(s)
    ...
    default : generate_statement(s)
  endcase
endgenerate
```

where the case expression must be known at elaboration (i.e. when the design hierarchy is created). A label is not required for a conditional generate as only one branch will be active.

Rules and Examples

```
generate
  case (SIZE)
    1: adder_1bit x1(co, sum, a, b, ci);
    2: adder_2bit x1(co, sum, a, b, ci);
    default:
       cla #(SIZE) x1(co, sum, a, b, ci);
  endcase
endgenerate
```

Here the parameter `SIZE` determines whether an instance of module `adder_1bit, adder_2bit, or cla` is instantiated.

Nested Generates

```
generate
  begin
  genvar j;
  for (j=0; j<18; j=j+1)
    begin: nestgen
    if( j=0)
      count1[j] = count0[j];
    else
      count1[j] = count0[j] & count1[j];
    case (j)
     1,2:     alu1 U1 (a[j], b[j], c[j]);
     default: alu2 U2 (a[j], b[j], c[j]);
    endcase
    end
  end
endgenerate
```

Generates can be nested and loop and conditional generates combined within the same statement.

Although conditional generates do not require a named block like the loop generate, implicit block names will be used by the simulator to reference each scope. These implicit names cannot be referenced in code. If hierarchical access to a generate block is required, the block must be explicitly named.

Synthesis Issues

Generates are synthesizable.

See Also

`genvar`

Verilog2001 declaration which defines the loop variable used by a `for` generate statement (see `generate` on page 78).

Syntax

```
genvar identifier;
```

Rules and Examples

```
genvar i;
...
generate
  for (i=0; i<3; i=i+1)
  begin : gendevice  //required label
    reg r1 (din(i), clk, rst, dout(i));
  end
endgenerate
```

Here the `genvar i` is used to create 3 instantiations of the module `reg`, and to index variables for the port connections of the modules.

A variable that is only used in a generate statement must be declared as a `genvar`. Primarily this applies to for (loop) generate variables.

A `genvar` must be declared within the module where it is used, either inside or outside of the generate scope. A `genvar` is an integer. It is an error if the `genvar` is set to a negative value, or if any bit is set to an X or Z.

Genvars are only used to elaborate the generate statements and do not exist during simulation.

A `genvar` can be referenced in any context where a `parameter` can be referenced.

See Also

`generate`

if ... else if ... else

A conditional procedural statement.

Syntax

```
if (condition)
  procedural_statement(s)
else if (condition)
  procedural_statement(s)
else
  procedural_statement(s)
```

where the `else if` and `else` branches are optional. The condition must be enclosed in brackets `()`. Multiple procedural statements in each branch must be enclosed in `begin-end`.

Rules and Examples

```
if (d == 4’b0000)
  y = a;
else if (d <= 4'b0101)
  y = b;
else
  begin
  y = c;
  valid = 1’b0;
  end
```

Each if condition is tested in sequence. The first condition that evaluates to logic 1 executes the statements in the branch of that condition. Branch conditions can overlap to create priority behaviour.

```
always @(posedge clk or posedge rst)
  if (rst)
    count <= 4’b0000;
  else if (load)
    count <= ip;
  else if (enable)
    if (count == maxcnt)
      count <= 4’b0000;
    else
      count <= count + 1;
```

An if statement can have any number of else if branches, but only one else branch. If statement branches can contain any procedural statement, including nested if statements.

Simple if statements can be replaced by the conditional operator (see `Operators` on page 102).

```
if (sel == 0)
  opr = a;
else if (sel <= 5)
  opr = b;
else
  opr = c;
```

```
opr = (sel == 0) ? a:((sel <= 5) ? b : c);
```

Synthesis Issues

The if statement is generally synthesizable.

If statements are used in clocked procedure templates to infer reset behavior:-

```
always @(posedge clk or posedge rst)
  // active high asynchronous reset
  if (rst)
    q <= 4'b0000;
  else
    q <= d;
```

If statements usually infer multiplexor logic in combination procedures. If statements without `else` branches can cause incomplete assignment for combinational processes in RTL code. This will infer transparent latches in synthesis. The following process infers a transparent latch:

```
always @(ctrl or a)
  if (ctrl)
    b = a;
```

Default assignments or unconditional else branches need to be added to avoid inferring latches.

```
always @(ctrl or a)
  if (ctrl)
    b = a;
  else
    b = 0;
```

See Also

`case, conditional operator`

An `initial` procedural block is a collection of sequential statements. Initial procedures execute *once* from the beginning of simulation.

Syntax

```
initial
  begin
    sequential_statement(s)
  end
```

An `initial` block containing a single sequential statement can omit the `begin..end` keywords.

Rules and Examples

An `initial` block executes concurrently with other procedural blocks and assign statements.

Initial blocks are primarily used for generating stimulus and initializing variables.

```
initial begin
  ip = 8'h00; //initialize
  #10 ip = 8'hf0; // pattern one
  #10 ip = 8'he1; // pattern two
  #10 ip = 8'hd2; // pattern three
  #10 ip = 8'hc3; // last pattern
end
```

Initial blocks do not have an event list like always, but can only contain embedded event expressions:-

```
initial
begin
  seq <= 3'b000;
  for (i = 0; i <= 4; i = i + 1)
    @(posedge clk)
      seq <= seq + 1;
end
```

Initial values

By default, variables have the value `x` at the start of simulation. Initial blocks are useful for defining alternate starting values for variables.

```
reg clk;

initial clk = 0;

always #50 clk = ~clk;
```

Initial blocks and initial values are either ignored or generate errors in synthesis and so should only be used in testbenches.

Verilog2001 Local Declarations

In Verilog2001, **named** initial blocks are allowed to declare local variables. These are only visible in the block where they are declared:-

```
initial
begin : IBLK   // named block
  integer i;   // local variable
  for(i = 0; i<=7; i = i+1)
    if (avec[i])
      count = count + 1;
end
```

Verilog2001 Initialization

Declaration and initialization of a variable can be combined in Verilog2001:-

```
reg clk = 0;

always #50 clk = ~clk;
```

Initial values are not allowed in synthesis code.

Synthesis Issues

Initial blocks are not synthesizable.

See Also

`always, delay #, event expression`

Used in the declaration of a module port or task argument, `inout` defines the direction of data sent via the port or argument as bi-directional.

Syntax

```
inout <[size]> name_list;
```

Where optional *`size`* defines bounds for vector types and *`name_list`* is a list of variables with the same direction and size.

Rules and Examples

```
module mymem (data, addr, read, write);
  inout [3:0] data;
  input [3:0] addr;
  input read, write;
  reg [3:0] mem [0:15];
assign data = (read ? mem[addr] : 4'bz);

always @(posedge write)
  mem[addr] = data;
endmodule
```

A module called `mymem` has an `inout` port named `data` of size 4. The `inout` port must be named in the port list and its size defined in an `inout` declaration. Inout ports are always net types and wire is the default type.

In Verilog2001 the port name, direction and type declarations can be merged.

```
module mymem (inout wire [3:0] data,
              input wire [3:0] addr
              input wire read, write);
  reg [3:0] mem [0:15];
...
```

Tasks can also have inout arguments. Functions are *not* allowed to have inout arguments

See Also

`function, inout, output, module, task.`

Used in the declaration of a module port or task or function argument, `input` defines that data sent via the port or argument is intended to be read inside the module, task or function.

Syntax

```
input <[size]> name_list;
```

Where optional *`size`* defines bounds for vector types and *`name_list`* is a list of variables with the same direction and size.

Rules and Examples

Module Ports

```
module mux (a, b, sel, op);
  input [7:0] a, b;
  input sel;
  output [7:0] op;

  assign op = sel ? a : b;
endmodule
```

A module called `mux` has input ports `a`, `b` of size 8 and a single bit input port `sel`. Each input port must be named in the port list and its size defined in an `input` declaration. Input ports are always net types and wire is the default type.

In Verilog2001 the port name, direction and type declarations can be merged.

```
module mux (input  wire [7:0] a,
            input  wire [7:0] b,
            input  wire sel,
            output wire [7:0] op);
  assign op = sel ? a : b;
endmodule
```

Note that it is not a compilation error if a module writes to an input port.

Task and Function Arguments

Both functions and tasks can have input arguments. Functions are *only* allowed to have input arguments.

```
function integer zero_count;
  input [7:0] in_bus;
  integer i;
begin
  zero_count = 0;
  for (i = 0; i < 8; i = i + 1)
    if (!in_bus[i])
      zero_count = zero_count + 1;
end
endfunction
```

Function `zero_count` has a single input formal argument named `in_bus`. When the function is called, the actual argument is passed into the function as a replacement for `in_bus`.

```
task zero_count;
  input [7:0] in_bus;
  output [3:0] count;
  integer i;
begin
  count = 0;
  for (i = 0; i < 8; i = i + 1)
    if (!in_bus[i])
      count = count + 1;
end
endtask
```

Task `zero_count` has a single input argument named `in_bus`. When the task is called, the actual argument is passed into the task as a replacement for `in_bus`.

Task arguments are static, which can lead to problems with task functionality. See `Static Task Arguments` on page 132 for more details.

In Verilog2001 the argument name, direction and type declarations can listed in brackets after the task name

```
task zero_count (input [7:0] in_bus,
                 output [3:0] count);
...
```

See Also

`function, inout, output, module, task`

Verilog2001 construct used in configurations to bind a specific object instantiation to a module or library search order.

Syntax

```
instance path use <lib.>name ;
```

```
instance path liblist <lib1> <lib2> ;
```

where `path` is a hierarchical path-name to an instance, *name* a library module or config and *lib*, *lib1*, *lib2* are optional library logical names.

Rules and Examples

```
config cfg;
 design rtlLib.top;
 default liblist gateLib rtlLib;
 instance top.u1 use rtlLib.adder;
 instance top.u2 liblist rtlLib gateLib;
endconfig
```

Configuration `cfg` identifies module `top` from the library `rtlLib` as the top-level module. It specifies a default library search order of `gateLib -> rtlLib` for all module instances. The two `instance` clauses over-ride the default library list for instantiations `top.u1` and `top.u2.` and `cpu` module instances.

The first `instance` binds all top.u1 in the design to the module `adder` in library `rtlLib`.

`top.u2` are bound in the second `instance` by searching `rtlLib` first, then `gateLib`, for a matching module name.

See Also

`cell, config, design, default, liblist, use`

Verilog2001 construct used in configurations to specify a library search order.

Syntax

```
liblist <lib1> <lib2> ;
```

where *lib1*, *lib2* are library logical names.

Rules and Examples

liblist can be used with a `default`, `cell` or `instance` configuration rule:-

```
config cfg;
 design rtlLib.top;
 default liblist gateLib rtlLib;
 cell cpu liblist rtlLib gateLib;
 instance top.u2 liblist rtlLib gateLib;
endconfig
```

Configuration `cfg` identifies module `top` from the library `rtlLib` as the top-level module. It specifies a default library search order of `gateLib -> rtlLib`, i.e. all instances will be bound by searching for a matching module in library `gateLib` first, then in `rtlLib` if a `gateLib` match is not found.

The `cell` clause defines a library search order of `rtlLib -> gateLib` for all instances of `cpu`.

The `instance` clause defines a library search order of `rtlLib -> gateLib` for the specific instance `top.u2`.

A `liblist` can contain a single library name, in which case only the library specified is searched for bindings.

Liblist Inheritance

Liblists are inherited hierarchically, i.e. unless an explicit binding is defined, all module instances below `top.u2` in the design above will inherit the library search order `rtlLib -> gateLib`.

See Also

`cell, config, design, default, instance, use`

Verilog2001 feature defining how design files are compiled into libraries for better management of the design using configurations.

Syntax

```
library name filepath;
```

where `name` is a library logical name and *filepath* a specifier for the source files to be compiled into the library.

Rules and Examples

Library constructs define the library into which source code is compiled. If the pathname of a source file matches the *filepath*, then the file is compiled into the specified library. Library constructs are defined in special compilation files (library map files). The name and compilation of a map file is tool specific.

```
library rtlLib "./*.v";
library gateLib "./*.vg";
```

Here all files in the current directory with the suffix `.v` are compiled into the `rtlLib` library, whereas all files ending in `.vg` are compiled into library `gateLib`.

The *filepath* can use the following wildcard options:-

`*`	Multiple character wildcard
`?`	Single character wildcard
`...`	Hierarchical wildcard (any directories)
`.`	Current directory
`..`	Parent directory

Library map files can also reference other map files using an `include` command

```
include "../mapfiles/lib.map";
```

Libraries

A library is a directory where compiled design units are stored and is referenced from within Verilog code using a library logical name. The creation and naming of libraries is performed by the simulator.

See Also

`config`

Literals

Definition of values in Verilog for use in assignment, expressions etc.

Syntax

```
<size>'<base>value
```

where `size` is the number of bits. `base` is one of `b` (binary), `o` (octal), `d` (decimal) or `h` (hexadecimal). `value` is any legal base number, plus `x`, `z` or underscore. `base` and `value` are *not* case-sensitive. `size` and `base` are optional.

Rules and Examples

If `size` is omitted, it defaults to 32-bits. If `base` is omitted, it defaults to decimal. If the most significant bit of `value` is `0`, `x` or `z`, then the bit will be automatically extended to the full width of `size`.

```
8'b1100_0001 // 8-bit binary
64'hff01     // 64-bit hexadecimal
12           // 32-bit decimal (default)
'h83a        // 32-bit hexadecimal
32'bz        // 32-bit z extension
```

It is good practice to size a literal to match the value, and the variable to which it is assigned. Remember over-long values are truncated, and over-short values padded with `0` from the most-significant bit.

```
reg [2:0] zbus;
...
// size/value too long, truncated:
zbus = 4'b1001;  // 001
// value too long, truncated:
zbus = 3'ha;     // 010
// value too short, padded:
zbus = 3'b10;    // 010
// size/value too short, padded:
zbus = 2'b1;    // 001
```

In Verilog2001, literal values can be qualified as signed (2's complement) using modifier `s` before `base`.

```
integer int = 4'sb1001  // -7
```

See Also

Net Data-types, Register Data-types

localparam

Verilog2001 object identical to a `parameter` except that a `localparam` cannot be modified with `defparam` or in an instantiation parameter assignment.

Syntax

```
localparam <type> name = expr;
```

where *type* is an optional type specifier, *name* is an identifier and *expr* a constant expression.

Rules and Examples

```
module us_mult (a, b, product);
  parameter size_a = 5;
  parameter size_b = 5;
  localparam size_op = size_a + size_b;
  input  [size_a-1:0] a;
  input  [size_b-1:0] b;
  output [size_op-1:0] product;
...
```

`size_a` and `size_b` are defined as parameters and can be modified for every instantiation of `us_mult`. `size_op` depends on `size_a` and `size_b`, and so must not be modified outside the module. Therefore it is declared as a `localparam` to prevent this.

Attempting to modify a localparam via a defparam or a parameter assignment in a module instantiation, is an elaboration error.

Localparam Types

With type or range defined, a `localparam` defaults to the type and range of the value assigned to it. If type, range or keyword signed are defined in the declaration, then the value assigned will take on the specified type, range or sign.

```
localparam w = 16, w2 = w/2;  // integers
localparam r = 5.7; // real
localparam s = 3'h2; // implied  [2:0]
localparam signed [3:0] mask = 0;
localparam [7:0] nash = 1'b1; // 8 bits
```

See Also

`parameter, module, signed`

module

The basic Verilog building block. A Verilog design is composed of a hierarchy of modules defining design functionality and communicating by port lists.

Syntax

```
module name (port_list);
  <declarations>
  procedural block(s)
  assign statement(s)
  instantiations
endmodule
```

where `port_list` is a list of in, out and bi-directional connections to the module. The contents of a module can appear in any order.

Rules and Examples

Module functionality is described by procedural blocks; assign statements and instantiations. Module connectivity is defined by the module port list.

```
module mux (a, b, sel, op);
  input [7:0] a, b;
  input sel;
  output [7:0] op;
  reg [7:0] op;

  always @(a or b or sel)
    op = sel ? a : b;

endmodule
```

A module called `mux` has input ports `a`, `b` and `sel`, and output `op`. Each port must be named in the port list; its size and direction defined in an `input`, `output` or `inout` declaration and finally the data-type defined. By default, all ports are of type wire, so register data-types must be explicitly defined. See also `Verilog2001 Port List` on page 96.

Module Instantiation

A design hierarchy is created by declaring module instances and connecting them by their ports.

```
mux U1(.a(a), .b(b), .sel(s1), .op(op1));
```

Named and Positional Port Connection

A connection list maps the module ports to the local variables and ports where the instantiation is declared.

Port connections can be named, where using the syntax

```
.<port name>(<variable_name>)
```

Alternatively, a positional port connection simply lists the local variables and ports and these are mapped to the ports of the module in order of appearance.

```
module muxes (a, b, s1, s2, op1, op2);
  input [7:0] a, b;
  input s1, s2;
  output [7:0] op1, op2;

  mux U1 (.a(a), .b(b),
          .sel(s1), .op(op1));
  mux U2 (a, b, s2, op2);

endmodule
```

Here instantiation `U1` uses named connection and `U2` uses positional connection. Positional connection is more error-prone as the connection list can be easily declared in the wrong order.

Parameterized Modules

Parameters can be used to define the width of ports in a module, allowing scalable modules to be created.

```
module muxn (a, b, sel, op);
  parameter WIDTH = 2;
  input [WIDTH-1:0] a, b;
  input sel;
  output [WIDTH-1:0] op;

  assign op = sel ? a : b;

endmodule
```

Parameter `WIDTH` defines the size of the input ports `a` and `b`, and output `out`. The parameter can be assigned a value in the module instantiation using `#()` or using the `defparam` statement. See `parameter` on page 111.

```
muxn #(8) U1 (.a(a), .b(b), .sel(select),
              .op(op1));
```

Verilog2001 Port List

In Verilog2001 the port name, direction and type declarations can be merged (sometimes called an ANSI C port declaration)

```
module mux (input  wire [7:0] a,
            input  wire [7:0] b,
            input  wire sel,
            output reg [7:0] op);

  always @(a or b or sel)
    op = sel ? a : b;
endmodule
```

Verilog2001 also allows parameters to be declared before the module name using the `#( )` construct. This is essential when using ANSI C style parameterized port declarations as the parameter must be declared before the port:-

```
module mux_size
  #(parameter WIDTH = 8)
  (input  wire [WIDTH-1:0] a,
   input  wire [WIDTH-1:0] b,
   input  wire sel,
   output reg [WIDTH-1:0] op);
  ...
endmodule
```

Verilog 2001 configuration

In Verilog95, the module name must be unique within the design hierarchy and an instantiated module is always linked to a module declaration of the same name.

In Verilog2001, each module instantiation can be linked to any declared module through the use of a configuration (see `config` on page 58).

See Also

`always, assign, config, initial, input, inout, output, parameter, primitive, defparam`

Used in an event expression to trigger the execution of a procedural block on a change of value of a variable or expression *towards* logic 0.

See also `Event Timing Control @` on page 69.

Syntax

```
@(negedge expression)
```

An event expression can contain multiple `negedge` expressions, combined with other event expressions, and separated by either a comma (Verilog2001) or the keyword `or`.

Rules and Examples

```
always @(negedge clk or negedge rst)
  if (rst)
    q <= 0;
  else
    q <= d;
```

`negedge` is used in synthesizable RTL code to imply clocked logic. The above synthesizes to a falling-edge clocked register with an active-low reset for `q`.

```
initial begin
  @(negedge enb, valid);
  ...
end
```

`negedge` is also used in verification code to control execution of initial blocks or tasks. Here the event control is triggered on a falling-edge on `enb` or any change in value on `valid`.

posedge and negedge are evaluated as follows:-

from to	0	1	x	z
0	no edge	posedge	posedge	posedge
1	negedge	no edge	negedge	negedge
x	negedge	posedge	no edge	no edge
z	negedge	posedge	no edge	no edge

See Also

posedge, event expression

Net Data Types

Net data types are used for variables which are continuously driven from a continuous `assign` statement or from an output port.

Syntax

```
type <[size]> name;
```

where `size` is optional and `type` is one of:.

`wire` `tri`	For standard interconnection (`wire` is the default)
`wand` `wor`	For multiple drivers that are Wire-ORed

`wand,`
`triand`
	For multiple drivers that are Wire-ANDed

`trireg`	For nets with capacitive storage

`tri0,`
`tri1`
	For nets that pull up or down when not driven

`supply0` `supply1`	For power or ground rails in netlists only

Rules and Examples

Net data-types are driven by continuous `assign` statements or by the output ports of module or primitive instances. They cannot be driven from procedural blocks.

```
wire nota;          // Scalar wire
wire [7:0] w1, w2; // Two 8-bit wires
wand c;            // Scalar wired-AND net
tri [15:0] busa;   // 16-bit tri-state bus

always @(a)
  nota = ~a;        // procedural           ☒

assign nota = ~a;       // continuous       ☑
```

Implicit Declarations

```
wire msel;
assign nsel = ~sel; // incorrect name
```

If a variable been explicitly declared, then a `wire` data type is implicitly created. As incorrectly typed identifiers will lead to new, implicitly declared wires, this behavior can be modified. The compiler directive `‘default_nettype` controls the type of implicit declarations and is set to `wire` by default. Verilog2001 allows the option `none` for the directive, which will disable implicit declaration and cause compiler errors for objects which are not explicitly declared.

Implicit Continuous Assignment

```
wire selb = sel & b;
wire nsela = nsel & a;
```

A `wire` declaration and an `assign` statement can be merged as above.

Synthesis Issues

`trireg`, `tri1`, `tri0`, `supply1`, `supply0` are not synthesizable.

See Also

`Register Data-types, ‘default_nettype`

Non-Blocking Assignment <=

Procedural assignment to a variable which is *scheduled*, i.e the variable takes its new value *after* the procedure suspends.

Syntax

```
target <= expression;
```

Rules and Examples

Non-blocking assignment is used in clocked procedures:-

```
always @(posedge clk or posedge rst)
  if (rst) // active high reset
    count <= 4'b0;
  else if (count < 9)
    count <= 4'b0;
  else
   count <= count + 4'b1;
```

Avoiding Race Conditions

Using blocking instead of non-blocking assignment in clocked procedures can lead to race conditions where a variable is written in one procedure and read in another and both procedures have the same clock.

```
always @(posedge clock)
bvar = avar + 1'b1;           ☒

always @(posedge clock)
cvar = bvar;                  ☒
```

Both procedures execute on the positive edge of `clock`. The final value of `cvar` depends on which procedure is executed first. However Verilog procedures can execute in any order and therefore the value of `cvar` cannot be determined.

With non-blocking, assignment to `bvar` and `cvar` are scheduled, i.e. the current value of the assignment expression is sampled and assigned to the variable after the procedure suspends:-

```
always @(posedge clock)
bvar <= avar + 1'b1;          ☑

always @(posedge clock)
cvar <= bvar;                 ☑
```

Non-Blocking Assignment <=

Non-blocking assignment must always be used in clocked procedures to infer registers and avoid race conditions.

Blocked assignment can only be used in clocked procedures for temporary variables, i.e. variables which are written first and then read. Temporary variables should not infer registers in synthesis.

```
always @(posedge clk)
begin
temp = a + b; // temporary
q <= temp + c;
end
```

Assignment Guidelines

- Use blocking assignments in combinational procedures
- In clocked procedures, only use blocking assignment for temporary variables (if at all).
- Use non-blocking assignment for all register inference in clocked procedures.

See Also

blocking assignment

Operators

Operators in Verilog fall into the following groups:-

• arithmetic	`+ - * / %`
• bit-wise	`~ & | ^ ~^`
• reduction	`& | ^ ~& ~| ~^`
• logical	`! && ||`
• shift	`<< >> <<< >>>`
• relational/equality	`< > <= >= == != === !==`
• conditional	`?:`
• concatenation/replication	`{} {{}}`
• indexed part-select	`+: -:`

Note: normal Verilog truncation and padding rules apply if the result of an expression is smaller or greater than the target variable of the assignment.

Arithmetic Operators

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`%`	modulus

Arithmetic on type `integer` is signed (2's complement). Arithmetic on `reg` variables and net types is unsigned (binary). The Verilog2001 keyword `signed` can be applied to a `reg` variable to specify that the variable contains signed data (see `signed`).

```
integer ans, int;
parameter FIVE = 5;
reg [3:0] rega, regb, num;
initial begin
  rega = 3; // 0011
  regb = 4'b1010;
  int = -3;

  ans = FIVE * int;    // ans = -15
  ans = (int + 5)/2;   // ans = 1
  num = rega + regb;   // num = 1101
  num = int - 1;       // num = 1100
end
```

Bit-Wise Operators

Compare each bit in one operand with its corresponding bit in the other operand to calculate each bit for the result. Normally used for vectors but works equally on scalar (single bit) values.

`~`	not
`&`	and
`|`	or
`^`	xor
`~^ ^~`	xnor

The operators can be used with operands of different sizes - the smaller operand is zero-extended to the size of the larger operand. Unknown bits in an operand do not necessarily lead to unknown bits in the result (e.g. 1'bx `& 1'b1 = 1'b1`)

```
reg [3:0] rega, regb, num;
initial begin
  rega = 4'b1001;
  regb = 4'b1010;

  num = ~rega;        // num = 0110
  num = rega & 0;     // num = 0000
  num = rega & regb;  // num = 1000
  num = rega | regb;  // num = 1011
end
```

Reduction Operators

Perform a bit-wise operation on all the bits of a single operand, returning a single bit `1'b1`, `1'b0` or `1'bx`.

`&`	and
`|`	or
`^`	xor
`~&`	nand
`~|`	nor
`~^ ^~`	xnor

`x` or `z` values in the operand can be hidden by OR-ing with 1 or AND-ing with 0. Any `x` or `z` value in a XOR or XNOR operand will always produce `x`.

Reduction Operators (continued)

```
reg val;
reg [3:0] rega;
initial begin
  rega = 4’b0100;

  val = &rega ;   // val = 0
  val = |rega ;   // val = 1
  val = ^rega ;   // val = 1
end
```

Logical Operators

Reduce both operands to a single bit, and then performs a single bit operation:.

!	not
&&	and
||	or

- Vectors containing *any* `1` reduce to `1’b1`
- Vectors containing *all* `0` reduce to `1’b0`
- Vectors containing any `x` or `z` with *only* `0` reduce to `1’bx`

```
reg ans;
reg [3:0] rega,regb,regc;
initial begin
  rega = 4’b0011;  // reduces to 1
  regb = 4’b10xz;  // reduces to 1
  regc = 4’b0z0x;  // reduces to x

  ans = !rega;         // ans = 0
  ans = rega && regb; // ans = 1
  ans = rega || 0;    // ans = 1
end
```

Shift Operators

Perform left or right bit shifts on the operand.

<<	shift left logical
>>	shift right logical
<<<	shift left arithmetic (Verilog2001)
>>>	shift right arithmetic (Verilog2001)

Shift Operators (continued)

Logical and arithmetic left shifts use 0 for extra bits. Arithmetic right shifts use the value of the most significant bit, i.e. they maintain sign data for 2's complement values.

```
reg [7:0] num, rega;
initial begin
  rega = 8'b1010_0101;
  num = rega << 1;  // = 010_01010
  num = rega <<< 1; // = 010_01010
  num = rega >> 3;  // = 0001010_0
  num = rega >>> 3; // = 1111010_0
end
```

Relational Operators

Comparison operators:

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

The result is:-

- `1'b1` if the condition is true
- `1'b0` if the condition is false
- `1'bx` if the condition cannot be resolved

If differently sized vectors are compared, the shorter is padded out with leading 0's to match the longer length operand. **Note** the less-than-or-equal-to operator is the same symbol as non-blocking assignment.

```
reg [3:0] rega, regb;
reg val, opq;
initial begin
  rega = 4'b0011;
  regb = 4'b1010;

  val = regb < rega ; // val = 0
  val = regb >= rega; // val = 1
end

always @(posedge clk)
  opq <= rega <= regb; // opq = 1
```

Equality Operators

There are two forms of equality, `logical` and `case`. The difference is in the handling of the `x` and `z` values:

==	logical equality
!=	logical inequality
===	case equality
!==	case inequality

With the logical equality, only 1 or 0 values are compared and an `x` or `z` in either operand generates an unknown result. With case (or identity) equality, all values are compared and an `x` in one operand will match an `x` in the same bit of the other operand.

```
reg [3:0] rega, regb;
reg val;
initial begin
  rega = 2'b1x;
  regb = 2'b1x;
  regc = 2'b0x;

  val = rega == regb ; // val = x
  val = rega === regb; // val = 1
  val = rega === regc; // val = 0
end
```

Conditional Operator ?

Short-hand form of a simple if statement:-

```
<condition>?<true_expr>:<false_expr>;
```

It is a compile error if each conditional operator does not have all three arguments. The `false_expr` serves as a default.

```
always @ (a or b or sel)
  out1 = sel ? a : b;
```

If `sel = 1`, then `out1 = a`, else `out1 = b`. Conditional operators can be nested.

```
out1 = sela ? a : (selb ? b: c);
```

If `sela = 1` then `out1 = a`, else if `selb = 1`, `out1 = b`, else `out1 = c`.

Concatenation Operator { }

Concatenates any number of *sized* expressions into a single vector. Concatenation can be used on either side of an assignment:-

```
reg [7:0] rega, regb, new;
reg [3:0] nib1, nib2;
initial begin
  rega = 8’b00000011;
  regb = 8’b00000100;
  new = {2’b11, regb[7:4], rega[1:0]};
  // new = 8’b11_0000_11
  new = {regb[4:0], regb[7:5]};
  // rotate regb right 3 places
  // new = 8’b00100_000
  {nib1, nib2} = rega; // 0000_0011
end
```

Replication Operator {{ }}

Repeats a sized expression a set number of times. Replication can be used and nested with concatenation.

```
rega = 1’b1;
regb = 4’b1001;
bus = {8{rega}};
// bus = 11111111
bus = { {4{rega}}, {2{regb[1:0]}} };
// bus = 1111_01_01
```

The first example repeats `rega` 8 times. The second example concatenates 4 repetitions of `rega` with 2 repetitions of the 2 lowest bits of `regb`.

Indexed Part-Select +: -:

Allows variable indexing of arrays:-

```
varr[base_expr +: width_expr] // positive

varr[base_expr -: width_expr] // negative
```

An indexed part select contains a base expression and a width expression. The operator defines if the width is added or subtracted from the base. The base expression can vary during simulation, e.g. be dependent on a variable, but the width expression must be constant. The width expression cannot be dependent on a run-time parameter assignment.

Indexed Part-Select +: -: (continued)

```
reg [63:0] varr;
reg [7:0] byte;
initial begin
  byte = varr[0 +: 8];  // varr[7:0]
  byte = varr[63 -: 8]; // varr[63:56];
  for (i=0; i<4; i=i+1)
      byte = varr[i*8 +: 8]);
      // i = 0 selects varr[7:0]);
      // i = 1 selects varr[15:8]);
      // i = 2 selects varr[23:16]);
      // i = 3 selects varr[31:24]);
  ...
```

The for loop example shows the advantage of the indexed part-select in reading `varr` by 8-bit slices.

Operator Precedence

From highest to lowest:

concatenation, replication	`{ } { { }}`
inversion (logical, bit-wise)	`! ~`
arithmetic	`* / % + -`
shift	`<< <<< >> >>>`
relational	`<= < > >=`
equality	`== ===`
bit-wise, reduction	`& ~& | ~| ^ ^~ ~^`
logical	`&& ||`
conditional	`?:`

Synthesis Issues

`%`, ===, `!==` operators are not synthesizable. / is only synthesieable if the second operand is a power of 2.

See Also

for loop,

Used in the declaration of a module port or task argument, `output` defines that data sent via the port or argument is intended to be written inside the module or task.

Syntax

```
output <[size]> name_list;
```

Where optional `size` defines bounds for vector types and *`name_list`* is a list of variables with the same direction and size.

Rules and Examples

Module Ports

```
module mux (a, b, sel, op);
  input [7:0] a, b;
  input sel;
  output [7:0] op;
  reg [7:0] op;

  always @(sel or a or b)
    op = sel ? a : b;
endmodule
```

A module called `mux` has a single output port `op` of size 8. The output port must be named in the port list and its size defined in an `output` declaration. If an output port is driven by a nested module output port or by an assign statement, then the port must be a net type and wire is the default type. If the port is driven from a procedural block, as in the example, it must be explicitly defined as a register type.

In Verilog2001 the port name, direction and type declarations can be merged.

```
module mux (input  wire [7:0] a,
            input  wire [7:0] b,
            input  wire sel,
            output reg [7:0] op);

  always @(sel or a or b)
    op = sel ? a : b;
endmodule
```

Note that output ports can be read from inside the module.

Task Arguments

Tasks can have output arguments. Functions are *not* allowed to have output arguments.

```
task zero_count;
  input [7:0] in_bus;
  output [3:0] count;
  integer i;
begin
  count = 0;
  for (i = 0; i < 8; i = i + 1)
    if (!in_bus[i])
      count = count + 1;
end
endtask
```

Task `zero_count` has a single output argument named `count`. When the task is called, the formal argument `count` is assigned to the actual argument in the task call at the end of the task execution.

In Verilog2001 the argument name, direction and type declarations can listed in brackets after the task name

```
task zero_count
 (input [7:0] in_bus,
  output [3:0] count);
  integer i;
begin
  ...
```

Task arguments are static, which can lead to problems with task functionality. See `Static Task Arguments` on page 132 for more details.

See Also

`function, input, inout, module, task`

parameter

Constants declared in a module, which can be set to a different value for every instantiation of the module. For constants which cannot be changed, see `localparam`.

Syntax

```
parameter <type> name = expr;
```

where *type* is an optional type specifier, *name* is an identifier and *expr* a constant expression.

Rules and Examples

```
module mux (a, b, sel, out);
  parameter WIDTH = 2;
  input [WIDTH-1:0] a, b;
  input sel;
  output [WIDTH-1:0] out;
...
```

`WIDTH` is defined as a parameter and can be modified for every instantiation of `mux`.

In Verilog2001, parameters can be defined as part of the ANSI-C style input/output declarations:-

```
module mux
  #(parameter WIDTH = 2)
  (input wire[WIDTH-1:0] a, b,
   input wire sl,
   output reg [WIDTH-1:0] out);
...
```

Parameter Modification

There are 3 ways of changing a parameter value:-

- Ordered assignment in a module instantiation
- Verilog2001 named assignment in a module instantiation
- A `defparam` statement (avoid if possible)

```
mux #(5) u1
  (.a(a), .b(b), .sl(sl), .out(out));
mux #(.WIDTH(5)) u2
  (.a(a), .b(b), .sl(sl), .out(out));
mux u3 (.a(a), .b(b), .sl(sl), .out(out));
defparam u3.WIDTH = 4;
```

Parameter Modification (continued)

In ordered assignment, values are mapped to parameters in the order of parameter declaration, e.g. for a module with 3 parameters, A, B and C:-

```
module mod (in1, in2, out);
  parameter A = 0, B = 0, C = 0;
...
```

The following sets A to 1, B to 2 and C to 3:-

```
mod #(1,2,3) u1 (in1, in2, out);
```

Not all parameters need be set, e.g. the following sets A to 5 and B to 6:-

```
mod #(5,6) u1 (in1, in2, out);
```

However ordered assignment cannot omit values, e.g. to set C without setting values for A and B. Verilog2001 named assignment must be used instead.

```
mod #(.B(7)) u1 (in1, in2, out);
```

In general, Verilog2001 named assignment is recommended for clarity.

defparam statements can be placed anywhere in a design hierarchy, and set a parameter via a relative hierarchical name of the parameter from the defparam statement location. They are a deprecated construct in SystemVerilog.

Parameter Types

Without type or range defined, a parameter defaults to the type and range of the final value assigned to it. If type, range or keyword signed are defined in the declaration, then the final value will take on the specified type, range or sign. This prevents bad assignment values changing the parameter type.

```
parameter w = 16, w2 = w/2;  // integers
parameter r = 5.7; // real
parameter s = 3'h2; // implied  [2:0]
parameter signed [3:0] mask = 0;
parameter [7:0] nash = 1'b1; // 8 bits
```

See Also

localparam, module, signed, specparam

Used in an event expression to trigger the execution of a procedural block on a change of value of a variable or expression *towards* logic 1.

See also `Event Timing Control @` on page 69.

Syntax

```
@(posedge expression)
```

An event expression can contain multiple posedge expressions, combined with other event expressions, and separated by either a comma (Verilog2001) or the keyword `or`.

Rules and Examples

```
always @(posedge clk or posedge rst)
  if (rst)
    q <= 0;
  else
    q <= d;
```

`posedge` is used in synthesizable RTL code to imply clocked logic. The above synthesizes to a rising-edge clocked register with an active-low reset for `q`.

```
initial begin
  @(posedge enb, valid);
  ...
end
```

`posedge` is also used in verification code to control execution of initial blocks or tasks. Here the event control is triggered on a rising-edge on `enb` or any change in value on `valid`.

posedge and negedge are evaluated as follows:-

from to	0	1	x	z
0	no edge	posedge	posedge	posedge
1	negedge	no edge	negedge	negedge
x	negedge	posedge	no edge	no edge
z	negedge	posedge	no edge	no edge

See Also

`negedge, event expression`

Primitive - built-in

Pre-defined low-level modules for gate-level modelling.

Syntax

`and (<out>,<in>,[<in>]*);`	logical and
`buf (<out>,<in>);`	simple buffer
`bufif0 (<out>,<in>, <nenable>);`	active-low enabled tri-state buffer†
`bufif1 (<out>,<in>, <enable>);`	active-high enabled tri-state buffer†
`nand (<out>,<in>,[<in>]*);`	logical nand
`nor (<out>,<in>,[<in>]*);`	logical nor
`not (<out>,<in>);`	logical not
`notif0 (<out>,<in>, <nenable>);`	active-low enabled tri-state inverter^
`notif1 (<out>,<in>, <enable>);`	active-high enabled tri-state inverter^
`or (<out>,<in>,[<in>]*);`	logical or
`xnor (<out>,<in>,[<in>]*);`	logical xnor
`xor (<out>,<in>,[<in>]*);`	logical xor
`pulldown (<signal>);`	pulldown
`pullup (<signal>);`	pullup

† when enable active, out = in, else out = Z

^when enable active, out = ~in, else out = Z

Rules and Examples

```
buf b1 (out1, out2, in);
// unnamed instance
and (out, in1, in2, in3, in4);
```

Primitives are instantiated like modules using named port connection only. It is optional to specify an instance name for a primitive instantiation. Output ports must be listed before inputs. Input ports for `and`, `nand`, `nor`, `or`, `xnor` and `xor` are expandable - any number of nets can be connected. Some synthesis tools support primitives, but their use in RTL code is rare.

See Also

`primitive` (UDP)

User-Defined Primitives (UDPs) provide an easy mechanism to extend the built in primitives by defining logic in tabular format. UDPs are declared and compiled separately, like a module.

Syntax

UDPs can be either combinational:-

```
primitive name (port_list);
  IO_declarations
  table
    input_level_list : output;
    ...
  endtable
endprimitive
```

or sequential:

```
primitive name (port_list);
  IO_declarations
  <initial_sequential_block>
  table
    input_seq_list : current : next;
    ...
  endtable
endprimitive
```

where `port_list` defines the name of a single output port first, followed by input port names. UDP behaviour is defined using a `table` construct only.

A combinational UDP table entry consists of input level values `input_level_list`, followed by the corresponding output value.

A sequential UDP table entry consists of input level or transition values `input_seq_list` followed the current state of the output `current` and finally the resulting output value `next`. Sequential UDPs can contain optional `initial` blocks for initialization.

Rules and Examples

UDPs are self contained, they cannot instantiate other modules or primitives. UDP behavior is described in a truth table using the `table` construct. UDPs can have only one output, which must be declared first, and from 1-10 inputs. All ports must be scalar and bi-directional ports are not allowed. UDPs do not support `z` values.

Combinational UDP

```
primitive multiplexer (o, a, b, s);
  output o;
  input a, b, s;
  table
  // a b s : o
     0 ? 1 : 0;
     1 ? 1 : 1;
     ? 0 0 : 0;
     ? 1 0 : 1;
     0 0 x : 0;
     1 1 x : 1;
  endtable
endprimitive
```

The table defines the output o for different combinations of inputs `a`, `b` and `s`. In the table, inputs are always listed in order of declaration with the input construct, and the single output value is defined last, after the colon.

A combinational table entry can only contain the following symbols for input values:

`0`	`1`	`x` or `X`	`?`	`b` or `B`
logic 0	logic 1	unknown	1, 0 or x	0 or 1

? is a wildcard value representing either `1`, `0` or `x`. For example in the first entry, if `a` = 0 and `s` = 1, then regardless of the value of `b`, output `o` = 0 (same as `a`).

Output values are restricted to `0`, `1`, `x` or `X`.

Note: the output becomes `x` for *any* input combination not specified in the table. The final two entries reduce x-pessimism by defining if `a = b`, then even if `s` = `x`, the output `o` is the same as `a` and `b`.

Sequential UDP

With sequential UDPs, the table entries can represent transitions on inputs as well as levels. A maximum of one input transition may be specified in any table entry. If a transition is specified for any input, every transition must be specified for that input. Any level entries will take precedence over transition entries as they are processed last. The output must be declared as `reg` for a sequential table. An optional `initial` block in the primitive allows initialization of the output.

Sequential UDP (continued)

```
primitive d_edge_ff (q, clk, data);
  output q;
  input clk, data;
  reg q;
  table
// clk data state next
    (01) 0 : ? : 0 ;
    (01) 1 : ? : 1 ;
    (0x) 1 : 1 : 1 ;
    (0x) 0 : 0 : 0 ;
    (x1) 0 : 0 : 0 ;
    (x1) 1 : 1 : 1 ;
  // ignore negative edge of clock
    (?0) ? : ? : - ;
    (1x) ? : ? : - ;
  //ignore data changes on steady clock
     ? (??) : ? : - ;
  endtable
endprimitive
```

A sequential table maps current and next state information for the output. For next state value only, the symbol – defines that the output does not change.

Transitions are defined with 2 values in brackets, e.g. `(01)` is a 0 -> 1 transition, `(??)` is *any* transition.

For `d_edge_ff`, a 0 to 1 transition on `clk` with `data` = 0 gives 0 as the next state on output `q` regardless of the current state of `q`.

For defining transitions, shorthand symbols can be used for better readability:.

Symbol	Interpretation	Description
`b`	`1` or `0`	any known value
`r`	`(01)`	`0->1` transition
`f`	`(10)`	`1->0` transition
`p`	`(01)` or `(x1)` or `(0x)`	positive edge inc. `x`
`n`	`(10)` or `(1x)` or `(x0)`	negative edge inc. `x`
`*`	`(??)`	any transition

```
// clk data state next
      r  0 : ? : 0 ;
      ?  * : ? : - ;
...
```

Initializing Sequential UDPs

```
primitive latch (q, clock, data);
  output q;
  reg q;
  input clock, data;

  initial
    q = 1'b1;

  table
 // clock data current next
      0 1 : ? : 1 ;
      0 0 : ? : 0 ;
      1 ? : ? : - ;
  endtable
endprimitive
```

The `latch` UDP uses an `initial` block to initialize the output `q`. Note that although this is a sequential UDP, the table is defined without transition entries.

Using UDPs

UDPs are instantiated like a built-in primitive.

```
multiplexer m1 (out, a, b, select);
d_edge_ff f1 (op, clock, data);
```

A UDP can only have one output, therefore a design requiring multiple outputs can only be created using multiple UDPs.

Synthesis Issues

UDPs are not synthesizable.

See Also

`primitives (built-in)`

Register Data Types

Register data types are used for variables which are assigned in always or initial procedural blocks. There are 4 types - `reg`, `integer`, `time` and `real`.

Syntax

```
reg <[size]> name;
integer name;
real name;
time name;
```

where `size` for `reg` types is optional.

Rules and Examples

There are four register data-types:

`reg`	Unsigned integer variable of varying bit width	Most common register type. Can be declared as scalar or vector.
`integer`	Signed integer variable, 32-bits wide.	Used for signed arithmetic and general use. Always 32 bits.
`time`	Unsigned integer variable, 64-bits wide.	Used for manipulating simulation time, delays etc
`real`	Signed floating-point variable, double precision.	Used for real numbers

Register types can only be updated from within a procedure. Procedures can only update register types. A register data-type cannot be driven from a continuous assign statement or from a module or primitive instance output port.

```
reg [3:0] vect; // 4-bit unsigned vector
reg [2:0] p, q; // 2 3-bit unsigned vector
integer aint;   // 32-bit signed integer
reg s;          // single bit reg
time delay;     // time value

always @(p) p = ~q;  // procedural    ☑

initial aint = 5;    // procedural    ☑

assign p = ~q;       // continuous    ☒
```

Integer Assignment

Arithmetic using `integer` variables is signed values (2's complement values). reg variables are unsigned (binary). reg and integer variables can be freely assigned to each other, with over-sized data being truncated, and under-sized data padded.

```
integer int;
reg [3:0] data;
initial
  begin
  int = 129;     // 11..1000001
  data = int;    // data = 0001
  int = data;    // int = 1
  int = -3;      // 11..1111101
  data = int;    // data = 1101
  int = data;    // int = 13
end
```

When a negative `integer` value is assigned to a `reg` variable, the `reg` variable will contain the 2's complement value, but as `reg` types are unsigned, the value is treated as a positive number. So if this value is then assigned back to an `integer` variable, the `integer` variable will contain a positive value.

Verilog2001 signed variables

The Verilog2001 `signed` keyword allows `reg` variables to be defined as signed (2's complement) data. See `signed` on page 123.

Synthesis Issues

`reg` and `integer` data-types are synthesizable. `time` and `real` types are not.

See Also

Net data-type, `always, initial, signed`

A procedural release removes the effect of an existing `force` statement.

Syntax

```
release variable_name;
```

Rules and Examples

Release is similar to the procedural `deassign`, except it can affect net types as well as register types.

```
assign d = a & b & c;

initial begin
  #10;
  force d = (a | b | c);
  #10;
  release d;
  #10;
   ...
```

The release procedural statement removes any existing force driver to a variable. If a procedural assign statement is active on the variable, the assignment immediately takes effect when the force is released. Otherwise the variable maintains it's current value until assigned by a procedural assignment.

In the `initial` block, the `force` statement makes `d` behave as the logical `or` of its inputs for the simulation time 10 to 20.

A procedural force statement overrides all procedural assignments to a register type or all drivers on a net variable, including module outputs.

Synthesis Issues

`release` is not synthesizable.

See Also

`assign (procedural), deassign, force`

repeat

A procedural loop statement where the number of iterations is set by a literal, variable or expression

Syntax

```
repeat (expression)
  begin
  repeat_loop_statement(s)
  end
```

where *expression* is a literal, variable or expression. Begin...end can be omitted for a single loop statement.

Rules and Examples

```
repeat (4)
  begin
  if (shift_opb[0])
    result = result + shift_opa;
  shift_opa = shift_opa << 1;
  shift_opb = shift_opb >> 1;
  end
```

Repeat can be used with a literal to loop for a set number of times.The repeat can also iterate for a number of times defined by the value of a variable or expression.

```
repeat(number_of_edges)
  @(negedge clk) ip <= ~ ip;
```

By adding event control in the loop, repeat can generate stimulus or count events on signals.

Synthesis Issues

Repeat loops are synthesizable if the number of iterations are fixed, i.e. known at compilation.

See Also

`for, forever, while,`

signed

A Verilog2001 type qualifier which defines a register or net vector type as representing signed data.

Syntax

```
reg signed <[size]> name;
net_type signed <[size]> name;
```

where *size* defines the upper and lower bounds of the vector. and *net_type* is any net data type (e.g. `wire`).

Rules and Examples

By default, only `integer` and `real` data types are signed (i.e. 2's complement representation). The `signed` qualifier allows variables of vector types such as `reg` and `wire` to represent signed data.

```
reg signed [3:0] sreg;
reg [3:0] usreg;
integer aint = -5;

initial begin
  sreg  = aint;  // sreg = 4'b1011 = -5
  aint  = sreg;  // aint = -5
  usreg = aint;  // usreg = 4'b1011 = +11
  aint  = usreg; // aint = +11
...
```

Rules for Signed expressions

- Decimal numbers are signed, but based numbers are unsigned, except where the `s` qualifier is used, e.g. `4'sb1001`
- Bit-select and part-select results are always unsigned, regardless of the operands and *even* if the part-select specifies the entire vector
- Concatenate results are always unsigned
- Comparison results (1, 0) are always unsigned
- If *any* operand is unsigned, the result is unsigned
- If *all* operands are signed, the result will be signed

See Also

Literals (`s` qualifier), Operators (arithmetic shifts), $System Tasks: Type Conversion (`$signed`, `$unsigned`)

Defines individual path-base propagation delays between the input and output ports of a module. A specify block can also include timing checks.

Syntax

```
specify
  <specparam_declaration>
  path_declaration
  timing_check
endspecify
```

where a `specify` block can contain optional `specparam` declarations, path delays or timing checks

Rules and Examples

A specify block is declared within a module. Path delays specify the delays from inputs or inouts to outputs or inouts of the module. The paths must be enclosed in parentheses.

```
module mone (out, a, b, c);
  output out;
  input a, b, c;
  ...
  specify
    (a => out) = 2;
    (b => out) = 3;
    (c => out) = 1;
  endspecify
endmodule
```

This defines the propagation delay from input `a` to output `out` as 2 time units, `b` to `out` as 3 time units and `c` to `out` as 1. Any change in the inputs is reflected at the output only after this delay. The time units are defined by the active `'timescale` compiler directive.

Parallel Connection

Parallel connection (=>)defines a path between each bit in the source and each corresponding destination bit. Source and destination must contain the same number of bits.

```
(a[1:0] => qa[1:0]) = 5;
// equivalent to
//(a[1] => qa[1]) = 5;
//(a[0[ => qa[0]) = 5;
```

Multiple sources and destinations are allowed

```
(a, b => qa, qb) = 5;
// equivalent to
//(a => qa) = 5;
//(b => qb) = 5;
```

Full Connection

Full connection (`*>`)defines a path between every bit in the source and every destination bit. Source and destination need not contain the same number of bits. Multiple sources and destinations are again allowed:-

```
(a, b *> qa, qb) = 5;
// equivalent to
//(a => qa) = 5;
//(a => qb) = 5;
//(b => qa) = 5;
//(b => qb) = 5;
```

Separate delay values can be defined for rise, fall and turn-off (transition to z). Maximum, typical and minimum delays can be defined for each transition. See `Delay #` on page 64

```
// min:typ:max for rise, fall, & turnoff
(b => y) = (2:3:4, 3:4:6, 4:5:8);
```

Edge Sensitive Paths

By using `posedge` or `negedge`, an edge-specific path delay can be defined. Edge-sensitive delays require a polarity operator to define whether the path is inverting (`-:`) or non-inverting (`+:`):

```
module mtwo (out, a, b, clk);
  output out;
  input a, b, clk;
  ...
  specify
    ( posedge clk => (out +: a) ) = 5;
    ( posedge clk => (out -: b) ) = 8;
  endspecify
endmodule
```

A rising-edge on `clk` creates a path with a delay of `5` from `a` to `out`, where `a` is not inverted. Also an inverting path from `b` to `out` is enabled with a delay of `8`.

State Dependent Paths

State dependent path delay are assigned to a module path *only* if a specific condition is true. Note - `else` is **not** allowed for a specify block `if` statement.

```
module mtwo (out, a, b);
  output out;
  input a, b;
  ...
  specify
    if (a)  (b=>op) = (5:6:7);
    if (!a) (b=>op) = (5:7:8);
    if (b)  (a=>op) = (4:5:7);
    if (!b) (a=>op) = (5:7:9);
  endspecify
endmodule
```

The delay from `a` or `b` to `op` is dependent on the state of the other input. State dependency can also be defined for edge-sensitive paths.

ifnone Condition

Defines a default state-dependent path delay when all other state-dependent conditions for the path are false

```
specify
  if (a)  (b=>op) = 10;
  ifnone  (b=>op) = 5;
endspecify
```

If `a` is high, the path delay from `b` to `op` is 10. If `a` is not high, the `ifnone` condition sets the path delay to 5.

It is an error to specify both an `ifnone` condition and an unconditional path for the same path.

Specify Parameters (specparam)

Normal parameters cannot be used in specify blocks to define delays. A Specify Parameter (`specparam`) must be declared inside a specify block for this purpose. Specparams cannot be over-ridden. See `specparam` on page 127.

Synthesis Issues

Specify blocks are not synthesizable.

See Also

`specparam, delay`

A parameter used within a specify block.

Syntax

```
specparam name = expr;
```

where *type* is an optional type specifier, *name* is an identifier and *expr* a constant expression.

Rules and Examples

```
module mone (out, a, b, c);
  output out;
  input a, b, c;
  ...
  specify
    specparam aop = 2,
              bop = 3,
              cop = 1;
    (a => out) = aop;
    (b => out) = bop;
    (c => out) = cop;
  endspecify
endmodule
```

A normal parameter cannot be used inside a `specify` block. A `specparam` must be used. A `specparam` must be declared inside a `specify` block and can only be used inside the `specify` block where it is declared. A `specparam` *cannot* be over-ridden like a normal parameter.

Synthesis Issues

Specify blocks are not synthesizable.

See Also

`specify, parameter`

Strength Specification

Allows logic strength specification in low-level modelling using primitives, switches and nets This level of detail is typically only used by component modelers, such as FPGA or ASIC library developers.

Syntax

Keyword	Strength Level
`supply0, supply1`	7
`strong0, strong1`	6
`pull0, pull1`	5
`large0, large1`	4
`weak0, weak1`	3
`medium0, medium1`	2
`small0, small1`	1
`highz0, highz1`	0

Rules and Examples

A strength specification can be attached to a primitive or switch instantiation, or a net declaration. The strength applies to any variable driven by the object:

```
// strength specification on a primitive
or (supply0,highz1) o1 (out,in1,in2,in3);
// strength specification on a net
trireg (small) t1;
// strength and delay on a primitive
nand (strong1,pull0) #(2:3:4) n1 (o,a,b);
```

If two or more variables of unequal strength combine in a wired net configuration, the stronger variable shall dominate all the weaker drivers and determine the result.

The combination of variables with unlike values and the same strength or ambiguous values (x, z) are complex. Please refer to the Verilog Language Reference Manual for more information.

Synthesis Issues

Strength specifications are not synthesizable.

See Also

`primitive, switch, delay`

String types are not directly supported in Verilog. However you mimic string behavior by creating `reg` vectors, where each 8-bit slice of the vector contains the ASCII encoding for a character.

Syntax

```
reg [8*<characters> : 1] name;
```

where *characters* is the number of characters in the string and *name* is an identifier. Note other forms of declaration are possible, but this is the simplest.

Rules and Examples

```
reg [4*8:1] strvar;
initial begin
  strvar = "AB";
  $display("strvar %s %h",strvar,strvar);
...
```

Creates the output:-

```
strvar AB 00004142
```

Where `41` is the ASCII code for character `A` and `42` for `B`. String variables can be displayed in ASCII using the `%s` format specifier, or as binary, octal or hex using other specifiers.

String assignment is just like normal vector assignment. Characters are loaded from the right-hand side of the vector, towards the left, with 8-bits for each character. Characters are padded with zero or truncated from the left if the string and variable width do not match.

String vectors can be defined and manipulated (tested and compared) to be viewed in a waveform window, or to produce more meaningful system task messages (see `$System Control: Display Tasks` on page 24). Strings can also be read from an external file, (see `$System Control: File Input` on page 28).

Special characters can also be embedded into strings, such as tabs or line-feeds (see `\ Special Characters` on page 22).

See Also

$System Control Tasks, \ Special Characters

Switch Devices

Built-in primitives for switch-level modeling of circuits. This level of detail is typically only used by component modelers, such as FPGA or ASIC library developers.

Syntax

`nmos (<out>,<in>, <enable>);`	NMOS transistor
`pmos (<out>,<in>, <enable>);`	PMOS transistor
`rnmos (<out>,<in>, <n_enable>);`	NMOS strength reduction
`rpmos (<out>,<in>, <p_enable>);`	PMOS strength reduction
`cmos (<out>,<in>, <n_enable>,<p_enable>);`	CMOS transistor
`rcmos (<out>,<in>, <n_enable>,<p_enable>);`	CMOS strength reduction
`tran (<io>,<io>) ;`	Bidirectional switch
`rtran (<io>,<io>) ;`	`tran` strength reduction
`tranif0 (<io>,<io>, <enable>) ;`	`tran` with active-low enable signal
`tranif1 (<io>,<io>, <enable>) ;`	`tran` with active-high enable signal

Rules and Examples

```
pmos p1 (out, data, control);
```

Creates an instance `p1` of a pmos switch with output `out`, data input `data`, and control input `control`. For truth tables defining the operation of switches, please consult the Verilog Language Reference manual.

Synthesis Issues

Switch devices are not synthesizable.

See Also

`primitive, strength`

A task is a subprogram, returning zero or more values, which is used as a procedural statement.

Syntax

```
task <automatic> task_name;
> (argument_list);
  <declarations;>
begin
  procedural_statement(s)
end
endtask
```

where `argument_list` defines the in, out and in-out arguments of the task and `declarations` are local variables used only within the task. Note the task statements **must** be enclosed in `begin-end`. `automatic` is an optional Verilog2001 qualifier.

Rules and Examples

A task is a group of procedural statements which is declared once, and used many times (with different variables if required) via a procedural task call.

```
task zero_count;
  input [7:0] in_bus;
  output [3:0] count;
  integer i;
begin
  count = 0;
  for (i = 0; i < 8; i = i + 1)
    if (!in_bus[i])
      count = count + 1;
end
endtask
```

The task named `zero_count` has a formal input argument named `in_bus` and formal output argument named `count`. `zero_count` returns a value by writing to the output argument `count`. When the task is called as a procedural statement, variables in the task call (actual arguments) are mapped to the formal arguments by position:-

```
always @ (posedge clk)
  begin
  zero_count(a_bus, a_count);
  ...
```

Tasks can be nested and can also contain functions, but a function cannot contain a task call.

Unlike functions, tasks can contain event control and delays, therefore they are customarily used for behavioral modeling in testbenches and stimulus models only.

Tasks have two major issues:-

- Task arguments are static.
- Task calls are static.

Static Task Arguments

```
module task_test;
reg req, ack;
reg [7:0] data;

task driver;
  input [7:0] write_data;
begin
  #30 req = 1'b1;
  wait (ack == 1'b1);
  #20 data = write_data;
  wait (ack == 1'b0);
  #20 data = 8'hzz;
  reg = 1'b0;
end
endtask

initial begin
  driver(8'hff);
  ...
```

Task arguments are static, i.e. passed by value. Input arguments are sampled when the task is called and output arguments are assigned only when the task completes. Arguments are not continuously updated.

If `ack` was declared as a formal input argument to the task, then a task call will wait forever for the condition `ack == 1’b1` to be true, as changes in `ack` during the lifetime of the task call will not seen inside the task.

This would not be a compilation error

The static argument issue is solved by exploiting the feature of tasks to access any variables local to the task call, without the variables being passed as arguments.

Static Task Calls

Task calls are not duplicated like conventional subprograms. Only one copy of the task exists. Simultaneous concurrent task calls can cause conflict with task variables and arguments and lead to indeterminate simulation behaviour.

```
task neg_clocks;
  input [31:0] edge_number;
begin
  repeat(edge_number) @(negedge clk);
end
endtask

initial begin
  neg_clocks(6);  // ERROR
  ...
end
always @(posedge trigger)
  begin
  neg_clocks(10); // ERROR
  ...
```

Here two simultaneous calls to the `neg_clocks` task from the initial and always blocks causes conflict.

Automatic Tasks (Verilog2001)

```
task automatic neg_clocks;
  input [31:0] edge_number;
begin
  repeat(edge_number) @(negedge clk);
end
endtask

initial begin
  neg_clocks(6);  // automatic task call
  ...
end
always @(posedge trigger)
  begin
  neg_clocks(10); // automatic task call
  ...
```

In Verilog2001 tasks can be defined as automatic. A unique copy of an automatic task is created for each call. This allows safe concurrent task calls as each has its own local copy of variables and arguments.

Automatic Tasks (Verilog2001) (continued)

Automatic tasks have a number of restrictions:-

- As multiple copies of a specific task may exist, hierarchical references to declarations within automatic tasks are not permitted.
- Local variables declared in an automatic task cannot be assigned with non-blocking assignment or accessed via constructs such as $monitor. Variables declared in automatic tasks are deallocated at the end of the task call, therefore, they cannot be used with such constructs that may refer to them after the task completes.

Disabling Tasks

A task call can be terminated from outside the task using the disable procedural statement.

```
initial begin
  cpu_driver(8'h00);
...
end
always @(posedge interrupt)
  begin
  disable cpu_driver;
  service_interrupt;
end
```

Here if `interrupt` is detected, the `cpu_driver` task is forced to exit and the interrupt can then be serviced. Disabling a task terminates all active calls to that task, but does not prevent subsequent calls. Disabling a task also disables all nested task calls within the disabled task.

Synthesis Issues

Tasks are synthesizable if they do not contain any delays or event control. However a task without timing can be replaced by a function (or number of functions) therefore tasks are rarely used in synthesizable code.

See Also

`automatic, function`

Timing checks are used in `specify` blocks *only* to check timing behavior.

Although timing checks begin with a `$` character, they are **not** considered as system tasks or functions. System tasks and functions are *not* allowed in `specify` blocks, and timing checks are *not* allowed in procedural code.

Timing checks are divided into 2 groups:

Stability Time Window Checks

These define a time window with respect to a reference signal, and check transitions on a data signal with respect to the time window.

Simplified syntax for the stability timing checks are:-

```
$setup (data_event, reference_event,
       setup_limit, <notify_reg>);

$hold (reference_event, data_event,
      hold_limit, <notify_reg>);

$setuphold (reference_event, data_event,
      setup_limit, hold_limit,
      <notify_reg>);

$removal (reference_event, data_event,
          removal_limit, <notify_reg>);

$recovery (reference_event, data_event,
           recovery_limit, <notify_reg>);

$recrem (reference_event, data_event,
          recovery_limit, removal_limit,
         <notify_reg>);
```

For example, `$setup` checks that a transition on `data_event` does not occur within `setup_limit` before the `reference_event`. `setup_limit` cannot be negative and if zero, the timing check never fails. The optional `notify_reg` toggles when a timing violation is found, and can be used to trigger code.

`$hold` is a separate hold-time check, and `$setuphold` combines a `$setup` and `$hold` into one check.

`$removal` and `$recovery` are setup and hold checks where the reference event is a control signal like a register reset, while the data event is a clock. `$recrem` is a combined `$recovery` and `$removal` check.

$setup Example

```
'timescale 1 ns/1 ns
module dff_nt (q, ck, d, rst);
input ck, d, rst;
output q;
reg nt;

u_ffd_rb i1 (q, d, ck, rst, nt);

specify
  specparam tsu = 2;
  (posedge ck => (q +: d)) =(2:3:4);
  $setup (d, posedge ck, tsu, nt);
endspecify
endmodule
```

if `d` changes value within 2ns (`tsu`) before a positive edge on `ck`, then a timing violation is reported and `nt` toggles. Rememeber a timing check can only be placed inside a `specify` block.

Clock and Control Signal Checks

The second group of timing checks accept one or two signals and verify that transitions on them are never separated by more than the limit:-

Simplified syntax for the stability timing checks are:-

```
$skew (reference_event, data_event,
       limit, <notify_reg>);

$timeskew (reference_event, data_event,
           limit, <notify_reg>);

$fullskew (reference_event, data_event,
           limit1, limit2, <notify_reg>);

$width (reference_event, limit,
        threshold, <notify_reg>);

$period (reference_event, limit,
         <notify_reg>);

$nochange (reference_event, data_event,
           start_offset, end_offset,
           <notify_reg>);
```

`$skew` checks that a data event occurs within *`limit`* time of a reference event. `$skew` is event-based - once a reference event is detected, `$skew` will wait indefinitely for a data event and will not report a violation until the data event occurs. If a new reference event occurs, the old check is cancelled and a new one started.

`$timeskew` is timer-based. If a data event does not occur within *`limit`* time after a reference event, a violation is reported immediately. `$timeskew` does not wait for a data event to occur.

`$fullskew` is identical to `$timeskew`, except the reference and data events can occur in any order. `limit1` is the maximum time by which the data event can follow the reference and `limit2`, the maximum time the reference event can follow the data.

`$width` performs a pulse-width check from the reference event (which must be edge triggered) to the *opposite edge* of the reference event. No violation is reported for glitches shorter than the `threshold`.

`$period` performs a pulse-width check from the reference event (which must be edge triggered) to the *next same edge* of the reference event.

`$nochange` reports a violation if a data event occurs during a specified level on the reference signal. The reference level is specified with `posedge` (high) or `negedge` (low). For example:-

```
$nochange( posedge clk, data, 0, 0);
```

Reports a violation if an event on `data` occurs while `clk` is high.

The offsets extend the timing window. A positive start offset extends the window to before the start event, a negative start offset to after the start event. Like wise a positive end offset extends the window to after the end event, a negative end offset to before the end event.

```
$nochange( posedge clk, data, 0.2, 0.3);
```

Reports a violation if an event on `data` occurs from 0.2ns before the rising edge of `clk` to 0.3ns after the falling edge of `clk`, assuming a time unit of 1ns.

See Also

`specify, 'timescale`

Verilog2001 construct used in configurations to specify the exact library and module for the binding of a selected module.

Syntax

```
use <lib.> name <:config>;
```

where *lib* is an optional library logical name and *name*, a module or configuration name from the library. If *name* is a configuration, then the `config` qualifier can be used.

Rules and Examples

A `use` clause can be used with a `cell` or `instance` configuration rule, but not a `default` rule:-

```
config cfg;
 design rtlLib.top;
 default liblist gateLib rtlLib;
 cell adder use gateLib.adder;
 instance top.u1 use rtlLib.adder;
endconfig
```

Configuration `cfg` identifies module `top` from the library `rtlLib` as the top-level module. It specifies a default library search order of `gateLib -> rtlLib`.

The `cell use` clause binds all instances of `adder` in the design to the module `adder` in library `gateLib`.

The `instance use` clause binds instance `top.u1` in the design to the module `adder` in library `rtlLib`.

The optional `config` suffix can be used to explicitly reference a `config` when it has the same name as a module in the library.

Library Inheritance

If no library is specified in a use clause, the library search order is inherited from the parent module.

See Also

`cell, config, design, default, instance, liblist`

A procedural statement which implements a level-sensitive event control.

Syntax

```
wait expression;
```
.

Rules and Examples

The wait statement blocks the procedure execution until the condition becomes true.

```
always
  begin : HANDSHAKE
  wait (req == 1);
  ack = 1;
...
```

Here the wait statement will suspend execution of the always block until the condition `req == 1` is true, at which time the procedure continues execution with assignment of `ack` to 1. If the condition is true when the wait statement is executed, the procedure is not suspended and the wait is effectively skipped.

```
always @(a or b)
  begin
  wait (!en)
  out = a + b;
  end
```

Waits can be combined with procedural event expressions. Here an event on `a` or `b` triggers the always block. If `en` is low, then the addition occurs immediately. Otherwise the always waits until `en` goes low. Note that if `a` or `b` change value while the always is waiting for `en`, then the block does not restart, but remains suspended at the `wait` statement until the condition is true, at which point the addition is executed.

Synthesis Issues

wait statement are non-synthesizable.

See Also

`event expression, posedge negedge, delay`

while

A procedural loop statement where the number of iterations is controlled by a condition.

Syntax

```
while (condition)
  begin
  sequential_statement(s)
  end
```

where the sequential statements are executed while `condition` evaluates to true (or non-zero). If `condition` is initially false, the statements are not executed. Begin-end is optional for a single statement.

Rules and Examples

```
while (count < 10)
  begin
  @(posedge clk)
  count = count + 1;
  end
...
```

The count condition is checked, and if true the while loop statements are executed. The condition is checked before each loop execution, and the loop statements are executed until the `while` condition is false. Multiple loop statements must be enclosed in `begin-end`.

Note if the condition is initially false, the loop is never executed.

If a standalone variable is used in a `while` loop condition, the variable size determines how many times the loop is executed:-

```
while (var1) //loops for length of var1
  begin
  if (tempreg[0])
    count = count + 1;
  tempreg = tempreg >> 1;
  end
```

Synthesis Issues

While loops are generally not synthesizable.

See Also

`for, forever, repeat`

Index

Symbols

- (subtract), 102
! (logical not), 104
!= (logical inequality), 106
!== (case inequality), 106
$bitstoreal, 41
$display, 24
$fclose
  Verilog2001, 27
  Verilog95, 26
$fdisplay, 31
$ferror, 30
$fflush, 30
$fgetc, 28
$fgets, 28
$finish, 37
$fmonitor, 31
$fopen
  Verilog2001, 27
  Verilog95, 26
$fread, 28
$fscanf, 32
$fseek, 30
$fstrobe, 31
$ftell, 30
$fullskew, 136
$fwrite, 31
$hold, 135
$itor, 41
$monitor, 24
$monitoroff, 24
$monitoron, 24
$nochange, 136
$period, 136
$printtimescale, 21, 39
$random, 35
$readmem, 33
$realtime, 39
$realtobits, 41
$recovery, 135
$recrem, 135
$removal, 135
$rewind, 30
$rtoi, 41
$sdf_annotate, 23, 36
$setup, 135
$setuphold, 135
$sformat, 38
$signed, 41
$skew, 136
$sscanf, 32
$stime, 39
$stop, 37
$strobe, 24
$swrite, 38
$test$plusargs, 34
$time, 39
$timeformat, 39
$timeskew, 136
$ungetc, 28
$unsigned, 41
$value$plusargs, 34
$width, 136
$write, 24
% (modulus), 102
% Format Specifier, 42
%t, 39
& (and), 103
&& (logical and), 104
* (multiply), 102
*> (specify block path), 125
+ (add), 102
+: (indexed part select), 107
+: (specify block path), 125
/ (divide), 102
-: (indexed part select), 107
-: (specify block path), 125
< (less than), 105
<< (shift), 104
<<< (arithmetic shift), 104
<= (less than or equal to), 105
<= Non-Blocking Assignment, 101
= Blocking Assignment, 52
== (logical equality), 106

=== (case equality), 106
=> (specify block path), 124
> (greater than), 105
-> event trigger, 68
>= (greater than or equal to), 105
>> (shift), 104
>>> (arithmetic shift), 104
? (conditional operator), 106
@ event timing control, 69
@(*) - Automatic Event List, 45
\ Special Characters, 22
^ (xor), 103
{ } (concatenation), 107
{{ }} (replication), 107
| (or), 103
|| (logical or), 104
~ (bit-wise not), 103
‘celldefine, 16
‘Compiler Directives, 16
‘default_nettype, 17, 46, 99
‘define, 18
‘else, 19
‘elsif, 19
‘ifdef, 19
‘ifndef, 19
‘include, 20
‘nounconnected_drive, 16
‘resetall, 16
‘timescale, 21
‘unconnected_drive, 16
‘undef, 18

A

Active Scheduling Region, 8
always, 44
and, 114
ANSI C
  Port Declaration, 96
Arithmetic Operator, 102
assign
  continuous, 46
  delay, 47
  Implicit, 99
  procedural, 48
  Strength, 47
Assignment
  Blocking, 52
  Complete, 10
  Continuous
    Implicit, 99
  Delay, 64
  Intra-Assignment
    delay, 65
  Non-Blocking, 100
automatic, 49
  function, 77
  Restrictions, 50
  task, 133

B

backslash, 22
begin, 51
Bit-wise Operator, 103
Blocking Assignment, 52
  Combinational
    Logic, 10
buf, 114
bufif0, 114
bufif1, 114

C

case, 11, 54
  default, 54, 61
  full, 11, 55
  generate, 79
  parallel, 11, 55
casex, 56
casez, 56
cell, 57, 59
cmos, 130
Concatenation, 107
Conditional Operator
  ?, 106
config, 58
configuration, 96
  default, 62

D

deassign, 60
Declaration
  Implicit, 99
  Local, 85
Declarations
  Implicit, 46
  Local, 45
default
  case, 54, 61
  configuration, 59, 62
defparam, 63, 111
Delay
  assign, 47
  Assignment, 64
  Gate, 65
  Intra-Assignment, 65
  Net, 65
design, 59, 66
disable, 67
  task, 134

E

else, 82
end, 51
Equality, 106
Event
  Scheduling, 8
event, 68
Event List, 44
  Comma Separated, 45, 70

F

File Designator (fd), 27
for, 71
  generate, 78
force, 72
forever, 73
fork, 74
Full case, 55
full case, 11
function, 76
  automatic, 77
  input, 88

G

Gate
  Delay, 65
generate, 78
  case, 79
  for, 78
  if, 79
genvar, 81

H

highz0, 128
highz1, 128

I

if, 82
  generate, 79
ifnone, 126
Implicit Declaration, 99
Implicit Declarations, 46
Inactive Scheduling Region, 9
include, 91
Indexed Part Select, 107
Inequality, 106
initial, 84
inout, 86
input
  function, 88
  module port, 87
  task, 88
instance, 59, 89
integer, 119

J

join, 74

L

large0, 128
large1, 128
liblist, 59, 90
library, 91
Literal, 92
Local Declarations, 45, 85
localparam, 93
  type, 93
Logical Operator, 104

M

medium0, 128
medium1, 128
Module
  Parameter, 95
  Port Declaration, 96
module, 94
Monitor Scheduling Region, 9, 24, 31
Multi-Channel Designator (MCD), 26

N

Named Block
  Block
    Named, 51
nand, 114
NBA Scheduling Region, 9
negedge, 12, 97
Net
  Delay, 65
Net Data Type, 98
new-line, 22
nmos, 130
Non-Blocking Assignment, 100
  Clocked Procedure, 12
nor, 114
not, 114
notif0, 114
notif1, 114

O

Operator, 102
  Arithmetic, 102
  Bit-Wise, 103
  Concatenation, 107
  Conditional, 106
  Equality, 106
  Indexed Part Select, 107
  Logical, 104
  Precedence, 108
  Reduction, 103
  Relational, 105
  Replication, 107
  Shift, 104
or, 44, 114
output, 109
  task, 110

P

Parallel case, 55
parallel case, 11
Parameter, 95
  Modification, 111
  Type, 112
parameter, 111
percentage, 22
pmos, 130
Port
  output, 109
Port Connection
  Named, 95
  Positional, 95
Port List
  Verilog2001, 96
posedge, 12, 113
Primitive
  Built-In, 114
  User-Defined (UDP), 115
printtimescale, 21
pull0, 128
pull1, 128
pulldown, 114
pullup, 114

Q

quote, 22

R

rcmos, 130
real, 119
Reduction Operator, 103
reg, 119
Register Data Type, 119
Relational Operator, 105
release, 121
repeat, 122
Replication, 107
Resets, 12
rnmos, 130

rpmos, 130
RTL Coding
  Guidelines, 10
rtran, 130

S

Sensitivity List
  Complete, 10
Shift Operator, 104
signed, 123
small0, 128
small1, 128
specify, 124
specparam, 126, 127
Standard Delay
  Format (SDF), 36
Strength
  assign, 47
Strength Specification,
  128
String, 129
strong0, 128
strong1, 128
supply0, 98, 128
supply1, 98, 128
Switch Devices, 130
Synthesis Directive, 55
synthesis directive, 11

T

tab, 22
table, 115
Task
  output, 110
task, 131
  automatic, 133
  disable, 134
  input, 88
time, 119
Timing Checks, 135
tran, 130
tranif0, 130
tranif1, 130
tri, 98
tri0, 98
tri1, 98
triand, 98
trireg, 98

U

UDP (User-Defined
  Primitive), 115
  Combinational, 116
  Sequential, 116
use, 59, 138

W

wait, 139
wand, 98
weak0, 128
weak1, 128
while, 140
wire, 98
wor, 98

X

xnor, 114
xor, 114