Verilog®-XL Reference
Product Version 12.1

20

The Value Change Dump File

This chapter describes the following:

Overview

Verilog-XL can produce a file called a value change dump (VCD) file that contains information about value changes on selected variables in a design. You can use this VCD file for developing various applications programs and postprocessing tools. Here are some examples:

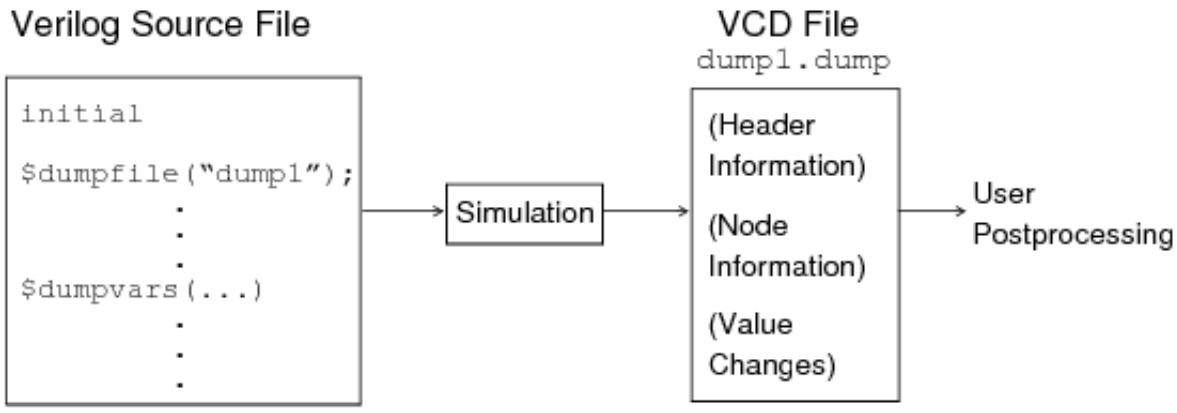
- An applications program can graphically display the results of an overnight simulation.
- A postprocessor can process the simulation results and forward them to a device tester or to a board tester.

The value change dumper is more efficient than the `$monitor` task, both in performance and in storage space. With the VCD feature, you can save the value changes of variables in any portion of the design hierarchy during any specified time interval. You can also save these results globally, without having to explicitly name all signals involved.

Creating the Value Change Dump File

The steps involved in creating the VCD file are listed below and illustrated in the following figure:

1. Insert the VCD system tasks in the Verilog source file to define the dump filename and to specify the variables to be dumped. You may also choose to invoke these tasks interactively during the simulation instead of adding them to your source file.
2. Run the simulation.



Verilog-XL produces an ASCII dump file that contains header information, variable definitions, and the value changes for all variables specified in the task calls.

Several system tasks can be inserted in the source description or invoked interactively to create the VCD file. The sections that follow describe the tasks listed in the following example:

```
$dumpfile(<filename> );
$dumpvars;
$dumpvars(<levels> <, <module_or_variable>>* );
$dumpoff;
$dumpon;
$dumpall;
$dumplimit( <filesize> );
$dumpflush;
```

Specifying the Dump File Name ($dumpfile)

Use the `$dumpfile` task to specify the name of the VCD file as follows:

```
initial
     $dumpfile ("module1.dump")
```

The filename is optional. If you do not specify a dump filename, the program uses the default name "`verilog.dump`".

The `$dumpfile` task can be invoked only once during the simulation, and it must precede all other dump file tasks, as described in the sections that follow.

Specifying Variables for Dumping ($dumpvars)

Use the `$dumpvars` task to determine which variables Verilog-XL dumps into the file specified by `$dumpfile`. You can invoke the `$dumpvars` task throughout the design as often as necessary but these

`$dumpvars` tasks must all execute at the same simulation time. You cannot add variables to be dumped once dumping begins.

You can use the `$dumpvars` task with or without arguments. The syntax for calling the task without arguments is as follows:

```
$dumpvars;
```

When invoked with no arguments, `$dumpvars` dumps all variables in the design, except those in source-protected regions, to the VCD file. If you want to include a variable from a source-protected region in the VCD file, you must include the variable in the `$dumpvars` argument list.

The syntax for calling the `$dumpvars` system task with arguments is as follows:

```
$dumpvars(<levels> <, <module_or_variable>>* );
```

When you specify `$dumpvars` with arguments, the *<levels>* variable indicates the number of hierarchical levels below each specified module instance that `$dumpvars` affects. Subsequent *<module_or_variable>* variable arguments specify which scopes of the design to dump. These subsequent arguments can specify entire modules or individual variables within a module. Here is an example:

```
$dumpvars (1, top);
```

Because the first argument in the previous example is a 1, this invocation dumps all variables within the module `top`; it does not dump variables in any of the modules instantiated by module `top`.

Setting the first argument to `0` causes a dump of all variables in the specified module and in all module instances below the specified module. Note that the argument `0` applies only to subsequent arguments that specify module instances, and not to individual variables.

In the following example, the `$dumpvars` task dumps all variables in the module `top` and in all module instances below module `top` in the design hierarchy:

```
$dumpvars (0, top);
```

The next example shows how the `$dumpvars` task can specify both modules and individual variables:

```
$dumpvars (0, top.mod1, top.mod2.net1);
```

This call dumps all variables in module `mod1` and in all module instances below `mod1`, along with variable `net1` in module `mod2`. Note that the argument 0 applies only to the module instance `top.mod1`, and not to the individual variable `top.mod2.net1`.

If you wish to dump individual bits of a vector net, first make sure that the net is expanded. Declaring a vector net with the keyword `scalared` guarantees that it is expanded. Using the `-x` command-line option expands all nets, but this procedure is not recommended due to its negative impact on memory usage and performance.

If the `$dumpvars` task is invoked with no arguments, all variables in the design except those in a source-protected region are dumped. However, if you include the name of a variable in a source-protected region in

the $dumpvars argument list, then that variable is dumped. For example, if the $dumpvars argument list contains the variable name 'top.mod2.net1', then that variable is dumped even though module 'top.mod2' may be source protected.

Stopping and Resuming the Dump ($dumpoff/$dumpon)

Executing the $dumpvars task causes value change dumping to start at the end of the current simulation time unit. To suspend the dump, invoke the $dumpoff task. To resume the dump, invoke $dumpon.

When $dumpoff is executed, a checkpoint is made in which every variable is dumped as an x value. When $dumpon is later executed, each variable is dumped with its value at that time. In the interval between $dumpoff and $dumpon, no value changes are dumped.

The $dumpoff and $dumpon tasks allow you to specify the simulation period during which the dump takes place. Here is an example:

```
initial
  begin
     #10       $dumpvars(0, top.mod1, top.mod2.net1);
     #200       $dumpoff;
     #800       $dumpon;
     #900       $dumpoff;
  end
```

This example starts the value change dumper after 10 time units, stops it 200 time units later (at time 210), restarts it again 800 time units later (at time 1010) and stops it again 900 time units later (at time 1910).

Generating a Checkpoint ($dumpall)

The $dumpall task creates a checkpoint in the dump file that shows the current value of all VCD variables. It has no arguments. An example is shown in "Sample Source Description Containing VCD Tasks".

When dumping is enabled, the value change dumper records the values of the variables that change during each time increment. Values of variables that do not change during a time increment are not dumped.

Periodically during a dump, an applications program might find it useful to check the values of all specified variables, as a convenient checkpoint. For example, a program can save considerable time obtaining a variable value by quickly backtracking to the most recent checkpoint, rather than returning to the last time the variable changed value.

Limiting the Size of the Dump File ($dumplimit)

Use $dumplimit to set the size of the VCD file as follows:

```
$dumplimit(<filesize>);
```

This task takes a single argument that specifies the maximum size of the dump file in bytes. When the dump file reaches this maximum size, the dumping stops and the system inserts a comment in the dump file

indicating that the dump limit was reached. The simulation continues uninterrupted.

Reading the Dump File During Simulation ($dumpflush)

The `$dumpflush` task empties the operating system's dump file buffer and ensures that all the data in that buffer is stored in the dump file. After executing a `$dumpflush` task, the system resumes dumping as before, so that no value changes are lost.

A common application is to call `$dumpflush` to update the dump file so that an applications program can read the file during a simulation.

Here is an example of using the `$dumpflush` task in a Verilog source file:

```
initial
   begin
      $dumpvars
      ...
      $dumpflush
      $(applications program)
   end
```

There are two ways of flushing the dump file buffer:

- Insert the `$dumpflush` task in the Verilog source description, as described above.
- Include a call to the PLI C-function `tf_dumpflush()` in your applications program C code.

The `$dumpflush` task and the PLI `tf_dumpflush()` functions are equivalent.

Sample Source Description Containing VCD Tasks

This section contains a simple source description example that produces a value change dump file. In this example, the name of the dump file is "`verilog.dump`". Verilog-XL dumps value changes for variables in the circuit. Dumping begins when event `do_dump` occurs. The dumping continues for 500 clock cycles, then stops and waits for event `do_dump` to be triggered again. At every 10000 time steps, the current values of all VCD variables are dumped.

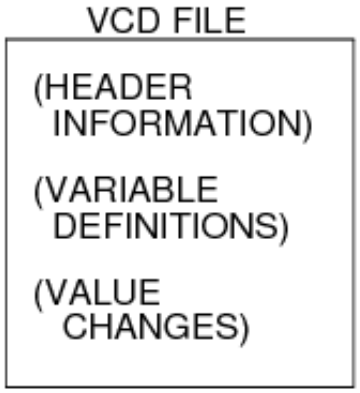
```
module dump;
   event do_dump;
initial $dumpfile("verilog.dump");       //Same as default file
initial @do_dump
$dumpvars;                               //Dump variables in the design
always @do_dump                          //To begin the dump at event do_dump
begin
   $dumpon;                              //No effect the first time through
   repeat (500) @(posedge clock);        //Dump for 500 cycles
   $dumpoff;                             //Stop the dump
end

initial    @(do_dump)
       forever #10000 $dumpall;          //Dump all variables for checkpoint
endmodule
```

Format of the Value Change Dump File

The information in this section pertains to users who write their own application programs to postprocess the VCD file.

Contents of the Dump File

As shown in the following figure, the VCD file starts with the header information (the date, the version number of Verilog-XL used for the simulation, and the timescale used). Next, the file lists the definitions of the scope and the type of variables being dumped, followed by the actual value changes at each time increment.



Only the variables that change value during a time increment are listed. Value changes for non-real variables are specified by 0, 1, X, or Z values. Value changes for real variables are specified by real numbers. Strength information and memories are not dumped.

Note: You cannot dump part of a vector. For example, you cannot dump only bits 8 through 15 (8:15) of a 16-bit vector. You must dump the entire vector (0:15). In addition, you cannot dump expressions, such as a + b.

Structure of the Dump File

The dump file is structured in a free format. White space is used to separate commands and to make the file easily readable by a text editor. Output data in the VCD file is case sensitive.

To simplify postprocessing of the VCD file, the value change dumper automatically generates 1- to 4-character identifier codes (taken from the visible ASCII characters) to represent variables. The examples in "Description of Keyword Commands" show these identifier codes, such as `*@` and `(k`.

Note that the value change dump file contains limited structural information, including information about the design hierarchy, but has no interconnection information. As a result, the VCD file by itself does not allow a postprocessor to display the drivers and loads on a net.

Formats of Dumped Variable Values

Variables may be either scalars or vectors. Each type has its own format in the VCD file. Dumps of value changes to scalar variables contain no white space between the value and the identifier code, as in this example:

```
1*@    // No space between the value 1 and the identifier code *@
```

Dumps of value changes to vectors contain no white space between the base letter and the value digits, but they do contain white space between the value digits and the identifier code, as in this example:

```
b1100x01z (k     //No space between the b and 1100x01z, but space between
b1100x01z and (k
```

The output format for each value is right-justified. Vector values appear in the shortest form possible: the VCD eliminates redundant bit values that result from left-extending values to fill a particular vector size.

The rules for left-extending vector values are as follows:

When the value is:	VCD left-extends with:
1	0
0	0
Z	Z
X	X

The following table shows how the VCD shortens values:

The binary value:	Extends to fill a 4-bit register as:	Appears in the VCD file as:
10	0010	b10
X10	XX10	bX10
ZX0	ZZX0	bZX0
0X10	0X10	b0X10

Events are dumped in the same format as scalars (for example, `1*%)`. For events, however, the value (1 in this example) is irrelevant. Only the identifier code `(*%` in this example) is significant. It appears in the VCD file as a marker to indicate that the event was triggered during the time step.

Using Keyword Commands

Much of the general information in the value change dump file is presented as a series of keyword commands that a postprocessor can parse. Refer to "Syntax of the VCD File" for information about keyword commands use.

If a postprocessor reads a keyword command that it does not recognize, it can perform error processing such as displaying a warning message and ignoring the text that appears between the keyword command and the `$end`.

Description of Keyword Commands

Keyword commands provide a means of inserting information into the VCD file. Keyword commands can be inserted either by the dumper or by you, as shown in the example in "Value Change Dump File Format Example". This section deals with the following keyword commands:

$comment	$dumpoff	$enddefinitions	$upscope
$date	$dumpon	$scope	$var
$dumpall	$dumpvars	$timescale	$version

Applications programs that read the value change dump file must be able to recognize and process the standard keyword commands defined in the following pages. You also can define additional keyword commands for each application.

$comment

The `$comment` keyword provides a means of inserting a comment in the VCD file.

Syntax

```
$comment

    <comment_text>

$end
```

Examples

```
$comment    This is a single-line comment    $end

$comment    This is a
multiple-line comment
$end
```

$date

The date stamp allows the system to indicate the date on which the VCD file was generated.

Syntax

```
$date
<date_text>
$end
```

Example

```
$date
    June 25, 1989 09:24:35
$end
```

$dumpall

`$dumpall` lists the current values of all the variables dumped.

Syntax

```
$dumpall $end
```

Example

```
$dumpall 1*@ x*# 0*$ bx (k $end
```

$dumpoff

`$dumpoff` lists all the variables dumped with X values and then stops dumping.

Syntax

```
$dumpoff $end
```

Example

```
$dumpoff x*@ x*# x*$ bx (k $end
```

$dumpon

`$dumpon` resumes dumping and list current values of all variables dumped.

Syntax

```
$dumpon $end
```

Example

```
$dumpon x*@ 0*# x*$ b1 (k $end
```

$dumpvars

`$dumpvars` lists the initial values of all the variables dumped.

Syntax

```
$dumpvars <value_changes>* $end
```

Example

```
$dumpvars x*@ z*$ b0 (k $end
```

$enddefinitions

`$enddefinitions` marks the end of the header information and definitions.

Syntax

```
$enddefinitions    $end
```

$scope

Scope definition defines the scope of the dump.

Syntax

```
$scope
<scope_type> <identifier>
$end
```

Definitions

<scope_type> is one of the following keywords:

`module`	for top-level module and module instances
`task`	for a task
`function`	for a function
`begin`	for named sequential blocks

`fork`	for named parallel blocks

Example

```
$scope
   module top
$end
```

$timescale

`$timescale` specifies the timescale that Verilog-XL used for the simulation.

Syntax

```
$timescale <number> <time_dimension> $end
```

Definitions

`<number>` is one of the following:

```
   1   10   100
```

`<time_dimension>` is one of the following:

```
   s   ms    us   ns   ps   fs
```

For more information see Chapter 17, "Timescales."

Example

```
$timescale 10 ns $end
```

$upscope

`$upscope` changes the scope to the next higher level in the design hierarchy.

Syntax

```
$upscope $end
```

$var

`$var` prints the names and identifier codes of the variables being dumped.

Syntax

```
$var
<var_type> <size> <identifier_code>
<reference>
$end
```

Definitions

<var_type> specifies the variable type and can be one of the following keywords:

```
event         integer         parameter       real        reg
supply0       supply1         time            tri         triand
trior         trireg          tri0            tri1        wand
wire           wor
```

<size> is a decimal number that specifies how many bits are in the variable.

<identifier_code> specifies the name of the variable using printable ASCII characters, as described in "Structure of the Dump File".

<*reference*> is a reference name you specify in the source file. More than one <reference> name may be mapped to the same <*identifier_code*>. For example, net10 and net15 may be interconnected in the circuit, and therefore will have the same <*identifier_code*>. A <*reference*> can have the following components:

```
::= <identifier>
||= <identifier> [ <bit_select_index> ]
||= <identifier> [ <MSI> : <LSI> ]
```

- *<identifier>* is the Verilog name of the saved variable, including any leading backslash (\) characters for escape identifiers.
- Verilog-XL dumps each bit of an expanded vector net individually. That is, each bit has its own *<identifier_code>* and is dumped only when it changes, not when other bits in the vector change.
- *<MSI>* indicates the most significant index; *<LSI>* indicates the least significant index.
- *<bit_select_index>*, *<MSI>*, and *<LSI>* are all decimal numbers.

Example

```
$var
integer 32 (2 index
$end
```

$version

`$version` indicates the version of the simulator that was used to produce the VCD file.

Syntax

```
$version
```

```
   <version_text>
$end
```

Example

```
$version
   VERILOG-XL 1.5a
$end
```

Syntax of the VCD File

The following example shows the syntax of the output VCD file:

```
<value_change_dump_definitions>
   := <declaration_command>*<simulation_command>*
<declaration_command>
   ::= <keyword_command>
   (NOTE: Keyword_commands are described in the next section.)
<simulation_command>
   ::= <keyword_command>
   ||= <simulation_time>
   ||= <value_change>
<keyword_command>
   ::= $<keyword> <command_text> $end
<simulation_time>
   ::= #<decimal_number>
<value_change>
   ::= <scalar_value_change>
   ||= <vector_value_change>
<scalar_value_change>
   ::= <value><identifier_code>
      <value> is one of the following: 0 1 x X z Z
<vector_value_change>
   ::= b<binary_number> <identifier_code>
   ::= B<binary_number> <identifier_code>
   ::= r<real_number> <identifier_code>
   ::= R<real_number> <identifier_code>
```

- *<binary_number>* is a number composed of the following characters: 0 1 x X z Z
- *<real_number>* is a real number will be dumped using a `%.16g printf()` format. This format preserves the precision of the number by outputting all 53 bits in the mantissa of a 64-bit C-code 'double'. Applications programs can read a real number using a `%g` format to `scanf()`.
- <identifier_code> is a code from 1 to 4 characters long composed of the printable characters that are in the ASCII character set from `!` to `~` (decimal 33 to 126).

Value Change Dump File Format Example

The following example illustrates the format of the value change dump file. A description of the file format follows the example. With the exception of the `$comment` command, all other keyword commands in this

example are generated by the value change dumper.

```
$date
   June 26, 1998 10:05:41
$end
$version
   VERILOG-XL 2.7
$end

$timescale
   1 ns
$end

$scope module top $end
$scope module m1 $end
$var trireg 1 *@ net1 $end
$var    trireg 1 *# net2 $end
$var    trireg 1 *$ net3 $end
$upscope $end
$scope task t1 $end
$var    reg 32 (k accumulator[31:0] $end
$var    integer 32 {2 index    $end
$upscope    $end
$upscope    $end
$enddefinitions    $end
$comment
   Note:      $dumpvars was executed at time '#500'.
         All initial values are dumped at this time.
$end

#500
$dumpvars x*@ x*# x*$ bx (k bx {2 $end

#505
0*@
1*#
1*$
b10zx1110x11100 (k b1111000101z01x {2
#510
0*$

#520
1*$
#530
0*$
bz (k
#535
$dumpall 0*@ 1*# 0*$
bz (k b1111000101z01x {2    $end
#540
1*$
#1000
$dumpoff x*@ x*# x*$ bx (k bx {2 $end
#2000
$dumpon z*@ 1*# 0*$ b0 (k bx {2 $end
#2010
1*$
```

Note: In general, the VCD does not automatically include comments in the dump file. An exception is when the dump file reaches the limit set by the `$dumplimit` task. Then, the VCD includes a comment to that effect.

The following sections describe the keyword commands, variables, and values that appear in the previous example.

$date. . .$end $version. . .$end $timescale. . .$end

These keyword commands are generated by the VCD to provide information about the dump file and the simulation.

$scope . . . $end

This keyword command indicates the scope of the defined signals that follow. In this example, the scope includes two modules (`top` and `m1`) and one task (`t1`).

$var . . . $end

This keyword command defines each variable that is dumped. It specifies the variable's type (`trireg`, `reg`, `integer`), size (1, 32, 32), identifier code
(`*@, (k, {2`), and reference name as specified by the user in the source file (`net1`, `accumulator`, `index`). To make the dump file machine-independent, compact, and capable of being edited, the VCD assigns each variable in the circuit a 1- to 4-character code called an identifier code. These characters are taken from the visible ASCII characters '!' to '~' (decimal 33 to 126).

$upscope . . . $end

For each `$scope` there is a matching `$upscope` to signify the end of that scope.

$enddefinitions . . . $end

This keyword command indicates the end of the header information and definitions, and marks the start of the value change data.

#500
$dumpvars x*@ x*# x*$ bx (k bx {2 $end

At time 500, `$dumpvars` is executed to show the initial values of all the variables dumped. Identification codes (such as `*@`, `*#`) are used for conciseness and are associated with user reference names in the `$var` ... `$end` sections of the VCD file. In this example, all initial values are X (unknown).

#505
0*@
1*#

1*$
b10zx1110x11100 (k b1111000101z01x {2

This display shows the new values of all the variables that changed at time 505: net1 (which has an identifier code of `*@`) changed to 0, `net2` (identifier code `*#`) and `net3` (identifier code `*$`) changed to 1, the vector `accumulator[31:0]` and the integer `index` changed to the binary values shown.

#510
0*$
#520
1*$
#530
0*$
bz (k

At time 510, only `net3` changed to a 0. All other variables remained unchanged. At time 520, `net3` changed to a 1, and at time 530 it changed back to a 0. Also at time 530, all bits of the vector `accumulator` changed to the high-impedance (Z) state.

#535
$dumpall 0*@ 1*# 0*$
bz (k b1111000101z01x {2 $end

The source file calls a `$dumpall` task at time 535 to dump the latest values of all the specified variables as a checkpoint.

#540
1*$

At time 540, `net3` changed to a 1.

#1000
$dumpoff x*@ x*# x*$ bx (k bx {2 $end

At time 1000, a `$dumpoff` is executed to dump all the variables as X values and to suspend dumping until the next `$dumpon`.

#2000
$dumpon z*@ 1*# 0*$ b0 (k bx {2 $end

Dumping resumes at time 2000; `$dumpon` dumps all the variables with their values at that time.

#2010
1*$

At time 2010, the value of `net3` changes to 1.

Using the $dumpports System Task

The procedures described in this section are deliberately broad and generic. The requirements for your specific design may dictate procedures slightly different from those described here.

$dumpports Syntax

The $dumpports system task scans the (`arg1`) ports of a module instance and monitors the ports for both value and drive level. The $dumpports system task also generates an output file that contains the value, direction, and strength of all the ports of a device. The output file generated by $dumpports is similar to the output file generated by the value change dump (VCD). For information about the VCD file, see "Overview" . The syntax for $dumpports is as follows:

```
$dumpports(<DUT> <,"filename"> <,ID>);
```

The table given below describes the arguments of `$dumpports`.

Arguments	Description
DUT	Device under test (DUT); the name of the module instance to be monitored.

filename	String containing the name of the output file. The filename argument is optional. If you do not specify a filename, then a `verilog.evcd` (the default filename) file will be generated. Consider the following examples:
	
	Example 1:
	
	`$dumpports(dut, , id);`
	
	In this example, the filename argument is not specified. No error will be reported and a `verilog.evcd` file will be generated.
	
	Example 2:
	
	`$dumpports(dut, "testVec.file", id);`
	
	In this example, as the filename argument is specified, a `testVec.file` will be generated.

ID	An integer data type that identifies a running `$dumpports` task with the `$dumpports_close` system task. For more information, see "$dumpports_close" .

Consider the example given below.

. . . .

```
....

module top;
reg A;
integer id;
.........
.........
initial
    begin
      $dumpports(dut, "testVec.file", id);
      #5 $dumpports_close(id);
    end
endmodule

....
....
```

$dumpports Output

The following example shows an output from using `$dumpports`.

```
#100
pDDBF 6566 0066 <1
```

The table given below describes each component of the output.

Component	Description
`#100`	Simulation time
`p`	Output type of the driver. `p` indicates the value, strength, and collision detection for the ports.
`DDBF`	Value of the port. See "Port Value Character Identifiers" for more information.
`6566`	0's strength component of the value. See "Strength Mapping" for more information.
`0066`	1's strength component of the value. See "Strength Mapping" for more information.
`<1`	Signal identifier

Port Names

Port names are recorded in the output file as follows:

- A port that is explicitly named is recorded in the output file.
- If a port is not explicitly named, the name of the object used in the port definition is recorded.

- If the name of a port cannot be determined from the object used int he port definition, the port index number is used (the first port being 0).

Drivers

A driver is anything that can drive a value onto a net including the following:

- primitives
- continuous assigns
- forces
- ports with objects of type other then net, such as the following:

```
module foo(out, ...)
  output out;
  reg out;
```

If a net is forced, a comment is placed into the output file stating that the net connected to the port is being forced, and the scope of the force is given. Forces are treated differently because the existence of a force is not permanent,even though a force is a driver.

While a force is active, driver collisions are ignored and the level part of the output is determined by the scope of the force definition.When the force is released, a note is again placed into the log file.

Port Value Character Identifiers

The following table shows the characters (middle column) that identify the value of a port in the `$dumpports` output:

Level	Char.	Value
DUT	L	(0) low
DUT	l	(0) low with more than 2 active drivers
DUT	H	(1) high
DUT	h	(1) high with more than 2 active drivers
DUT	T	(Z) tri-state
DUT	X	(X) unknown
DUT	x	(X) unknown because of a 1-0 collision
unknown	?	(X) unknown state
unknown	0	(0) unknown direction; both the input and the output are active with

		the same value.
unknown	1	(1) unknown direction; both the input and the output are active with the same value.
unknown	A	(0-1) Input is a 0 and output is a 1.
unknown	a	(0-X) Input is a 0 and output is an X
unknown	B	(1-0) Input is a 1 and output is a 0.
unknown	b	(1-X) Input is a 1 and output is an X.
unknown	C	(X-0) Input is a X and output is a 0.
unknown	c	(X-1) Input is a X and output is a 1.
unknown	F	(Z) tri stated, nothing is driving the net.
unknown	f	(Z) tri stated, both internal and external.
test fixture	D	(0) low
test fixture	d	(0) low with more than 2 active drivers
test fixture	U	(1) high
test fixture	u	(1) high with more than 2 active drivers
test fixture	N	(X) unknown
test fixture	n	(X) unknown because of a 1-0 collision
test fixture	Z	(Z) tri-state

The level of a driver is determined by the scope of the driver's placement on a net. Port type has no influence on the level of a signal. Any driver whose definition is outside of the scope of the DUT is at the test fixture level.

In the following example, because the driver is outside the scope of a DUT, the continuous assignment is at the test fixture level:

```
module top;
   reg regA;
   assign dut1.out = regA;
   dut dut1(out, ....);
   initial
      $dumpports(dut1, "testVec.file");
   ...
endmodule
module dut(out, ...
   output out;
   wire out;
   ...
endmodule
```

Strength Mapping

Strength values in the `$dumpports` output are as follows:

```
0 HiZ
1 Sm Cap
2 Md Cap
3 Weak
4 Lg Cap
5 Pull
6 Strong
7 Supply
```

$dumpports Restrictions

The following restrictions apply to the `$dumpports` system task:

- The `$dumpports` system task does not work with the `$save` and `$restart` system tasks.
- Continuous assignments cannot have delays.
- The following wire types are the only ones permitted to be connected to the ports:

wire, tri, tri0, tri1, reg, trireg

- Directional information can be lost when a port is driven by one or more drivers of the different tran elements (`tran`, `rtran`, `rtranif0`, ...).

  In the following example, the direction of the port `out` cannot be determined because the value that the `tran` gate is transporting from its other terminal is unknown.

```
bufif1 u1(out, 1'b1, 1'b1);
DUT udut(out);
...

module DUT(out)
   tran t1(out, int);
```

$dumpports_close

The `$dumpports_close` system task stops a running `$dumpports` system task. The syntax is as follows:

```
$dumpports_close(<ID>);
```

The optional *<ID>* argument is an integer that identifies a particular `$dumpports` system task. If only one `$dumpports` system task is running, the *<ID>* can be left blank. Valid *<ID>* values are specified in the syntax of the `$dumpports` system task.

Consider the example given below.

```
....
```

```
....

module top;
reg A;
integer id;
.........
.........
initial
    begin
      $dumpports(dut, "testVec.file", id);
      #5 $dumpports_close(id);
    end
endmodule

....
....
```

Return to top of page

Library Contents Index < Previous Next > View/Print PDF Feedback Help Exit

For support, see Cadence Online Support service.