ESP Topics

Version O-2018.06, June 2018

SYNOPSYS®

Contents

ADDING WILDCARDS TO A COMMAND 6
Clocks 15
Coding for X 18
Collections and Querying 24
Commands Listed by Functional Group 29
Coverage Topics 31
Code Coverage - Filtering Coverage Data 32
Code Coverage - Merging Coverage Data 40
Code Coverage - Overview 42
Code Coverage - Reporting Coverage Data 45
Code Coverage - System Tasks 54
Code Coverage - Usage 55
Custom Compiler 60
Galaxy Custom Designer (CD) 63
Device Model Simulation 64
Distributed Processing 67
Discovery Viewing Environment (DVE) 71
$esp_addrvar 73
$esp_choose_var 74
$esp_cmdvar 76
$esp_const 77
$esp_const_one 79
$esp_const_zero 81
$esp_contains 82
$esp_context 84
$esp_ctrlvar 85
$esp_datavar 86
$esp_drive_zero_delay 87
$esp_equation 88
$esp_equation_size 90
$esp_equation_symbols 91
$esp_error 92
$esp_exclcovscope 93
$esp_exclusive 95
$esp_gen 96
$esp_multi_select 98
$esp_onehotindex 100
$esp_retire 102
ESP Shell Flow 103

$esp_smminit . . . 116
$esp_timevar . . . 118
$espvar . . . 119
$espcovfile . . . 121
$espcovoff . . . 123
$espcovoff/$espcovon . . . 125
$espcovscope . . . 127
exit . . . 130
Frequently asked questions: SPICE . . . 132
Frequently Asked Questions: SystemVerilog . . . 135
Code Coverage - Frequently Asked Questions . . . 146
GUI Usage Tips . . . 147
UNIX Printing . . . 148
FAQ - Regular Expressions . . . 149
FAQ Supported SPICE Device Models . . . 151
FAQ Top . . . 152
Efficient Coding Practices . . . 153
Unsupported Verilog Constructs . . . 162
Fast Signal Data Base (FSDB) . . . 168
help . . . 174
Interactive Signal Tracing . . . 176
Cell Library Verification . . . 177
Limitations . . . 188
Logical to SPICE mapping . . . 192
man . . . 195
Net Attributes . . . 197
Output Checkers . . . 199
Pattern File . . . 200
Perl Regular Expression Syntax . . . 205
Pin Functions . . . 216
Power Integrity Verification . . . 218
quit . . . 220
Redundancy Verification . . . 222
Scan Chain Verification . . . 224
source . . . 227
SPICE to SPICE . . . 229
Startup Command Files . . . 231
ESP Specific System Tasks . . . 233
Testbench Customization . . . 235
Timescale . . . 244
Transistor-only Simulation . . . 245
Using Online Help . . . 247
Variables Listed by Functional Group . . . 250
Variables . . . 252
Value Change Dump (VCD) . . . 254

VCS Options 255
Verilog to Verilog 278

ADDING WILDCARDS TO A COMMAND

Note:

This application note shows how to modify the **set_supply_net_pattern** command to add wildcard support. Starting with the 2016.12 release of the ESP tool the **set_supply_net_pattern** is not available. This command has been replaced by the **set_supply_net** command which supports the use of regular expressions. The techniques shown in this application note are still relevant as most commands in the ESP tool do not support regular expressions or wildcards.

How to Add Wildcards to a Command

This application note will show how to create a Tcl command that extends an existing command to handle a wildcard pattern. The example will be the **set_supply_net_pattern** command.

Background

The **set_supply_net_pattern** command defines a net name that is a supply net. The command specifies the logical value (0 or 1), the type (virtual or real) and the name of the net. Anywhere in the design hierarchy where a net of the specified name appears it will be treated as a supply of the stated logic value and type. Many users think of *pattern* as a wildcard (using * and ?) or as a full regular expression (using () [] +? - ...). The *pattern* in supply net pattern is the full net name that is searched for throughout the design hierachy.

The rest of this application note will show how to create a true pattern version of **set_supply_net_pattern** .

Objective

Write a version of the **set_supply_net_pattern** that accepts regular expressions or patterns for the name of the supply net.

This new command will be called **set_supply_net_by_pattern**.

The switch *-regexp* will be used to specify when a pattern is a regular expression or a wildcard pattern. Wildcard patterns will use ***** to represent one or more characters and **?** to represent any single character.

When the switch *-regexp* is present then the net name shall be interpreted as a full regular expression. Net names that match the expression will be operated upon.

A net name without *-regexp* and with any ***** or **?** characters will be interpreted the same as if **set_supply_net_pattern** had been used.

All other options (*-type*, *-logic*, *-i* and *-r*) to **set_supply_net_pattern** will be accepted and operate the same.

Working With Collections

In this section we will explore the use of collections within ESP shell.

Synopsys applications that support Tcl typically extend the Tcl shell to support a data object called a collection. A collection is a group of data objects that have been made visible to the Tcl shell.

Commands that return a collection look as if they return a normal Tcl list. In fact, these commands return a collection and display a list of (up to 100) objects that define the collection that is returned. The length of this displayed list is controlled by the variable **collection_result_display_limit** . This distinction is important. Special commands are needed to manipulate the data held within a collection.

The **foreach_in_collection** command is used to iterate through each object in a collection. Do not use the Tcl **foreach** command to iterate over a collection. The Tcl **foreach** command will delete the collection. Also, collections are not Tcl lists.

For example, the following code displays the names of all the instances within the current instance.

```
> foreach_in_collection ci [get_instances -i] {puts [get_attribute $ci name]}
 XI1
 XI2
 XI4
 XI5
 XI6
 XDI0
 XDI1
 XDI2
 XDI3
 XDI4
 XDI5
 XDI6
 XDI7
 XOE
 XCLRFF
 XCLKD
 XCLRIIFF
 XCLRINV
```

To operate on only part of a collection, you can iterate over all of the collection with **foreach_in_collection** or you can use the **filter_collection** command and then use **foreach_in_collection** . The **filter_collection** command creates a new collection without modifying the original collection. The **filter_collection** command can use complex expressions, patterns or even regular expressions for filters.

The **copy_collection** command can be used to create a copy of a collection. For example:

```
set nets [get_nets -i]   # create a collection of nets
set nets_ref $nets       # create a reference to nets
set nets_copy [copy_collection $nets] # create a copy of nets
```

Note:
In the above example the variable *nets_ref* is just a reference to the collection of nets referenced by *nets*. The actual collection of nets will be destroyed only after *nets* and *nets_ref* no longer reference the collection or they are both destroyed. Either *nets* or *nets_ref* can be used to access or change the collection that is referenced.

Note:
The collection of nets referenced by *nets_copy* is a unique collection. Changes to *nets_copy* do not affect *nets* or *nets_ref*.

The **copy_collection** command can also create an empty collection. For example:

```
copy_collection ""  # creates an empty collection
```

The **add_to_collection** command will add objects to an existing collection. With the *-unique* switch duplicate objects will not be added to the collection. For example:

```
set ports [get_inputs -r]  # create a collection of inputs
set ports [add_to_collection [get_outputs -r]]  # add outputs
```

Given a collection, **foreach_in_collection** and **add_to_collection** could be used to filter a collection to keep only the objects that meet a specific selection criteria. Since this is a common operation for collections, the **filter_collection** command was created to be highly efficient at filtering a collection.

Here is a Tcl procedure to print out names of the objects in a collection.

Note:
It does not provide a line wrap capability. The output is a single line.

```
proc putCollection {cobj} {
  #
  # Print a collection
  #   actually just names of the objects
  #
  set names {}

  foreach_in_collection obj $cobj {
    lappend names [get_attribute $obj name]
  }
  puts $names
}
```

The ESP shell has a set of commands (get*) that return collections of specific data objects. The three commands needed to write the **set_supply_net_by_pattern** command are:

get_nets	Returns a collection of nets for the current instance
get_instances	Returns a collection of instances that are instantiated by the current instance.
get_designs	Returns a collection of designs for current container.

The other ESP shell commands that will be needed are:

current_instance	sets and returns the current instance.To return the name of the current instance without setting the current instance use **current_instance** .
set_supply_net_pattern	sets the supply information for nets that match the net name pattern. This is the command that is being enhanced.

Solution

Now that the objective has been set and all the background information needed has been identified we can proceed to produce a solution.

There are many possible ways to enhance **set_supply_net_pattern** as specified by the objective. The approach shown here will:

- Interate over the entire design tree using **foreach_in_collection** and **get_instances** .
- Filter the collection of nets at each level with **filter_collection** . Only unique net names will be added to the list of net names.
- Call **set_supply_net_pattern** with the explicit net names determined by the previous steps.

Traversing the Design

The following code will traverse the entire design hierarchy.

```
proc isPathTop {path} {
  return [expr [string equal {$path} {.}] || [string equal {$path} {}]]
}

proc traverseDesign { {container -i} {start_instance .} {count 0} } {


  if { ![isPathTop $start_instance] } {
    if {[catch {current_instance $container $start_instance}]} {
      current_instance $container ..

      return
    }
  } else {
   return
  }

  puts stderr "Devices:"
  putCollection [get_devices $container]

  puts stderr "Instances:"
  set myInstances [get_instances $container]
  putCollection $myInstances

  foreach_in_collection instanceObj $myInstances {
    set instanceName [get_attribute $instanceObj name]
    puts stderr "\tInstance $instanceName"
```

```
      traverseDesign $container $instanceName [expr $count + 1]
      puts stderr "\t\t End $start_instance"

    }

    puts stderr  "========================================================"
    puts stderr  "Foo: |$start_instance|"

    if { ! [isPathTop $start_instance] } {
      current_instance $container ..
    }


  }
```

Note

the use of **suppress_message** and **unsuppress_message** commands. These commands are used to remove some messages about get* and current* calls that issue information messages that clutter the output.

There are some issues with this code for our application.

1. This traverses the entire design hierarchy and could be slow.
2. The final **current_instance** call to return to the top of the design is not really needed. It is hard to stop it from happening.

Therefore, a different approach is needed for an elegant solution to the stated problem. Instead of trying to trace the design hierarchy, we will simply trace the nets in each of the SPICE subcircuits. For this we can use **get_designs** to get all the SPICE subcircuit names.

So we create a procedure that will return all of the nets in the implementation. This version has the issue that it can return net names that are not in the actual hierarchy of the design. This happens if the SPICE design file has subcircuits that never appear in the actual design.

Getting all the nets in all the designs in a container is generally faster than getting all the nets in the design hieararchy. This is because each design (module/subcircuit) could appear multiple times in the design hierarchy.

```
  proc getDesignNets {{container "-i"}} {
    #
    # Return a collection of all nets in container
    #
    # default to implementation container
    #

    set oldDesign [current_design $container]

    set allNets [copy_collection ""]

    foreach_in_collection design [get_designs $container] {
       current_design $container [get_attribute $design name];
       foreach_in_collection net [get_nets $container] {
         set allNets [add_to_collection $allNets $net -unique]
       }
    }

    return [copy_collection $allNets]
 }
```

Now we write a Tcl procedure that will allow us to filter all the nets in the container.

```
proc getDesignNetsByPattern {pattern {container {-i}} {switches {}}} {
   #
   # Return collection of all design nets
   #
   set allNets [getDesignNets $container]

   if { [string equal $switches ""] } {
     return [filter_collection $allNets "name=~$pattern"]
   } else {
     return [filter_collection $allNets "name=~$pattern" $switches]
   }
}
```

The last step is to create the actual command stated in the objective section. We will take one short cut. Our **set_supply_net_by_pattern** command will not fully parse the command line. We will require the syntax to be:

```
 set_supply_net_by_pattern [options] pattern
```

Where options can be any of the standard options to **set_supply_net_pattern** and also can include the switch *-regexp*.

Our solution is here:

```
proc set_supply_net_by_pattern {args} {
  #  We falsely assume and do not check
  #   that the last argument is the pattern
  #

  set pattern [lindex $args end]
  set args [lreplace $args end end]

  #
  # -regexp ?
  #
  if {[set regexp [lsearch $args -regexp]] > -1} {
    set args [lreplace $args $regexp $regexp]
    set switch {-regexp}
  } else {
    set switch {}
  }

  #
  # pull container switch
  #
  if {[set idx [lsearch -regexp $args {-[ir]} ]] > -1} {
    set container [lindex $args $idx]
    set args [lreplace $args $idx $idx]
  } else {
    set container [current_container]
  }

  if { [string match {*[*?.+^-]*} $pattern]} {
    #
    # Highly likely there is a pattern here
    #

    foreach_in_collection netObj [getDesignNetsByPattern $pattern $container $switch] {
       set nets([get_attribute $netObj name]) 0
    }
```

```
        foreach {key value} [array get nets] {
          puts "Set $key"
          eval set_supply_net_pattern $container $args $key
        }

      } else {
        #
        # Use unaltered call to set_supply_net_pattern
        #
        eval set_supply_net_pattern $container $args $pattern
      }
    }
```

Note

the use of **eval** to create the command to execute. This is needed since *args* is a list of command arguments. The **eval** command invokes the Tcl interpreter to process the command string and flattens *args* into separate arguments. Without **eval** , *$args* is treated as one argument instead of two or more arguments.

Putting It All Together

Now that we have shown all the pieces to the solution, how do you use it ?

1. Copy all the code at the end of this application note to a file. We will assume you called the file *extension.tcl*. You could use any file name. We will also assume that the file is in the current working directory.
2. Start **esp_shell** on your design, load the Verilog and SPICE files, complete matching.
3. This step can be done anytime after **esp_shell** is started but before the next step. Load the Tcl script.`source extension.tcl`
4. Use the new command to specify the supply nets. In our example the supply net is DVDD.

```
> set_supply_net_by_pattern -logic 1 -type real *VDD*
Set DVDD

    OR

> set_supply_net_by_pattern -logic 1 -type real -regexp .*VDD.*
Set DVDD
```

   The two commands shown above do the exact same thing. They set any net in the implementation container meeting the pattern to be a real supply of logic value 1 (Verilog supply1).

   Both commands report the name of the nets that match the pattern:

```
"Set DVDD"
```

5. Check the results

```
report_supply_net_pattern -i

*********************
Report : report_supply_net_patterns
Design : ram1r1w
Version: Z-2006.12
Date   : Thu Jan 18 10:05:23 2007
*********************

     Net_Name        type     Logic Value      Voltage
     DVDD    0       1           0.00
```

Exercises

Now try your hand at the following exercises.

Exercise 1 - Add the *-nocase* switch

The filter_collection command has a *-nocase* switch that requires the use of *-regexp*. The *-nocase* switch makes matching case insensitive.

Exercise 2 - Fully parse the command line

Extend the implementation to remove the requirement that the pattern be the last argument.

Exercise 3 - Is the pattern check complete ?

Is **[string match {*[*?.+^-]*} $pattern]** sufficient to detect all legal patterns ?

Exercise 4 - Print the design hierarchy

The current output of **traverseDesign** is not very useful. Modify **traverseDesign** to display the design hierarchy with appropriate indentation.

Code

```
proc getDesignNets {{container "-i"}} {
   #
   # Return a collection of all nets in container
   #
   # default to implementation container
   #

   set oldDesign [current_design $container]

   set allNets [copy_collection ""]

   foreach_in_collection design [get_designs $container] {
     current_design $container [get_attribute $design name];
     foreach_in_collection net [get_nets $container] {
       set allNets [add_to_collection $allNets $net -unique]
     }
   }

   return [copy_collection $allNets]
 }
```

```
proc getDesignNetsByPattern {pattern {container {-i}} {switches {}}} {
   #
   # Return collection of all design nets
   #
   set allNets [getDesignNets $container]

   if { [string equal $switches ""] } {
     return [filter_collection $allNets "name=~$pattern"]
   } else {
     return [filter_collection $allNets "name=~$pattern" $switches]
   }
}

proc set_supply_net_by_pattern {args} {
  #  We falsely assume and do not check
  #   that the last argument is the pattern
  #

  set pattern [lindex $args end]
  set args [lreplace $args end end]

  #
  # -regexp ?
  #
  if {[set regexp [lsearch $args -regexp]] > -1} {
    set args [lreplace $args $regexp $regexp]
    set switch {-regexp}
  } else {
    set switch {}
  }

  #
  # pull container switch
  #
  if {[set idx [lsearch -regexp $args {-[ir]} ]] > -1} {
    set container [lindex $args $idx]
    set args [lreplace $args $idx $idx]
  } else {
    set container [current_container]
  }

  if { [string match {*[*?.+^-]*} $pattern]} {
    #
    # Highly likely there is a pattern here
    #

    foreach_in_collection netObj           [getDesignNetsByPattern $pattern $container $switch] {
       set nets([get_attribute $netObj name]) 0
    }

    foreach {key value} [array get nets] {
      puts "Set $key"
      eval set_supply_net_pattern $container $args $key
    }

  } else {
    #
    # Use unaltered call to set_supply_net_pattern
    #
    eval set_supply_net_pattern $container $args $pattern
  }
}
```

Clocks

The ESP tool creates testbenches that have defined test cycles. The binary, symoblic and flush cycles are defined by one test clock period. The init cycle consists of multiple test clocks.

The test clock is the first clock created by the create_clock command or the -function clock option of the set_testbench_pin_attributes command. If there is no clock defined, the ESP tool uses the clock name inno_clock.

Clocks are defined with several attributes. These attributes have default values. The attributes are:

Attribute	Default	Description
name	inno_clock	name of the clock
period	40[1]	Clock period in nanoseconds
setup	5	Setup time in nanoseconds. The minumum time before the clock edge where inputs must be stable
delay[2]	0	The delay (in nanoseconds) between the primary clock and this clock
phase[2]	1/2 period	The width (in nanoseconds) of the first clock phase. The second clock phase will be the period less the clock phase
initial	1	initial value of the clock. Clocks start at the active level.
clktype[3]	primary[4]	Clock type (primary, secondary or buffered).
constraint[3]	none	

1. For the library style testbench the default period is 50ns
2. delay and phase ignored for all testbenches except for the clock wave (*.mpt) testbench
3. clktype and constraint are ignored for all testbenches except for the library cell (*.ltb) testbench
4. Only the library cell (*.ltb) testbench has default of primary. For all other testbenches the clktype is ignored.

Testbench styles

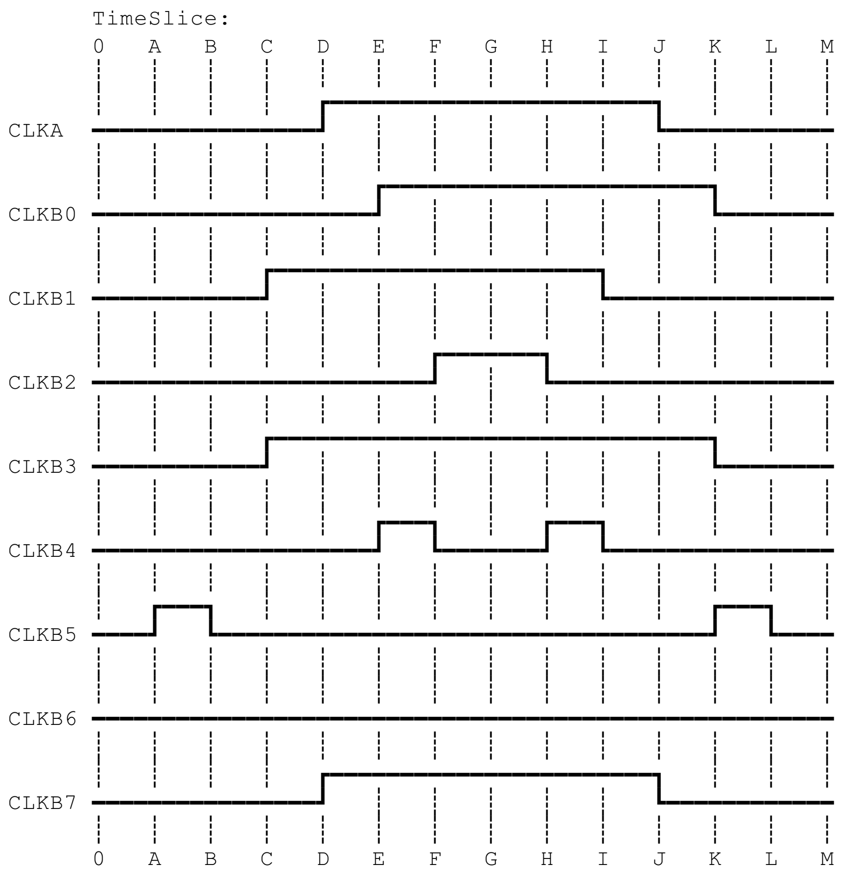
Clock Wave

The clock wave test is the only test that uses the delay and phase attributes of a clock definition. To create a clock wave test you should assign device pins to port groups. Each port group should have a clock. Pins not assigned to a port group will be in the port group of the first clock defined.

Each clock in the clock wave is independent of all the other clocks. Inputs in a port group are changed at the setup time for the first clock defined in a port group.

Clock Enum

The clock enum test will explore the clock edge relationships. The user specified clock times are ignored. The first clock defined will be "CLKA". The second clock defined will be "CLKB". The clock waveforms look like:



If there are more than 2 clocks defined,

- A warning message will be issued
- Additional clocks will match the second clock.

Because of the way that clock enum creates the second and subsequent clocks, they will have symbolic net coverage. The first clock will be binary. If you do not run the clock enum testbench, then all clocks will be binary and will appear as non-symbolic nets in the coverage report.

Library Cell

The library cell testbench uses the clktype and constraint attributes of the clock definition. The clock is held at the inactive state when the clock constraint condition is true.

SEE ALSO

Commands: **create_clock**,

```
          report_clock,
          set_testbench_pin_attributes
Variables: testbench_style
Topics:   ,

          set_testbench_pin_attributes
Variables: testbench_style
Topics:   ,
```

Coding for X

Overview

Both the Verilog and SPICE formats handle X values differently. Verilog code handles an X or Z value depending upon the context. In SPICE code, an X value does not exist.

The ESP tool handles X and Z values for Verilog nets in accordance with the Verilog Language Reference Manual. The ESP tool handles X and Z values for SPICE nets similar to how Verilog handles X and Z for integral expressions with the conditional operator. This topic explores these differences.

Throughout this topic the term **LRM** refers to the IEEE Standad 1800-2012 Standard for SystemVerilog and the SystemVerilog unsized X constant 'X' is used to represent a result in which all bits are X.

Guidelines

This section discusses the recommended coding guidelines for Verilog. These guidelines pertain to writing Verilog code that is more predictable in handling X and Z values. These guidelines are intended to better match the SPICE behavior under the same X or Z input conditions.

Following these guidelines does not prevent all differences in X and Z handling in the Verilog and SPICE design representations. The rationale for these guidelines are explained in later sections which describe how to handle X and Z values in both Verilog and SPICE designs.

Use of flags

Use the 2-state data types for all flag values. Use the `bit` type for single-bit flag values. If you use the `reg` type, then be explicit in comparison.

Table 9-1

Use	Do Not Use
`if (flag===1'b1)` or `if (flag!===1'b0)`	`if flag` or `if flag==1'b1`
`if (flag===1'b0)` or `if (flag!===1'b1)`	`if !flag` or `if ~flag` or `if flag==1'b0`

Array Access

This feature allows you to check the index on all write operations to an array. If the index contains any X or Z bits, write 'X' into the contents of the entire array. This is closer to the write operation of array write in the SPICE netlist. This corruption is pessimistic but it is faster than exactly matching the SPICE behavior which could be extremely complex depending upon the actual architecture used in the design.

If the write enable decodes into the word line in the SPICE design, and the write enable bit is X or Z, then write 'X' into the memory location on a write operation.

If-then-else Statements

Prepare all the `if-then-else` statements such that the "exception" case is in the `else` clause. The exception case often writes X values to reg or net variables. By coding the exception case as the `else` clause, if the condition part of the `if` statement evaluates to X or Z, the code evaluates the `else` clause of the `if` which is most likely the result you want.

Case Statements

Use the `default:` clause to assign X values to reg or net variables.

Note:
Usage of the `default:` clause to assign 'X' is not compatible with synthesis or logic equivalence checking guidelines.

Use the `case` statement instead of the `if then else` statement when you require separate actions based upon a logical expression returning 0, 1 X, or Z values.

"===" Versus "=="

You can use "`===`" or "`!==`" when checking for equivalence or non-equivalance. Using "`==`" or "`!=`" for comparison can result in 0,1, or X. If any bit of either of these operations is X or Z, these comparisions return the value X.

Loops

A value of X for the looping condition is interpreted as false which causes the loop to exit. Code to avoid this condition.

Testing for X or Z Bits

Use the reduction exclusive or operator to check if any bits of an expression are X or Z. To check whether any bits of address and data are X or Z, use the following example::

```
if (^{address,data}===1'bX) ...
```

X and Z in Verilog

This section describes how various aspects of the Verilog language handles and creates X and Z values.

Edge Transitions

Table 9-2 in the LRM defines postive-edge and negative-edge transitions.

Table 9-2

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

To suppress X and Z transitions from the Verilog code, first monitor the past value of a net. Then check whether both the previous value and the current value are not X or Z.

```
  // Example to check for falling edge of RESET
  reg RESET_prev;
  initial begin
    RESET_prev = RESET;
  end
  always @(RESET) begin
    case(RESET_prev)
     1'b1: begin
       // 1->?
       if (RESET===1'b0) begin
         // code for falling edge of RESET
         //  ignores 1->X and 1->Z
       end
       end
     1'b0: begin
       // 0->?
     1'bX: begin
       // X->?
       end
     1'bZ: begin
       // Z->?
       end
    endcase
    // now that all RESET processing done
    //  save as previous value
    RESET_prev = RESET;
 end
```

Vector and Array Indexing

When assigning to a vector or array with an index value that contains any X or Z bits, Verilog ignores the assignment. All the expressions in the statement execute except for the actual assignment.

When reading from a vector or array with an index value that contains any X or Z bits, the results are determined based on the vector or array type. LRM clause 7.4.6 and Table 7-1 define the results when reading from a nonexistent array entry.

LRM clause 7.8.6 establishes that the default value is returned for a nonexistent index of an associative array when a default value has been defined. If there is no default value, then reading from a nonexistent index of an associative array follows the rules shown in Table 7-2.

Table 9-3

Type of Array	Value Read

4-state integral type	'X'
2-state integral type	'0'
enumeration	Value specified in this table for the enumerations base type
real,shortreal	0.0
string	"" (empty string)
class, event, chanlde	null
virtual interface	null
Variable-size unpacked array (dynamic, queue, associative)	Array of size zero (no elements)
Fixed-size upacked array	Array, all of whose elements have the value specified in this table for that array's element type
Unpacked struct	struct, each of whose members has the value specified in this table for that member's type, unless the member has an initial assignment as part of its declaration, in whic case the member's value shall be as given by its initial assignment
Unpacked union	Value specifed in this table for the type of the first member of the union

Operators

Assignment to a 2-state type converts any X or Z bits to 0. Assignment using arithmetic assignment operators first completes the arithmetic operation as per the rules for the operator. Then, if the assignment is to a 2-state type, the tool converts the X or Z bits to 0.

Arithmetic Assignment Operators

+=	-=	*=	/=
|=	&=	^=	%=
<<=	>>=	<<<=	>>>=

Arithmetic Operators

Operator	Description	X or Z Handling
++ --	increment/decrement	If any bit is X or Z, the result is 'X
+ -	Unary positive/negative	If any bit is X or Z, the result is 'X
+ - *	addition, subtraction, multiplication	If any bit in either operand is X or Z, the result is 'X
<< >> <<< >>>	logical left shift, logic right shift, arithmetic left shift, arithmetic right shift	If any bit in the right side operand is X or Z, the result is 'X
/ %	division remainder	If second operand is 0, the result is 'X. If any bit in either operand is X or Z, the result is 'X
**	power	If neither operand is real or short real, and the first operand is 0 and the second operand is negative, the result is 'X. If any bit of either operand is X or Z, the result is 'X

Relational Operators

Operator	Description	X or Z Handling
< <= > >=	less than, less than or equal, greater than,	If any bit in either operand is X or Z, the result is 1'bX

	greater than or equal	
===	case equality	If all bits in each operand match, even X or Z bits result is 1'b1 else result is 1'b0
!==	case inequality	If all bits in each operand match, even X or Z bits result is 1'b0 else result is 1'b1
==	logical equality	If any bit of either operand is X or Z, the result is 1'bX
!=	logical inequality	If any bit of either operand is X or Z, the result is 1'bX
==? !=?	wildcard equality and wildcard inequality **Not supported in ESP**	Similar to case equality and case inequality, except that X or Z bits on the right side operand are treated as wildcards and match 0,1,X and Z bit in the left side operand in the same position

Logical Operators

Operator	Description	X or Z Handling
&&	short circuit and	First operand is always evaluated. If the first operand is logically false, then the second operand is not evaluated and the result is 1'b0.
||	short circuit or	First operand is always evaluated. If the first operand is logically true, then the second operand is not evaluated and the result is 1'b1.
!	logical negation	
->	logical implication	logically equivalent to **(!expression1||expression2)** except that expression1 and expression2 are each evaluated just one time, there is no short circuit operation
<->	logical equivalence	logically equivalent to **((expression1->expression2))&&((expression2->expression1))** except that expression1 and expression2 are each evaluated just one time, there is no short circuit operation

Conditional Operator

```
conditionExpression ? trueExpression : falseExpression
```

If *conditionExpression* is true, the return value is the *trueExpression* without evaluating the *falseExpression*. If *conditionExpression* is false, then the return value is the *falseExpression* without evaluating the *trueExpression*.

Gate Primitives

Gate Primitives

Gate	Description	X or Z handling
and nand nor or xor xnor		X unless non-X input can determine result Can have more than 2 inputs. Can have 0,1 or 2 delays. Output transition to X will use the smallest of specified delays

and	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

nand	0	1	X	Z
0	1	1	1	1
1	1	0	X	X
X	1	X	X	X
Z	1	X	X	X

or	0	1	X	Z
0	0	1	X	X
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X

nor	0	1	X	Z
0	1	0	X	X
1	0	0	0	0
X	X	0	X	X
Z	X	0	X	X

xor	0	1	X	Z
0	0	1	X	X
1	1	0	X	X
X	X	X	X	X
Z	X	X	X	X

xnor	0	1	X	Z
0	1	0	X	X
1	0	1	X	X
X	X	X	X	X
Z	X	X	X	X

buf not

buf	
Input	Output
0	0
1	1
X	x
Z	X

not	
Input	Output
0	1
1	0
X	X
Z	X

Can have 0,1 or 2 delays. Output transition to X will use the smallest of specified delays

X and Z in SPICE

This section describes how X and Z values are handled and created by various aspects of SPICE netlists as evaluated in the ESP tool.

Net Initial Value

The initial value of a SPICE net is determined by the ESP tool at the beginning of simulation. This value is controlled by the set_initialization and the set_net_initial_value commands. The set_initialization command controls how SPICE net initial values are determined globally. The set_net_initial_value command is used to override the global net initial values.

The default global net initialization is "DC". This mode is similar to SPICE simulator DC operating point calculations. Most nets will initialize to logic 0. Some nets will initialize to 1. If the supply defnitions are missing then most nets will be undriven and have a logic value of Z. This initialization mode is different then the Verilog simulator initialization of X for all nets.

Contention Conditions

When a net in a SPICE design has multiple drivers, the ESP tool resolves the fight by picking the smaller of the pull-up and pull-down resistances. If the resistances are close, the tool sets the net to the X value.

The set_drive_fight_x_range command defines the threshold which defines the X range. The default range is 0.499 to 0.501. This number is the ratio of the pull-down resistance to the total resistance as expressed in the following equation:

Floating Nets

When a net in a SPICE design has no drivers, the ESP tool sets the net value to Z. The value is set after the global RC decay time specified by the set_rcdecay_time command. The default decay time is 5 ns.

Collections and Querying

This section describes the methodology for creating collections of objects and querying objects in the database.

DESCRIPTION

Synopsys applications build an internal database of the netlist and the attributes applied to it. This database consists of several classes of objects, including designs, libraries, ports, cells, nets, pins, and clocks. Most commands operate on these objects.

By definition:

A collection is a group of objects exported to the Tcl user interface.

Collections have an internal representation (the objects) and, sometimes, a string representation. The string representation is generally used only for error messages.

A set of commands to create and manipulate collections is provided as an integral part of the user interface. The collection commands encompass two categories: those that create collections of objects for use by another command, and one that queries objects for viewing. The result of a command that creates a collection is a Tcl object that can be passed along to another command. For a query command, although the visible output looks like a list of objects (a list of object names is displayed), the result is an empty string.

An empty string "" is equivalent to the empty collection, that is, a collection with zero elements.

Homogeneous and Heterogeneous Collections

A homogeneous collection contains only one type of object. A heterogeneous collection can contain more than one type of object. Commands that accept collections as arguments can accept either type of collection.

Lifetime of a Collection

Collections are active only as long as they are referenced. Typically, a collection is referenced when a variable is set to the result of a command that creates it or when it is passed as an argument to a command or a procedure. For example, if in ESP you save a collection of ports:

```
> set ports [get_ports ]
```

then either of the following two commands deletes the collection referenced by the ports variable:

```
> unset ports
> set ports "value"
```

Collections can be implicitly deleted when they go out of scope. Collections go out of scope when the parent (or other antecedent) of the objects within the collection is deleted. For example, if our collection of ports is owned by a design, it is implicitly deleted when the design that owns the ports is deleted. When a collection is implicitly deleted, the variable that referenced the collection still holds a string representation of the collection. However, this value is useless because the collection is gone, as illustrated in the following ESP example:

```
> current_design
{"TOP"}
> set ports [get_ports in*]
{"in0", "in1"}
> remove_design TOP
Removing design 'TOP'...
> query_objects $ports
Error: No such collection '_sel26' (SEL-001)
```

Iteration

To iterate over the objects in a collection, use the **foreach_in_collection** command. You cannot use the Tcl- supplied foreach iterator to iterate over the objects in a collection, because foreach requires a list; and a collection is not a list. In fact, if you use foreach on a collection, it will destroy the collection.

The arguments to **foreach_in_collection** are similar to those of foreach: an iterator variable, the collections over which to iterate, and the script to apply at each iteration. Note that unlike foreach, the **foreach_in_collection** command does not accept a list of iterator variables.

The following example is an iterative way to perform a query. For details, see the **foreach_in_collection** man page or the user guide.

```
>   foreach_in_collection s1 $collection {
    echo [get_object_name $s1]
   }
```

Manipulating Collections

A variety of commands are provided to manipulate collections.

o **add_to_collection** - Takes a base collection and a list of element names or collections that you want to add to it. The base collection can be the empty collection. The result is a new collection. In addition, the **add_to_collection** command allows you to remove duplicate objects from the collection by using the -unique option.

o **remove_from_collection** - Takes a base collection and a list of element names or collections that you want to remove from it. For example, in PrimeTime:

```
> set dports     [remove_from_collection [get_inputs] CLK]
{"in1", "in2", "in3"}
```

o **compare_collections** - Verifies that two collections contain the same objects (optionally, in the same order). The result is "0" on success.

o **copy_collection** - Creates a new collection containing the same objects (in the same order) as a given collection. Not all collections can be copied.

o **index_collection** - Extracts a single object from a collection and creates a new collection containing that object. Not all collections can be indexed.

o **sizeof_collection** - Returns the number of objects in a collection.

Filtering

You can filter any collection by using the **filter_collection** command. It takes a base collection and creates a new collection that includes only those objects that match an expression.

Many of the commands that create collections support a -filter option, which allows objects to be filtered out before they are ever included in the collection. Frequently this is more efficient than filtering after the they are included in the collection. The following example prints out a list of all the vector ports:

```
> filter_collection [get_ports] "port_width > 1"
{"A", "DI", "DOUT"}
```

The basic form of a filter expression is a series of relations joined together with AND and OR operators. Parentheses are also supported. The basic relation contrasts an attribute name with a value through a relational operator. In the previous example, *is_hierarchical* is the attribute, == is the relational operator, and *true* is the value.

The relational operators are

```
  ==   Equal
```

```
!=   Not equal
>    Greater than
<    Less than
>=   Greater than or equal to
<=   Less than or equal to
=~   Matches pattern
!~   Does not match pattern
```

The basic relational rules are

o String attributes can be compared with any operator.

o Numeric attributes cannot be compared with pattern match operators.

o Boolean attributes can be compared only with == and !=. The value can be only true or false.

Additionally, existence relations determine if an attribute is defined or not defined, for the object. For example,

```
sense == setup_clk_rise and defined(sdf_cond)
```

The existence operators are

defined undefined

These operators apply to any attribute as long as it is valid for the object class. See the appropriate man pages for complete details.

Sorting Collections

You can sort a collection by using the **sort_collection** command. It takes a base collection and a list of attributes as sort keys. The result is a copy of the base collection sorted by the given keys. Sorting is ascending, by default, or descending when you specify the -descending option. In the following example, the **esp_shell** command sorts the ports by direction and then by full name.

```
> sort_collection [get_ports] {port_direction name}
{"in1", "in2", "out1", "out2"}
```

Implicit Query of Collections

All commands that create collections implicitly query the collection when the command is used at the command line. The number of objects displayed is controlled by the **collection_result_display_limit** variable. Consider the following examples from PrimeTime:

```
pt_shell> set_input_delay 3.0 [get_ports in*]
1
pt_shell> get_ports in*
{"in0", "in1", "in2"}
pt_shell> query_objects -verbose [get_ports in*]
{"port:in0", "port:in1", "port:in2"}
pt_shell> set iports [get_ports in*]
{"in0", "in1", "in2"}
```

In the first example, the **get_ports** command creates a collection of ports that is passed to the **set_input_delay** command. This collection is not the result of the primary command (set_input_delay), so it is not queried.

The second example shows how a command that creates a collection automatically queries the collection when that command is used as a primary command.

The third example shows the verbose feature of the **query_objects** command, which is not available with implicit query.

Finally, the fourth example sets the variable iports to the result of the **get_ports** command.

Only in the final example does the collection persist to future commands until iports is overwritten, unset, or goes out of scope.

SEE ALSO

Commands: **add_to_collection** , **compare_collections** , **copy_collection** , **filter_collection** , **foreach_in_c**

Variables: **collection_result_display_limit**

Commands Listed by Functional Group

Modeling SPICE Devices

add_device_model

create_model_library

read_spice_behavioral_model

report_devices

report_process

reset_device_model

set_device_model

set_process

Reading Source Files

reset

read_verilog

read_spice

read_spice_behavioral_model

Matching Design Ports

set_top_design

match_design_ports

remove_matched_ports

set_matched_ports

report_matched_ports

report_unmatched_ports

Testbench Pin Functionality

set_testbench_pin_attributes

report_testbench_pins

rename_testbench_pin

create_clock

reset_clock

report_clock

Managing Testbenches

write_testbench

set_testbench

set_active_testbench

list_active_testbench

Verification

check_design

verify

debug_design

report_status

report_log

Debug Design Mismatches

debug_design

explore_with_viewer

start_waveform_viewer

write_spice_debug_testbench

start_explore

start_spice_simulator

Shell Commands

reset

exit

quit

help

man

source

save_session

restore_session

Interactive Signal Trace (IST) Commands

start_explore

stop_explore

get_net_inputs

get_net_transitions

get_net_triggers

is_net_explorable

print_net_backtrace

print_net_trace

Collection Commands

get_designs

get_matched_designs

get_devices

get_inouts

get_inputs

get_outputs

get_instances

get_nets

get_pins

get_ports

Coverage Topics

Coverage Overview

Coverage Usage

Coverage Tasks

Merging Coverage Reports

Reporting Coverage

Code Coverage - Filtering Coverage Data

Overview

The coverage specification file defines specific coverage data reported by the **report_coverage** command.

Coverage Specification File Statements

The following sections provide details about file statements that control the information in your summary report.

Comments

Comments start with the character # in column 1, for example:

```
#Comment line
```

Select

Selects a list of modules, other than top-level modules, to report in the output file. The usage is as follows:

```
select <scope> [<scope>+]
<scope>::= (<level>, <instancepath>)
```

The hierarchical name used in the **select** command must be the complete hierarchical name from the root and not a relative hierarchical name. This is different from $espcovscope, where relative hierarchical names are allowed. This is because the select command does not have any local scope and hence the absence of a complete hierarchical name from the root can lead to ambiguity unless you have mapped the root of your design hierarchy. If you have a merged database with two new top-level modules, ximp and xref, use the following example:

```
select (0,xref)
select (0,ximp)

select (0,inno_tb_top.xref)
select (0,inno_tb_top.ximp.I1)
select (0,inno_tb_top.ximp.I2)
select (1,top.xref) (0,top.ximp)
```

printdetail

Reports detailed coverage information for all scopes. The usage is as follows:

```
printdetail <covtypelist>
```

If you do not specify the printdetail option, the tool reports only the summary for all the coverage types by default. With this option, you indicate the coverage type (or types) whose detailed information needs to be listed. To specify all the keywords available in the specification file, use printdetail all or specify one among the SA, DS, NS, or LC selections.

Consider the following examples:

```
 printdetail SA DS NS

 printdetail NS
 printdetail DS
```

filter and filtertree Commands

Filters remove unwanted information from the coverage report. The usage of these commands is as follows:

```
filter <covtype> <fully_scoped_identifier> [<missingtoggle>]
filtertree <covtype> <fully_scoped_identifier> [<missingtoggle>]
```

where
`<covtype>` is NS, DS, SA, or a combination of these, separated by spaces.
`<fully_scoped_identifier> ::= <instancepath>'.' <net_name> [<bit_enum>]`
`` <bit_enum> ::= `[` <enum_range> [ `,' <enum_range>]`]' ``
`` <enum_range> ::= <num> | <num> `-' <numgt; ``
`<missingtoggle> ::= norise | nofall`, and only applies to stuck-at (SA) coverage.

For example, when analyzing the results of real-world test cases, some signals are always nonsymbolic, such as clocks. In other cases, you can apply symbols to inputs just to ensure that they are not used (for example, data input symbols on a read operation). These symbols show up in the output report as dropped symbols. To mask these symbols out of the report, two filter operations are provided: filter and filtertree.

The `filter` command takes a list of signal names, which can include wild cards, and a coverage metric type (NS, DS, SA, or LC). It prevents the selected signals from being reported for that coverage type. Since the LC coverage type uses a completely different format, it is covered separately in the section "Filtering for LC Coverage Type". Notice that symbols are implemented as signals in the testbench.

The `filtertree` command, which is applicable only for SA and NS coverage metrics, helps in filtering all buffer- and inverter-connected trees associated with a signal. This convenience can sometimes drastically reduce the number of `filter` commands or the length of the list of a `filter` command in the *spec* file.

If you do not specify a value for `<missingtoggle>`, both missing rise and missing fall transitions are ignored. If you specify `norise`, missing rise transitions are ignored in the report. Similarly, `nofall` ignores missing fall transitions.

Defining a Fully-Scoped Identifier

A fully-scoped identifier has the form `<instancepath>'.' <net_name> [<bit_enum>]`.

Rules for defining the `instancepath` are explained in the following section. The `net_name` supports bit number enumerations similar to those used in the coverage report (for example, `net[20-5,3-1]`).

Defining an Instance Path

The `instancepath` must begin with a module name, which can be the top module name, but is not necessary (*inno_tb_top* in the case of a testbench created by the **write_testbench** command). This usage of `instancepath` is more restrictive than the one allowed in $espcovscope.

Each element in the `instancepath` represents a level of hierarchy. It is separated from the other elements by the "." character. Each element can also contain wildcard characters. The `instancepath` should never be empty in the context of the `filter` command. If the net name does not exist in the coverage database, the tool displays an error message.

Wildcard Characters in Filter Instance Paths

This feature is exclusive to the `filter` and `filtertree` commands and allows concise representation for filtering a large number of signals. The filtered signal name and all the identifiers in the instance path including the starting module name can use wildcard characters.

The SA, NS, and DS filters have the form `hier.net`, where `hier` is the hierarchical path to the net and `net` is the name of the net being filtered.

The hierarchical path is restricted to a *module name* followed by zero or more *instance names*. You can apply the following wildcards to *hier* as well as *net* :

`*`	Match any string of zero or more characters
`?`	Match any single character
`(ab|cd|ef)`	Match the alternative selections between parentheses

The following wildcard can be applied only to part1 (`hier`) of the filter:

`[xyA-Z0-9]`	Match any one of the enclosed characters; a hyphen can be used to specify a range (for example, a-z, A-Z, 0-9)
`[^xyz]`	Match any character not among the enclosed characters

The following wildcard can be applied only to part2 (`net`) of the filter:

`23-25,12-9`	Match any number between the specified range. In this example 23, 24, 25, 12, 11, 10 and 9 are matches.

Example: Filtering All Bits of a Net

```
filter DS inno_tb_top.net[*]
```

In this example, the module name is *inno_tb_top*, the net name is *net* and entire bus bits are specified with '[*]'.

Example: Filtering Specific Bits in a Group of Modules

```
filter DS *ABC*.rst.X*B.hn*[30-8,5-0]
```

In this example, the module name is all modules that have the substring *ABC* in their names. The instance path starts with the instance *rst* under each of the selected modules. The *rst* instance must have a child whose name begins with *X* and ends with *B* (specified by *X*B*). The net name is all the bus signals that begin with *hn* and have the specified bits.

Example: More Wildcards

```
filter NS A(pq|rs)*.*bc*.a??[a-zA-Z]*b?[^0-9]*.net[15-11,9-2]
filter SA module.net[*]
filtertree NS module.net[*]
```

Follow these rules while using wildcards:

1. The * matches 0 or more characters.
2. The *module_name* cannot be * (an exhaustive search is too expensive). For example, *filter NS *.clk* is not allowed, but *filter NS M*.clk* is allowed).
3. The top-level instance name (for example, inno_tb_top) is also treated as a module name for filters and wild cards.
4. Wildcard characters cannot include the hierarchical separator ".". They cannot cross hierarchical boundaries. You must explicitly specify each of the levels of hierarchy in the path. For example, *a.b.net* can be represented by *a.*.n** but not by *a.**.
5. When using wildcards inside a Verilog escaped name (starts with a backslash, ends with a space) the wildcards need to be added with a backslash. For example, consider the following nets:

   ```
   inno_tb_top.ximp.abc.\xyz[0] .jkl
   inno_tb_top.ximp.abc.\xyz[1] .jkl
   inno_tb_top.ximp.abc.\xyz[2] .jkl
   ```

   You can filter them using any of the following examples:

   ```
   filter NS inno_tb_top.ximp.abc.\xyz[\[0-2\]] .jkl
   filter NS inno_tb_top.ximp.abc.\xyz[\*] .jkl
   filter NS inno_tb_top.ximp.abc.\xyz[\?] .jkl
   ```

6. Remove arrayed module instance brackets from filter lines.

   For example, you could create a Verilog array of modules as follows:

   ```
   module tb;
       ...
       testmod inst[1:0] ({arg1,argb1}, {arg2,argb2});
       ...
     endmodule
     module testmod (in1, out1);
       ...
     endmodule
   ```

   In order to differentiate arrayed module instances from wildcards, remove their brackets as follows:

   ```
   filter NS tb.inst\[1\].in1
   filter NS tb.inst\[0\].out1
   ```

Filtering for LC Coverage Type

Line coverage filtering identifies and removes source code that you know does not require coverage reporting. The usage is as follows:

`filter LC <filename>:<range_of_lines>`

where

`filename` is the source file you want to report

`range_of_lines` gives the lines of source code that are filtered from the report.

Example Filtering Lines of Source Code

```
filter LC abc.gv:5
filter LC ../ab*.v:5-8
```

```
filter LC abc.gv:8-10
```

Filtering Examples

This section illustrates various `filter` and `filtertree` command forms.

The following example shows `filter` command statements for various coverage types:

```
filter NS inno_tb_top.xref.clk
filter NS inno_tb_top.xref.MX0.clk
filter SA inno_tb_top.xref.MX0.clk norise
filter SA inno_tb_top.xref.MX1.clk nofall
filter SA inno_tb_top.xref.MX2.clk
filter DS inno_tb_top.a_sym1[0-2]
```

Clock Filtering Examples

Consider the following example where the instance path starts with a module name that is not the top-level module name.

```
filter NS inner_module_name.clk
```

This helps in identifying `clk` as a nonsymbolic signal for all instances of `inner_module_name`. You have to make sure that this is always a clock signal in all instances where the module is used:

```
filter SA inner_module_name.stuck_net_name
```

Otherwise, you could filter out signals that you should examine. These two examples are the best in terms of efficiency and ease of use.

Alternatively, you could enumerate all instantiations of `inner_module_name` as follows:

```
filter NS inno_tb_top.ximp.I1.inst1.clk
filter NS inno_tb_top.ximp.I2.inst1.clk
```

This could be a laborious task, and lead to a long list.

Yet another way is to use wildcards, but this is expensive in terms of the runtime of the coverage tool.

```
filter NS inno_tb_top.ximp.I*.inst*.clk
```

This assumes that all the module instantiations are covered by the preceding regular expression.

Finally, another way to filter the specified clk signal is by using the `filtertree` command approach:

```
filtertree NS inno_tb_top.ximp.clk
```

This works only if the `clk` signal of all the instantiations of inner_module_name are connected through a buffer or inverter tree to inno_tb_top.ximp.clk.

Detecting and Levelizing bufinv Trees

The **report_coverage** command automatically detects bufinv trees and prints them in a levelized manner for nonsymbolic (NS) and toggle (SA) coverage. Use the `-filterdetail` option of the **report_coverage** command to remove this information in the coverage report.

In the nonsymbolic coverage report section, the regular nodes that do not not receive any symbols during simulation are printed first, followed by bufinv nodes. In the toggle coverage section of the report, the regular nodes that do not have a rise or fall transition or both, are printed first followed by the bufinv nodes. The source node of the bufinv tree is printed first, marked with >>. Rise, fall, and type information are provided to enable you to identify why the bufinv tree did not toggle. The driven nodes (net name only) follow the source node and are indented.

The ESP tool creates a specification file called bufinv.spec for each of the bufinv nodes after running the **report_coverage** command. This bufinv.spec file contains all the bufinv nodes with a *filtertree* for the source node and a *filter* for all the driven nodes. By default, the *filtertree* lines are commented out with the '#' sign and the *filter* lines are left active (uncommented). You can save this file to a different file name and use that file (or an edited version) in a subsequent run of the **report_coverage** command with the `-spec` option and filter out all the driven nodes except the master or the source node to reduce debugging the number of nodes. Do not use the bufinv.spec file directly as the filter file as it gets overwritten each time you run the **report_coverage** command.

The bufinv.spec file contains *filtertree* and *filter* lines for each of the coverage metrics, NS, and SA coverage depending on how you used the `-printdetail` option in the initial run. If you run `espcov -printdetail 'SA'`, the bufinv.spec file contains the `filtertree` and `filter` commands with the 'SA' specification. If you specify both NS and SA using the `-printdetail` option, the `filtertree` and filter commands are listed for both metrics.

With each execution of the **report_coverage** command, the tool creates or overwrites the bufinv.spec file based on how the `filtertree` and `filter` commands are applied to those nodes. When running ESP Cov, if all the bufinv nodes are filtered (by passing in a previously copied bufinv.spec file with the filter lines uncommented) the bufinv.spec file stays empty at the end of this run.

The examples in this section use the following top.v file with Verilog definitions:

```
module myand(z,a,b) ;
  input a,b;
  output z;
  and a10(z,a,b);
endmodule
module myinv(o,i,j);
  input i;
  input j;
  output o;
  wire w1,w2,w3;
  wire za0, ma1;
  supply1 innovcc;
  supply0 innovss;
  pmos MP1(o,innovcc,i);
  nmos MN1(o,innovss,i);
  not n11(w2,w1);
  not n21(w3,w2);
  and an1(za0,j,ma1);
endmodule
module top;
  reg ss;
  wire a1,a2,a3;
  wire b1,b2,b3;
  wire c1;
  wire a0;
  wire d1,d2,d3;
  wire f1,f2,f3;
  wire e1,e2;
  not n1(a0,a1);
  not n2(a3,a0);
  myinv i3(b2,b1,f1);
  myinv i4(b3,b2,f2);
  myinv i5(e2,e1,f3);
  myand i9(d3,d2,e2);
  initial begin
    $esp_var(ss);
    $espcovfile("top.cov");
    $espcovscope(0, top);
  end
endmodule
```

Run the following commands:

```
espcv top.v -hc high
espcov top.cov -printdetail SA
```

The espcov.log file includes the following information.

```
espcov <header info>
...
*************************************
and2_a (and2_a)
Symbolic Net Coverage: 0%
 All nets nonsymbolic
Propagated Symbol Coverage: not applicable
 Propagated Symbols: 0 propagated / 0 symbols
 Dropped Symbols: 0 dropped / 0 symbols
Toggle Coverage:    0%
Untoggled Nets: 31 / Total Nets 31
Missing toggles rise    fall     type name
X       X       B       a2
X       X       B       c1
X       X       B       d1
X       X       B       d2
X       X       B       d3
X       X       B       f1
X       X       B       f2
X       X       B       f3
X       X       B       i3.ma1
X       X       B       i3.za0
X       X       B       i4.ma1
X       X       B       i4.za0
X       X       B       i5.ma1
X       X       B       i5.za0
>>      X       X       B       a1
 a0
 a3
>>      X       X       B       b1
 b2
 b3
>>      X       X       B       e1
 e2
>>      X       X       B       i3.w1
 i3.w2
 i3.w3
>>      X       X       B       i4.w1
 i4.w2
 i4.w3
>>      X       X       B       i5.w1
 i5.w2
 i5.w3
```

Note: All the bufinv nodes start with `>>` so you can identify them easily.

The bufinv.spec file that the ESP tool generates after the espcov.log file is as follows:

```
#***************************************
#filter information for bufinv structures (and2_a)
```

```
printdetail SA
#filtertree SA and2_a.a1
filter SA and2_a.a0
filter SA and2_a.a3
#filtertree SA and2_a.b1
filter SA and2_a.b2
filter SA and2_a.b3
#filtertree SA and2_a.e1
filter SA and2_a.e2
#filtertree SA and2_a.i3.w1
filter SA and2_a.i3.w2
filter SA and2_a.i3.w3
#filtertree SA and2_a.i4.w1
filter SA and2_a.i4.w2
filter SA and2_a.i4.w3
#filtertree SA and2_a.i5.w1
filter SA and2_a.i5.w2
filter SA and2_a.i5.w3
```

Note: The master and source nodes are listed with *filtertree* and the driven nodes are listed with *filter*.

The code coverage filter feature has the following limitations:

- Coverage reporting does not include true source node information for a bufinv tree during toggle coverage reporting when hierarchical compression is used. In one manifestation, if the source node of the bufinv tree is supply1 or supply0, coverage reporting treats the source node as a constant signal and there is no rise, fall, or type information for that node. Coverage reporting reports the first driven node as the master node, indicated with >> and noted with an NM indicating it is not a true master (source) node.
- During coverage reporting with the `select` command set to a lower-level instance in the design hierarchy than the top-module instance, if the source node of the bufinv tree inside the lower-level module is determined to be at the top-most level, the rise, fall, or type information for this source node is unavailable, and coverage reporting makes the first driven node the master (source) node. Coverage reporting marks these nodes in the toggle section with NM. You can remedy this limitation by removing the `select` command from the specification file.
- Another limitation deals with non-bufinv tree nodes appearing in the bufinv tree levelization section in HC mode. This is caused by the determining master (source) node of nets. For a driven node in this instance, the master or source node is the starting node of the tree and the driven node is different from the source node. For non-bufinv tree nodes, the driven node and the master or source node are identical. Although distinctions are made, situations can arise where the master node for a particular non-bufinv tree node is derived to be different than the actual node. For example, this could occur if the input to the non-bufinv type gate is also an input to a bufinv tree section. If this input is passed from a higher level in the design hierarchy, and you place a `select` command on an instance lower in the hierarchy where the non-bufinv type gate and the bufinv tree are present, you can eliminate the problem by removing the `select` command.

Code Coverage - Merging Coverage Data

Merging Coverage Data

Merging of coverage databases is required when you run multiple simulations to verify a design and want to know the overall coverage of the design. For example, you might have a protocol verification and a data integrity test, and you want to know how much of the design is covered with both of these tests.

The **report_coverage** command requires you to specify a testbench with the `-testbench` option or use the `-all` option merges all the coverage information from the testbenches on the active testbench list.

In the ESP tool, the **set_active_testbench** and the **remove_testbench** commands maintain the active testbench list. The **list_active_testbench** command provides a list of all active testbenches.

Note: The two phase (*.2ph) test does not use symbols in a manner that is compatible with the other tests that are created by the *sram* style testbench. You should always remove the *.2ph from the list of active testbenches before creating a coverage report.

Merge Failure

Merging of coverage databases can fail in a number of ways:

- Using different databases for differently-named testbenches
  By default, the ESP tool creates a top-level testbench named `inno_tb_top`. The **testbench_module_name** application variable can change the name of the top-level testbench. If generated testbenches are used, ensure that **testbench_module_name** has the same value throughout the script. If you create your own testbenches, ensure that the top-level testbench module always has the same name.
- Using different +defines for some testbenches when simulated.
  This can result in different structural netlists and internal nodes which can prevent merging.
- Different values of the **verify_use_specify** application variable used in separate testbenches.
  This can result in different structural netlists and internal nodes that can prevent merging. Always run verification with the same setting for the **verify_use_specify** application variable when merging coverage.
- Using a different reference model or implementation model for the different coverage databases that are merged
  This is almost certain to prevent merging of coverage databases
- Using different versions of the ESP tool to create coverage data that is to be merged
  The tool does not support mixing different versions of coverage data.

SEE ALSO

Commands: **list_active_testbench** , **set_active_testbench** , **remove_testbench** , **report_coverage**

Topics: **coverage_filters** , **coverage_overview** , **coverage_reports** , **coverage_tasks** , **coverage_usage**

Code Coverage - Overview

Overview

When doing symbolic simulation, you might have questions about coverage and symbolic simulation, such as

- How do I ensure that symbolic simulation is happening and reasonable checks are being made?
- How do I ensure that I am running enough cycles and not waiting for more cycles to complete than are required?
- How do I know that I am not over constraining the testbench? What are the effects of the constraints provided?

This topic demonstrates how to use the coverage tool and interpret the reports to answer these questions. The following sections introduce the coverage metrics and some topics pertaining to symbolic simulation:

- Concepts
- Terminology
- How Symbolic Coverage Helps

Concepts

A symbolic simulator like ESP is different from standard Verilog simulations in that every net can have equations associated with it besides the fixed values of 0, 1, X, or Z you normally expect. There are four coverage metrics that the ESP coverage tool reports.

- Propagated or dropped symbol coverage

  Propagated or dropped symbol coverage measures the percentage of symbols that go into a scope but do not come out. All dropped symbols should be investigated. In a complete set of tests, this metric should be near 100 percent. The measurement is a ratio of the number of symbols propagated divided by the total of the propagated symbols and dropped symbols.

- Symbolic net coverage

  Symbolic net coverage measures the percentage of all nets that received symbols at any time during the simulation.

- Toggle coverage (or stuck-at coverage)

  This is the traditional toggle coverage available with most Verilog simulators. Toggle coverage measures the percentage of all nets that achieved both 0 and 1 value at any point during the simulation. The ESP tool considers symbolic values when recording toggle coverage data. Any nets that never achieve 0 or 1 during the simulation are assumed nontoggled or stuck-at and reported under that coverage metric.

- Line coverage

  This is the traditional line coverage availabe with most Verilog simulators. Line coverage measures the percentage of all lines of design code touched and executed by the symbolic simulation. This metric is meaningful for behavioral and RTL-level code. This metric has little value for gate level netlists and SPICE netlists.

Collectively, these metrics provide information about the effectiveness of the symbolic testbenches. While toggle and line coverage are used extensively in binary simulation coverage, propagated and symbolic net coverage specifically address the coverage problem in the symbolic simulation domain.

Terminology

In practice, the mnemonics, used in commands and outputs, represent the negation of ESP metrics to report the coverage results (except line coverage).

Specifically,

- DS (Dropped Symbols): Represents the propagated and dropped symbol coverage
- NS (Non Symbolic): Represents symbolic net coverage
- SA (Stuck-At): Represents toggle coverage
- LC (Line Coverage): Represents line coverage information

In the coverage output, there are a number of symbolic statistics that use the following terms:

Nonsymbolic nets
: A list of nets that never had symbols. These nets always had binary content (0,1,x,z) and possibly were stuck at one value.

Propagated symbols
: A list of symbols that were driven out of a particular scope. The scope might have bidirectional nets; the metric accounts for directionality.

Dropped symbols
: A list of symbols that entered, but did not come out of the specified scope. Similar to propagated symbols.

The use of symbolic coverage does not guarantee complete coverage. It tells you what has not been covered. Coverage is a summation of simulation results over the simulation period. The following examples show the shortcomings of symbolic coverage:

- If a net has a symbol at any point, it is considered symbolic. It does not matter whether the symbolic value is propagated to the output to be observed.
- A net that has a symbolic value for only one of the testbench cycles is still reported as a symbolic net.

Thus, symbolic coverage (and coverage in general) must be used as a negative check to ensure that the testbenches in question do not lack in the ability to verify a design. In other words, high coverage numbers do not imply the absence of bugs in a design, but only the authenticity of the testbench. Conversely, if the coverage metrics reports low numbers, the testbench needs improvement.

The following example shows a symbolic coverage output:

```
inno_tb_top.ximp (RAM1R1W)
Symbolic Net Coverage: 100%
 Non-Symbolic Nets: 0 / Total Nets 455
 42 Non-symbolic Nets Ignored by Filter
Propagated Symbol Coverage: 100%
```

```
 Propagated Symbols: 52 propagated / 52 symbols
        inno_tb_top.A_sym1[2-0]
...
        inno_tb_top.inno_addr_sym3[2-0]
 Dropped Symbols: 0 dropped / 52 symbols
Toggle Coverage: 100%
 Untoggled Nets: 0 / Total Nets 455
```

How Symbolic Coverage Helps

Theise answers to the questions in the *Overview* demonstrate how to use the coverage tool effectively.

How do I ensure that symbolic simulation is happening and reasonable checks are being made?

- Dropped symbols, at the top level of a design, report situations where symbols go in to a design but do not come out of the design. These symbols are never observed on the outputs of the design.
- If you use the ESP tool, the `$display` functions that are already part of a generated testbench identify when no signal values except `x` are coming out. The ESP tool created testbenches will fail if an output is always `x` and the comparator is `` `ESPCOMMONLIBMODULE.comparex``. An `x` on the output could indicate:
  - Missing supply definitions for the SPICE design.
  - Incorrete supply definitions for the SPICE or power aware Verilog design.
  - Missing constraints in the testbench.
  - Gross timing errors in the SPICE design. Be sure and use a device model file if you suspect gross timing issues. The ESP tool is not SPICE timing accurate. The ESP tool has only functional timing accuracy.
  - A flaw in the SPICE or Verilog design
- If the symbolic simulation is slightly active for the design under test, large parts of the design might not take on symbolic values. In this case, symbolic net coverage numbers help to show that the simulation is deficient (or that the design is incorrect).

How do I ensure that I am running enough cycles and not waiting for more cycles to complete than are required?

This problem occurs frequently in pipelined and datapath designs where there is uncertainty on the number of cycles to be run. By looking at the dropped symbols, you can find out whether any of the cycles have not come out yet, and you require more flush cycles to propagate them out. (A flush cycle is a stimulus cycle in which the input stimulus is kept binary to allow the design an extra cycle to propagate interesting values to the output.)

A net is considered symbolic even if it gets a symbolic value in only one testbench cycle. You can use symbolic nets to determine if no symbols ever reach an area of a design or a stage of the pipeline. Symbolic nets are not as useful as propagated or dropped symbols.

How do I know that I am not over constraining the testbench? What are the effects of the constraints provided?

The report on nonsymbolic nets addresses this issue. Dropped symbols can help because over-constrained designs swallow symbols and they are not propagated through the design. However, they might not enter the design in the first place. Therefore, they are not dropped. This can happen if the testbench constrains the input stimulus and kills certain symbols before they enter the design.

An over-constrained design has large parts in it that do not get symbolic values. Toggle and line coverage can be secondary indicators to help identify regions of low activity due to extra constraints.

Code Coverage - Reporting Coverage Data

Coverage Reporting

The **report_coverage** command provides a summary report as the default. The summary report prints just the scope, module name, and the coverage percentage for each of the metrics reported.

The coverage report is scope-based. It might be impractical to expect all scopes to be printed out, so you can select the scopes to be printed. For example, a coverage report for a typical hierarchical SRAM would list all the core cells if everything were printed.

The mechanism to select the scopes that are reported is the specification file, which lists the Verilog scopes to report. It needs to be specified from the top (using full paths). There can be multiple scopes in the specification file.

If no scope is specified, the top-level scopes where data was gathered in the testbench is reported. In the case of a testbench generated by **write_testbench** , these would be *xref* and *ximp*.

To obtain more information, use the optional *-spec* command-line option to choose the information to display, to request more information about other scopes, or to filter the coverage results.

Using specification file commands, you can turn on or off the detailed reporting for each of the individual sections in the report file. Each is controlled independently from the other.

Refer to **coverage_filters** for information on the format and usage of the specification file.

Interpreting the Report Format

The significant sections of the coverage report are the propagated symbol coverage followed by the symbolic net coverage sections. You want these numbers to be as high as possible. In particular, if the propagated symbol coverage is not 100%, all dropped symbols must be logically reasoned and subsequently filtered with the help of the specification file (for example, data in symbols of a read cycle). You might also need to use filtering to remove known nonsymbolic signals (such as clock signals and their fanouts). filtertree can help filter clock trees and other uninteresting buffer or inverter connected signal trees in a single specification file command.

The format of the coverage report output is shown below. There might be one or more <scope> sections depending on the number of $espcovscope statements in your Verilog code or the number of scopes selected in your espcov specification input file.

If you have not specified printdetail in the specification file, only the summary information appears in the report. The lists under each summary header is not printed.

The meaning of fields in the report are as follows:

<scope>	Full hierarchical path to the scope in question.
<modulename>	Name of module
<TotalNets>	Total number of nets as seen by the simulator

The basic template for a coverage report:

```
Welcome, ESP COV Version <version>, <compiledate>
Copyright (C) 1998-2012 Synopsys, Inc. All rights reserved.
command line arguments:
 <commandlineused>
   ESP Coverage Database Version 2.0 read on <currentdate>
*************************************
<scope> (<modulename>)
Symbolic Net Coverage: XXX%
 Nonsymbolic Nets: <NumberOfNets> / Total Nets <TotalNets>
    <list of nets which have never had symbols applied>
 <number> Nonsymbolic Nets Ignored by Filter
    <list of nets which have been removed by the filter>
Propagated Symbol Coverage: XXX%
 Propagated Symbols: <NumberOfSymbols> propagated / <TotalSymbolsUsed>
    <list of symbols which entered and exited this scope>
 Dropped Symbols: <NumberOfSymbols> dropped / <TotalSymbolsUsed>
    <list of symbols which entered but did not exit this scope>
 <number> Dropped Symbols Ignored by Filter
    <list of nets which have been removed by the filter>
 <NumberOfNets> Binary nets with symbolic fan in
    <list of nets which always have binary values but have symbolic inputs>
Toggle Coverage: XXX%
 Untoggled Nets: <NumberOfNets> / Total Nets <TotalNets>
    <list of nets which never go to 1 or 0>
 <number> Untoggled Nets Ignored by Filter
    <list of nets which have been removed by the filter>
*************************************
Line Coverage: XXX%
 Total lines: <TotalLines>, Uncovered lines: <NumberOfUncoveredLines>
    <file name> has the following uncovered lines
   <line> <list of instances which do not have this line covered>
 <number> Uncovered Lines Ignored by Filter
    <list of nets which have been removed by the filter>
Unused Filters in <specfile>: <numberunused>/<totalfilters>
    <list of filters which were never activated>
    An example of a summary output for the coverage generated by the
    standard testbench is as follows:
```

Example of a summary report:

```
Welcome, ESP COV Version 5.0.0, Mon Sep 2 21:38:37 PDT 2002
Copyright (C) 1998-2012 Synopsys, Inc. All rights reserved.
command line arguments:
 espcov esp.cov
ESP Coverage Database Version 1.1 read on January 23, 2002 10:22:06
*************************************
inno_tb_top.xref (Reference)
Symbolic Net Coverage: 90.32%
```

```
  Nonsymbolic Nets: 3 / Total Nets 31
 Propagated Symbol Coverage: 90.47%
  Propagated Symbols: 57 propagated / 63 symbols
  Dropped Symbols: 6 dropped / 63 symbols
  0 Binary nets with symbolic fan in
 Toggle Coverage: 100%
  Untoggled Nets: 0 / Total Nets 31
 *************************************
 inno_tb_top.ximp (implement)
 Symbolic Net Coverage: 90.55%
  Nonsymbolic Nets: 12 / Total Nets 127
 Propagated Symbol Coverage: 90.47%
  Propagated Symbols: 57 propagated / 63 symbols
  Dropped Symbols: 6 dropped / 63 symbols
  0 Binary nets with symbolic fan in
 Toggle Coverage: 100%
  Untoggled Nets: 0 / Total Nets 127
 *************************************
 Line Coverage: 100%
 Total lines: 1479, Uncovered lines: 0
```

No filters were applied in this case, so the filter information is not applicable. For information about how the detailed output looks for each section or how to interpret the data, see the appropriate sections that follow.

Symbolic Net Coverage Results

This section reports the nets in every module that do not have symbolic values assigned to them at any time during the simulation. To add details for this section, specify printdetail NS in the specification file.

The default is to dump information for all modules below the specified scope. You can limit the information dumped per scope by specifying how many levels below that you want dumped using the select specification file command.

The following example shows a report of symbolic net coverage.

```
*************************************
test.a.x (blk5)
Symbolic Net Coverage: 85%
 Nonsymbolic Nets: 6 / 40
   bResult
   clk
   I1.cb
   I2.dat[4,2-0]
```

This report shows that under the instance test.a.x, where blk5 is the module name, bResult and clk are always binary. There is also an instance I1 within blk5 that has the signal cb that is always binary. In instance I2, only bits 4 and bits 2 to 0 of net dat are nonsymbolic. All other bits of the bus are symbolic. The format of the bit range is little endian, not necessarily the original bus order.

Notes:

1. If no $esp_var exists in the testbench, ESP runs as if NS and DS are not present in $espcovfile. The sections Symbolic Net Coverage, and Propagated and Dropped Symbols are not shown in the report.
2. Similarly, if $esp_var has no arguments (so it looks like $esp_var()), it does not report these sections either.
3. If $esp_var has arguments but the arguments are not used, it generates a report for symbolic net coverage and propagated symbol coverage. In this case, the symbolic net coverage is

expected to be 0 percent, and propagated symbol coverage 100 percent(0 propagated out of 0 symbols).

Propagated Symbol and Dropped Symbol Results

This section reports symbols in modules that are propagated through the module and reports symbols that come in, but do not go out of the module as dropped symbols.

Add the command printdetail DS in your specification file to enable the detailed output for this section.

An example of a detailed report file for DS is shown here.

```
*************************************
inno_tb_top.ximp (rand)
  :
  :
Propagated Symbol Coverage: 80%
 Propagated Symbols: 4 propagated / 5 symbols in
  inno_tb_top.a_sym2
  inno_tb_top.a_sym3
  inno_tb_top.b_sym2
  inno_tb_top.b_sym3
 Dropped Symbols: 1 dropped / 5 symbols in
  inno_tb_top.b_sym1
 0 Binary nets with symbolic fan in
```

In this example, the symbols a_sym2, a_sym3, b_sym2 and b_sym3 were applied to the inputs of the module and were propagated by the simulation to the outputs of the module.

The symbol b_sym1 was applied as an input but it did not reach an output pin at any point in the simulation.

Propagated Symbol Coverage information is recorded for all the sub-instances subject to the $espcovscope directive. For example, if you specify $espcovscope(0,ximp), all the sub-instances under module ximp, propagated and dropped symbol information is recorded. The reporting at any specific level includes all the nets in that scope as well as all the nets of all its children and their children down to the lowest leaf cells. Notice that $espcovscope can limit the number of generations of children that the data is gathered for, which can exclude the leaf cells.

Setting the Scope for Dropped Symbol Reporting

Collecting dropped symbol data during a symbolic simulation is costly in runtime. As you gather data for lower and lower levels of modules, this operation becomes more expensive. Furthermore, if you run with -hc, it becomes so expensive that it can cause the symbolic simulation run to randomize or run out of memory. However, usually you do not need to collect the DS data for lower levels. By default, DS coverage is done only for the top-most level. More detailed data is required only for debugging and locating the logic that dropped a symbol. If the propagated symbol coverage is 100 percent or the dropped symbols are easily explainable (which is usually the case), the lower level data would be redundant.

The levels of dropped symbols data collection can be controlled using the shell variable **verify_coverage_max_dropped_symbol_levels** You should leave this variable at the default level of one. If you need information for lower levels, change the level to 0 to dump all levels, or choose a level number that dumps the area you are interested in. For information about how to handle capacity issues under coverage, see the coverage "Frequently Asked Questions" **faq_coverage** .

If you want to find out exactly where a symbol was dropped, look at the dropped symbol information for the sub-instances of the top-level scope that drops the symbol. Locate the sub-instance that drops this symbol. You can repeat this process successively to zero in on the logic that actually dropped the symbol.

The dropped symbol information loses its accuracy in the following cases:

Some or all the ports of a module are inout	In this scenario, ESP treats an inout port as an input as well as an output port. Thus any symbol that "comes in" on such a port is also assumed to be "going out". For such modules, the dropped symbol information is optimistic and incorrect (a symbol might be assumed as propagated where it was actually dropped).
A port of a module is multi-driven and at least one of the drivers does not belong to this module	This typically happens at fine granularity where an instantiation is a transistor and the output of the transistor is wire-anded or wire-ored with outputs of other transistors. For such scenarios, a symbol that is not at the input port of a transistor might appear at the output port through another transistor. Even if one transistor drops a symbol, another one might propagate it.

You should set the scope of your investigation of dropped symbols down only if the ports of the lower-level module do not violate the preceding criteria. ESP determines the direction of an inout port (whether it is input or output) on a best effort basis.

Binary Nets With Symbolic Fanin (No Hierarchical Compression)

A binary net is where a net can be toggling between binary values 0, 1, X, Z but never gets a symbolic equation value. For example, if you have a 2-input AND that has a data input with the binary value 0 but the other data input is symbolic, the output is reported in this subsection because a symbolic value was not propagated on the output of the AND. This section does not show up when using hierarchical compression because the information cannot be determined in that environment.

Stuck At (Toggle) Coverage Results

This reports the nets in every module that do not have complete toggle coverage. It reports, per module, the nets within that module that do not have a rise or a fall transition. From that you can infer that the net is stuck at a particular value because the transition to that value has not occurred.

A rise transition is any transition to the value 1. This includes the transitions from 0, X, or Z to 1. A fall transition is any transition to the value 0. This differs from the Verilog definition of posedge and negedge.

Nets are also marked as being S (symbolic) or B (binary). The definition of binary is obvious in that the net has never had a symbolic value associated with it. The net has a value of 0, 1, X or Z.

The definition of symbolic is more complex. If a net has been marked as symbolic, an X in the fall column means that a transition to 0 never occurred and the net has a symbolic value on it at some point. For example, assume a net has an initial value of 0 at time 0 and has a formula (F) that contains only 0, X or Z at time t. If the value changes to ~F at time t+1, only a rise transition could have occurred.

The initial value of a net is determined as follows: at time 0, simulation starts with all nets as X. A number of initial values can be asserted at time 0 that cause some nets to get assigned a value other than X before the next time step is started. At the point where the next time step after time 0 is begun, coverage samples all the net values and uses that value as the initial value of the net.

In another example of a symbolic net, assume the net has an equation, F, at time t that can be 0 or 1 or X or Z. If the formula changes to ~F at t+1, both a rise and fall transition have occurred at the same time. If a signal has both a rise and fall, it is not reported as having a missing toggle.

Add the command printdetail SA to your specification file to enable the detailed output for this section.

An example of the report is shown below. The top level being reported on is test. The hierarchical path to the module named mod1 in this example is test.a.b and the path to the module gate2 is test.c.

```
     ************************************
     test (test)
 :
 :
     Toggle Coverage: 50%
      Untoggled Nets: 7 / Total Nets 14
 Missing toggles
 rise fall type name
    X    S   a.b.Co
  X    X    B   a.b.b
  X B   a.b.y
    X    S   c.net103
    X    S   c.net106
  X S   c.net136
  X S   c.net89
```

This report shows that there is a signal named a.b.Co, which is a symbolic signal and it is stuck at non-zero. The X under the fall column indicates that a fall transition has not occurred or is missing. From that, you can infer that it has transitioned to a nonzero value since it never had a fall transition.

The signal named a.b.b never had a rise or fall on it. The initial X or Z value remains throughout the simulation.

Another signal named a.b.y is a binary signal and it is stuck at non-one. It had a fall transition from X to 0 but never had a rise transition.

If a net has been marked as symbolic (S), an X in the fall column still means that a transition to 0 never occurred. In the case of module gate2, the signal c.net103 could have changed from X -> 1 -> X -> 1 -> Z -> 1 when evaluating one possible solution to the equation, but it did not get assigned the 0 value.

Similarly, c.net89 had a fall transition of X -> 0 -> X -> 0 but never had a transition to the value 1. A symbolic negedge is defined as a transition from X or Z to 0.

Toggle Coverage for Primitive Cells

If you have a cell with internal nodes that you do not want coverage to report on, use the `innocov_off `... endinnocov_off construct to ignore those nodes. This also affects line coverage. Any uncovered lines within the primitive cell is reported as the primitive instance line being uncovered.

```
`ifdef _ESP_
      `innocov_off
      `endif
module inv (y, a);
      input a;
      output y;
      m1 y a vcc vcc p
      m1 y a vss vss n
endmodule
`ifdef _ESP_
`endinnocov_off
`end if
```

If you need to run simulation using these cells in a simulator other than in a Synopsys product, you need to qualify these commands using the _ESP_ compile definition.

Line Coverage Results

This section reports the lines that have never been simulated. For behavioral code, you can determine which branches have not been tested. For switch-level models, you can find out which transistors have gates that are always off during the part of simulation that is gathering coverage data.

This section is only scope-based with regard to gathering the data. The output has no scope and all uncovered lines are identified. Only one line coverage section is shown in the output of espcov even if more than one scope has been specified with $espcovscope.

Line coverage is reported as the percentage of lines covered per total lines present. It does not matter how many times a module has been instantiated.

The line coverage report identifies which specific instances have not had a line covered, rather than which instances have that line covered. Unused modules are not reported as uncovered.

If different paths are used for the same file names and you are trying to merge the data, the full path name is used to match the coverage information to be recorded.

Example Coverage run 1 references file f.v using a/b/c/d/f.v. Coverage run 2 references the same file using a/lnk/d/f.v. The line coverage report shows them as two separate files after the merge.

If two identical designs are represented by two different unidentical files, the information is merged, with both files being considered as separate files and reported separately.

If you are doing divide-and-conquer simulations in different directories, the line coverage might not be accurate unless the file name and path are identical. (Relative file paths are recognized when used.)

Add the command printdetail LC to your specification file to enable the detailed output for this section.

The coverage output consists of one or more sections based on the files being compiled. The first part of each section indicates the file name and path and is followed by one or more line numbers that have never been encountered during simulation. The line number is followed by one or more hierarchical instances that reference that line and have not been executed.

```
      <file name> has the following uncovered lines
  <line> <list of instances which do not have this line
covered>
```

Line coverage is reported by blocks of sequential RTL code. Lines within the block are not reported as uncovered if the first line is reported as uncovered. Since all lines have to be executed sequentially, you can assume that the following lines in that block also have not been executed.

Example section of code:

```
      initial begin // line 200 in test.q (module is ex1)
 X = 0;
 #10 $finish();
      end
      initial begin
 A = 1;
 B = 2;
 #100
 C = 0;
 D = 1;
```

```
      end
```

Example of a report file:

```
 Line Coverage: 99.7%
  Total lines: 1000, Uncovered lines: 3
tb_esp.v
 35    test.a.b
208    test.q.c
```

In this example, the file tb_esp.v line 35 has never been encountered during simulation. The hierarchical instance path, which includes this line, is reported as test.a.b. Also, lines 208 and 209 have never been executed and these are reported via the report for tb_esp.v:208 line. Since C and D are executed together and cannot be interrupted, only the first line of that code block is reported. However, the number of uncovered lines does count all of the uncovered lines. In this example, three lines are reported as uncovered.

Example of Filtered Output File Format

Filtering the symbolic net coverage results increases the coverage metric so that you can identify what is not covered. This reduces the amount of output generated, revealing unexpected uncovered nets.

Filtering of results sometimes requires feedback on the effectiveness of the filter. You might want to report on things filtered out and filter statements not used. Reporting on these is not the default because the idea is to reduce the amount of data that is being reported. These sections also show up only if a filter for a section has been specified in the spec file. Reporting of the filter details is controlled by the command-line option -filterdetail. If that option is not specified, only the summary result appears, not the detailed results.

The symbols, nets, and lines that get filtered out are the same format as if they were not removed by the filter. They are just placed in another section of the report. The total number of Nets/Symbols remains the same with or without filtering. The number of (nonsymbolic/dropped symbol/untoggled net/ uncovered lines) is reduced by the filtered number. If you add those two numbers, you should get the unfiltered value.

If filter specifications have not been used to filter the coverage report data, these are reported in the unused filter section of the report. This section is printed once at the end of the report and only if the command-line option -unusedfilters has been specified to report it.

Example of output for unused filters:

```
Unused Filters in specfile: 4 / 20
   filter NS inno_tb_top.xref.clk
   filter SA inno_tb_top.xref.MX0.clk norise
   filter DS inno_tb_top.a_sym1[0-2]
   filter LC abc.gv:5
```

Example

```
module inno_tb_top ();
   `ifdef ESPCOV
   initial begin
        $espcovfile("esp.cov");
        $espcovscope(0,ximp);
        $espcovscope(0,xref);
```

```
      end
      `endif
   endmodule
```

Limitations

The following sections describe the limitations of the coverage report.

Randomization of Symbols

Randomization is the conversion of randomly selected input symbols into constant binary values to work around capacity problems. Randomized symbols can result in nets being reported as nonsymbolic. The binary value on these signals can also cause other symbols to be dropped because they are now disabled due to use of the binary value.

Legacy:
Signals in the esp.Random file that are read by the -r command-line option are not reported as dropped or propagated.

Output Checking

When analyzing coverage, consider whether the output signals are checked at the appropriate time. You can have symbols propagated throughout the design and your coverage numbers look good, but if you are not checking the outputs at the appropriate time, or not checking them at all, you do not detect any mismatches that are present. Therefore, coverage does not help you determine if your testbench is complete in that respect. This issue is not addressed by the current coverage implementation.

Code Coverage - System Tasks

Overview

You can control ESP coverage results similar to the VCD dump interface. There are four ESP coverage system tasks that control the generation of the coverage database:

- **$espcovfile**
- **$espcovscope**
- **$espcovoff**
- **$espcovon**

These system tasks are normally placed within an initial statement and can dynamically control the generation of coverage data.

Refer to the **Coverage Overview** secton or the tasks listed for more information.

Code Coverage - Usage

Coverage Flow

Symbolic coverage analysis is done in two steps. The first step generates raw coverage data in a database file during simulation. The information collected can be controlled by the $espcovscope function. Typically, you would dump data for only the reference (xref) and implementation (ximp) models.

The second step generates a summary report on the data collected in the first step using the **report_coverage**. Multiple simulation results can be merged during this step to generate the summary report. The amount of information reported is ,by default, only the top level information, but a specification (spec) file can be used to request more detailed information be dumped.

It is expected that prior to reporting coverage, you have simulated a testbench without detecting any errors and without having randomization in the simulation results. Either of these causes the coverage results to be invalid or severely handicapped.

Note:

If your set of testbenches includes the two phase test, you should remove the *.2ph testbench from the list of testbenches to be reported by **report_coverage** . The **remove_testbench** command can be used to remove the testbench after passing verification but before reporting coverage information.

The usage model for running symbolic coverage is as follows:

Run verification with **coverage** turned on	`set_app_var coverage on`
Check that verification passed with **report_status**	`report_status`
Remove *.2ph test from list of active tests.	If testbench name is VT `remove_testbench -testbench VT.2ph`
Generate coverage report.	If the filter specification is cov.spec then `report_coverage -all -spec cov.spec -filterdetail`

Failing Verification

```
 ...
set_app_var coverage true
verify
 ...
Verify testbench tb0 esp.tb.bin ......
Testbench 0 finished
Verify testbench tb1 esp.tb.ptl ......
-Testbench 1 finished
```

```
Error: Verification aborted due to capacity. (ESPUI-032)
Information: Use command "report_log" for more information. (ESPUI-078)
0
report_status
****************************************
Report : report_status
Design : r128x128
Version: Y-2006.06-SP1
Date   : Tue Jul 17 12:54:12 2007
****************************************
Reference Design:      r:/WORK/r128x128
Implementation Design: i:/WORK/r128x128
Verification:
   Status:        INCONCLUSIVE
  tb0: pass   ESP_WORK/r128x128/esp.tb.bin
     Failing Points:     0
     Passing Points:     9
     Aborted Points:     0
  tb1: abort   ESP_WORK/r128x128/esp.tb.ptl
     Failing Points:     0
     Passing Points:     0
     Aborted Points:     9
  tb2: pend   ESP_WORK/r128x128/esp.tb.dit
     Failing Points:     0
     Passing Points:     0
     Aborted Points:     0
  tb3: pend   ESP_WORK/r128x128/esp.tb.2ph
     Failing Points:     0
     Passing Points:     0
     Aborted Points:     0
```

Successful Verification

```
 ...
> write_testbench VT
 ...
> verify
Information: Verification succeeded. (ESPUI-033)
> report_status
****************************************
Report : report_status
Design : ram1r1w
Version: Z-2007.06
Date   : Wed Jul 18 16:09:54 2007
****************************************
Reference Design:      r:/WORK/ram1r1w
Implementation Design: i:/WORK/RAM1R1W
Verification:
   Status:        SUCCEEDED
  tb0: pass   VT.bin
     Failing Points:     0
     Passing Points:     8
     Aborted Points:     0
...
  tb3: pass   VT.2ph
     Failing Points:     0
     Passing Points:     8
     Aborted Points:     0
> remove_testbench -testbench VT.2ph
> report_coverage -all -spec cov.spec -filterdetail
****************************************
Report : report_coverage
```

```
Design : ram1r1w
Version: Z-2007.06
Date   : Wed Jul 18 16:09:54 2007
****************************************
Information: Used coverage data for tb0 in merge. (ESPCOV-009)
Information: Used coverage data for tb1 in merge. (ESPCOV-009)
Information: Used coverage data for tb2 in merge. (ESPCOV-009)
Information: Coverage data for 3 of 3 testbenches merged
             together. (ESPCOV-006)
Welcome, ESP COV Version Z-2007.06, Wed May 16 20:00:47 PDT 2007
Copyright (C) 1998-2012 Synopsys, Inc. All rights reserved.
command line arguments:
Hostname: teclsrv12 (linux)
ESPCOV Info:  Detailed Info is in the log file
   ESP_WORK/coverage_report.log.
 ESP Coverage Database Version 3.0 read on
           July 18, 2007     16:09:54
*************************************
inno_tb_top.xref (ram1r1w)
Symbolic Net Coverage: 100%
 Non-Symbolic Nets: 0 / Total Nets 29
 1 Non-symbolic Nets Ignored by Filter
Propagated Symbol Coverage: 100%
 Propagated Symbols: 52 propagated / 52 symbols
        inno_tb_top.A_sym1[2-0]
 ...
        inno_tb_top.inno_addr_sym3[2-0]
 Dropped Symbols: 0 dropped / 52 symbols
Toggle Coverage: 100%
 Untoggled Nets: 0 / Total Nets 29
*************************************
inno_tb_top.ximp (RAM1R1W)
Symbolic Net Coverage: 100%
 Non-Symbolic Nets: 0 / Total Nets 455
 42 Non-symbolic Nets Ignored by Filter
Propagated Symbol Coverage: 100%
 Propagated Symbols: 52 propagated / 52 symbols
        inno_tb_top.A_sym1[2-0]
...
        inno_tb_top.inno_addr_sym3[2-0]
 Dropped Symbols: 0 dropped / 52 symbols
Toggle Coverage: 100%
 Untoggled Nets: 0 / Total Nets 455
*************************************
Line Coverage:   100%
Total lines: 2622, Uncovered lines: 0
(for efficiency, a multi line block may report only the first line in that block)
All lines covered
*************************************
inno_tb_top.xref (ram1r1w)
Symbolic Net Coverage: 100%
 Non-Symbolic Nets: 0 / Total Nets 29
 1 Non-symbolic Nets Ignored by Filter
Propagated Symbol Coverage: 100%
 Propagated Symbols: 52 propagated / 52 symbols
 Dropped Symbols: 0 dropped / 52 symbols
Toggle Coverage: 100%
 Untoggled Nets: 0 / Total Nets 29
*************************************
inno_tb_top.ximp (RAM1R1W)
Symbolic Net Coverage: 100%
 Non-Symbolic Nets: 0 / Total Nets 455
 42 Non-symbolic Nets Ignored by Filter
Propagated Symbol Coverage: 100%
```

```
 Propagated Symbols: 52 propagated / 52 symbols
 Dropped Symbols: 0 dropped / 52 symbols
Toggle Coverage: 100%
 Untoggled Nets: 0 / Total Nets 455
*************************************
Line Coverage:   100%
Total lines: 2622, Uncovered lines: 0
```

Enabling Coverage in Testbenches

Coverage is turned off by default. To enable coverage use **coverage** shell variable. Setting **coverage** to true enables gathering of code coverage data. If you are going to run code coverage then you should always set **coverage** to true.

Merging Coverage Data From Multiple Simulations

Merging of multiple coverage simulation results is done by using the espcovmerge command to create a new database file. You then use espcov to report coverage information about this new consolidated database.

The rules that you must follow are

1. The models referenced by the $espcovscope instance path must be identical for all simulations generating coverage data. This includes code optionally included using command-line +define options. The same +define should be used on all simulations.
2. For the simple merge used here, the top-level testbench module name must be identical and all the symbol names used for each test must be identical. For example, A_sym1, A_sym2, A_sym3 are used in all testbenches for the same A input pin.
3. If used, the -hc command-line option must be consistent across all simulations being merged. For example use -hc high (the default) across all simulations to be merged.

A simple merge command is

```
espcovmerge db1.cov db2.cov -db esp.cov -log espmerge.log
```

This example shows how a simple merge of results can be done. In the previous simulation runs, the coverage database files db1.cov and db2.cov were created. They could have been given those names in the **$espcovfile** system task or even more simply been renamed after the simulation completed. In this case, the merged output file is called esp.cov and you have directed the log file output to the file espmerge.log.

An optional merge specification file can be used to identify equivalent instance and module names in multiple simulation runs. However, if your module names remain the same in all the testbenches used, you do not have to use the specification file. The specification file commands for merging are documented in section **Merge Specification File Commands** .

Reporting Coverage Results

After simulation (or database merge) is finished, you can examine coverage results in the file esp.cov by using:

```
espcov esp.cov
```

You should now have a report file, called espcov.log by default, created in the local directory showing each type of coverage information that was gathered. By default, the report shows only top-level summary information for each $espcovscope specified in the coverage simulation (ximp and xref in the preceding example). To obtain more information about each of the coverage types in the output report file, use the -printdetail command-line option.

```
espcov -printdetail "NS DS SA LC" esp.cov -log espcov.log
 all Use in lieu of specifying all coverage types on the command line (i.e. -printdetail all)
 NS  Nonsymbolic
 DS  Dropped Symbols and propagated symbols
 SA  Stuck At binary values and toggle coverage
 LC  Line Coverage
```

The detailed report file reports the name and number of symbols that entered and exited each scope, thereby determining the amount of coverage that was achieved. It also reports symbols that entered a scope but did not exit as dropped.

If you turned off hieararchical compression during verification, it also reports which nets had symbols coming in, but did not get propagated to the output (dropped symbols). When you use hierarchical compression, this information cannot be generated.

You can specify a specification file using espcov -spec to control the information that is to be reported. You can specify different modules to report on and how many levels to report.

Custom Compiler

The Custom Compier environment can be used to design custom logic. In that environment you can use ESP to functionally verify your design.

Setup

To enable ESP as a verification tool within Custom Compiler you must do the following:

- setenv ESP_INSTALL_DIR < *Your ESP installation* >
- setenv SYNOPSYS_CUSTOM_SITE $ESP_INSTALL_DIR/auxx/esp/cdesigner
- Add ESP, Custom Compiler and WaveView to your PATH environment variable

Note:
ESP_INSTALL_DIR must be in your environment. This is not just for convenience.

Use

- Start Custom Compiler as normal and open a schematic for verification.
- Select menu item *Tools* ⇒ *Simulation*
- Select menu item *Simulation* ⇒ *Initialize* ⇒ *New...* (or *Load ...*)
    - Set *Simulator* to ESP and change the *Run Directory* if desired
        - Press *Create*
- Select menu item *Simulation* ⇒ *Inputs* ⇒ *Edit...* (or *Load...*)
    - Modify the control script as needed.
- Select menu item *Simulation* ⇒ *Verify with ESP*
    - Enter a file name in *Verilog Options*. This file is a Verilog *-f* control file and typically contains the name of the Verilog behavioral model and any other Verilog command options such as *-y* or *-v*.
        - Press *OK*
- The ESP shell will appear. Complete your work within ESP shell as needed.
- Exit the ESP shell with the `exit` command.
- You can now look at outputs within Custom Compiler. (See below)

Create/Edit Control Script

When using ESP from Custom Compiler, there is a Tcl control script that defines how verification is to happen within ESP. This control script is divided into a set of procedures that match the recommended ESP verification flow.

By default, if you have not previously created the control script a default script template is created. This default script includes comments to show commands that might be useful within the specific flow steps.

- **NOTE:** Do not remove any procedure declarations in this default script.
- **NOTE:** Do not add an exit or quit command to this control script.

Edit your control script as needed for successful verification

Viewing ESP results

The menu item *Simulation ⇒ Outputs* allows you to view the ESP log file, the error markers and the waveforms from an ESP verification.

ESP Log file

Select the menu item *Simulation ⇒ Outputs ⇒ View Simulator Outputs* to view the ESP Run log.

Error Markers

If verification fails ESP creates a set of error markers to help you debug any verification failures.

To turn on the ESP markers select the menu item *Simulation ⇒ Outputs ⇒ View Error Markers* .

Select the error marker you wish to investigate.

Viewing Waveforms in Waveview

If verification fails ESP will create a set of binary waveform data. This data can be viewed using WaveView.

To access this data select the menu item *Simulation ⇒ Outputs ⇒ Plot Waveforms* .

Now when you select a net within the schematic it will automatically be added to the WaveView plot window.

SEE ALSO

`Commands:` **`start_waveform_viewer`** **`explore_with_viewer`**

Galaxy Custom Designer (CD)

See **Custom Compiler**

Device Model Simulation

DESCRIPTION

By default, ESP uses it's own internal process library as the basis for transistor models. This approach scales well down to about 45nm. Below 45nm the default functional accuracy of ESP may lead to incorrect verification results.

If the default internal process library does not have the functional accuracy needed then there are several alternative solutions:

Method	Description
Device model simulation	Generate technology data from SPICE libraries for multiple device sizes. This is the preferred solution.
Strength overrides	Use **set_instance_strength_multiplier** to explicitly set transistor strength.
set_process	Explicit process values can be specified with the **set_process** command. This involves the most work.

Usage

The recommended approach to create accurate device models for use in ESP is to use the **add_device_model** and **create_model_library** commands to simulate your SPICE library and create an ESP device models file. This process is done once for a library.

Verification runs load the ESP device models file with the **set_process** command:

```
# replace tech.edm with the name of the actual ESP device models file
set_process -i -technology_file tech.edm
```

When using an ESP device models file, the delay rounding should be set to 1ps using:

```
set_app_var verify_delay_round_multiple 1
```

Frequently Asked Questions

How do I characterize and use devices with 5(or more) terminals?

ESP expects transistor models to have four ports. The base connection is ignored. To handle 5 or more ports, the ESP netlist must instantiate the device using X card syntax.

```
XM0 D G S B P1 P2 pfet6port L=31n W=500n
```

```
XM1 D G S B P1 nfet5port L=31n W=500n
```

When using the **add_device_model** command specify a surrogate device for the NFET and PFET devices. In our example the devices with 5 and 6 ports are called nfet5port and pfet6port. We will use the names espNfet5 and espPfet6 as the surrogate devices. Example simulation script:

```
add_device_model -cpoints {{31n 500n}} -ptype espPfet6 -ntype espNfet5 -voltage 1.0 -macromodel
create_model_library -spicelib mytech.sp -edmfile mtech.edm
```

Contents of mytech.sp

```
* Example library
.include '/global/vendor/myfoundary/models.l' TT

.subckt espNfet5 D G S B
XM0 D G S B VDD nfet5port L=l W=w M=1 AS=0 AD=0 PD=0 PS=0
.ends

.subckt espPfet6 D G S B
XM0 D G S B GND VDD pfet6port L=l W=w M=1 AS=0 AD=0 PD=0 PS=0
.ends
```

To verify a netlist design.sp create a wrapper netlist that provides a mapping from the 5 and 6 terminal devices and includes the actual design netlist. Read the ESP device models file with the **set_process** command. Read the wrapper file with **read_spice** instead of the actual design netlist. Example wrapper:

```
* Example wrapper

.subckt nfet5port D G S B psub
XM0 D G S B espNfet5 L=L W=W M=M NF=NF
.ends

.subckt pfet6port D G S B epi psub
XM0 D G S B GND VDD espPfet6 L=L W=W M=M NF=NF
.ends

.include 'design.sp'
```

Add the following code into the **esp_shell** command script used for verification.

```
set_app_var verify_delay_round_multiple 1
set_process -i -library_technology_file mtech.edm
read_spice -i wrapper.sp
```

How do I characterize and use devices that do not follow standard port order of: drain gate source base?

ESP expects transistor devices in a netlist to have four ports specified in the order drain gate source base. If a device has a different port order then the netlist must use X card syntax to instantiate the device.

```
XM1 B D G S pfetAlpha L=31n W=500n
```

Similar to handling 5 or more ports, simulate a surrogate device, mapping the standard port order to the actual device port order. Then in verification create a wrapper SPICE file that maps the ports for the actual device onto the surrogate device that was characterized.

How do I characterize and use FinFET devices?

Use the *-frange* option of the **add_device_model** command to specify the NFIN range for FinFet devices. Do not use the *-wrange* option of the **add_device_model** command For a single size FinFet device use the *-fcpoints* option of the **add_device_model** command.

SEE ALSO

Commands: add_device_model
create_model_library
set_instance_strength_multiplier
set_process

Variables: netlist_case
verify_delay_round_multiple
 verify_rcdelay_unit

Topics:

Distributed Processing

The ESP tool tasks such as verifying multiple testbenches or performing **Device Model Simulation (DMS)** to extract transistor device characteristics can be performed in parallel using distributed processing, greatly reducing the elapsed time needed to perform the tasks. The ESP tool distributes tasks to multiple hosts using the Synopsys **Common Distributed Processing Library (CDPL)**, which is included with every ESP installation. Several utility programs are available to help you set up and monitor distributed processes. For more information about **CDPL** and the utility programs, see the user documents located in the `ESP_install_root_directory/cdpl/doc` directory. The documents are the **CDPL Users Manual**, the **DP Manager User Guide** and **DP Manager Frequently Asked Questions**.

Distributed processing is configured with the **add_distributed_processors** command:

- Use the *-hosts* option to specify the host file.
- Use the *-max_active* option to specify the maximum number of tasks to run in parallel, unless otherwise specified in the host file. Note that this option corresponds to the maximum number of relevant licenses that may be checked out to perform the task: ESP licenses for distributed testbench verification; circuit simulator licenses (HSPICE, FineSim, etc.) for **DMS.**
- Set the `CDPL_FARM_DISABLE_SHELL_INTERPRETER` environment variable if your computing environment does not accept scripts that begin with the #! characters. Set the variable to any value; the important distinction is whether the variable is set or not.

Host Files for Distributed Processing

A host file defines the distributed processing environment. In the host file, you specify information about the machines in the compute farm and the way jobs are to be launched. You must set up the computing environment before starting an ESP session that uses **CDPL**.

The following protocols are supported:

- RSH (remote shell)
- SSH (secure shell)
- SGE (Oracle Grid Engine, formerly Sun Grid Engine)
- LSF (Load Sharing Facility from Platform Computing)
- SH (shell process on the local host)
- PBS (Portable Batch System)
- RTDA (Runtime Design Automation Network)
- NETBATCH (Intel Netbatch)
- CUSTOM

The host file is an ASCII file in which each line follows the same format and provides information about one entity. Comment lines begin with a pound sign (#).

Each line in the host file must observe the following rules:

- The format for each line is: flag | hostname | num_slots | tmpDir | mode | command
- Each field is terminated by a pipe symbol ("|") except the last field.
- A space must be present for an empty field.
- Each line must be a single unwrapped line.
- The fields contain information as follows:
    - *flag*: 1 to enable; 0 to disable
    - *hostname*: Machine name; empty for some environments
    - *num_slots*: number of worker slots on this host or farm; -1 indicates unlimited
    - *tmpDir*: (optional) temporary work directory
    - *mode*: RSH, SSH, LSF, SGE, SH, PBS, RTDA, NETBATCH or CUSTOM
    - *command and options*: string containing the command to connect to worker machines (slave processes), written on a single unwrapped line in the host file. The string in this field can also be the file name (optionally including a path) of a script file that contains the submission command. The contents of the file must be a single unwrapped line of text.

Examples

RSH infrastructure with two host machines and allowing a total of 10 worker slots (4 on one machine and 6 on the other)

```
# host file for RSH
1|engr_lab-x9|4|/remote/users/tmp |RSH| rsh
1|engr_lab-x2|6|/remote/users/tmp |RSH| rsh
```

SSH infrastructure with 2 machines and allowing a total of 12 worker slots. This environment requires login without passwords.

```
# host file for SSH
1|engr_lab-x15|4|/remote/users/tmp |SSH| ssh
1|engr_lab-x21|8|/remote/users/tmp |SSH| ssh
```

SGE compute farm. The hostname field is empty for SGE farms.

```
# host file for SGE
1| |-1| |SGE| qsub -cwd -V -P bnormal -l mem_free=1G
```

SGE compute farm in which the number of launched jobs is limited to 3 and a directory for temporary files is specified.

```
# host file for SGE
1| |3| /remote/users/tmp |SGE| qsub -cwd -V -P bnormal arch=glinux
```

LSF compute farm allowing an unlimited number of worker slots. The hostname field is empty for LSF farms.

```
# host file for LSF
1| |-1| |LSF| bsub -R rusage[mem=8000] -R arch=glinux
```

LSF compute farm allowing an unlimited number of worker slots and using a script file to provide the submit command. The file named my_bsub contains the following line:

```
/lsf/bin/bsub -R rusage[mem=8000] -R arch=glinux
```

The host file contains the following:

```
# host file for LSF
1| |-1| |LSF| ./my_bsub
```

The SH protocol can be used to distribute processing over multiple cores on the same machine when

RSH and SSH are not available. The hostname field is not required, but in this example, localhost is entered as a reminder.

```
# host file for SH
1|localhost |8|/remote/users/tmp |SH| sh
```

PBS infrastructure with one machine and four worker slots. The hostname field is empty for PBS environments.

```
# host file for PBS
1| |4|/remote/users/tmp |PBS| qsub -q bnormal -l arch=glinux
```

RTDA infrastructure with one machine and eight worker slots. The hostname field is empty for RTDA environments.

```
# host file for RTDA
1| |8|/remote/users/tmp |RTDA| nc run
```

For more hostfile protocol examples, refer to the **CDPL UsersManual** in the `ESP_install_root_directory/cdpl/doc` directory.

Testing the Computing Environment

Before you run an ESP script that uses parallel analysis, you should verify that the host file for distributed processing is correct for your computing environment.

Every installation is different, but the general steps are as follows:

1. Set the `CDPL_HOME` environment variable to the cdpl subdirectory of the ESP installation root directory. For example:

   ```
   % setenv CDPL_HOME CDPL_home_dir
   ```

   where `CDPL_home_dir` is `ESP_install_root_directory/cdpl`.
2. Update the path variable to include the **CDPL utilities**, as follows:

   ```
   % set path = ($CDPL_HOME/bin $path)
   ```

3. Create a script file to configure your compute farm environment or obtain a script from your Information Technology support group. The objective is to configure a submit host that submits a distributed processing run to a compute farm. Determine the name of the compute farm and check that the machine you are using is a submit host for the farm. Use the `dpvalidate` command and the `dpcheck -m` command with the appropriate argument (lsf, sge, pbs, or rtda) to validate the infrastructure. For SSH or RSH protocols, all Linux hosts are submit hosts.
4. Verify that the selected queue has appropriate capabilities (such as memory and duration) for the planned ESP runs.
5. Test the farm setup with a simple command such as the `ls` command or the date command. For example, if you are using an LSF farm:

   ```
   % bsub -R "rusage[mem=1000]" date
   ```

6. Create and test the host file. For example:

```
% dpcheck -host your_host_file
```

7. Update your ESP script to specify the host file name using **add_distributed_processors**.
8. (Optional) To monitor and control a distributed processing run, use the DP Analyzer feature in the **dpmanager** utility (located in `$CDPL_HOME/bin`).

For more information about **dpmanager**, see the **DP Manager User Guide** and **DP Manager Frequently Asked Questions** documents, located in the `ESP_install_root_ directory/cdpl/doc` directory. For more information about **CDPL** and other utility programs, see the **CDPL Users Guide** located in the same directory.

For assistance with your computing environment, contact your Information Technology support group.

SEE ALSO

Commands: **verify** **add_distributed_processors** **create_model_library**

Variables:

Topics: **CDPL Users Manual**
DP Manager User Guide
DP Manager Frequently Asked Questions

Discovery Viewing Environment (DVE)

The Discovery Viewing Environment (DVE) can be used to view ESP results.

This is not an integration of DVE with ESP. This flow uses VCS to compile the design files and create a DVE environment. A VCS license is needed to use this flow.

The general flow is:

- Generate a **vcd** file from ESP.
- Compile source files in VCS with *-debug_all* .
- Start simulation: simv -gui
- Open *dump.vcd* from the ESP simulation.

This flow has been implemented by the **start_dve** command. The rest of this topic shows how to access DVE using the **start_waveform_viewer** command which is used by the GUI.

Accessing DVE from start_waveform_viewer

The following command will change ESP to use DVE as the waveform viewer.

```
set_app_var waveform_viewer {vcs %s
  $SYNOPSYS/auxx/esp/primitives/inno
  $SYNOPSYS/auxx/gui/dve/espPli.c
  -P $SYNOPSYS/auxx/gui/dve/espPli.tab
  -debug_all -R -gui}
```

ESP provides a special set of symbols for use with DVE. To access these symbols, set the DVE_SYMLIBS environment variable to point to the symbols

```
set env(DVE_SYMLIBS) $env(SYNOPSYS)/auxx/gui/dve/esp.sdb
```

After setting **waveform_viewer** and env(DVE_SYMLIBS), DVE is started with **start_waveform_viewer** command.

DVE will start up using the inter.vpd database. Use *File->Open Database* to open the file *dump.vcd* or any other **vcd** or VPD waveform database.

Viewing Schematics

Schematics can only be viewed when the *inter.vpd* database is active. Annotations to the source and schematic from the *dump.vcd* file are not supported. This is a limitation of this flow.

If you use **start_dve** then you can get source and schematic annotations.

Files needed

The extra files needed to make this flow work are found in the directory

<ESP installation directory>/auxx/gui/dve/

From the **esp_shell** this directory is:

$env(SYNOPSYS)/auxx/gui/dve/

The files needed are:

- esp.sdb - schematic symbols for use with ESP
- espPli.c - ESP PLI routine stubs
- espPli.tab - ESP PLI tab file for VCS

SEE ALSO

Commands: **debug_design** , **start_dve** , **start_waveform_viewer** , **stop_dve**

Variables: **waveform_format** , **waveform_viewer**

Topics: **esp_shell_flow** , **vcd**

$esp_addrvar

SYNTAX

$esp_addrvar (list of symbols);

DESCRIPTION

The **$esp_addrvar** system task identifies a symbol as an address variable. The optimum ordering of the symbols for a typical memory circuit is to put the priority of the address variables immediately after that of the control variables.

This system task is most often used in conjunction with **$esp_ctrlvar** and **$esp_datavar** and replaces **$esp_var** .

EXAMPLES

Declare symbols for ADDRA and ADDRB for 3 cycles.

```
reg [31:0] ADDRA_SYM1, ADDRA_SYM2, ADDRA_SYM3,
           ADDRB_SYM1, ADDRB_SYM2, ADDRB_SYM3 ;
$esp_addrvar(ADDRA_SYM1, ADDRA_SYM2, ADDRA_SYM3,
             ADDRB_SYM1, ADDRB_SYM2, ADDRB_SYM3) ;
```

SEE ALSO

Topics: **esp_cmdvar** , **esp_const** , **esp_const_one** , **esp_const_zero** , **esp_contains** , **esp_ctrlvar** , **esp_dat**

$esp_choose_var

SYNTAX

$esp_choose_var (list of symbols);

DESCRIPTION

The $esp_choose_var system task defines a special kind of symbol to the ESP tool. When verification would fail, the ESP tool attempts to find binary (0 or 1) values for all symbols created by calls to the $esp_choose_var system task that allow verification to pass. Verification is marked as failing only if no binary values for the symbols created by calls to $esp_choose_var system task allow verification to pass.

The **report_symbols_to_pass** command creates a report of the binary values, that allow verification to pass, for each symbol created by calls to the $esp_choose_var system task.

The ESP tool creates calls to $esp_choose_var in a generated testbench when the **set_symbol_to_pass** command is used in the Tcl script read by the ESP tool.

Variables that are specified in a $esp_choose_var task take on both a 1 and 0 throughout the simulation by becoming symbols. These symbols are tracked across the design, eventually reaching checkers that verify whether the two designs (reference and implementation) are producing identical equations.

You can call $esp_choose_var multiple times and from many locations, such as tasks, functions and named blocks. However, you should not assign a value to variables that are symbolic because it loses its symbolic characteristics. Use the $esp_choose_var system task to assign a value to another variable. Local variables (defined inside of a named block) should not be arguments to $esp_choose_var.

When you run ESP in binary mode (-b) or symbolic binary mode (-sb), $esp_choose_var no longer declares symbolic variables. Instead, it looks for the *esp.TestVector* file, which is created by the $esp_error statement. The *esp.TestVector* file is read and all the symbolic values declared by all $esp_choose_var statements are assigned a value from this file. If the file does not exist, a symbolic value is missing, or an extra symbol is defined, an error occurs and simulation is terminated.

Best Practice

Use the **set_symbol_to_pass** command for symbols used on design ports. Use the $esp_choose_var system task in a **declarartion file** for internal symbols such as symbols needed for serial loading of redundancy controls.

EXAMPLES

SEE ALSO

```
Commands: report_symbols_to_pass set_symbol_to_pass
Topics: esp_addrvar , esp_cmdvar , esp_const , esp_const_one , esp_const_zero , esp_contains , esp_ctr
```

$esp_cmdvar

SYNTAX

$esp_cmdvar (list of symbols);

DESCRIPTION

The $esp_cmdvar system task instructs ESP to treat the symbols as command (or opcode) variables. Those symbols have a medium priority in ESP. The main use of this system task is for operation codes of DRAM, FLASH memory, or CPU.

This system task is most often used in conjunction with **$esp_ctrlvar** and **$esp_addrvar** and replaces **$esp_var** .

EXAMPLES

SEE ALSO

`Topics:` **esp_addrvar** , **esp_const** , **esp_const_one** , **esp_const_zero** , **esp_contains** , **esp_ctrlvar** , **esp_da**

$esp_const

SYNTAX

$esp_const (variable expression);

DESCRIPTION

The **$esp_const** system task works like **$esp_const_one** and **$esp_const_zero** . It tests whether the variable expression is a constant (0, 1, X, or Z). If it is, the system function returns a value of 1; otherwise it returns a value of 0.

The variable expression can be a scalar, a bit select, a part select, a complete vector, or an expression containing these. An application for this system function would be to test whether a signal or signal bus becomes symbolic, giving insight into the testbench.

ESP supports two-dimensional arrays bit select and part select. For a Verilog register declared as `reg [a:b] mysample[c:d];`, ESP supports bit select *mysample[I][f]* and part select *mysample [I][f:g]*.

Note:
Do not use $esp_const, $esp_const_one or $esp_const_zero for any other purpose than to print a message using the Verilog $display() command. For the net R use this Verilog expression to check if R is a constant value:

```
if ((R===1'b1 && R!== 1'b0 && R!==1'bZ && R!=1'bX) || // $esp_const_one
    (R===1'b0 && R!== 1'b1 && R!==1'bZ && R!=1'bX) || // $esp_const_zero
    (R===1'bZ && R!== 1'b0 && R!==1'b1 && R!=1'bX) ||
    (R===1'bX && R!== 1'b0 && R!==1'bZ && R!=1'b1))
  $display($time, " R is a constant %b",R);
```

Using the expanded Verilog expression, you can safely change other net values.

EXAMPLES

For example,

```
  if ($esp_const(r))
    $display($time. " r is a constant %b",r);
```

This example prints out a message showing the value of signal r only when the value of r does not have a symbolic equation associated with it.

SEE ALSO

Topics: **esp_addrvar** **esp_cmdvar** **esp_const_one** **esp_const_zero** **esp_contains** **esp_ctrlvar** **esp_datavar** **esp_e**

$esp_const_one

SYNTAX

$esp_const_one (variable expression);

DESCRIPTION

The $esp_const_one system task tests to see whether the argument is identically one, meaning that it has no symbolic content. This system task is typically used to guard against something, such as a finish.

Note:

>Do not use $esp_const, $esp_const_one or $esp_const_zero for any other purpose than to print a message using the Verilog $display() command. For the net R use this Verilog expression to check if R is a constant value:

```
if ((R===1'b1 && R!== 1'b0 && R!==1'bZ && R!=1'bX) || // $esp_const_one
    (R===1'b0 && R!== 1'b1 && R!==1'bZ && R!=1'bX) || // $esp_const_zero
    (R===1'bZ && R!== 1'b0 && R!==1'b1 && R!=1'bX) ||
    (R===1'bX && R!== 1'b0 && R!==1'bZ && R!=1'b1))
  $display($time, " R is a constant %b",R);
```

Using the expanded Verilog expression, you can safely change other net values.

EXAMPLES

For example,

```
...
if (done_flag)
$display($time,"Finished all threads");
...
```

The *done_flag* option is set in another location in the code. It is likely that *done_flag* option is both true and false at some time as a result of the symbols going through the design. However, after the *done_flag* option is true, it hits the $display statement and starts displaying the finish time for multiple threads. The last display corresponds to the longest thread.

Change the code as follows:

```
...
if ( $esp_const_one (done_flag === 1'b1) )
$display($time,"Finished all threads");
...
```

The simulation waits until the *done_flag* option is completely true and there are no simulation threads that have it as untrue. A clean display occurs in this case. This display identifies the time taken by the longest thread being simulated.

An alternative that works in all Verilog simulators is to use the following:

```
...
if (done_flag === 1'b1 && done_flag !== 1'b0 &&
    done_flag !== 1'bx && done_flag !== 1'bz)
  $display($time,"Finished all threads");
```

SEE ALSO

Topics: **esp_addrvar** , **esp_cmdvar** , **esp_const** , **esp_const_zero** , **esp_contains** , **esp_ctrlvar** , **esp_datav**

$esp_const_zero

SYNTAX

$esp_const_zero (variable expression);

DESCRIPTION

The $esp_const_zero system task is similar to **$esp_const_one** , only it checks to make sure that the expression is identically zero. Its usage is similar to that of **$esp_const_one** .

Note:
Do not use $esp_const, $esp_const_one or $esp_const_zero for any other purpose than to print a message using the Verilog $display() command. For the net R use this Verilog expression to check if R is a constant value:

```
if ((R===1'b1  && R!== 1'b0  && R!==1'bZ  && R!=1'bX) || // $esp_const_one
     (R===1'b0  && R!== 1'b1  && R!==1'bZ  && R!=1'bX) || // $esp_const_zero
     (R===1'bZ  && R!== 1'b0  && R!==1'b1  && R!=1'bX) ||
     (R===1'bX  && R!== 1'b0  && R!==1'bZ  && R!=1'b1))
  $display($time, " R is a constant %b",R);
```

Using the expanded Verilog expression, you can safely change other net values.

EXAMPLES

SEE ALSO

Topics: **esp_addrvar** , **esp_cmdvar** , **esp_const** , **esp_const_one** , **esp_contains** , **esp_ctrlvar** , **esp_datava**

$esp_contains

SYNTAX

$esp_contains(signal, symbol);

DESCRIPTION

The $esp_contains system task tests whether the specified symbol is contained in the equation of the specified signal. For example, $esp_contains(a,b) tests whether symbol b is contained in the equation of signal a. Symbol b has to be a symbolic variable (declared using **$esp_var** or its variants) and a has to be a 1-bit signal. This task can test whether a variable affects the logic at a particular intermediate signal or output signal at a given time.

EXAMPLES

This example shows how to use the $esp_contains task.

```
module top;
 reg[1:0] r;
 reg[1:0] v;
initial begin
 $esp_var(v);
 #2;
 r = v[1] & v[0];
 #2;
 r = v[1];
 #2;
 r = v[1] || ~v[1];
 #2;
 r = v[0] && ~v[0];
 #2;
 r = 2'b0;
 #2;
 r = 2'b1;
end // initial begin
```

```
always @(r) begin
 if ($esp_const_one(r))
        $display($time, " r is real one binary %b", r);
 else if ($esp_const_zero(r))
        $display($time, " r is real zero binary %b", r);
 else if ($esp_const(r))
        $display($time, " r is real X or Z binary %b", r);
 if ($esp_contains(r[0],v[0]))
  $display($time, " r[0] contains v[0]");
 if ($esp_contains(r[1],v[0]))
  $display($time, " r[1] contains v[0]");
 if ($esp_contains(r[0],v[1]))
      $display($time, " r[0] contains v[1]");
 if ($esp_contains(r[1],v[1]))
  $display($time, " r[1] contains v[1]");
end // always @(r)
endmodule // test
```

SEE ALSO

Topics: **esp_addrvar** **esp_cmdvar** **esp_const** **esp_const_one** **esp_const_zero** **esp_ctrlvar** **esp_datavar** **esp_errc**

$esp_context

SYNTAX

$esp_context(top_level_signal);

DESCRIPTION

The $esp_context system task is used with **$esp_error** . $esp_context records the context for the next call to **$esp_error** . This information is used by the ESP Shell to report the cause of the error in the **report_failing_points** command.

The context is the name of a testbench pin for automated testbenches and the name of a top level signal for the simulation only flow.

EXAMPLES

SEE ALSO

Commands: **report_failing_points**

Topics: **$esp_error**

$esp_ctrlvar

SYNTAX

$esp_ctrlvar (list of symbols);

DESCRIPTION

The $esp_ctrlvar system task is used in lieu of the **$esp_var** system task to identify to ESP that the symbols listed control the state of the circuit being tested. In general, control variables need to be pushed to the top of the ESP internal variable ordering so that minimal equations are produced.

What makes a variable a control variable? It is easiest to determine this from the behavioral code. If a signal appears in an if statement or case construct, it is most likely a control signal. Write enables, read enables, output enables, and tristate control signals are examples of control variables.

This system task is most often used in conjunction with **$esp_addrvar** and **$esp_datavar** and replaces **$esp_var** .

EXAMPLES

SEE ALSO

Topics: **esp_addrvar** , **esp_cmdvar** , **esp_const** , **esp_const_one** , **esp_const_zero** , **esp_contains** , **esp_dat**

$esp_datavar

SYNTAX

$esp_datavar (list of symbols);

DESCRIPTION

The $esp_datavar system task instructs ESP to put the lowest priority on the symbols listed for internal ordering. Data variables typically have minimal effect on the capacity of ESP. By identifying these variables as data variables, ESP can prioritize the control and address variables and stretch the symbolic capacity.

This system task is most often used in conjunction with **$esp_ctrlvar** and **$esp_addrvar** and replaces **$esp_var** .

EXAMPLES

SEE ALSO

`Topics:` **esp_addrvar** , **esp_cmdvar** , **esp_const** , **esp_const_one** , **esp_const_zero** , **esp_contains** , **esp_ctr**

$esp_drive_zero_delay

SYNTAX

```
$esp_drive_zero_delay ();
```

DESCRIPTION

Internal use for library cell verification.

EXAMPLES

SEE ALSO

Topics: esp_addrvar esp_cmdvar esp_const_one esp_const_zero esp_contains esp_ctrlvar esp_datavar esp_e

$esp_equation

SYNTAX

```
$esp_equation (expr [, size [, mask]]);
```

DESCRIPTION

The $esp_equation system task returns a string that is the sum-of-products form of the equation that currently is the value *expr*.

expr is a Verilog expression that returns a value. Normally, this would be a wire, a net or a register.

As many as four equations can be part of the string that is returned. These equations are for getting a value of 1, 0, X or Z from *expr*.

The equations will be truncated if they are too long to display. By default this is after 20 product terms have been written. Truncated equations end with "...".

size specifies the number of terms that are output. This parameter overrides the system wide default of 20 product terms per equation.

mask specifies which equations to output. The default value of *mask* is 15. The possible legal values for mask are:

Value	Interpretation
1	Equations: 0
2	Equations: 1
3	Equations: 0,1
4	Equations: X
5	Equations: 0,X
6	Equations: 1,X
7	Equations: 0,1,X
8	Equations: Z
9	Equations: 0,Z
10	Equations: 1,Z

11	Equations: 0,1,Z
12	Equations: X,Z
13	Equations: 0,X,Z
14	Equations: 1,X,Z
15	Equations: 0,1,X,Z

EXAMPLES

Display the equation for DOUT.

```
    $display("%s",$esp_equation(DOUT));
```

Creates output similar to:

```
{
1: ~test.A_sym1&&test.B_sym1
0: test.A_sym1&&test.B_sym1
x: test.A_sym1&&~test.B_sym1
z: ~test.A_sym1&&~test.B_sym1
}
```

SEE ALSO

Topic:
esp_equation_size
esp_equation_symbols

$esp_equation_size

SYNTAX

$esp_equation_size (expr);

DESCRIPTION

The $esp_equation_size system task returns an integer that is the maximum number of terms in the sum-of-products form for all of the equations (0,1,Z,x) used to represent the value of *expr*.

expr is a Verilog expression that returns a value. Normally, this would be a wire, a net or a register.

EXAMPLES

SEE ALSO

Topics: **esp_equation** , **esp_equation_symbols**

$esp_equation_symbols

SYNTAX

$esp_equation_symbols (expr);

DESCRIPTION

The $esp_equation_symbols system task returns a string that is the symbols used in the equation that currently is the value *expr*.

expr is a Verilog expression that returns a value. Normally, this would be a wire, a net or a register.

EXAMPLES

Display the symbols in the equation for DOUT.

```
$display("%s",$esp_equation_symbols(DOUT));
```

The output is similar to:

```
{
test.A_sym1 test.B_sym1
}
```

SEE ALSO

Topics: **esp_equation** **esp_equation_size**

$esp_error

SYNTAX

$esp_error ("error string");

DESCRIPTION

When simulation reaches the $esp_error system task, it prints out the string specified as an argument.

ESP keeps track of the number of times this system task is executed. If the number of times executed is less than or equal to the **verify_max_number_error_vectors** variable then counter-example is also created. The first counter-example is provided in the `esp.TestVector` file and contains the binary values necessary to reach the $esp_error statement. To use counter-examples other than the first, use the -*vector_num* option on the **debug_design** , **start_explore** or **write_spice_debug_testbench** commands.

EXAMPLES

SEE ALSO

`Commands:` **debug_design** **start_explore** **write_spice_debug_testbench**

`Topics:` **esp_addrvar** **esp_cmdvar** **esp_const** **esp_const_one** **esp_const_zero** **esp_contains** **esp_ctrlvar** **esp_dat**

$esp_exclcovscope

SYNTAX

$esp_exclcovscope(level, instancepath);

level	Number of levels to record. Legal values: 0 tp 2,147,483,647 Default: 0
instancepath	Path to instance to exclude

DESCRIPTION

If ESP should not record coverage data for one or more instances, use the $esp_exclcovscope() option. This is useful when you are generating a small coverage database. It enhances debugging the coverage report by excluding unwanted instances. Additionally, you can include specific instances for coverage recording within an excluded instance.

To exclude scope features, specify $esp_exclcovscope() in the testbench. To exclude scope feature on a specific instance but include one or more of its children, specify $esp_exclcovscope() on the instance that you want to exclude followed by $espcovscope() on the child instances you that want to include. The level parameter is the number of levels of exclusion. The instancepath is the hierarchical instance name from which you do not want ESP Cov to record coverage data.

EXAMPLES

For example, to exclude a particular instance within a top-level instance, you would use the following:

```
$espcovscope (0,ximp); //dump coverage data for all design
$esp_exclcovscope(0,ximp.XI1) //do not generate coverage
//data for instance XI1 and
//all of its children
$esp_exclcovscope(2,ximp.A) //exclude only the instance
//"ximp.A" and its immediate
//children
```

To exclude a particular instance but generate coverage data for one of its child instances, you can use

```
$espcovscope (0,ximp); //dump coverage data for all design
$esp_exclcovscope(0,ximp.XI1) //do not generate coverage
//data for instance XI1 and
//all of its children
$espcovscope(0,ximp.XI1.XA1) //exclude all instances of
//ximp.XI1 but generate
//coverage for ximp.Xi1.XA1
```

Consider the following items when merging two coverage databases generated from two different testbenches involving the $excl_exclcovscope() system task.

- If you attempt to merge two coverage databases from two different testbenches, one having $esp_exclcovscope() and the other not having it, **espcovmerge** issues a warning message similar to the following and exits:

```
ESPCOV Warning:
 Warning: file testbench_file dropped from the
   merged database due to inconsistent sizes for the
   existing file.
```

- As a general rule, you must use identical exclude scopes in simulation runs where you want to merge coverage data. For more information about merging coverage data, see **$espcovscope** .

SEE ALSO

Topics: **espcovfile**
espcovoff
espcovon
espcovscope

$esp_exclusive

SYNTAX

$esp_exclusive (bus1, bus2, bus3...);

DESCRIPTION

The $esp_exclusive task ensures that the buses specified have unique values. It is typically used when there are multiple addresses that can be specified (possibly on multiple ports) and the addresses must remain different from one another.

Verilog code can accomplish this same function. However, the Verilog code produces input that is complex. That complexity typically goes into a design where it finds further complexity. The advantage of this function is that the constraint is met, but it is done internally and more efficiently from a symbolic capacity standpoint.

EXAMPLES

SEE ALSO

Topics: esp_addrvar , esp_cmdvar , esp_const , esp_const_one , esp_const_zero , esp_contains , esp_ctr

$esp_gen

SYNTAX

$esp_gen ("file","Heading string");

DESCRIPTION

ESP creates a file containing a counter-example the first time the **$esp_error** statement is executed. This file specifies a binary value for all the symbols that would reproduce the condition that causes the **$esp_error** statement to be executed. Simulation is then switched to run in binary mode to complete the simulation.

The $esp_gen statement is similar, but it creates a file and continues execution in symbolic mode. This allows multiple test vector files to be generated. Each file captures one set of symbol values that would cause the simulation to execute the $esp_gen statement that generated the file.

The heading string in the second parameter is placed as a comment in the first line of the file generated.

If the statement is executed multiple times, only the first occurrence generates a test vector.

To use one of the files generated by $esp_gen, copy the *esp.TestVector* file and then run the simulation in binary mode using the -b command-line option.

EXAMPLES

For example,

```
if (state1 === 1'b1)
  $esp_gen("state1.tv", "Execute State 1");
```

When the state1 variable gets a 1 value and the $esp_gen statement is executed, the values of all symbols that caused this condition are output to the state1.tv file. Only the first set of symbol values are output. All other possible values are ignored and further executions of this statement are skipped.

SEE ALSO

Topics: **esp_addrvar** , **esp_cmdvar** , **esp_const** , **esp_const_one** , **esp_const_zero** , **esp_contains** , **esp_ctr**

$esp_multi_select

SYNTAX

$esp_multi_select (0|1|2);

DESCRIPTION

The $esp_multi_select system task is an aid for debugging switch-level netlists by locating Xs in a design and attempting to determine the cause of the unknown. The argument specifies to the task what type of X to chase.

0: Looks for hard conflicts, where there are two drivers on a net, one driving to 0, the other driving to 1 to produce the X.
1: Looks for hard conflicts (covered by 0), where a hard value is conflicting with an X (0 on one driver X, on the other, or 1 on one driver, X on the other).
2: Looks for all causes of X, including hard conflicts (0), soft conflicts (1), and cases where both drivers are X.

Option 2 gives the most information, but it is often difficult to see the cause of the X through all the data. Option 0 points to hard errors that are often related to strength-modeling.

To avoid printing too much information, $esp_multi_select prints only the first 10 unique conflicts that meet the criteria set by the argument. Conflicts beyond 10 unique ones are ignored.

EXAMPLES

SEE ALSO

```
Topics:
esp_addrvar
esp_cmdvar
esp_const
esp_const_one
esp_const_zero
esp_contains
esp_ctrlvar
esp_datavar
esp_error
esp_exclusive
esp_gen
esp_onehotindex
esp_retire
esp_smminit
esp_timevar
esp_var
```

$esp_onehotindex

SYNTAX

$esp_onehotindex (wordlinebus);

DESCRIPTION

The $esp_onehotindex system task is used for memory core modeling within reference designs. This system task is a function that takes a one-hot word line bus and returns an integer identifying what address was being used.

EXAMPLES

For example

```
i = $esp_onehotindex(bus);
```

The previous statement is interpreted as follows:

If bus contains any Z or X, the return value is X. else if bus isn't one hot, the return value is X. else if bus is all 0, the return value is 0. else return value is the index of the (only) 1 in the bus. The indexes are 1, 2, ...

For example:

```
$esp_onehotindex(5'b00001) returns 32'd1
$esp_onehotindex(5'b00100) returns 32'd3
$esp_onehotindex(5'b00000) returns 32'd0
$esp_onehotindex(5'b01010) returns 32'bx
$esp_onehotindex(5'b000z0) returns 32'bx
$esp_onehotindex(5'b00x00) returns 32'bx
```

SEE ALSO

Topics: esp_addrvar , esp_cmdvar , esp_const , esp_const_one , esp_const_zero , esp_contains , esp_ctr

$esp_retire

SYNTAX

$esp_retire (list of symbols);

DESCRIPTION

The $esp_retire system task removes all references of the variables listed from the equations stored in ESP. You should use it only when you know for certain that a symbol is no longer needed.

For example, a symbol might not be needed anymore for a pipelined datapath test. If a design has latches or flip-flops to hold the state, a symbol is no longer needed after it has been propagated through the datapath block. By calling $esp_retire, some overhead internal to ESP can be cleaned up to increase the capacity.

EXAMPLES

Remove the reset symbols after time 10000.

```
initial begin
  #10000;
  $esp_retire(reset_SYM1, reset_SYM2, reset_SYM3);
end
```

SEE ALSO

Topics: esp_addrvar , esp_cmdvar , esp_const , esp_const_one , esp_const_zero , esp_contains , esp_ctr

ESP Shell Flow

Contents

Overview
Compare Flow
Power Integrity Flow
Simulate Only Flow
Cell Library Flow
Redundancy Verification
Verify more than one design
Verilog to Verilog Verification
SPICE to SPICE Verification

DESCRIPTION

Overview

The ESP tool is a symbolic simulator that can be used to verify designs written in Verilog and SPICE.

The **esp_shell** command borrows concepts and commands from the Synopsys PrimeTime and Formality tools. These concepts and commands have been adapted to the unique needs of the ESP tool verification flow. Users familiar with other Synopsys Tcl shell tools should quickly become productive with **esp_shell** .

Two key organizational concepts in **esp_shell** are:

- Containers - Hold design objects
- Collections - Hold collections of complex data objects.

Each of these concepts are described in more detail later in this manual page.

Redirecting command output

The **redirect** command performs the same function as the traditional Unix-style redirection operators > and >>. The command string must be rigidly quoted (that is, enclosed in curly braces) in order for the operation to succeed.

Command naming conventions

The ESP tool Tcl commands for **esp_shell** follow a verb-noun-modifier format. Several of the noun forms may appear in singular or plural forms.

The primary verbs are:

- check
- create
- debug
- get
- is
- match
- print
- read
- remove
- report
- set
- start
- stop

The primary nouns are:

- clock
- container
- design
- device
- instance
- net
- pin
- port
- testbench

The exceptions to this naming convention are the current_* commands:

- **current_container**
- **current_design**
- **current_instance**

Containers

esp_shell supports two separate container objects. Containers hold design data for a specific design. The two containers are called the reference and implementation containers.

esp_shell tracks the default (current) container. When **esp_shell** is first started the current container is the reference container.

Many **esp_shell** commands operate on one or both containers. Commands that operate on only one container will operate on the "current container" unless the *-i* or *-r* command switch is used.

Each container keeps track of:

- The design data
- The top design name
- The current design
- The current instance

The general idea within the ESP tool is that the reference container is compared with the implementation container to prove equivalent functionality. To accomplish this a testbench can be automatically created. This testbench runs a symbolic simulation to prove that all possible inputs will produce the same results in the implementation container as the reference container. In this flow, the reference container is considered to be the "correct" design.

Using **esp_shell** for a verification flow with only one container is supported with the "simulate" mode (see **set_verify_mode**. Automatic testbenches cannot be created for this verification flow style.

Collections

Collections are a Tcl data type that hold information about various objects in the **esp_shell** database. Some of these objects include:

- matched_design
- device
- design
- instance
- net
- pin
- port

Each of these objects has a set of attributes that have values. See the **collections** man page for more information about collections and the commands that operate on them.

The get_* commands return collections.

Collections are useful in writing advanced scripts. There are collections for matched designs, container design information and debug explore results.

DESIGN LANGUAGES

The ESP tool supports:

- Verilog 1995 with some 2001 extensions. Refer to **faq_verilog_unsupported** for a list of Verilog language features that are not supported.
- A very small subset of SystemVerilog 2012. The supported subset is limited to design constructs and assertion support. Refer to **faq_SystemVerilog** for details on the supported subset of SystemVerilog.
- A SPICE netlist with a sub-circuit for the top level of the design. The ESP tool does not support the top level of the design outside of a .SUBCKT . Refer to **faq_SPICE** for details on the supported subset of SPICE.
- Reading a design from a Synopsys Library Compiler internal database. The **read_db** command is used to read the internal database into the reference container. This is only supported in the Cell Library Flow.

Reference vs. Implementation

Both the reference container and implementation container can contain Verilog designs and SPICE netlists. If Verilog source files are read, the container is a Verilog container and all top designs must be in the

Verilog design. If there is no Verilog designs read and a SPICE netlist is read, then the container is a SPICE container and all top designs must be in the SPICE netlist. Reading both Verilog and SPICE into a single container is not supported.

Verification Flow

The ESP tool supports three verification flows: compare, power integrity and simulation-only.

The compare flow consists of a reference design and an implementation design. The ESP tool can build an automated testbench that will compare the two implementations. Multiple testbenches are supported in compare mode.

The compare flow supports verification of Verilog to SPICE, Verilog to Verilog, and SPICE to SPICE.

The power integrity flow is similar to the compare flow but it adds analysis and reporting of power integrity. This flow only makes sense to compare Verilog to SPICE netlists.

The simulation-only flow consists of a reference design. The verification engineer or designer must supply a testbench. This flow operates like a traditional simulator such as VCS. Only a single testbench is supported in simulation-only flow.

The compare and power integrity flows use two design containers called reference and implementation. Commands that work on data in a container have the switches *-r* and *-i*. Only one of these switches can be used on a command at a time.

The ESP tool also supports the concept of a current container which is set with the **current_container** command. By default the current container is the reference container. When you want to operate on the current container you do not have to specify the *-r* or *-i* container switch. However, it is a best practice to always specify the *-r* or *-i* container switch on commands in a script.

The compare flow is the default flow. To change the verification mode use the **set_verify_mode** command.

In all the flows you need read in design files in Verilog or SPICE format. You also need to specify the top module or subcircuit for the design with the **set_top_design** command.

Read Verilog design

The **read_verilog** command is used to read Verilog source files into a container. The *-r* or *-i* switches are used to select the reference and implementation containers.

Only one **read_verilog** command per container can be used in a verification session. Use **reset** to start a new verification session.

To read more than one Verilog source file, do one of the following:

- Specify the files on the command as a list

```
read_verilog -r [list file1.v file2.v file3.v]
```

- Specify the files in a list file and use the *-f* option:

```
        read_verilog -r -f files.f
```

Read SPICE design

To simulate SPICE designs the ESP tool has to understand the various device models that are used in the design. The ESP tool has default models for transistors, diodes, resistors and capacitors. The ESP tool default transistor model includes support for FinFet technologies.

The **set_process** command is used to specify models for transistors. Use the *-technology_file* option to provide the ESP device file (EDM) that defines the transistor models. **Best Practice** Use of a device model file is the preferred way to specify the transistor models for designs that are at 28nm or smaller.

The first character on a line in a SPICE netlist defines the time of command that is being used on the line. Lines that start with M,D,R or C are instantiating transistor, diode, resistor or capacitor devices in the netlist.

For transistor devices there is a specific device model later in the line. By default if that model name starts with the letter n or N the ESP tool will assume that the device is an NMOS transistor. If the model name starts with the letter p or P the ESP tool will assume that the device is a PMOS transistor. If the first character on a line in a SPICE netlist is an X then the line is instantiating a macro model. Many SPICE netlists use macro models for the transistor devices. These macro models are not actually part of the SPICE netlist but are instead part of the device library. The ESP tool will try and match any macro model lines with known transistor models if there is no corresponding subcircuit definition in the SPICE netlist. Effectvely, the ESP tool treats the X lines as if they start with M when there is no subcircuit with the same name present in the SPICE netlist. This behavior is the same as HSPICE when *.OPTION MACMOD=3* is specified.

The **netlist_bus_extraction_style** variable controls the conversion of SPICE nets into buses. The default value is "%s<%d>". This must be set before reading in SPICE source files.

The **read_spice** command is used to read SPICE source files into the implementation container.

Only one **read_spice** command per container can be used in a verification session. Use **reset** to start a new verification session.

The **get_designs** command is used to display a list of the designs (SPICE sub-circuits) that have been read with the **read_spice** command.

The **set_top_design** command is used to specify the top design (i.e. the top SPICE sub-circuit) in the implementation container.

Supplies must be configured when a SPICE netlist is in a container. The ESP tool only knows about three special supply nets. These are gnd, !gnd and 0. All three nets are always real supplies and have a logic value of 0. All other supplies must be explicitly defined. Supplies in a Verilog design cannot be defined. The tool will issue an error message if the set_supply_net command is used on a container that does not have a SPICE netlist.

In the tool, a supply is either of type real or type virtual. When supplies are defined, the tool can use some performance optimizations. Without any supply definitions, verification will run slowly.

Real supplies have a fixed logic value and will not change logic value during verification. Real supplies cannot be constrained.

Virtual supplies have an on logic value and can change logic value during verification. Virtual supplies can be constrained.

Defining Supply Nets

To define all nets that start with VDD as a real power supply regardless of case in the implementation container, use the following example

```
  > set_supply_net -i (?)vdd.* -type real -logic 1
```

Defining Supply Nets

Defining Subcircuit-Based Supplies

To define VDD25 as a virtual supply within the decode subcircuit use:

```
 > set_supply_net -i VDD25 -type virtual -logic 1 -design decode
```

For more details, see the manual page for the set_supply_net command.

Finding Supplies in a Design

The report_potential_supply command can be used to find potential supplies in a design. Use the report_potential_supply command to get a report of the number of connections on all the nets within a design. A net that has a high number of connections could be a supply. A net that has a high number of connections could also be a clock net, a reset net or a bit-line.

Example for the RAM1R1W sample design.

```
 > read_spice -i ram.sp
 > report_potential_supply -i
****************************************
Report : report_potential_supply
        -i
Version: M-2016.12
Date   : Fri Oct 14 15:18:00 2016
****************************************

  NET               CONNECTIONS  MAJORITY  BULK  SETTING    DESIGN
  ----------------------------------------------------------------------
  DVDD              22           pmos      pmos             INV_D1
  DVSS              22           nmos      nmos             INV_D1
  DVDD              22           pmos      pmos             INV_D2
  DVSS              22           nmos      nmos             INV_D2
  ----------------------------------------------------------------------
  Summary:
  Median connection_count: 3
  Maximum connection_count: 22
  Reporting only candidates with connection_count >= 14
1
```

Any net that connects to a bulk terminal is a good candidate as a supply net. Connections to a P type device are typically a logic 1 or power net. Connections to an N type device are typically a logic 0 or ground net. In the report DVDD is a power net and DVSS is a ground net. The following supply definitions are recommended:

```
 > set_supply_net -i DVDD -logic 1 -type real
 > set_supply_net -i DVSS -logic 0 -type real
```

For more details, see the manual page for the report_potential_supply command.

Compare Flow

The compare flow is the default flow in the ESP tool. To ensure that compare flow is active issue the following command:

```
> set_verify_mode compare
```

The recommended compare flow in **esp_shell** is:

1. Read the reference design into the reference container.
2. Read the implementation design into the implementation container.
3. Match designs between the reference and implementation containers.
4. Match ports between the reference and implementation of matched designs.
5. Define the clock
6. Configure the SPICE design information
7. Configure the testbench
8. Generate/write testbenches
9. Run verification
10. Debug verification failures

Each of the above steps consists of smaller steps defined in the following subsections.

The prompt in **esp_shell** helps to indicate what part of the recommended flow is currently active. The defined modes (and their prompts) are:

Mode	**Prompt**
Setup	esp_shell (setup)>
Match	esp_shell (match)>
Configure	esp_shell (config)>
Verify	esp_shell (verify)>
Debug	esp_shell (explore)>

Refer to **verilog_to_verilog** for information on using the compare flow for Verilog to Verilog comparisons.

Refer to **spice_to_spice** for information on using the compare flow for SPICE to SPICE comparisons.

Refer to **redundancy** for information on using the compare flow for redundancy verification.

The compare flow supports library cell verification. Refer to **library_verification** for information on using the compare flow for library cell verification.

Matching Designs

Match designs allows the user to define multiple designs that are to be verified. It is used mostly by **library_verification** but is applicable to non-library compare and power integrity verification as well.

Matching of designs occurs before port matching. If design matching is not done, an implicit design match occurs for the current top design.

The command **match_designs** automatically matches designs in the reference and implementation container based on matching names. The **set_matched_designs** command creates a manual match of designs which may not match by name.

The **report_matched_designs** and **report_unmatched_designs** commands create reports of the matched and unmatched designs.

The **remove_matched_designs** command is used to remove a design match that has already been established by the automatic or manual matching methods.

Most commands have the option -matched_designs to specify one or more matched designs which the command operates on. The collections command **get_matched_designs** returns a collection of matched designs which can be used with the -matched_designs option on commands.

The **set_top_design** command with the -matched_designs option, identifies a matched design from which the top level reference module and top level implementation module are set as the top designs.

Configure SPICE

The following commands are related to SPICE modeling within the ESP tool:

- **get_devices**
- **remove_supply_net**
- **report_design_spice_mode**
- **report_devices**
- **report_process**
- **report_supply_net**
- **reset_design_spice_mode**
- **reset_device_model**
- **reset_process**
- **set_design_spice_mode**
- **set_device_parameter_multiplier**
- **set_instance_delay**
- **set_instance_spice_mode**
- **set_instance_strength_multiplier**
- **set_model_suffix**
- **set_net_initial_value**
- **set_net_no_decay**
- **set_process**
- **set_rcdecay_time**
- **set_supply_net**
- **set_transistor_type_autodetect**

The following variables are related to SPICE modeling within the ESP tool:

- **flattening_device_count**
- **mos_transconductance_ratio**
- **netlist_aggressive_net_compression**
- **netlist_aggressive_port_compression**
- **netlist_bus_extraction_style**
- **netlist_inverter_chain_extraction**
- **netlist_unwrap_transistors**
- **netlist_use_verilog_escape**
- **threshold_high_impedance_resistance**
- **threshold_warn_big_rc_delay**

- **verify_delay_round_multiple**

Matching ports

The ports must match between the reference container and the implementation container in order to set port attributes and then use the command **write_testbench** .

The command **match_design_ports** will automatically match the ports between the reference container and the implementation container. If **match_design_ports** fails, then manual matching can be used to get a complete mapping of the ports.

The **report_matched_ports** and **report_unmatched_ports** commands create reports of the matched and unmatched ports.

The **set_matched_ports** command creates an manual matching of ports between the reference container and the implementation container. This command is especially useful to resolve issues with bit ordering on buses.

The **remove_matched_ports** command is used to remove a port match that has already been established by the automatic or manual matching methods.

Define the clock

At least one clock should be defined for the testbench.

The **create_clock** command is used to define a clock for the testbench.

Support for more than one clock is limited for the automatically created testbenches. All clocks will be synchronous and either in-phase or 180 degrees out of phase.

Configure testbench

Configuring the specific testbench that gets created involves setting several Tcl variables or setting matched design attributes for each design. The following Tcl variables influence the testbench that will be written by the **write_testbench** command:

- **testbench_binary_cycles** Default: *2*)
- **testbench_constraint_file** Default: "")
- **testbench_dump_symbolic_waveform** Default: *false*)
- **testbench_flush_cycles** Default: *1*)
- **testbench_initialization_file** Default: "")
- **testbench_symbolic_cycles** Default: *3*)
- **testbench_implementation_instance** Default: *ximp*)
- **testbench_module_name** Default: *inno_tb_top*)
- **testbench_output_checks** Default: *3check*)
- **testbench_reference_instance** Default: *xref*)
- **testbench_style** Default: *symbolic*)

The **set_matched_design_attributes** command allows setting testbench attributes on one or more designs during setup and then uses this information when generating the testbench for each of the designs. This command has the most application to **library_verification** but is applicable to non-library compare and power integrity verification as well.

The matched design attributes options available are:

- -constraint specifies a constraint file to be used for a design
- -declaration specifies a declaration file to be used for a design
- -setup specifies a setup file to be used for a design
- -verilogdrive to specify the library verification Verilog drive strength.
- -channellength to specify size of library verification SPICE testbench drivers
- -indrive to specify strength of library verification SPICE testbench drivers
- -outload to specify load of library verification SPICE netlist outputs
- -allow specifies the default allowed input values on all inputs
- -checker specifies the default checker to be used on all outputs
- -weak_threshold specifies the weak threshold for eqw output checker
- -checkmode specifies library verification output checker mode
- -initialization specifies how library verification does initialization
- -vectormode specifies library verification input vector type
- -iomode specifies library verification handling of bidirectional ports

Creating and using testbenches

Testbenches can be generated or created manually.

The **write_testbench** command will generate a set of testbenches. This command will also set the list of active testbenches to a list of the generated testbenches.

The **set_testbench** command can be used to read in custom testbenches instead of having the ESP tool generate a testbench. The user is responsible for ensuring that all the design inputs are stimulated correctly.

The **list_active_testbench** command returns a list of the current active testbenches.

Running Verification

The **verify** command runs the testbenches in the order specified by the active testbench list. If a testbench fails, verification is halted.

Note:
Change the ESP tool default delay rounding to 1ps for technology nodes below 45nm. Use the following command before **verify** :

```
set_app_var verify_delay_round_multiple 1
```

The following commands can be used to modify the list of known and active testbenches:

- **list_active_testbench**
- **remove_testbench**
- **set_active_testbench**
- **set_testbench**
- **write_testbench**

The **verify** command automatically runs **check_design** as a quick sanity check on the testbenches to be verified.

The **report_status** command can be used to get a quick summary of the verification run.

The **report_log** command displays the last simulation log file. Switches can be used to get the log file from a specific testbench.

Code coverage is controlled by the **coverage** variable. By default **coverage** is *false* and code coverage is not run.

Debugging Failures

Most verification failures will be debugged using *binary* simulation mode. After the **verify** command finds a failure the failure can be analyzed: using traditional Verilog debugging techniques or **interactive signal tracing (IST)** mode.

Traditional Verilog techniques use a waveform viewer. Use the **debug_design** command to get a waveform file. Use the **start_waveform_viewer** command to start the preferred waveform viewer.

Set the **waveform_viewer** variable to control which waveform viewer **start_waveform_viewer** will start. The default value is "vfast %v; verdi %s -ssy -ssv -ssz -ssf %v.fsdb".

Interactive Signal Tracing (IST) is a debug mode unique to the ESP tool. **IST** mode allows the interactive exploration of the SPICE parts of a design.

See the **IST** man page for more information about **IST** mode and the commands that can be used in **IST** mode.

The SPICE language portion of the design can be verified in a circuit simulator with the help of the **write_spice_debug_testbench** , **export_spice_debug_testbench** and **start_spice_simulator** commands.

Power Integrity Flow

To enter power integrity mode issue the following command:

```
> set_verify_mode inspector
```

Follow instructions for compare flow with the following changes:

After matching has been completed,

- specify all power domains with **set_supply_net_pattern** and **set_power_domain**
- specify additional constraints with **set_constraint**
- specify power integrity checkers with **set_inspector_rules**

After **verify** , report results with **report_inspector_results** .

For more details see the topic Power Integrity Flow

Simulate-Only Flow

Refer to the ESP Shell Help User Guide for detailed information on using Simulation-Only Flow.

To enter simulation mode issue the following command:

```
 > set_verify_mode simulate
```

Only one **read_verilog** or **read_spice** command can be used in simulate mode.

To read more than one Verilog source file, create a file that is a list of all the source files and any switches that are needed. This file is much like the command file you would give VCS. The ESP tool supports a small subset of the VCS switches. If the file is called file.list then you can use:

```
> read_verilog -r -f file.list
```

The simulate mode assumes that you provide all the needed Verilog files. You should provide simulation files that produce any needed analysis information. Waveform files such as VCD are not created unless the Verilog files include calls to dumping tasks such as $dumpvar().

Methodology

1. Execute **set_verify_mode simulate** .
2. Ensure that your testbench provides the expected verification results and incorporates the ESP tool specific system functions to generate symbols and check the expected symbolic equations. See **output_checkers** , **system_tasks_in_esp**
3. Use the **read_verilog** command to read your design files and testbench file. In this flow you specify all the Verilog source files to be simulated using the **read_verilog** command. You can specify every file used in the simulation in the Verilog command option file *vfile* or you can specify a list of files on the **read_verilog** command. Include the testbench file in the list of files.
4. Specify your top design. Use the **set_top_design** command to identify the top module of the design being tested. If you do not set your top design, the tool uses the first non-referenced module name as the top level design name. In order to obtain coverage data, you must set the top level design correctly. This should not be the testbench top module name, but should be the top design instantiated by the testbench. This design name will be used in the work directory to store results and it will be used in the report headers. You can only set the top design for one container and it must be the reference container.
5. To enable coverage checking and reporting, set the **coverage** variable to on.
6. Set your testbench module name using the **testbench_module_name** variable. If you do not specify a testbench module name, you will get a compilation error if either waveform dumping or coverage is enabled.
7. Set your testbench instance name using the **testbench_design_instance** variable. If you do not specify a testbench name, the tool assigns a default testbench id of tb0 and identifies all internal result files with the testbench id of tb0. The testbench instance name can be a hierarchical path from the top level testbench to a lower level instance. For example, you could set the variable to the value *x1.x15.ref* . The **testbench_design_instance** variable is defined as an empty string. If the variable has a non empty string value, this will be interpreted as an instance name. The specified instance name will be used to gather symbolic coverage data. If it has no value, the Verilog source code will be searched to find the instance name for the design specified by the **set_top_design** command.
8. Verify the design. In simulate mode, the **verify** Tcl command runs symbolic simulation using just the reference container. This command assumes that everything is in one container. When coverage is requested, the path to the design being tested must be known. If no value can be found for the top level design instance name and the **testbench_design_instance** variable is not found, an error is returned.
9. Review the **report_status** report. The **report_status** command must be able to report errors generated by custom testbenches. The errors that it can detect and report depend on how you implement your error checking in the Verilog source code. See **output_checkers** .

10. Review the **report_log** report.
11. Review the **report_coverage** report. For an example report, see **coverage_reports** The Tcl shell adds symbolic coverage during the verify command when the **coverage** variable is set to the value *on*. To allow the Tcl shell to automatically gather symbolic coverage data, do the following:
    1. Specify the top level testbench name using the Tcl variable **testbench_module_name** .
12. Run the **debug_design** command, if needed. In simulate mode, the **debug_design** command uses only one container when generating the VCD dump files. The command uses Tcl variables to determine what the top level testbench name is for dumping purposes.

Coverage

In order to obtain proper code coverage data in the simulation-only flow, the **set_top_design** command must be used to set the name of the module that is the top of the design. This should not be the module that is the testbench. By default the ESP tool will use the first module it finds that is not referenced by any other module as the top design. The ESP tool default is rarely the correct top design for collecting code coverage statistics.

$esp_smminit

SYNTAX

$esp_smminit ("memory_name", "value");

memory_name	Quoted string that is the instance path to a memory
value	Quoted string that is the value to write to all memory locations. One of: "0", "1", "X", "Z", "ID"

DESCRIPTION

To speed up some Verilog simulations, users often choose to initialize a memory into a known state. Because ESP uses a sparse memory model to represent large memories, initializing the memory with a large for loop is inefficient and slow.

The $esp_smminit system task efficiently loads a memory with the value of 0, 1, X, Z, or its ID number. ID number means that the i'th location is given a value of i.

EXAMPLES

SEE ALSO

```
Topics:
esp_addrvar
esp_cmdvar
esp_const
esp_const_one
```

- esp_const_zero
- esp_contains
- esp_ctrlvar
- esp_datavar
- esp_error
- esp_exclusive
- esp_gen
- esp_multi_select
- esp_onehotindex
- esp_retire
- esp_timevar
- esp_var

$esp_timevar

SYNTAX

$esp_timevar (list of symbols);

DESCRIPTION

The $esp_timevar function is typically called at time 0 from an initial block to specify to ESP that a variable is being used to symbolically specify an amount of time. These variables have the most control over the internal equations and are given the highest priority in the ordering of the variables, so as to minimize capacity issues.

EXAMPLES

An example of a symbolic time variable is as follows:

```
for (I=0;I<timevar_sym;I=I+1) begin
   @posedge (clk);
   some action
end
```

What makes timevar_sym a symbolic time variable is that the length of time or number of posedges of this block is controlled by its value.

SEE ALSO

Topics: **esp_addrvar** , **esp_cmdvar** , **esp_const** , **esp_const_one** , **esp_const_zero** , **esp_contains** , **esp_ctr**

$espvar

SYNTAX

$esp_var (list of symbols);

DESCRIPTION

The $esp_var system task notifies ESP that the specified register is symbolic rather than a binary variable. It, coupled with $esp_error (described in **esp_error** , are the minimal set of changes necessary to make a testbench symbolic.

Variables that are specified in a $esp_var task take on both a 1 and 0 throughout the simulation by becoming symbols. These symbols are tracked across the design, eventually reaching checkers that verify whether the two designs (reference and implementation) are producing identical equations.

There is a limit on the total number of symbols that can be declared within a single simulation. If the total number of symbols declared by all the $esp_var and $esp_timevar, $esp_ctrlvar, $esp_addrvar, and $esp_datavar statements exceeds the limit, the ESP simulation randomly chooses binary values for each symbol over that limit before starting simulation. These random binary values are placed in the esp.Random file (see Figure 5-1). Note that every bit of a bus is counted as a separate symbol.

If too many variables are randomized, 10 warning messages are issued and the total number of symbolic variables randomized is also provided.

By default, ESP generates a testbench that already has the symbolic registers specified as arguments to $esp_var. You can write your own testbench instead of using ESP . In that case, the testbench needs to declare a set of registers for the symbols and then pass those registers into $esp_var to make them symbolic.

You can call $esp_var multiple times and from many locations, such as tasks, functions and named blocks. However, you should not assign a value to variables that are symbolic because it loses its symbolic characteristics. Use the $esp_var system task to assign a value to another variable. Local variables (defined inside of a named block) should not be arguments to $esp_var.

When you run ESP in binary mode (-b) or symbolic binary mode (-sb), $esp_var no longer declares symbolic variables. Instead, it looks for the *esp.TestVector* file, which is created by the $esp_error statement. The *esp.TestVector* file is read and all the symbolic values declared by all $esp_var statements are assigned a value from this file. If the file does not exist, a symbolic value is missing, or an extra symbol is defined, an error occurs and simulation is terminated.

EXAMPLES

SEE ALSO

Topics: esp_addrvar , esp_cmdvar , esp_const , esp_const_one , esp_const_zero , esp_contains , esp_ctr

$espcovfile

SYNTAX

$espcovfile(espcovfile [, listofcovtypes]) ;

```
  listofcovtypes := NS Nonsymbolic information
                    DS Dropped symbol data
                    SA Stuck at symbol/binary
                    LC Line coverage
```

DESCRIPTION

The $espcovfile statement specifies the file that the coverage information is to be written into. Optionally, it also specifies the type of coverage information that should be gathered. The default is to output all coverage data.

Specify only one $espcovfile. Otherwise a compile error occurs.

EXAMPLES

For example:

```
    $espcovfile("cov.db");
    $espcovfile("gotyou.covered", "DS" );
```

The first statement specifies that all coverage is to be written to the file cov.db. The second statement, provided it is in a different simulation, specifies that only the dropped symbol and nonsymbolic data are written to the *gotyou.covered* coverage database file.

SEE ALSO

Topics: **esp_exclcovscope** , **espcovoff** , **espcovon** , **espcovscope**

$espcovoff

NAME

$espcovoff

Turn off gathering of coverage statistics.

SYNTAX

$espcovoff ;

DESCRIPTION

Dynamically turns coverage recording off for stuck-at and symbolic data gathering. Line coverage data is not affected by this command.

$espcovoff and $espcovon can be used to check only at certain times. However, doing this can cause the coverage results to report inaccurate symbolic coverage results for dropped symbols and symbolic net coverage.

EXAMPLES

Turn coverage off until after a reset that bypasses normal chip operation procedures.

```
initial begin
  $espcovoff;
  // Use XMRs to force internal regs to reset
  // This is a fast reset that does not need
  // to wait the normal 23 clock cycles.
  // So we disable coverage to remove improper
```

```
    // coverage data.
    xref.I1.reset34 = 0;
    ximp.I1.reset34 = 0;
    xref.I1.I2.I123 = 0;
    ...
    reset = 0;
    #1;
    $espcovon;
  end
```

SEE ALSO

Topics: **esp_exclcovscope**
espcovfile
espcovon
espcovscope

$espcovoff/$espcovon

NAME

$espcovon

Turn on gathering of coverage statistics.

SYNTAX

$espcovon ;

DESCRIPTION

Dynamically turns coverage recording on for stuck-at and symbolic data gathering. Line coverage data is not affected by this command.

$espcovoff and $espcovon can be used to check only at certain times. However, doing this can cause the coverage results to report inaccurate symbolic coverage results for dropped symbols and symbolic net coverage.

EXAMPLES

Turn coverage off until after a reset that bypasses normal chip operation procedures.

```
initial begin
  $espcovoff;
  // Use XMRs to force internal regs to reset
  // This is a fast reset that does not need
  // to wait the normal 23 clock cycles.
  // So we disable coverage to remove improper
```

```
    // coverage data.
    xref.I1.reset34 = 0;
    ximp.I1.reset34 = 0;
    xref.I1.I2.I123 = 0;
    ...
    reset = 0;
    #1;
    $espcovon;
  end
```

SEE ALSO

Topics: **esp_exclcovscope**
espcovfile
espcovoff
espcovscope

$espcovscope

SYNTAX

$espcovscope(level, moduletorecord);

level	Number of levels to record. Legal values: 0 tp 2,147,483,647 Default: 0
moduletorecord	Starting module name

DESCRIPTION

The $espcovscope statement specifies the scope of coverage data to record. If no scope is specified, coverage data for the entire design is recorded. This statement works in the same way as the $dumpvars statement.

The *level* parameter specifies the number of levels to record. If the value is 0, all levels starting at the specified path and below are recorded. If a level of 2 is specified, only two levels are recorded, that is the specified scope and all its children. The expected value for most simulations is 0. Specifying anything other than 0 limits the data usefulness. A value larger than 0 should only be used if too much data is being collected.

Manually Specified Scopes

By default, when coverage is enabled the ESP tool will record coverage data for the reference and implementation designs. If you have created testbenches manually or if there is too much coverage data, you can manually specify your own coverage recording using $espcovscope.

To manually control coverage, you should add $espcovscope to your initialization file. Do not turn coverage on in the shell.

If overlapping scopes are present in two different testbenches, you can have problems merging the coverage data. Overlapping scope is defined to be the use of an instance path in one $espcovscope that causes coverage data to be gathered on nets that are being covered by an instance path used in another $espcovscope statement. Overlapping scope is not intended to happen in the normal operation and you are strongly encouraged not to use it.

In the following example, only the top three levels in one simulation are dumped, along with all the levels below one particular instance at the third level down in another simulation. The coverage results cannot be merged and the following error is issued:

Scopes in all files should not have ancestor-offspring relationships.

```
  $espcovscope(3,ximp);// In testbench 1
  $espcovscope(0,ximp.a.b); // In testbench 2
```

You must use identical scopes in simulation runs where you want to merge coverage data. In the aforementioned case, you could look at coverage data from separate runs, but you still cannot merge the results.

EXAMPLES

This example records coverage data for all levels of the design starting from the module top.

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

This example dumps all levels of the entire design.

```
  $espcovscope;
    or
  $espcovscope();
```

Dumps all levels starting at instance a.b.c .

```
  $espcovscope(0, a.b.c);
```

Dumps only the instance xref and its children and nothing below that.

```
$espcovscope(2,xref);
```

Dumps every level below the instance ximp

```
$espcovscope(0,ximp);
```

The typical usage of $espcovscope in a testbench is that the specified scope is the xref and ximp model instances, but not the testbench. This is shown by the following example:

```
  $espcovscope(0,ximp);
  $espcovscope(0,xref);
```

SEE ALSO

Topics: **esp_exclcovscope**
espcovfile
espcovoff
espcovon

exit

NAME

exit

Terminate the application.

SYNTAX

```
exit [exit_code]
```

ARGUMENTS

exit_code	Return code to the operating system. Default is 0.

DESCRIPTION

This command exits from the application. You have the option to specify a code to return to the operating system.

The shell reports the maximum memory used and the total CPU time used.

EXAMPLES

The following example exits the current session and returns the code 5 to the operating system. At a UNIX operating system prompt, verify the return code as shown.

```
shell> exit 5
Maximum memory usage for this session: 12 MB
CPU usage for this session: 0.01 seconds
% echo $status
5
```

SEE ALSO

Commands: **quit**

Frequently asked questions: SPICE

SPICE as a language

SPICE does not have an official language definition. There are many variations of SPICE.

CDL

CDL is a SPICE netlist variant used by physical verification and extraction tools. Most schematic systems can write this variant of SPICE netlist.

One of the key extensions in CDL is the use of *.PININFO comments to declare the direction of each port. In regular SPICE ports are always bidirectional.

ESP and the SPICE language

The ESP tool works with a subset of the SPICE language that is used by the HSPICE tool.

The ESP tool works with digital SPICE netlists. Analog or RF effects are not part of the simplified model of the SPICE netlist used by the ESP tool.

General Issues

Handling of current

The ESP tool does not simulate current within the netlist.

Analysis modes

The ESP tool is not a SPICE simulator. The ESP tool does not support the analysis modes available in most SPICE simulators such as HSPICE.

Supported Features

Supported SPICE device models

The ESP tool supports the following device models:

Element	Description	Comment
D	diode	
C	capacitor	
M	MOSFET transistor	**Use device model simulation**
R	resistor	
X	subcircuit or macro model	

Supported SPICE commands

The ESP tool accepts and uses the following list of commands:

Command	Description	Comment
.global	global net	ignore
.include	include file	
.param[eter]	parameter definition	
.subckt	sub-circuit or macro circuit definition	
.ends	end of .subckt section	
.option MACMOD	macro model handling for MOS devices	ignored. The ESP tool uses .option MACMOD=3.

Supported Other

Unsupported Features

Any element, command or option not explicitly shown as supported above is not supported by the ESP tool. This section lists many of the known elements, commands or options that are not supported by the ESP tool.

Unsupported SPICE device models

The ESP tool does not support sources, inductors, BJT transistors and other elements as listed in this table:

Element	Description	Comment
B	IBIS I/O buffer	
E	voltage controlled voltage source	ideal op-amp or transformer
F	current controlled current source	
G	voltage controlled current source	
H	current controlled voltage source	
I	current source	
J	JFET or MESFET	
K	mutual inductor or transformer	
L	inductor	
N	TMI dummy device	
P	port	
Q	bipolar junction transistor BJT	
S	Scattering parameter data	
U	lumped transmission line	

V	voltage source
W	distributed transmission line

Unsupported SPICE commands

The ESP tool does not support use of commands shown in the following table:

Command	**Description**	**Comment**
.ac	AC analysis	
.alter		
.data	in-line data	
.dc	DC analysis	
.del	delete	.del lib tt
.dout	digital output	
.meas[ure]	user defined analysis	
.model_info		
.mosra		
.op	operating point analysis	
.option		
.plot		
.print		
.probe		
.stim[uli]	stimulus save	
.tran	transient analysis	

Unsupported Other

Frequently Asked Questions: SystemVerilog

Introduction

The ESP parser is the same parser as used by the Synopsys VCS simulator version 2014.03-SP1.

The ESP tool has a very limited support for SystemVerilog. The details are listed in the following sections. The ESP tool support for SystemVerilog is based upon the IEEE Standard 1800-2012. The IEEE Standard 1800-2012 extends the IEEE Standard 1364 (the Verilog language standard) with new languages features.

This FAQ references specific clauses in the IEEE Standard 1800-2012 For example:

```
The ESP tool does not support clause 6.14 Chandle data type.
The 6.14 refers to page 69 in Part One of the IEEE Standard 1800-2012.
```

Using SystemVerilog

To use SystemVerilog code you must use the `-sverilog` option to the **read_verilog** shell command.

Supported Features

Base Verilog

The ESP tool supports the Verilog 1995 language with 2001 extensions. See **faq_verilog_unsupported** for more information on the base Verilog language support.

Data Types Clause 6

The ESP tool supports the 2-state SystemVerilog types shortint, short, longint, byte and bit.

The ESP tool supports the 4-state SystemVerilog types logic, reg, integer and time. The 4-state types can have Z and Z bit values as well as 0 and 1.

The ESP tool supports `shortreal` as a `real`.

Structured Procedures Clause 9

The ESP supports `initial` and `always` procedures as defined for Verilog.

The ESP tool has partial support for `always_comb`, `always_latch` and `always_ff` procedures. Effectively, the ESP tool implements these procedures as if they are implemented as always @* . These procedures are defined in the standard to execute once at time zero after all other initial and always procedures have been started. The ESP tool does not guarantee these procedures will execute at time zero after other procedures.

The ESP tool does not support the `final` procedure. (9.2.3)

Assignment statements Clause 10

The ESP tool supports continuous assignment to net or variable data types. (10.3.2). This includes wire, reg, logic, bit, integer and real types.

Operators Clause 11

The ESP tool supports the new SystemVerilog assignment operators: +=, -=, *=, /=, %=, &=, |=, ^=, >>=, <<=, >>>= and <<<=.

The ESP tool supports the SystemVerilog arithmetic shift operators: >>> and <<<.

The ESP tool is compliant with the SystemVerilog interpretation of the power operator ** per clause 11.4.3. This is different than the definiton of the power operator in the IEEE Standard 1364-2001.

For example using the following code:

```
module test ;
  real   ff_1 = (10**12);
  real ff_2 = (10**3.1);
  initial begin
    $display ("ff_1 is %f", ff_1);
    $display ("ff_2 is %f", ff_2);
  end
endmodule
```

```
The Verilog 2001 compliant implementations should produce the following output:

ff_1 is 1000000000000.000000
ff_2 is 1258.925412
```

The SystemVerilog compliant implementations should produce the following output:

```
ff_1 is -727379968.000000
ff_2 is 1258.925412
```

The ESP tool supports the unary increment and decrement operators: -- ++.

The ESP tool supports the `let` construct per clause 11.13. **Note:** Let works in the ESP tool when used with other ESP tool supported constructs.

Example:

```
module m;
  bit clk, a, b;
  logic p, q, r;
  // let with formal arguments and default value on y
  let eq(x, y = b) = x == y;
  // without parameters, binds to a, b above
  let tmp = a && b;
  // ...
  a1: assert property (@(posedge clk) eq(p,q));
  always_comb begin
    a2: assert (eq(r)); // use default for y
    a3: assert (tmp);
  end
endmodule : m
```

The equivalent code with out let is:

```
module m;
  bit clk, a, b;
  logic p, q, r;
  // let eq(x, y = b) = x == y;
  // let tmp = a && b;
  //...
  a1: assert property (@(posedge clk) (m.p == m.q));
  always_comb begin
    a2: assert ((m.r == m.b)); // use default for y
    a3: assert ((m.a && m.b));
  end
endmodule : m
```

Loop Statements Clause 12.7

The ESP tool supports declaration of the loop index variable in the for statement when in SystemVerilog.

The ESP tool supports `break` and `continue` statements in loops when in SystemVerilog.

Assertions Clause 16

The ESP tool supports boolean assertions using the `assert` and `assume` statements.

The ESP tool supports deferred assertions. (Clause 16.4) The ESP tool supports `assert #0` and `assert final`.

The ESP tool does not support use of `disable` with deferred assertions. (Clause 16.4.4)

The ESP tool supports the `assert` statement.

The ESP tool treats all `assume` statements as constraints. Constraints created by assume only work for symbolic inputs. Binary input values are not constrained by the SystemVerilog assume statements. Inputs used in assume statements must not have any other constraints applied using the Tcl **set_constraint** command nor other Verilog code in the testbench.

The ESP tool supports single clock assertions.

The ESP tool supports the system functions $onehot, $onehot0, $isunknown and $countones.

The ESP tool supports typed formal arguments in property declarations.

The ESP tool supports the overlapping implication operator |->.

The ESP tool does not support sequential statements for assert or assume. Sequences and the expect statement are not supported. See **Unsupported Features** for more detailed information on supported SystemVerilog constructs.

The ESP tool supports concurrent assertions in procedures and modules.

System tasks and system functions

The ESP tools supports some of the SystemVerilog system tasks and system functions as described in the following table

Name	Supported	Comment
	Simulation control tasks. Clause 20.2	
$finish	Yes	
$stop	Yes	
$exit	Yes	
	Simulation time functions. Clause 20.3	
$realtime	Yes	
$stime	Yes	
$time	Yes	
	Timescale tasks. Clause 20.4	
$printtimscale	No	
$timeformat	No	Results not correctly scaled
	Conversion functions. Clause 20.5	
$bitstoshortreal	No	
$bitstoreal	Yes	
$cast	No	
$itor	Yes	
$shortrealtobits	No	
$signed	No	
$realtobits	Yes	
$rtoi	Yes	

Name	Supported	Comment
$unsigned	No	
Data query functions. Clause 20.6		
$bits	No	
$isunbounded	No	
$typename	No	
Array query functions. Clause 20.7		
$dimensions	No	
$high	No	
$increment	No	
$left	No	
$low	No	
$size	No	
$right	No	
$unpacked_dimensions	No	
Math functions. Clause 20.8		
$acos	No	
$acosh	No	
$asin	No	
$asinh	No	
$atan	No	
$atan2	No	
$atanh	No	
$ceil	No	
$clog2	Yes	
$cos	No	
$cosh	No	
$exp	No	
$floor	No	
$hypot	No	
$ln	No	
$log10	No	
$pow	No	
$sin	No	
$sinh	No	
$sqrt	No	
$tan	No	
$tanh	No	
Bit vector system functions. Clause 20.9		
$countbits	Yes	
$countones	Yes	
$onehot	Yes	
$onehot0	Yes	
$isunknown	Yes	

Name	Supported	Comment
Severity tasks. Clause 20.10		
$error	Yes	
$fatal	Yes	
$info	Yes	
$warning	Yes	
Elaboration tasks. Clause 20.11		
$error	No	SystemVerilog. Location information is hard to understand
$fatal	No	SystemVerilog. Location information is hard to understand
$info	No	SystemVerilog. Location information is hard to understand
$warning	No	SystemVerilog. Location information is hard to understand
Assertion control tasks. Clause 20.12		
$assertcontrol	No	SystemVerilog.
$assertfailoff	No	SystemVerilog.
$assertfailon	No	SystemVerilog.
$assertkill	No	SystemVerilog.
$assertnonvacuouson	No	SystemVerilog.
$assertoff	No	SystemVerilog.
$asserton	No	SystemVerilog.
$assertpassoff	No	SystemVerilog.
$assertpasson	No	SystemVerilog.
$assertvacuousoff	No	SystemVerilog.
Sampled value functions. Clause 20.13		
$changed	No	SystemVerilog
$changed_gclk	No	SystemVerilog
$changing_gclk	No	SystemVerilog
$falling_gclk	No	SystemVerilog
$fell	No	SystemVerilog
$fell_gclk	No	SystemVerilog
$future_gclk	No	SystemVerilog
$past	No	SystemVerilog
$past_gclk	No	SystemVerilog
$rising_gclk	No	SystemVerilog
$rose	No	SystemVerilog
$rose_gclk	No	SystemVerilog
$sampled	Yes	SystemVerilog
$stable	No	SystemVerilog
$stable_gclk	No	SystemVerilog
$steady_gclk	No	SystemVerilog
Coverage control functions. Clause 20.14		
$coverage_control	No	SystemVerilog
$coverage_get	No	SystemVerilog
$coverage_get_max	No	SystemVerilog
$coverage_merge	No	SystemVerilog

Name	Supported		Comment
$coverage_save	No	SystemVerilog	
$get_coverage	No	SystemVerilog	
$load_coverage_db	No	SystemVerilog	
$set_coverage_db_name	No	SystemVerilog	
Probabilistic distribution functions. Clause 20.14			
$dist_chi_square	No		
$dist_erlang	No		
$dist_exponential	No		
$dist_normal	No		
$dist_poisson	No		
$dist_t	No		
$dist_uniform	No		
$random	No		
Stochastic analysis tasks and functions. Clause 20.16			
$q_add	No		
$q_exam	No		
$q_full	No		
$q_initialize	No		
$q_remove	No		
PLA modeling tasks. Clause 20.17			
$async$and$array	No		
$async$and$plane	No		
$async$nand$array	No		
$async$nand$plane	No		
$async$nor$array	No		
$async$nor$plane	No		
$async$or$array	No		
$async$or$plane	No		
$sync$and$array	No		
$sync$and$plane	No		
$sync$nand$array	No		
$sync$nand$plane	No		
$sync$nor$array	No		
$sync$nor$plane	No		
$sync$or$array	No		
$sync$or$plane	No		
Miscellaneous tasks and functions. Clause 20.18			
$system	Yes		
Display tasks and functions. Clause 21.2			
$display	Yes		
$displayb	Yes		
$displayh	Yes		
$displayo	Yes		

Name	Supported	Comment
$monitor	Yes	
$monitorb	Yes	
$monitorh	Yes	
$monitoro	Yes	
$monitoroff	Yes	
$monitoron	Yes	
$strobe	Yes	
$strobeb	Yes	
$strobeh	Yes	
$strobeo	Yes	
$write	Yes	
$writeb	Yes	
$writeh	Yes	
$writeo	Yes	
	File IO tasks and functions. Clause 21.3	
$fclose	Yes	
$fdisplay	Yes	
$fdisplayb	Yes	
$fdisplayh	Yes	
$fdisplayo	Yes	
$feof	Yes	
$ferror	Yes	
$fflush	Yes	
$fgetc	Yes	
$fgets	Yes	
$fmonitor	Yes	
$fmonitorb	Yes	
$fmonitorh	Yes	
$fmonitoro	Yes	
$fopen	Yes	
$fread	No	
$fscanf	No	
$fseek	Yes	
$fstrobe	Yes	
$fstrobeb	Yes	
$fstrobeh	Yes	
$fstrobeo	Yes	
$ftell	Yes	
$fwrite	Yes	
$fwriteb	Yes	
$fwriteh	Yes	
$fwriteo	Yes	
$rewind	Yes	

Name	Supported	Comment
$sformat	No	
$sformatf	No	
$sscanf	No	
$swrite	No	
$swriteb	No	
$swriteh	No	
$swriteo	No	
$ungetc	Yes	
Memory load tasks. Clause 21.4		
$readmemb	Yes	
$readmemh	Yes	
Memory dump tasks. Clause 21.5		
$writememb	Yes	
$writememh	Yes	
Command line input. Clause 21.6		
$test$plusargs	Yes	
$value$plusargs	Yes	
VCD tasks. Clause 21.7		
$dumpall	Yes	
$dumpfile	Yes	
$dumpflush	Yes	
$dumplimit	Yes	
$dumpoff	Yes	
$dumpon	Yes	
$dumpports	Yes	
$dumpportsall	Yes	
$dumpportsoff	Yes	
$dumpportson	Yes	
$dumpvars	Yes	
Optional system tasks and functions. Clause D		
$countdrivers	No	
$getpattern	No	
$incsave	No	
$input	No	
$key	No	
$list	No	
$log	No	
$nokey	No	
$nolog	No	
$reset	No	
$reset_count	No	
$reset_value	No	

Name	Supported	Comment
$restart	No	
$save	No	
$scale	No	
$scope	No	
$showscopes	No	
$showvars	No	
$sreadmemb	No	
$sreadmemh	No	

Protected Envelopes. Clause 34

The ESP tool supports the use of protected envelops at the same level as VCS release 2014.03-SP1. The ESP tool supports use of `protect and `protect128 directives.

Unsupported Features

The ESP tool does not support

- Structure literals. Clause 5.10
- Array literals. Clause 5.11
- Attributes. Attributes are ignored. Clause 5.12
- Built-in methods. Clause 5.13.
- Void data type. Clause 6.13
- Chandle data type. Clause 6.14
- Class. Clause 6.15
- String data type. Clause 6.16
- User-defined types. *typedef* Clause 6.18
- Enumerations. Clause 6.19
- Data scope and lifetime - static automatic. Clause 6.21
- Casting. Clause 6.24.1
- $cast dynamic casting. Clause 6.24.2
- Bit-stream casting. Clause 6.24.3
- Structures and unions. *struct*, *union* Clause 7.2/7.3
- Dynamic arrays. Clause 7.5
- Associative arrays. Clause 7.8/7.9
- Queues. Clause 7.10/7.11
- Array methods 7.12
- classes. Clause 8
- Final Procedures. Clause 9.2.3
- join_any, join_none. Clause 9.3.2
- Level-sensitive sequence controls. Clause 9.4.4
- Process control - wait fork. Clause 9.6.1
- Fine-grain process control. Clause 9.7

- Wildcard equality - ==? !=?. Clause 11.4.6
- Tagged union expressions - tagged. Clause 11.9
- Conditional operator - matches &&&. Clause 11.4.11
- Set membership - inside. Clause 11.4.13
- Streaming operators - >>> <<< with. Clause 11.4.14
- Operator overloading - bind function. Clause 11.11
- unique/priority/matches/inside. Clause 12.4.2/12.5.3/12.5.4
- foreach. Clause 12.7.3
- Task/Function import/export. Clause 13.6
- Clocking blocks. Clause 14
- semaphores. Clause 15.3
- mailboxes. Clause 15.4
- Persistent trigger: triggered property Clause 15.5.3
- wait_order(). Clause 15.5.4
- Operations on name event variables Clause 15.5.5
- Deferred Assertions. Clause 16.4
- Sequences. Clause 16.7/16.8/16.9
- Declaring properties outside a module 16.3
- Multiclock properties or sequences 16.14
- Binding properties. Clause 23.11
- Expect statement. Clause 16.18
- Checkers. Clause 17
- random constraints. Clause 18
- Coverage statements. *covergroup*, *endgroup*, *bins*, *wildcard type_option*, *with function sample*, *coverpoint* Clause 19
- String version of Input/Output system tasks and functions ($sformat, $sformatf, $sscanf, $swrite, $swriteb, $swriteh, $swriteo) Clause 21
- Nested modules. Clause 23.4
- Extern modules. Clause 23.5
- Name spaces. Clause 23.9
- Programs. *program*, *endprogram*, *extern* Clause 24
- Interfaces. *interface*,*endinterface*,*modport* Clause 25
- Packages. *package*, *import* Clause 26
- Configuring the contents of a design. Clause 33
- DPI. Clause 35
- VPI. Clause 36/37/38
- Assertion API. Clause 39
- Coverage API. Clause 40
- Data Read API. Clause 41

Code Coverage - Frequently Asked Questions

Coverage Frequently Asked Questions (FAQs)

Q) Some nets of my design were not reported in the coverage results. Why?

By default, ESP has a gate extraction optimization. When the tool is run, some nets are optimized away. This improves its capacity.

Q) What can I do to force coverage results for all my nets (in other words, no nets should be optimized)?

To generate coverage for an exact netlist with no net optimization under coverage, use the switch -noopt, which shuts off gate extraction. Adding -noopt results in a decrease in performance and capacity. You should not use it in a simulation run with coverage turned on, if you did not use it in a simulation run without coverage.

Q) When I run esp_shell with coverage enabled, it randomizes or goes out of memory. The normal run without coverage does not. What can I do?

It is only logical to expect that the coverage enabled run affects the capacity of ESP negatively. Extra bookkeeping and computation might result in out of memory or randomization. The following workarounds are suggested for the coverage run to get around this problem:

1. Always use "set_app_var verify_randomize_variables sysmem" in the coverage run.
2. If coverage still randomizes, use divide and conquer techniques to reduce the number of symbols in the testbench and run more testbenches to improve the coverage results.
3. Disable the dumping of symbol (DS) information collection with the $espcovfile command. (See **$espcovfile** for a description of this command.)

   $espcovfile("<espcovfile>", "NS SA LC")

GUI Usage Tips

Using the GUI

This section describes tips on viewing the ESP GUI.

Using Keys in the Command Line

Ctrl→M - Displays a multi-line submit box in which you can copy/paste and run multiple commands.

Ctrl→V - Pastes the current copy buffer into the command input.

Ctrl→X - Cuts selected text in the paste buffer. To select text, see LMB description below for the Transcript Window.

Saving Preferences

The ESP GUI allows you to save GUI preferences so that the settings persist the next time you invoke the ESP GUI.

When you invoke the ESP GUI, it reads some of the default preferences from the synopsys_espgui_sysdef_prefs.tcl file. The ESP GUI has a Preferences dialog to change the default settings to control the appearance and behavior of the GUI. These default settings control the size of main window, window geometry, application font, size, other preferences. If you change the appearance and behavior of the GUI using the Preferences dialog, ESP saves your changes in $(HOME)/.synopsys_esp_prefs.tcl file before it exits.

The next time you invoke ESP, it:

1. Reads the default preferences from synopsys_espgui_sysdef_prefs.tcl

2. Reads the preferences from the $(HOME)/.synopsys_esp_prefs.tcl file. For the preferences that are listed in the synopsys_espgui_sysdef_prefs.tcl file, the $(HOME)/.synopsys_esp_prefs.tcl file has precedence over the synopsys_espgui_sysdef_prefs.tcl file. For all other GUI preferences, ESP uses the values from the synopsys_espgui_sysdef_prefs.tcl file to define the appearance and behavior of the GUI.

UNIX Printing

Printing Help Pages

When you do the first print of a page from a session it may take a while for the print setup form to appear. The wait depends on how many local printers you have available. When the _Setup Printer_ form pops up it will show you the list of local printers that are available.

The default printer is a print to file in PostScript format.

After the first print, the list of printers is cached and the _Setup Printer_ form will pop up quickly.

To change printers and printer options

In the _Setup Printer_ form, you can change the printer that will be used by clicking on the "Print to printer" select button and then select the printer you want to send the page to.

1. Choose File→Print to bring up the _Setup Printer_ form.

2a. If setting up for printing to a file:

- Select _Print to file_
- Type the name of the file or select *Browse*

2b. If setup up to print from a connected printer:

- Select _Print to printer_
- Click on the desired printer to be used.

3. Set print options:

- In _Paper format_, choose *Portrait* or *Landscape*.
- In _Number of Copies_, type the number of copies to print.
- Click *OK*.

FAQ - Regular Expressions

Limited Regular Expressions

When an ESP command or option indicates that you may specify a regular expression in place of a specific string to match, then use the following information to help you construct a regular expression to achieve your desired string matching goals.

Regular Expression Meta-Characters

.	matches any single character
*	matches preceding item 0 or more times
+	matches preceding item 1 or more times
?	matches preceding item 0 or 1 times
^	matches start of string (optional)
[a-d]	matches a single character within a range of chars: a,b,c,d
[~a-d]	matches a single character not within a range of chars
\c	escapes next char 'c', whatever that may be, except within []'s

Usage Notes

"" appearing as the first character of a string, escapes the entire string and none of the remaining characters are considered meta-characters.

"~" must appear as the first character of a [...] range to invert the range match or it is taken literally.

"-" should appear as the first or last character of a [...] range or it is taken as a range separator.

"\t" is not a tab, but the letter 't'.

A regular expression enclosed in matched double quotes "...." will have those quotes removed before being evaluated.

The caret '^' appearing anywhere except the start of the string is taken literally as that character.

Be aware of what your hierarchical separator character is.

Examples

Examples:	Will Match:
.*_pad	ABC_pad go[123]_pad pad_pad_pad _pad
..._pad	ABC_pad zyx_pad
a.*z	az abc%xyz az%123%z
[a-z]+\[[0-9]+	abc[12] alpha[1] a[123456789]
[a-zA-Z_]+\[[0-9]+\]	A_B[12] Dog_Cat[9]
[a-z]+\[[0-9]+\].*	abc[2]xyz bat[121]bat[121] cat[9]dog
[A-C]+_[pad]+	ABC_pad ABC_a ABC_p ABC_ppaadd BAA_dad
[A-C]+_[pad]+	ABC_pad C_p CC_pp
[A-C]?_[pad]?	A_p B_a _p C_
[A-C]*_[pad]+	_pad AAAAA_p BB_aaaa CABCAB_papa
[a-zA-Z0-9_]*z	abRAcadabra_9z_zzz_abc_y_z z Z9z
[a-z_]*[0-9_]*.+z	abracadabro_z_123_abc_y_z y_200z yz z
[aA].*[zZ]	Az aZ abc%xyz az[123]/z
data[12]	data1 data2
data\[12\]	data[12]

FAQ Supported SPICE Device Models

Models

The ESP tool uses built-in models for resistors, capacitors and diodes.

The ESP tool built-in MOS transistor model is based upon published generic process data. This MOS model supports planar and FinFET based technologies. The FinFET model is used when the nfin parameter is found in the SPICE or CDL netlist.

The ESP tool does not support bipolar transistor devices.

Default Models

The default ESP transistor model does not support different Vth devices. The ESP tool uses the smallest length found to select a technology node from an internal table. All transistor models are then scaled from those model parameters.

If your design uses multiple different Vth devices, use the **ESP device simulation methodology** to create models for the transistors in your design.

Device Simulation Methodology

Using the **ESP device simulation methodology** , all 4 terminal MOSFET transistor devices supported by the circuit simulator will work in ESP. This includes planar MOS and multi-gate (FinFet) devices. Encrypted device models also work.

ESP does not support creating device models for resistor, capacitor, or diode devices.

FAQ Top

Frequently Asked Questions

- **esp_shell_flow** How to use ESP shell
- **faq_printing** Printing Help Pages
- **faq_gui_usage_tips** GUI Usage Tips
- **faq_regular_expressions** Limited Regular Expressions
- **faq_supported_devices** Supported SPICE Device Models
- **variables** Using Variables
- **faq_verilog_unsupported** Unsupported Verilog Constructs
- **faq_verilog_coding** Writing Efficient Verilog
- **faq_SystemVerilog** SystemVerilog Support

Efficient Coding Practices

Introduction

The capacity and runtime of symbolic simulation is sensitive to the coding style of the design under test as well as the testbench. Over time a number of such issues have emerged at customer sites or during evaluation of customer cases. This section summarizes some of the most commonly encountered bottlenecks due to coding styles that are not efficient for ESP. This section describes the following:

- always Processes and Sensitivity Lists
- Multidriven Signals and Temporary Variables
- Usage of Functions and Tasks
- Bused Versus Bit-Blown Logic
- Feedback Loops in the Logic and the Effect on Symbolic Simulation
- Imposing Input Constraints
- Output Checker and Output Constraints
- Symbolic Time Recoding
- Symbolic Time Reduction
- Optimizing Symbolic Memory Operations
- Using Delays in Tasks and always Statements

always Processes and Sensitivity Lists

If the set of right-hand side signals of a process is contained within the sensitivity list, ESP performs an optimization that has a good impact on the symbolic capacity. For example,

```
always@(a or b)
  c = c+1; //not optimized
versus
always@(a or b)
  c = a+b; //optimized
```

As a special case, if a left-hand side signal appears on the right-hand side, but not in the sensitivity list, ESP does not optimize the always processes for symbolic simulation.

For example,

```
always@(a or b)
  c = a ? c : b; //not optimized
versus
always@(a or b)
```

```
  if (a) c = b; //optimized
```

As a special case, always processes with delays inside them are expensive to handle for symbolic simulation.

For example,

```
always@(a or b)
  #1 c = a + b;
versus
assign #1 c = tmp;
always@(a or b)
  tmp = a + b;
```

Multidriven Signals and Temporary Variables

The following example shows multidriven signals and temporary variables:

```
always@(reset)
  a = 0;
always@(b)
  a = b;
```

In this example, *a* is a multidriven signal. To ensure correctness of symbolic simulation, an expensive computation is performed. The simulator has no way of knowing that *reset* and *b* are never transitional together because an arbitrary testbench could set it up to work together. Rewriting the previous logic as follows eliminates this problem and leads to better simulation efficiency:

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

Closely related to multidriven signals is the issue of temporary registers and variables often used in RTL coding.

For example,

```
reg [63:0] tmp;
always@(addr1 or mode)
begin
  tmp = mem[addr1];
  if (mode) out1 = tmp[31:0];
  else out1 = tmp[63:32];
end
always@(addr2 or mode)
begin
  tmp = mem[addr2];
  if (mode) out2 = tmp[31:0];
  else out2 = tmp[63:32];
end
```

The intention is to use *tmp* only as temporary storage, but its use for multiple read ports makes it a multidriven signal. On top of doing expensive simulation, there is added overhead to maintain the correct value of *tmp* in all possible symbolic conditions. This leads to extremely complex equations spanning different always blocks even though it looks as if each always block is being computed under its own independent set of variables.

To continue the same example, you might think that *addr1* and *addr2* are driven by independent symbolic inputs. Nevertheless, using the same *tmp* mixes these seemingly unrelated symbolic inputs and entangles them in complex unnecessary equations.

The following is a workaround:

```
reg [63:0] tmp1,tmp2;
always@(addr1 or mode)
begin
  tmp1 = mem[addr1];
  if (mode) out1 = tmp1[31:0];
  else out1 = tmp1[63:32];
end
always@(addr2 or mode)
begin
  tmp2 = mem[addr2];
  if (mode) out2 = tmp2[31:0];
  else out2 = tmp2[63:32];
end
```

In summary, using separate temporary variables can help ease certain issues.

Usage of Functions and Tasks

If a task or function is called multiple times, especially under symbolic conditions, its internal variables behave like temporary variables described in the preceding section. For example,

```
function [3:0] f;
  input in1,in2;
  integer i;
  i = in2;
  if (in1) f = i*i;
  else f = i+i;
endfunction
initial
begin
  for(i=0;i < S1; i=i+1)
    @(posedge clk);
    result = f(a,b);
end
initial
begin
  for(j=0;j < S2; j=j+1)
    @(posedge clk);
  result = f(a1,b1);
end
```

In this example, loop limits *S1*, *S2* are symbolic and ranges from [0:31] => that the *f* function is called 32*2 times in the initial blocks once for each possibility that leads to termination of the for loops.

Therefore, the symbolic condition under which each invocation of *f* happens is symbolic. Further, if *a* / *a1* and *b* / *b1* are symbolic, the resulting scenario becomes more complex. The internal variable of *f*, which is *i*, has a complex equation that becomes worse with each invocation. ESP is unable to conclude that the previous state of *i* is no longer needed and can be overwritten. Therefore a history of *i*'s value is maintained inherently in its equations.

The workaround is as follows:

1. Use in-line code for the task or function or rewrite to eliminate its need.
2. Avoid too many internal variables. For example, in the previous case *i* could be replaced by *in2* and the issue would not occur.
3. Create a separate function for each invocation or usage. Fundamentally, any optimization relies on the internal variables of a function or task not being used as state variables. The previous values are not needed for their next computation.

Bused Versus Bit-Blown Logic

ESP performs much better on bused operations than on the corresponding bit-blown implementation.

For example,

```
always@(a or b or c[31:0] or d[31:0])
  out = (a&b) ? c : (a&~b) ? d : (~a & b) ? c^d : 32'bx;
is much better than
always@(a or b or c[31:0] or d[31:0])
begin
  out[0] = (a&b) ? c[0] : (a&~b) ?  d[0] :
     (~a & b) ?  c[0]^d[0] : 1'bx;
  . . .
  out[31] = (a&b) ? c[31] : (a&~b) ? d[31] :
     (~a & b) ?  c[31]^d[31] : 1'bx;
end
```

The reason is the amortization effect of not having to recompute potentially expensive conditions and expressions for each bit. In this case, if *a* and *b* are complex symbolic bits, computing _(a & b)_ 32 times leads to significantly slower progress. This implies that currently the tool does not have common subexpression elimination and similar compiler optimizations. However, the aforementioned guideline (bus versus bit-blown) would hold if they were available.

Feedback Loops in the Logic and the Effect on Symbolic Simulation

In the following example, *cond_x[1]* depends on *q[i]* and *q[i]* depends on *cond_x[i]*.

```
cond_x[0] = (en[0]===1'bx) ||
   (en[0]===1'b0 && q[0]!==d[0]);
assign . . assign
cond_x[31] = (en1[31]===1'bx) ||
   (en[31]===1'b0 && q[31]!==d[31]);
always@(en or d or cond_x)
begin
```

```
  q[0] <= (cond_x) ? 1'bx : (en[0] ? d[0] : q[0]);
  . . .
  q[31] <= (cond_x) ? 1'bx : (en[31] ? d[31] : q[31]);
end
```

As the ESP tool evaluates *cond_x[i]*, it schedules the always process for execution. After the execution of the always process, it schedules all the *cond_x* assignments for execution.

A worst-case execution trace would be *cond_x[0]*, *always process*, *cond_x[1]*, *always process*, ... , *cond_x[31]*, *always process*. This leads to redundant execution of the always block 31 out of 32 times.

A best-case trace would be *cond_x[0]*, *cond_x[1]* ..., *cond_x[31]*, *always process*. This trace has no redundant execution.

A real case would actually be an average run with a few redundant executions. Each execution can be expensive because all the inputs might be symbolic expressions. You can code the above example again as follows:

```
always@(en or d) begin
  cond_x[0] = (en[0]===1'bx) || (en[0]===1'b0 && q[0]!==d[0]);
  . . .
  cond_x[31] = (en1[31]===1'bx) || (en[31]===1'b0 && q[31]!==d[31]);
  q[0] <= (cond_x) ? 1'bx : (en[0] ? d[0] : q[0]);
  . . .
  q[31] <= (cond_x) ? 1'bx : (en[31] ? d[31] : q[31]);
end
```

This eliminates redundant executions because the order implied by the intent of the design is now coded serially inside the always block. Note: This example has been derived from a capacity problem seen in a customer design.

Imposing Input Constraints

Care should be taken while writing input constraints as to not make the inputs too complex or not mix too many variables together in one input equation.

For example,

```
reg [15:0] sym1,sym2,sym3,sym4;
rd1 = inno.one_hot(sym1);
rd2 = inno.one_hot(sym2);
rd3 = inno.one_hot(sym3);
wr = inno.one_hot(sym4);
for(i=0;i<15;i=i+1)
  if (rd1[i] || rd2[i] || rd3[i]) wr[i] = 0;
```

This is inefficient because it first mixes 16 symbolic inputs each in the equations of *rd1[0]* ... *rd1[15]*, *rd2[0]* ... *rd2[15]*, *rd3[0]* ... *rd3[15]*, *wr[0]* ... *wr[15]* through application of *one_hot* constraints.

Subsequently it computes a complex equation for *wr[i]* in terms of 16+16+16 variables in *rd1[i]*, *rd2[i]*, *rd3[i]* as well as its own 16 variables in *wr[i]*.

In a real customer example, this complex constraint on the *wr* port caused a blowup and randomization on a simple design. The solution was to rewrite the constraint using auxiliary variable *wen_sym[15:0]* as

```
reg [15:0] wen_sym;
reg [15:0] sym1,sym2,sym3,sym4;
rd1 = inno.one_hot(sym1);
rd2 = inno.one_hot(sym2);
rd3 = inno.one_hot(sym3);
wr = inno.one_hot(sym4);
wr = wr & wen_sym; //16 bit bit-wise AND
rd1 = rd1 & ~wen_sym;
rd2 = rd2 & ~wen_sym;
rd3 = rd3 & ~wen_sym;
end
```

This guarantees the desired exclusivity between read and write on slice *i* and ensures that the equation of the input write port is much simpler (only 17 variables versus 64 variables in each equation `wr[i]`).

Output Checker and Output Constraints

A divide-and-conquer methodology approach on checking outputs is effective. For example, it is inefficient if you direct the ESP tool to compute a complex logical OR of an already complex checking conditions.

```
if (cond1 || cond2 || cond3 ...... || cond_n)
  $esp_error("ERROR");
```

Instead, rewriting as *n* different *if* expressions can save the tool a redundant computation that could have led to randomization. As a side effect, it tells you exactly which condition failed.

For example,

```
checks :
  if (cond1)
    $esp_error("ERR_1");
    . . .
  if (cond_n)
    $esp_error("ERR_n");
```

A related issue is checking the equality under the constraint that reference signal *xref* is not *x*. However, in this check `(xref!==ximp)` can blow up even before the constraint `(xref!==1'bx)` gets computed and added with it.

```
if ((xref !== ximp) && (xref!==1'bx))
  $esp_error("ERROR");
```

The following alternative coding style avoids this potential blowup if the blowup was under the symbolic threads leading to an *x* condition on the *xref* (again a real test case experience):

```
if (xref!==1'bx)
  if (xref!==ximp)
    $esp_error("ERROR");
```

Changing the order of the AND operation does not help, but converting it to nested if condition does.

In summary, a *logical and* computation at the checker `($esp_error())` can be converted to nested-if's with the least complex condition as the *outermost if* and the most complex condition as the *innermost if*.

This is a recommended coding style for ESP.

Symbolic Time Recoding

Delay expressions that involve symbolic equations may trigger an error message from ESP about unsupported symbolic time. Arbitrary symbolic time values can be expensive to simulate. Symbolic time values should be constrained to a constant value during any single simulation. Run multiple simulations to validate the possible values.

If you must use symbolic time values then recode the delays as for loops with an if expression to guard the symbolic time value.

For example:

```
always @(clk) begin
  clkd <= #(delay) clk;
end
```

should be recoded as:

```
// use a named block to allow local loop variable
always @(clk) begin: clk1
  integer i;
  for (i=0; i<=delay; i=i+1)
    if (i==delay) clkd <= #(i) clk;
end
```

Symbolic Time Reduction

Use a for loop or while loop with large intervals instead of *#S*. For example, if you need to wait for any number of clock cycles between 0 and 15 and the clock period is 10 time units, a simple way to code this in ESP is

```
reg [7:0] time_sym; // range 0-255
$esp_var(time_sym);
initial
begin
  wait_time = (time_sym>150) ? 0 : time_sym;
  #(wait_time);
  ...
end
```

The above is inefficient because the *wait_time* command can take 150 possible values and one thread of execution starts for each of the previous possibilities. Increasing the granularity of wait_time can help tremendously:

```
reg [3:0] time_sym; // range 0-15
$esp_var(time_sym);
initial
begin
  for(i=0;i<time_sym;i=i+1)
    @posedge(clk);
  ...
end
```

This coding style allows only 16 threads of execution on the boundary of the positive edge of the clock. Thus the granularity in this case is 10 time units. The complexity is reduced by a factor of 10.

Note: It is important to understand different situations in which symbolic time can happen directly or indirectly in the testbench. A common indirect cause can be the symbolic length of a packet where processing time is proportional to its length. In this case, increasing the granularity of the symbolic length to cover the cases of interest can help.

Optimizing Symbolic Memory Operations

Symbolic memory writes (where address and/or data are symbolic) are expensive operations. Optimizing them can increase efficiency. For example,

```
case 2'b01: mem[addr1] = a;
     2'b10: mem[addr2] = b;
     ...
endcase
```

can be transformed to

```
case
  2'b01: begin
           adr = addr1;
           data = a;
         end
  2'b10: begin
           adr = addr2;
           data = b;
         end
  ...
endcase
mem[adr] = data;
```

Using Delays in Tasks and always Statements

Using delays is expensive and inefficient in various contexts, such as tasks, always statements, and within blocking or nonblocking assignments. Wherever possible the delay usage should be avoided or coded again. For example,

```
always@(a or b)
  #1 c = a + b;
```

is more expensive than

```
assign #1 c = tmp;
always@(a or b)
  tmp = a + b;
```

And

```
always @(...)
  a <= #3 mysample;
```

is more expensive than

```
assign #3 tmp = mysample;
always @(...)
  a <= tmp;
```

Unsupported Verilog Constructs

Introduction

This chapter lists the Verilog 1995 and 2001 HDL expressions and statements that are not supported by the ESP tool. It includes the following sections:

```
Specify Blocks
Hierarchical Names
Recursive Functions and Tasks
Data Types
System Calls
PLI
VCD
Math Operators
Compiler Directives
```

Specify Blocks

The ESP tool supports commonly used specify path delays and timing checks as documented in the *IEEE Standard 1364-95 Verilog Language Reference Manual*. However, there are a few constructs that are not widely used and not supported. These unsupported constructs are listed in the following categories:

```
specify path delay
Unsupported features not included in the IEEE standard
Timing checks
```

If an unsupported statement is encountered while compiling, a warning is issued and the statement is ignored.

The $nochange timing check task is not supported.

specify Path Delay

The pulse limit values using *PATHPULSE$* syntax construct is not supported.

Unsupported Nonstandard Features

Some Verilog simulator products use features that are not part of the IEEE 1364 standard, but they are expected to work. These features are invoked through command-line options and Verilog system tasks. The ESP tool does not support the following nonstandard features:

Pulse handling through the Cadence Verilog XL product

```
+transport_path_delays
+x_transport_pessimism
+alt_path_delays
```

Global pulse control on specify module paths

```
+pulse_e/error_percentage
+pulse_r/reject_percentage
+pulse_int_e/error_percentage
+pulse_int_r/reject_percentage
+transport_int_delays
```

Local pulse control invoking $PATHPULSES to be effective

```
+pathpulse
```

Pulse filtering using command line or Verilog system tasks

```
+pulse_e_style_on_event
+pulse_e_style_on_detect
$pulseestyle_onevent
$pulseestyle_ondetect
```

Controlling whether canceled scheduled events are displayed, using command line or Verilog system tasks

```
+show_cancelled_e
+noshow_cancelled_e
$showcancelled
$noshowcancelled
```

accu_path algorithm to change the path delay accuracy

```
+accu_path_delay
$eventcond
$noeventcond
```

Timing Checks

There are no unsupported timing checks in the ESP tool.

Hierarchical Names

In Verilog, a hierarchical path is made up of a list of identifiers separated by periods. For example, assume the hierarchical name, *a.b.c.w* .

When you use hierarchical compression with the ESP tool, you cannot access a task or function in a compressed module. For example, suppose module top has children *x*, *a1*, and *a2* with *a1* and *a2* being compressed together. You cannot hierarchically access *top.a1.task_mytaskx* or *top.a2.func_mytasky*, although you can locally access *task_mytaskx* or *func_mytasky* inside *a1(a2)*.

With the ESP tool you can disable a hierarchical task, but you cannot specify the disabling of any identifier within the task.

Recursive Functions and Tasks

The ESP tool does not support recursive functions and tasks. During compilation, an error is reported that indicates where the recursion occurred and stopped at the end of the compile phase.

Data Types

The ESP tool does not support arrays of real or realtime type.

System Calls

The ESP tool supports all display tasks, I/O tasks, and simulation control tasks, as well as *$random*, *$time*, and *$stime*.

Generally, using a non-supported system call within the ESP tool will result in a compilation error. However, the following non-supported system calls will be ignored during compilation and only result in an error if they are actually used during simulation:

```
$countdrivers   $reset_count
$getpattern     $reset_value
$incsave        $restart
$input          $save
$key            $scale
$list           $scope
$log            $showvars
$nokey          $sreadmemb
$nolog          $sreadmemh
$reset
```

This allows code that contains these functions to be accepted by the ESP tool as long as these functions are not actually used during simulation. For instance a design containing the following code would otherwise be rejected at compile time:

```
always @(some_sig)
  begin
    if (!$test$plusargs("dont_count_drivers"))
      if ($countdrivers(some_sig))
        $display("%t Too many drivers: %m", $realtime);
  end
```

Instead, defining dont_count_drivers will allow the design to be compiled and run, since $countdrivers() will never actually be executed. So using the following command will allow simulation.

```
read_verilog -r verilog.v -vcs "+dont_count_drivers"
```

The ESP tool does not support the following system tasks and functions.

Math functions

ESP does not support the following system tasks and functions:

```
$ln     $log10   $exp    $sqrt   $pow
$floor  $ceil   $sin     $cos    $tan    $asin
$acos   $atan   $atan2   $hypot  $sinh   $cosh
$tanh   $asinh  $acosh   $atanh
```

Timescale tasks

The (path) option of *$printtimescale* is not supported, but *$printtimescale* without options is supported.

Stochastic analysis tasks

Unsupported tasks include *$q_add $q_exam $q_full $q_initialize $q_remove*

Conversion functions for reals

Unsupported functions include *$itor* and *$rtoi*.

Probabilistic distribution functions

Unsupported functions include *$dist_exponential $dist_chi_square $dist_erlang $dist_normal $dist_poisson $dist_t $dist_uniform*

Miscellaneous system functions

Other unsupported system functions include *$countdrivers $plus$val*

Note: You can find more detailed explanations for these tasks and functions in Section 14 of the Verilog LRM.

PLI

The ESP tool does not support the use of PLI, VPI or DPI.

VCD

The ESP tool does not fully support VCD. The following information is not recorded in a VCD file:

Symbolic variables

Hierarchically compressed real numbers

Power Operator

The ESP tool supports the power operator ** per the SystemVerilog interpretation. This is different than the IEEE 1364-2001 definition.

For example using the following code:

```
module test ;
  real   ff_1 = (10**12);
  real ff_2 = (10**3.1);
  initial begin
    $display ("ff_1 is %f", ff_1);
    $display ("ff_2 is %f", ff_2);
  end
endmodule
```

IEEE Standard 1364-2001 compliant implementations should produce the following output:

```
ff_1 is 1000000000000.000000
ff_2 is 1258.925412
```

IEEE Standard 1800 SystemVerilog compliant implementations should produce the following output:

```
ff_1 is -727379968.000000
ff_2 is 1258.925412
```

The ESP tool produces the IEEE Standard 1800 SystemVerilog compliant results.

Compiler Directives

All of the compiler directives defined in the IEEE Standard 1364-2001 Verilog LRM are supported. These include:

```
`celldefine `endcelldefine
`ifdef      `ifndef         `endif
`else       `elseif
`define     `undef          `include
`line       `timescale      `resetall
`unconnected_drive
`nonconnected_drive
`default_nettype
```

Use of an unknown compiler directive will result in an error. To avoid this error, either remove the compiler directive from the code or define the compiler directive as a macro.

For example, the following code will not compile in the ESP tool as the compiler directive `switch is not implemented in the ESP tool.

```
// begin test.v
`switch
module test;
endmodule
```

To coerce this to compile under the ESP tool add "+define+switch" to the Verilog command line options.

VCS Directives

The following VCS compiler directives are not supported:

```
`inline   `noinline   `portcoerce   `noportcoerce
```

Use of these directives will results in the ESP tool issuing a message about undefined macros. Use +define to define the compiler directive.

For example if your Verilog code uses `inline, add +define+inline to the VCS options for read_verilog in the shell. If your command was:

```
 read_verilog -r test.v
```

it now will be:

```
 read_verilog -r -vcs "+define+inline" test.v
```

Verilog-XL Directives

The following Verilog-XL compiler directives are not supported.

Use of these directives will be ignored.

```
`accelerate        `noaccelerate
`default_rswitch_strength,
`default_switch_strength
`default_trireg_strength,
`pre_16a_paths,    `end_pre_16a_paths,
`remove_gatenames  `noremove_gatenames,
`remove_netnames   `noremove_netnames,
`rs_technology
`switch
```

Fast Signal Data Base (FSDB)

OVERVIEW

If you are using Verdi to view waveforms in your VCD files, you can dump a Verdi Fast Signal Data Base (FSDB) file directly from ESP. Otherwise Verdi, upon loading of files in the VCD format, runs a converter to convert these files to FSDB format. FSDB enables the functionality of VCD file format with a smaller file size. Additional features of this format include the dumping of strength for each signal.

ESP uses FSDB format version 5.1 so Verdi 2014.12 or newer is recommended to view the FSDB files created by ESP.

The following table shows various releases of Verdi and the version of the FSDB format supported by that release. All releases of Verdi support FSDB formats from prior releases but some functionality may be limited as older formats may not contain the data needed to support the functionality.

Verdi versions	FSDB versions
Verdi 2013.04, Verdi 2014.12	FSDB 5.1
Verdi 2015.09	FSDB 5.3
Verdi 2016.06	FSDB 5.4
Verdi 2017.03	FSDB 5.5
Verdi 2017.12	FSDB 5.6

The contents of the FSDB file are controlled in the testbench through calls to $fsdbDumpfile, and $fsdbDumpvars. Briefly, $fsdbDumpfile specifies the filename for the FSDB database. The $fsdbDumpvars system task defines what signals, or more likely, scopes that should be recorded into the FSDB database.

Reasons to use FSDB include:

- VCD dumps in ESP are limited to 8,000,000 nets. FSDB has no limit to the number of nets in the database
- The compressed FSDB file is smaller than the ASCII VCD format.
- You can enable strength dumping in FSDB mode only, ESP currently does not support VCD dumping with strength.
- Sometimes Verdi spends a long time in VCD to FSDB conversion. Dumping to FSDB directly can eliminate this overhead.

DESCRIPTION

The FSDB dump file format is used by the Verdi waveform viewer. The compressed file format takes up less disk space than the VCD file format. The trade-off is that it takes longer to dump these files since file compression is done dynamically while the file is being written. This section describes the ESP interface to the Verdi FSDB dump file format. The format of the dump data that is generated by ESP is only supported by Verdi base technology version 5.3 and greater.

Some benefits of this format are

- File size is reduced
- Strength information can be dumped
- Dump scope can be specified at run time rather than at compile time
- Dump files can be changed during run time

Dumping Strength Data

By default, strength data is not dumped. You can enable strength data dumping if you specify *+dump+strength* to **read_verilog** or by using $fsdbDumpStrength system function. You can specify *+dump+strength* using one of the following arguments to **read_verilog** :

```
   -f <file>
      where file contains +dump+strength
 or
   -vcs "+dump+strength"
```

You can also turn on strength dumping within a simulation by using the $fsdbDumpStrength system task. Once strength data dumping is turned on, there is no way to disable it within that simulation.

To see the strength values, use $fsdbDumpfile so waveforms are dumped in FSDB format. You can specify the net colors in the RC modeled portions of the design. Normal(strong) net values (such as full-rail nets) in the waveform can be displayed as one color; partially discharged nets (such as slow-changing nets) can appear as another color in the waveform. In the Verilog portions of the design, the usual Verilog strength values will be dumped.

$fsdbDumpfile

Usage: $fsdbDumpfile("dumpfile");

This command specifies the file name that the FSDB data will be written to during simulation. If no name is provided, the default file name of verilog.fsdb will be used.

Example:

```
  initial begin
    $fsdbDumpfile("dump.fsdb")
    $fsdbDumpvars;
  end
```

The example shows that the file dump.fsdb is to be used to place the FSDB dump file data.

$fsdbDumpvars

Usage: $fsdbDumpvars(dumplevels, dumppath) ;

This command works in the same way as the standard $dumpvars function. It specifies the instance path to the module to be dumped and how many levels are to be dumped starting at that module. A dumplevel of 0 specifies that all levels at and below this module are to be dumped.

When no options are specified on this command, all levels are dumped. This is shown in the first example. The second example dumps the module pointed to by the instance path a.p. All levels below this are also dumped because a dumplevel of 0 was specified.

$fsdbDumpStrength

Usage: $fsdbDumpStrength ;

The $fsdbDumpStrength command enables signal strength dumping for any $fsdbDumpvars statements evaluated after encountering the $fsdbDumpStrength command. If any $fsdbDumpvars statements were evaluated prior to the $fsdbDumpStrength statement, these would still not dump strength data. (Note that if you changed the default strength dumping with the command-line option +dump+strength, all $fsdbDumpvars will write out strength information.) By default, no strength value are dumped.

Example:

```
$fsdbDumpStrength;
```

$fsdbDumpoff

Usage: $fsdbDumpoff ;

The $fsdbDumpoff command turns off all FSDB dumping that is currently enabled. Any new $fsdbDumpvars encountered will not begin dumping until enabled by the evaluation of another $fsdbDumpon statement.

Example:

```
$fsdbDumpoff;
```

$fsdbDumpon

Usage: $fsdbDumpon ;

The $fsdbDumpon command enables the dumping of FSDB data if it has been previously turned off.

Example:

```
$fsdbDumpon;
```

$fsdbDumpflush

Usage: $fsdbDumpflush ;

Dumping FSDB data does not automatically flush the data to the output file. It buffers the data internally so that performance is not affected by file I/O. The $fsdbDumpflush command is useful if you are dynamically viewing the dumpfile while simulation is still in progress. If you do not force a flush of the dump file at periodic intervals, you may not be able to view the current signals in simulation because the data has not been written to the dumpfile yet.

Example:

```
$fsdbDumpflush;
```

$fsdbSwitchDumpfile

Usage: $fsdbSwitchDumpfile("fsdbfile") ;

The $fsdbSwitchDumpfile command dynamically switches which dump file you are currently writing to.

Example:

```
$fsdbSwitchDumpfile("dump.fsdb");
```

$fsdbAutoSwitchDumpfile

Usage: $fsdbAutoSwitchDumpfile(dumpfilesize, dumpfile, maxfilecount);

The $fsdbAutoSwitchDumpfile command dumps up to dumpfilesize Mbytes of data to a series of files based on the file name present in the dumpfile parameter. A maximum of maxfilecount files will be dumped.

The name of the file used as the dump file is constructed from the dumpfile parameter and an index number. The index number is appended to the dumpfile name along with an underscore character just prior to the last period in the name. If there is no period, it is appended to the end of the name. For example, if you had a file name of foo.fsdb and a maxfilecount of 4, the file names used would be foo_1.fsdb, foo_2.fsdb, foo_3.fsdb, and foo_4.fsdb.

Do not use a small dumpfilesize value. To be safe, use something greater than 200Mbytes. Otherwise, you risk creating a number of files which only have the header data written to the file. The header data would be the design's signal name indexes and the size would depend on the size of the design. If all the simulation dump data does not require all the allowed files in this statement, the remaining unused files will not be created.

Example:

```
$fsdbAutoSwitchDumpfile(500,"dumpfile",4);
```

In this example, the first 500Mbytes of data will be dumped to the file dumpfile_1, the next 500Mbytes of data to the file dumpfile_2, and so on up to the last file dumpfile_4. Dumping will stop if the maxfilecount number of files have been dumped.

Examples

The following examples give a more comprehensive idea of how these commands work.

```
Example 1        Standard Usage
  initial begin
    $fsdbDumpfile("esp.fsdb");
    $fsdbDumpvars;
  end
```

Example 1 begins dumping all of the signals in the design to the file esp.fsdb. Strength data is not dumped by default. To dump strength data, invoke **read_verilog** with the option +*dump*+*strength*.

```
Example 2        Time- and signal-dependent dumping
    initial begin
      $fsdbDumpfile("esp.fsdb");
      $fsdbDumpvars(0, a.p);
      $fsdbDumpvars(0, a.q);
      #10;
      $fsdbDumpStrength;
      $fsdbDumpvars(0, a.r)
      $fsdbDumpvars(0, a.s)
      #10;
        $fsdbDumpoff;
      #10;
        $fsdbDumpon;
      #10
    end
```

At time 0, the instances a.p and a.q are dumped to the file esp.fsdb. Instances a.r and a.s dump with strength data after 10 time scale units. After another time 10 time units, dumping to all instances is disabled and after another 10 time units, dumping of all instances is enabled again. At the end of simulation, the internal dump buffers are flushed to the dump file automatically.

```
Example 3  Dumping to different files
   initial begin
     $fsdbDumpfile("CPU.fsdb");
     $fsdbDumpvars(0, top.cpu);
     #10;
```

```
      $fsdbSwitchDumpfile("ALU.fsdb");
      $fsdbDumpvars(0, top.alu);
    end
```

When the $fsdbSwitchDumpfile command is encountered, the file CPU.fsdb is closed and another file named ALU.fsdb is opened. After 10 time scale units, dumping occurs to the file ALU.fsdb.

```
Example 4  Limiting dump file size
   initial begin
     $fsdbAutoSwitchDumpfile(500, "esp.fsdb", 4);
     $fsdbDumpvars;
   end
```

The FSDB file size limit is 500 Mbytes. Once the size of esp_1.fsdb reaches 500 Mbytes, it automatically closes esp_1.fsdb, opens esp_2.fsdb and continues dumping.

Similarly, when esp_2.fsdb reaches 500 Mbytes, it is closed and esp_3.fsdb opened. Since the number of files is limited is 4, the last file to be dumped is esp_4.fsdb.

SEE ALSO

Variables: **waveform_dump_control** , **waveform_format** , **waveform_viewer**

Topics: **dve** , **vcd**

help

NAME

help

Displays quick help for one or more commands.

SYNTAX

```
help [-verbose] [-groups] [pattern]
```

ARGUMENTS

-verbose	Displays options, for example "command -help".
-groups	Displays a list of command groups only.
pattern	Displays commands matching pattern string.

DESCRIPTION

The Tcl shell **help** command is used to get quick help for one or more commands or procedures. This is not the same as the **man** command which displays reference manual pages for a command in a Web browser.

SPECIAL Note: Every command supports a *-help* option to provide more information on command options. So in addition to using `help read_verilog` you could also do `read_verilog -help`.

There are many levels of help with the Tcl shell command.

By typing the **help** command, commands are listed in paragraph format by command group. There are dedicated groups for Built-in commands and Procedures. Other groups are defined by the application.

List all of the commands in a group by typing the group name as the argument to help. Each command is followed by a one-line description of the command.

You can get a one-line description help for a single command by typing help followed by the command name. You can specify a wild-card pattern for the name. For example, all commands containing the string "alias". Finally, get syntax help for one or more commands by adding the *-verbose* option.

List only the command groups in the application by passing only the *-groups* option.

EXAMPLES

```
shell> help
Procedures:
 ls, sh,
Builtins:
 alias, append, array, break, catch, cd, close, concat, continue
 ...
shell> help Procedures
 ls                 # List files
 sh                 # Execute a shell command
shell> help a*
 alias                # Create a command which expands to words.
 append               # Builtin
 array                # Builtin
shell> help -verbose source
 source               # Read a file and execute it as a script
   [-echo]                (Echo all commands)
   [-verbose]             (Display intermediate results)
   file_name              (Script file to read)
```

SEE ALSO

Commands: **man**

Topics: **command_group** List of primary commands by functional group

Interactive Signal Tracing

DESCRIPTION

This section describes the interactive signal tracing debug flow available within the ESP shell.

Interactive Signal Tracing (IST) can be used for debugging a design after the verification run fails.

The primary IST commands are: **start_explore** , **stop_explore** , **get_net_transitions** , **get_net_triggers** , **print_net_backtrace** , **print_net_trace** , and **is_net_explorable** .

SEE ALSO

```
Commands: debug_design , get_net_transitions , get_net_triggers , is_net_explorable , print_net_backtr
Topics:
```

Cell Library Verification

Description

Cell library verification is a subset of the ESP tool support for verification of multiple designs The tool can verify the Verilog, SystemVerilog, SPICE and Liberty formats of a cell description. See the man page **SystemVerilog FAQ** for specific information on the supported subset of SystemVerilog.

The cell library verification flow uses a testbench which does a more detailed analysis of functionality that is generally better suited to typical library cells. This analysis includes support for input and output loading for the SPICE design. This analysis is also better suited for asynchronous type cells. The default ESP symbolic testbench is targeted at synchronous design styles.

Flow

The cell library verification flow is similar to the general memory verification flow. A design is any Verilog module, Verilog UDP, SPICE subcircuit or Liberty cell. The general memory verification flow works with a single matched design pair. The cell library verification flow works with multiple matched design pairs.

Two steps are added for the cell library verification flow:

1. Set defaults for testbench style and other ESP shell variables to those most commonly used for cell library verification. This step is done before reading any design data.
2. Match all the design pairs. This step is done after all the design information is read into the reference and implementation containers.

The full cell library verification flow is:

1. Set Defaults
2. Set Technology
3. Read Designs
4. Match Designs
5. Define Supplies
6. Match Ports
7. Set Design Port Attributes
8. Understanding Constraints
9. Verify Designs
10. Debug Design Differences
11. Analyze Code Coverage

Set Defaults

The **set_verification_defaults** command sets default values for cell library verification. Use:

```
set_verification_defaults -library_verification
```

This command changes the default values for the application variables **testbench_reference_instance** ,**testbench_implementation_instance** ,**testbench_style** ,**testbench_binary_cycles** ,**testbench_symbolic_cycles** ,**testbench_flush_cycles** ,testbench_output_checker ,**verify_use_specify** ,**verify_dynamic_reorder** and **verify_hierarchical_compression**.

Set Technology

The **set_process** command with the *-technology* option loads an ESP device model simulation file to define transistor device models.

Read Designs

The reference design can be Verilog text , SystemVerilog text or a Liberty binary database. The **read_verilog** command reads Verilor or SystemVerilog text. The **read_db** command reads a Liberty binary database. A Liberty binary database is created from a Liberty text file using the lc_shell command from the Design Compiler tool suite.

The **read_spice** command reads one or more SPICE text file(s).

Match Designs

Several ESP shell commands are used in the verification of cell libraries. Many of these commands work using the concept of a matched design pair. A matched design pair is a reference cell and an implementation cell that have the same functionality. The ESP tool provides commands to create and manage the matched pairs.

The **match_designs** command will scan the reference and implementation containers and create matched design pairs for designs that have the same name. This match is a case insensitvie match. Options are provided to enforce port naming and name case sensitivity.

The **set_matched_designs** command manually matches the specified pair of designs.

The **remove_matched_designs** command unmatches a pair of designs.

The **report_matched_designs** command reports the matched design pairs.

The **report_unmatched_designs** command reports the unmatched designs in the reference and implementation containers.

The **get_matched_designs** command returns a collection of matched designs. The return value from **get_matched_designs** can be used as the value to the *-matched_designs* switch available on other commands.

How to get a restricted set of matched designs to operate on all the DFF cells

After design matching is completed, you can get a collection of all the cells that contain DFF in the name using the **filter_collection** command. For example to save the collection to the variable allDFF use the following code:

```
set allDFF [filter_collection -regexp [get_matched_designs] {name=~.*DFF.*}]
```

This collection can then be used with the `-matched_designs` option on many other commands. For example to define CLK as the clock for all the DFF cells use the following code:

```
# after port matching is completed
set allDFF [filter_collection -regexp [get_matched_designs] {name=~.*DFF.*}]
create_clock -matched_designs $allDFF CLK
```

Switch -matched_designs

Library verification provides the ESP tool the capability to set attributes on more than one top-level design during setup.

The following commnands include the `-matched_designs` option to support cell library verification:

- **check_design**
- **get_testbench_ports**
- **match_design_ports**
- **remove_clock**
- **remove_constraint**
- **remove_matched_ports**
- **remove_testbench**
- **rename_testbench_pin**
- **report_aborted_points**
- **report_clock**
- **report_constraints**
- **report_failing_points**
- **report_log**
- **report_matched_ports**
- **report_passing_points**
- **report_status**
- **report_testbench_pins**
- **report_unmatched_ports**
- **reset_clock**
- **reset_testbench_pin_attributes**

- **set_active_testbench**
- **set_constraint**
- **set_initialization**
- **set_matched_ports**
- **set_testbench**
- **set_top_design**
- **verify**
- **write_esp_db**
- **write_testbench**

Define Supplies

Match Ports

Port matching for multiple designs uses the `-matched_designs` option for the port matching commands:

- **set_matched_ports**
- **match_design_ports**
- **remove_matched_ports**
- **report_matched_ports**
- **report_unmatched_ports**
- **get_testbench_ports**

The automatic port matching performed by the **match_design_ports** command is case insensitive. Add the `-exact` option to match ports only when they have the exact same case.

The **set_matched_ports** command matches ports regardless of case, name or bit size. The test generator uses the reference port name in the testbench as the net to drive or check. The `-tbpin` option will change the name of the net used in the testbench. **Note:**Using the `-tbpin` option can affect commands like **set_constraint** or **set_testbench_pin_attributes** that work with testbench nets.

The **get_testbench_ports** command returns a collection of testbench pins. The *-matched_designs* option controls which collection of testbench pins is returned. **Not supported in current release**.

Set Design Port Attributes

Certain testbench attributes are specified at the design level, others are set for each testbench pin. The following table lists all the testbench attributes and show which are design level and which are pin level.

Option	design	pin	Description

-allow	Y	Y	Allowed input values. One or more of 1 0 X Z
-channellength	Y		testbench driver channel length in microns. Default: 0.18 microns
-checkdelay	Y		Delay used with continuous output check in nS. Default: 1nS
-checker	Y	Y	Output checker. Default: eq in library verfication
-checkmode	Y		Output checker mode. Legal values: discrete, discrete2 or continuous. Default: discrete
-constraint	Y		Global constraint file in library verification. Equivalent to **testbench_constraint_file** application variable in single design verification
-declaration	Y		Declaration file in library verification. Equilvalent to **testbench_declaration_file** application variable in single design verification
-direction		Y	Testbench port direction. Legal values: input, output or inout. Default: inout or direction of design in Verilog reference container
-function		Y	Pin functions. Default: other. See **pin_functions**
-indrive	Y		Size of input driver in microns for all SPICE inputs
-initialization	Y		Predefined initialization vector type or initialization file. Initialization file is equivalent to **testbench_initialization_file** application variable in single design verification
-ioenable	Y	Y	Equation that defines when device port is in output mode and is driving the testbench
-iomode	Y	Y	Bidirection port mode. Legal values: allowX, allowZ, noZ, inputonly, outputonly. Default: allowX
-maxphasevectors	Y		Maximum number of inputs that change per phase
-outload	Y		Size of output load in microns for all SPICE outputs
-portgroup		Y	One or more portgroups as defined by the **set_portgroup** command
-power_domain		Y	power domain as defined by the **set_power_domain** command
-radix		Y	Display radix: Legal values: binary, hex, octal, decimal. Default: binary
-setup	Y		Setup file in library verification mode. Equilvalent to **testbench_setup_file** application variable in single design verification
-symbolic_static		Y	Input will have same symbolic value in every symbolic cycle
-vectormode	Y		Type of input vector to apply to inputs. Legal values: functional, skewmode, derace. Default is functional
-verilogdrive	Y		Verilog drive strength for input driver. Legal values: supply, strong, pull, or weak. Default: strong
-voltage		Y	Real supply voltage.
-weak_threshold	Y		Weak threshold for eqw checker in KOhms Default: 50 KOhms

Design attributes are set with the **set_matched_design_attributes** command. Pin attributes are set with the **set_testbench_pin_attributes** command.

Defining Clocks

The library cell testbench generator uses information about primary clocks, buffered clocks and secondary independent clocks when creating a testbench for a cell. The *-clktype* and *-constraint* options to the **create_clock** command guide the ESP tool when creating testbenches for the *library* style testbench.

The option *-clktype* specifies which clock is the primary clock, a secondary clock or a buffered clock in the testbench. The primary clock is the only clock that is toggled during the binary and flush cycles. The primary clock is the first input changed in the `INIT` task of the testbench. The option *-clktype* is ignored for all testbench styles except for *library*. An error message is generated if more than one primary clock is defined for a cell.

A buffered clock is a copy of another clock. A buffered clock can be an inversion of another clock. The *-constraint* option defines which clock is to be copied or inverted for a buffered clock.

The *-constraint* option defines the condition under which a secondary clock can change value.

Note:

- The *-delay* and *-phase* options of the **create_clock** command are ignored for the *library* style testbench.
- The *-period* and *-setup* options of the **create_clock** command can only be used on *primary* clocks in the *library* style testbench.

Understanding Constraints in Library Testbenches

Definitions:

Active value of a clock
: The value of a clock input after the active edge of that clock has occurred. For a latch, the active clock value is the value that enables the transparent phase of the latch.

Primary clock
: This is the clock used by the design during work mode.

Secondary clock
: The secondary clocks should be used for scan clocks or test clocks.

To understand how clock constraints and data constraints are used in library testbenches, you have to understand how clocks are specified and when clock inputs are applied to designs being verified in relation to when data inputs are applied.

The initial value of a clock specified in the create_clock command is the inactive clock value. The default inactive value, if not specified, is a value of 0. All clocks are initialized in the testbench with the inactive value at time 0. Specifying a pin as a clock with the set_testbench_pin_attributes command, uses the value specified as the initial clock value which is the inactive clock value. For non-clock inputs the value specified in set_testbench_pin_attributes is the active value.

In the example below, the first command sets the sclk1 input initial value as 0 with an active value as 1 (high) The second command sets the sclk2 initial value as 1 with an active clock value of 0 (low).

```
Example:
create_clock -clktype secondary sclk1 -constraint "clk && scanmode" -initial 0
```

```
set_testbench_pin_attribute sclk2 -function clock -value high
set_testbench_pin_attribute we -function enable -value high
report_testbench_pins -all
Testbench         Dir Size Active Input InOut    Output   Func     Clk  Clk
 PortName                  Value  Value Mode     Checker           Type Constraint
 sclk1            in     1  Low    01   n/a      n/a      Clock    Sec  D&&B
 sclk2            in     1  High   01   n/a      n/a      Clock    Sec
 we               in     1  High   01   n/a      n/a      Enable   n/a  n/a
```

The report for testbench clock pins shows the initial value (or inactive value) rather than the active value, as stated in the report header, which is the opposite of what is reported for all other pin types.

While a standard symbolic testbench applies a new input test vector once per clock period which covers both phases of the clock, the library testbench applies a new input test vector once per clock phase, i.e. twice per clock period. For each vector, the clock transitions first, followed by the data transitioning a hold time later.

Only one primary or secondary clock changes at a time in the library testbench which is different from the symbolic testbench where every clock changes at the same time. Buffered clocks are set after the primary and secondary clocks change since their value depends on the values of primary and secondary clocks. In the library testbench, the INIT cycle toggles each clock input separately. During the binary and flush cycles, only the primary clock is toggled. During the symbolic cycles, the clock becomes symbolic with only one of them changing during each clock phase. The testbench signal inno_clk_select is used to determine which clock input changes and it is symbolic during the symbolic cycles. The library testbench has more symbolic cycles than the symbolic testbench to allow more clock waveform combinations to be exercised.

Global constraints are invoked twice in each clock phase, both using the same apply_global_constraints task. The first call is after the clocks change and the next after data input vectors are changed. The first constraint call is required to be able to use a constraint that says if the clock is a certain value then the data input has to be constrained. This constraint has to occur when the clock changes. An example of this is provided later on in this document.

There are no testbench buffers on clock inputs. So once the clock value is changed, that value is immediately propagated to the devices under test. This is not the same as data input changes which are delayed during apply_vector until hold time after the clock changes using the task assign_buffer_value.

The create_clock command allows clock constraints to be added to each clock for library testbenches. The option -constraint identifies the conditions under which the clock is allowed to change. This is especially important if you have multiple clocks and there are cases where both clocks can not be active at the same time or the clock can not toggle when a particular data pin is active. Clock constraints only apply to clocks, not data. Clock constraints are applied at the time that clocks are changed and not when data is constrained in apply_global_constraints. The clock constraint specifies the legal conditions under which the clock can change.

When multiple clock constraints are present, all clock constraints have to be true or no clock can change state. If just the clock constraints are considered, a condition can occur that causes no clocks to be able to toggle. To prevent this, an extra term is added to the constraint for each clock that allows the constraint to be true if the next value of the clock is the inactive value. An example of this is shown in a following example. The generic clock constraint looks like:

```
  if ((clockconstraint) || inactiveclockvalue)  { finalclockvalue = nextclockvalue; }

For two clock constraints this condition results in:

  if ((clock1constraint) || inactiveclock1value) &&
      (clock2constraint) || inactiveclock2value)) {
```

```
    finalclock1value = nextclock1value;
    finalclock2value = nextclock2value;
  }
```

during the binary cycles, if the constraints specified is a condition that is different from the inactive clock values, then the primary clock will not change during the binary cycles

The following example shows a condition that would prevent the primary clock from toggling if only the constraint was used to determine whether any clocks could change. During the binary cycles both sclk and clk start out inactive. Since neither is high then no clock could toggle based on just the constraints being met. The addition of the inactive clock term to the clocks constraint, allows the equation to be true if the next clock value is inactive. Use of this additional term assumes that having all clocks at their inactive value is not an illegal condition.

```
Example:
create_clock -clktype primary clk -constraint sclk -initial 0
create_clock -clktype secondary sclk -constraint clk -initial 0
```

Note that during the INIT cycle all clocks are inactive and only one clock at a time is toggled to active and back to inactive along with all the data inputs. During the binary cycles only the primary clock is toggled and the secondary clocks are kept inactive. During the symbolic cycles, only one clock is selected to change during each phase.

The clock constraint equation can include simple logical operators like >, <, &, |, ^, ~, !. ?:, ==, !=, ===, !==, and parenthesis like (.). The equation can also include some shortcut macros that are identified by starting with a dollar sign in the name and use a single clock as an argument. These macros only work for clock signals and are $nochange, $change, $rise, and $fall. The $change(clkname) macro returns true when the previous value of clkname is not the same as the next value of clkname. The following example shows two clocks with constraints. If scanmode is high, clk is not changing and if clk is high, then sclk is allowed to change.

```
Example:
create_clock -clktype primary clk -constraint "scanmode && \$nochange(clk)"
create_clock -clktype secondary sclk -constraint 'clk'
```

The set_constraint Tcl command should be used for setting the state of non-clock inputs. It should not be used for changing the value of a clock signal. If it does change a clock input, there will be glitches introduced into the clock signal. The macros allowed for clock constraints are not allowed in set_constraint conditions.

Before any new vector is generated, the previous values are saved in the set of innopv_SIGNAL variables. These variables can be used when setting constraints to cause an input signal to not change or to determine whether a signal is not changing or is changing, rising or falling.

The set_constraint conditions are incorporated into the apply_global_constraints task and are used to constrain data inputs. However the data constraints are only applied a short time after clocks are changed using the assign_buffer_value task. i Data inputs are connected to the devices being tested by an array of innont_<port> registers. The assign_buffer_value task applies constrained inputs through innont_<port> signals to both reference and implementation design at the same time. To allow global constraints to affect data inputs at the same time as a clock changes, you need to change the associated innont_<port> signal for that input port.

If you do not want all data signals to happen at the same time, then you can use "set_matched_design_attributes -vectormode derace" option to force all inputs to happen at different times. If there are any input conditions that require two signals to not be active at the same time, then "-vectormode derace" should not be used. The other option is to use the "set_input_delay" Tcl command to specify an offset delay to a signal to be used when applying inputs during assign_buffer_value task.

The following example shows two different ways to constrain the next value of a data input. The constraint file allows more complex constraint conditions to be defined then can be set by the set_constraint command. There could be multiple inputs constrained at the same time, or you could define a data constraint which changes the data input at clock time to emulate precharge conditions.

Both of these commands result in changes to the apply_global_constraints task. The constraintfile constraints will be applied first, followed by the constraints generated by the set_constraint Tcl commands.

```
Example:
set_constraint scanin -set innopv_scanin -if "(sclk === 1'b1)" -matched_designs abc

set_matched_design_attributes -constraint constraintfile -matched_designs abc

where constraintfile contains

if (sclk === 1'b1) begin
  scanin = innopv_scanin;
end
```

An example on how to handle precharge inputs to a library cell will show how to handle data inputs when a clock changes. A precharge input must always be a logic 1 level when the clock is inactive. It is only allowed to go to a logic 0 value when the clock is active.

```
Example:
create_clock C1 -clktype primary -initial 0
set_constraint P1 -set ?1; innont_P1 = 1? -if ?(C1===0)&&(P1!===1)?
```

The first call to data constraints is called after the clock change is determined. The testbench signal P1 retains the previous constrained value which is equal to the previous value captured just prior to the clock change. This allows you to use P1 in the -if condition at clock time. But we have to change the current P1 value being applied to the devices being verified. To do that we change the signal innont_P1 which drives the inputs on both models. This invalidates the previous value captured but that value is not needed again. So now we have the P1 signal being precharged to value 1 when the clock goes inactive.

The next set of input vectors are applied and P1 may now be set to 1 or 0. If the clock is still inactive, then the next data constraint check will ensure that the next value of P1 is set to the precharge value 1. However if the clock is active, the next value of P1 can either be 1 or 0 as determined by the input vector being applied. The fact that we also set the innont_P1 value during the constraint does not matter because the next thing that happens is the assign_buffer_value setting the next value of P1. It does bypass any input delays that are present in the assign_buffer_value task.

Verify Designs

The **verify** command will verify the current design unless the *-matched_design* option is specified. The **verify** command will stop verification on the first failing testbench. To run all testbenches regardless of completion status use **distributed processing** or write a loop and run each testbench or cell separately.

The following example shows part of the output from a cell library verfication run in parallel.

```
...
verify -matched_designs [get_matched_designs]
Information: Minimum channel width is 0.12u (ESPSPC-219)
Information: Minimum channel length is 0.035u (ESPSPC-220)
```

```
Generating testbench   ...... ESP_WORK/ADD_FULL/esp.tb.ltb
Information: Minimum channel width is 0.12u (ESPSPC-219)
Information: Minimum channel length is 0.035u (ESPSPC-220)
Generating testbench   ...... ESP_WORK/ADD_HALF/esp.tb.ltb
...
Information: Starting distributed job using CDPL...

    PASS    FAIL   OTHER  |     Jobs      Done  Running  Pending   Killed  Unknown
       0       0       0  |      951         0        0      951        0        0
       0       0       0  |      951         0        0      951        0        0
       0       0       0  |      951         0        0      951        0        0
     164       0       0  |      951        69      478      404        0        0
     945       0       0  |      951       857       94        0        0        0
     951       0       0  |      951       951        0        0        0        0

Information: Distributed processing job completed.
Information: Verification succeeded. (ESPUI-033)
1
```

The **add_distributed_processors** command enables **distributed processing**. The following example shows how to use **distributed processing** and then generate debug information for any failing cells. The example uses a simple LSF setup file myLSF.setup.

Example myLSF.setup file allowing 10 verifications in parallel.

```
1| |10| |LSF| bsub -R rusage[mem=8000] -R arch=glinux
```

Example section of run.tcl script.

```
add_distributed_processors -host myLSF.setup
...
verify -matched_designs [get_matched_designs]
set failed [report_status -fail]
foreach tb $failed {
   set dir [file dirname $tb]
   set tbFile [file tail $tb]
   set cell [file tail [file dirname $tb]]
   set_top_design -matched_design $cell
   debug_design
   file rename dump.vcd $dir/dump.vcd
   save_session $cell
}
```

The following example shows how to loop through all the matched cells and save a debug session if verification fails.

```
foreach_in_collection cell [get_matched_designs] {
  if ![verify -matched_designs $cell] {
    set cellName [get_attribute $cell name]
    set_top_design -matched_design $cell
    debug_design
    file rename dump.vcd ESP_WORK/$cellName/dump.vcd
    save_session $cellName
  }
}
```

Debug Design Differences

Cell library verification works on multiple designs. The debug factilities in the ESP tool work with single testbenches and designs. To debug a failing cell, specify the top design to be debugged with the -*matched_design* option of the **set_top_design** command. Examples of using the **debug_design** command are shown above.

Analyze Code Coverage

Code coverage can be enabled when performing cell library verification.

```
set_app_var coverage true
```

After verification is completed you can review the coverage data for each cell using the **report_coverage** command with the *-all* and *-matched_designs* options. Loop through each cell and report the results separately. If the *-matched_designs* option is specified with more than one design, the ESP tool will try and merge all the reports into one coverage report and a merge error is likely to occur.

The following example shows how to get coverage information for every cell.

```
foreach_in_collection cell [get_matched_designs] {
  report_coverage -all -matched_designs $cell > ${cell}.coverage.summary
}
```

SEE ALSO

Commands: **get_matched_designs** **match_designs** **match_design_ports** **remove_matched_designs** **report_matched_d**

Variables: **testbench_max_vectors_in_phase**
testbench_output_checker
testbench_style
testbench_weak_threshold

Topics:

Limitations

DESCRIPTION

This manual page describes known ESP tool limitations. When possible, general workarounds are suggested.

Performance

When used only as a Verilog simulator, the ESP tool is generally not as fast as other Verilog simulators. The ESP tool can show some performance gains for highly replicated designs.

Capacity

Designs with a sequential depth beyond 4 may not work with the ESP tool. It is not possible to predict when the ESP tool will hit a capacity limit.

When the ESP tool reaches a "symbolic capacity" limit it will try and continue by randomly replacing symbols with a 1 or a 0. This is called "randomization". The threshold for randomization is controlled by the application variable **verify_randomize_variables**.

Best Practice
Set the value to `sysmem` to allow the ESP tool to use the most memory.

```
set_app_var verify_randomize_variables sysmem
```

Shell

netlist_case

When the **netlist_case** variable is set to *lower*, commands that have net names or other SPICE object names must use lower case names. Such names are not automatically converted to lower case.

You can write a Tcl procedure to automatically lower case a name based upon the setting of **netlist_case** . Use this procedure to protect names in your script. Example:

```
proc CorrectCase{s} {
  if {string equal "preserve" [get_app_var netlist_case]} {
    return $s;
  } else {
    return [string tolower $s]
  }
}
set_net_delay -delay 10 -design [CorrectCase Mux221] [CorrectCase NET31]
```

Language Compliance

Verilog

The ESP tool supports most of the Verilog language as defined by the IEEE 1364-1995 language standard.

A limited set of features of the Verilog language as added in the IEEE 1364-2001 language standard are also supported.

Newer features of of the Verilog language as added in the IEEE 1364-2005 language standard are not supported.

The ESP tool does not support the PLI access routines.

For a detailed listing of unsupported Verilog language features refer to **Unsupported Verilog Constructs FAQ** .

SystemVerilog

The ESP tool supports a very small subset of SystemVerilog.

See **SystemVerilog FAQ** for a description of how to enable SystemVerilog and the specific limitations.

VHDL

VHDL is not supported.

SystemC

SystemC is not supported.

SPICE

There is no official SPICE language definition. Therefore, any SPICE language is described in terms of a specific implementation.

The ESP tool supports the Berkeley SPICE and HSPICE dialects.

The ESP tool does not support the .CONNECT command from HSPICE.

The ESP tool supports the design parts of SPICE and not the simulation control portions of the SPICE language. Commands such as .PRINT and .PLOT are not supported.

The ESP tool does not support any of the following design elements:

- sources: E,F,G,H,I or V
- inductors: K or L
- multi-terminal linear: S, U or W
- ports: P
- IBIS buffers: B
- Bipolar Juntion Transistors (BJT): Q
- JFETS and MesFETS: J

These elements must be removed from the design netlist.

SPICE design files should not contain .MODEL or .LIBRARY statements. These statements can cause the **read_spice** command to fail.

The work around is to comment out any .MODEL or .LIBRARY statements in the design files.

The ESP tool supports three and four terminal transistor models. The ESP tool ignores the bulk (fourth) connection of a four terminal device.

Five or more terminal transistor devices are not supported. Such devices must be instantiated using 'X' card syntax and a subcircuit must be provided to map all the terminals to a four terminal device.

See **SPICE FAQ** for detailed description of the ESP tool support for SPICE.

General Issues

The automated testbenches create names for tasks, functions, registers and wires. These names are not checked for collisions with top level design port names or design names. If a design name is the same as a name generated by the ESP tool, compilation errors can occur.

There are three work arounds for this issue:

- Create a wrapper module and subckt for the design to change the name of the port that is in conflict with the ESP name.
- Rename the ports in the design.
- Create a custom testbench. Modifications as detailed in **testbench_customization** may not be possible for some name conflicts. You may have to hand edit the actual generated testbenches to work around name conflicts.

Clocks in Automated Testbenches

The ESP tool generates automated testbenches that assume synchonous clocks. All clocks have the same clock period and a 50% duty cycle. You can control the initial delay to a clock and the starting value for the clock.

For more than one clock, it must be the same as the first clock defined or an inversion of the first clock defined.

Inputs are applied before the first change on the first clock. This is controlled by the *-setup* option of the **create_clock** command.

More complex clocking schemes require the creation of custom `apply_vector` and `INIT` tasks. For more information see **Customizing your Testbench** .

SEE ALSO

Commands: **read_spice** **set_process**

Variables: **verify_randomize_variables**

Topics: **faq_verilog_unsupported** **faq_SystemVerilog** **testbench_customization**

Logical to SPICE mapping

Overview

This topic discusses how to use the ESP tool to create a logical to SPICE mapping file. Converting the logical to SPICE mapping file into a logical to physical mapping file is beyond the scope of this topic.

The general flow is to use the mapping style testbench with code injected into the SPICE file to print out mapping information.

Methodology

Let's say you want the ESP tool to report information that allows you to determine the external and logical address and data mapping for each internal and physical implementation memory cell instance. The ESP tool does not automatically produce this mapping; however, you can do this with appropriate "annotation" of the memory's core cell and a custom script that processes the report_log output.

General approach:

1. Ensure that the testbench does a symbolic write cycle after the following has been added to the implementation memory core cell:

   ```
   always @(wordline)
   if  (bitline==0) $display("%m addr=%b data=%b", addr, data);
   ```

2. Use a script to process the resulting lines that show the cell instance name (from the %m), address, and "0" within the symbolic data field indicating what bit within the data word corresponds to that cell. For example,

   ```
   inno_tb_top.xtor.XI5.XBANK0.X0.X7 addr=111 data=sssssss0
   inno_tb_top.xtor.XI5.XBANK1.X0.X7 addr=111 data=sssss0ss
   ...
   ```

Example

Specific example (using the ESP Tcl Shell) and the ram1r1w design: (This is an eight-by-eight memory, i.e.

three address bits and each word being eight bits wide.)

Start the ESP Tcl Shell:

```
esp_shell
```

Read Verilog, SPICE, and set pin attibutes:

```
read_verilog   -r ram.v
set_top_design -r ram1r1w

read_spice     -i ram.sp
set_top_design -i RAM1R1W

match_design_ports

set_testbench_pin_attributes CLK -function Clock   -value High
set_testbench_pin_attributes A   -function Address -value High
set_testbench_pin_attributes DI  -function Data    -value High
set_testbench_pin_attributes WE  -function Write   -value High
set_testbench_pin_attributes RE  -function Read    -value High

set_app_var verify_testbench_style mapping
set_app_var testbench_symbolic_cycles 1

# write in 1st symbolic cycle
set_constraint WE -set {1'b1} -cycle SYMCYCLE1

# inject code into MEMCELL module (the memory cell)
#  to display needed information when a 0 value is written
#  to the cell.
#   In our example WBL=0 when writing a 0
#
set_net_value -i -design MEMCELL -code {
   always @(WWL) begin
     # only write mapping in the 1st symbolic cycle
     if (inno_tb_top.inno_cycle === "SYMCYCLE1") begin
       if (WBL===0) begin
         $display("%m addr=%b data=%b", inno_tb_top.A, inno_tb_top.DI);
       end
     end
   end
}

write_testbench ram

verify
report_log

quit
```

Post-process output. There are 64 lines that are similar to the following:

```
inno_tb_top.xtor.XI5.XBANK0.X0.X7 addr=111 data=sssssss0
inno_tb_top.xtor.XI5.XBANK1.X0.X7 addr=111 data=sssss0ss
inno_tb_top.xtor.XI5.XBANK2.X0.X7 addr=111 data=sss0ssss
inno_tb_top.xtor.XI5.XBANK3.X0.X7 addr=111 data=s0ssssss
inno_tb_top.xtor.XI5.XBANK0.X1.X7 addr=111 data=ssssss0s
inno_tb_top.xtor.XI5.XBANK1.X1.X7 addr=111 data=ssss0sss
inno_tb_top.xtor.XI5.XBANK2.X1.X7 addr=111 data=ss0sssss
inno_tb_top.xtor.XI5.XBANK3.X1.X7 addr=111 data=0sssssss
inno_tb_top.xtor.XI5.XBANK0.X0.X6 addr=110 data=sssssss0
```

```
 inno_tb_top.xtor.XI5.XBANK1.X0.X6 addr=110 data=sssss0ss
  ...
```

For this particular RAM, these lines indicate that the XI5.XBANK0.X0.X7 cell corresponds to address 111 and data bus bit 0 while XI5.XBANK1.X0.X6 corresponds to address 110 and data bus bit 2, etc.

Create a custom script to put this information in the format you need.

man

NAME

man

Displays reference manual pages.

SYNTAX

```
man topic
```

ARGUMENTS

topic Specifies the subject to display. Available topics include commands, variables, and error messages

DESCRIPTION

For the Tcl shell, the man command displays the on-line manual page for a command, variable, error messages, and other ESP related help information. Users can write man pages for their own Tcl procedures and access them with the man command by setting the **sh_user_man_path** variable to an appropriate value. See the man page for **sh_user_man_path** for details.

From the ESP GUI this man viewer can also be invoked using the pull-down menu _Help→Man Pages_. You can also use the pull-down menu *Help→Help* to invoke the GUI Help viewer. The man viewer cannot be invoked from the ESP GUI console command line.

EXAMPLES

The following command displays manual page for the echo command.

```
shell> man echo
```

The following command displays the manual page for the error message CMD-025.

```
shell> man CMD-025
```

SEE ALSO

`Commands: help`

Net Attributes

NAME

net_attributes
Describes the predefined application attributes for net objects.

Description

is_supply

Type:
string
Value:
true if the net is a supply net, *false* otherwise.

name

Type:
string
Value:
name of the net

supply_from

If *is_supply* attribute is *false* the attribute value is not defined.

Type:
string
Value:

- *auto* for a net automatically identified by the ESP tool as a supply
- *user* for a net identified via the **set_supply_net** command as supply

supply_type

If *is_supply* attribute is *false* the attribute value is not defined.

Type:

string

Value:

real or *virtual*.*real* supplies do not change logic value in verification.

supply_value

If *is_supply* attribute is *false* the attribute value is not defined.

Type:

string

Value:

The "on" state for a supply net. The "on" state is specified by the -*logic* option of the **set_supply_net** command.

supply_id

If *is_supply* attribute is *false* the attribute value is not defined.

Type:

integer

Value:

The ID number of the supply net definition as reported by the **report_supply_nets** command that results in this net identified as a supply.

SEE ALSO

Commands:
debug_design , **start_waveform_viewer**

Variables:
waveform_format , **waveform_viewer**

Topics:
dve , **esp_shell_flow** , **fsdb** ,

Output Checkers

DESCRIPTION

ESP provides a series of automatic checkers to verify that two signals are identical. All the checkers are written in standard Verilog and can be simulated in other simulators. The checkers are located in the file: <release>/auxx/esp/primitives/inno In that file you will find the following tasks:

compare	Verifies that two signal/buses are exactly equivalent.
comparex	Verifies that two signal/buses are equivalent, treating X as a don't care if it is in the reference signal.
comparez	Exactly the same as comparex but uses Z as the don't care instead of X.
comparexz	The same as comparex, but treats both X and Z as don't care.
compare_zmx	Similar to the compare checker but treats Z in the reference model to be equal to X in the implementation model. This is useful when the reference model propagates Z values but the implementation can only represent this case as an X due to buffers or inverters within the data path from input to output.
comparex_zmx	Combines the comparex and the compare_zmx functionality.

SEE ALSO

Commands: **set_testbench_pin_attributes**

Variables:

Topics:

Pattern File

DESCRIPTION

ESP pattern matching allows you to match portions of a SPICE netlist and apply properties to matched nets or device nodes in the netlist. The users can define any pattern combined with rules. Pattern matching is not limited to pre-defined settings. The patterns are specified using the SPICE subcircuit syntax, and can be based on circuit topology only, on specific transistor sizes, or even on particular net names.

Overview

Pattern matching is the same as applying various shell commands against a subcircuit. However, pattern matching allows you to re-use patterns from one design to the next, and reduces switch-level modeling requirements as your pattern library grows. Several other situations may also call for use of pattern matching.

You may want to consider using pattern matching in the following cases:

- You wish to deploy a methodology to your design group, which can automatically locate and model commonly used cells or circuit structures.
- Your circuit has the same topology in many different subcircuits.
- Your SPICE netlist is flat.
- You routinely create new copies of existing cells for each new application.

You probably do NOT want to use pattern matching in the following cases:

- The topology being matched is too simple, and may cause many false matches elsewhere in your design.
- You have a methodology in place to annotate schematics with modeling commands (they will likely propagate to new copies anyway).
- You are using a memory or datapath compiler that can generate the appropriate modeling commands.

Using Pattern Matching

Pattern matching occurs in esps2v, using the -pattern <file> option, where <file> is a SPICE netlist containing various patterns to be matched, specified as subcircuits. You may specify multiple patterns in one file or multiple files containing a separate pattern in each by using multiple -pattern <file> pairs.

esps2v -pattern filename

The resulting Verilog switch-level netlist created, will be altered if pattern matching was successful. The user can provide as many patterns as needed. The pattern may even contain hierarchy. The hierarchy will be flattened out entirely before the pattern is used.

Pattern Definition

A pattern is represented by a (small) SPICE netlist. The port map of this pattern netlist defines the boundaries of the pattern within the big design. The pattern will contain transistors with or without parameters being wired to represent an electric circuit. While running, the pattern matching algorithm will execute two tasks.

The first task will try to find patterns in the design. Rules can be specified to reduce the possible matches. The rules are not just on design topology. If transistors come with parameters in the pattern only those will be matched in the design which have the same parameters.

The second task alters the netlist by applying properties to the matched devices. An example is to set a transistor to be weak.

Special Features to Configure the Pattern Matching Task

The Matching Part

Parameter matching: Simply add parameters to the transistors in the pattern.

Name matching: Occasionally topological matching is not enough, only specific instance names of transistors or net names should be considered. Add one or more of the following statements to the pattern on top.

```
**innoprop pmmatch specialdev <subckt> <inst>
**innoprop pmmatch specialnet <netname>
```

Transistor model matching: If your design contains more than two transistor model definitions (mostly the default n or p polarity models) you can explicitly specify via the *.model* statement in SPICE.

Now the pattern matcher will limit the matches to these models only and all other potential topological fits will be disregarded.

The Action Part

Property application: Default behavior, no extra config required. The properties to be applied on transistor devices are

delay weak/weaker/weakest remove blackbox swap bidir

The properties to be applied on nets are

net_type (supply1/0, wand, wor, tri1/0, bitline, bitlineb)

Creating a Pattern File

Create a SPICE netlist which represents the pattern. The SPICE netlist can also contain **innoprop commands to specify modeling properties to be applied with the recognition of this pattern.

Note: Make sure all boundary connections are shown as port signals in the pattern.

Example 1 shows an example pattern placed in the file mypattern.sp.

Example 1 Example pattern

```
.subckt bitcell wl bl blb
mn1 bit bitb vss vss n
mp1 bit bitb vdd vdd p
mn2 bitb bit vss vss n
mp2 bitb bit vdd vdd p
mn3 bl wl bit vss n
mn4 bitb wl blb vss n
.ends
```

Enable pattern matching when running esps2v with the -pattern option:

```
esps2v -spice design.sp -pattern mypattern.sp
       -cfg my.cfg -verilog design.v -verbose
```

Observe the -verbose option at the end of the command line.

Initially you might find it hard to compose the pattern. Please continue with No Match when expecting a Match for more information on how to debug. Later you should remove the -verbose option since it slows down the matching algorithm.

Upon completion check the esp_s2v.log file. It will tell you how many patterns have been matched on an instance basis. If the match succeeded at least once, modify the pattern by adding **innoprop properties to it. For example, substitute **innoprop inst weak <subckt> <inst>, as shown here, in the previous example pattern.

```
**innoprop inst weak bitcell mp1
**innoprop inst weak bitcell mp2
```

This will mark the pFETs of the subcircuit bitcell as weak.

Rerun esps2v and study the changed output netlist. All transistors being matched will be changed into weak. (in Verilog, this is represented by altering the name from pmos to rpmos or tranif0 to rtranif0). You can check the esp_s2v.log file to see how many matches were found.

No Match when expecting a Match.

This is very common at the beginning of the process. Likely causes are small differences in the wiring of the pattern circuit as well as missing port map signals for defining the boundaries. If your pattern carries parameters for the transistors make sure that they are properly instantiated with the correct values in the design. If you are interested in just a topological match, add this option to the pattern file with **innoprop:

```
**innoprop pmmatch parameter off;
```

This option may lead to unexpected results. You could potentially match portions of a netlist which are topologically the same, but the transistors sizes cause it to perform a different function.

Another source of problems is the recognition of .model definitions of transistors. If the pattern carries such a definition, the matcher will be limited to these models. If these models are different in the main design to be matched, you will receive an early abort.

If you still have no success in matching the pattern, study the log file carefully. Make sure the -verbose option is turned on. A dump of the extra information might look like this:

```
S2V Pattern Info: Failure with initial node phi[1]_L (1.
out of 2). 28 percent of pattern inverter are matched.
Pattern node a matched with Design node phi[1]_L
Pattern node mp0 matched with Design node mp0
Last examined nodes from pattern:
out
Vdd
Last examined nodes from design:
srcell_0/T1
in
```

The pattern matcher works its way from the inner to the outer boundaries of the pattern to be matched. It will find candidate nodes from which the matching process begins. Nodes can be nets or transistor devices. In the previous example, the two initial candidates were nets a from the pattern and phi[1]_L from the main design.

The next two nodes successfully matched are mp0 and mp0 which are both transistors. The matching algorithm is incremental. A net match is always followed by a device match and vice versa. The matcher incrementally adds nodes until the pattern match is successful or fails. The last nodes shown in the previous example caused the pattern matcher to fail and abort. Start your debugging efforts by examining the surroundings of the failed nodes. In the example, consider out from the pattern with srccell_0/T1 from the design.

Last, but not least, even if all patterns are matched correctly as expected and you run with -verbose on, you will still find some failures reported in the log file. This is not a problem. The pattern matcher algorithm tries to match more candidates than you might perceive.

Other Issues. If the pattern has been correctly specified but the result shows less (or no) matches than expected, look at the following:

1. Check if some ports of the pattern are connected to VDD/GND. These are special nets and will lead to mismatches if ports are connected to them. For example, a 3 NAND gate pattern with an input connected to VDD becomes a 2 NAND gate and thus violates the original logic of the pattern.
2. Check if a transistor model definition has been placed in the pattern itself. If this is the case, the pattern matcher limits itself to match only those models in the main design and disregards other correct topological patterns. After pattern matching, some .subckt names might have changed to P_ ... prefix. Because some patterns depend on correct transistor sizes (parameterized), the hierarchy might be changed. This changed hierarchy is reflected by changes in .subckt or module names.

SEE ALSO

```
Commands:

Variables:

Topics:
```

Perl Regular Expression Syntax

In Perl regular expressions, all characters match themselves except for the following special characters:

```
.[{}()\*+?|^$
```

Wildcard

The single character '.' when used outside of a character set will match any single character except:

- The NULL character when the flag match_not_dot_null is passed to the matching algorithms.
- The newline character when the flag match_not_dot_newline is passed to the matching algorithms.

Anchors

A '^' character shall match the start of a line.

A '$' character shall match the end of a line.

Marked Sub-expression

A section beginning (and ending) acts as a marked sub-expression. Whatever matched the sub-expression is split out in a separate field by the matching algorithms. Marked sub-expressions can also repeated, or referred to by a back-reference.

Non-marking Grouping

A marked sub-expression is useful to lexically group part of a regular expression, but has the side-effect of spitting out an extra field in the result. As an alternative you can lexically group part of a regular expression, without generating a marked sub-expression by using (?: and) , for example (?:ab)+ will repeat ab without splitting out any separate sub-expressions.

Repeats

Any atom (a single character, a marked sub-expression, or a character class) can be repeated with the *, +, ?, and {} operators.

The * operator will match the preceding atom zero or more times, for example the expression a*b will match any of the following:

```
b
ab
aaaaaaaab
```

The + operator will match the preceding atom one or more times, for example the expression a+b will match any of the following:

```
ab
aaaaaaaab
```

But will not match:

```
b
```

The ? operator will match the preceding atom zero or one times, for example the expression ca?b will match any of the following:

```
cb
cab
```

But will not match:

```
caab
```

An atom can also be repeated with a bounded repeat:

`a{n}` Matches 'a' repeated exactly n times.

`a{n,}` Matches 'a' repeated n or more times.

`a{n, m}` Matches 'a' repeated between n and m times inclusive.

For example:

```
^a{2,3}$
```

Will match either of:

```
aa
aaa
```

But neither of:

```
a
aaaa
```

It is an error to use a repeat operator, if the preceding construct can not be repeated, for example:

```
a(*)
```

Will raise an error, as there is nothing for the * operator to be applied to.

Non-greedy Repeats

The normal repeat operators are "greedy", that is to say they will consume as much input as possible. There are non-greedy versions available that will consume as little input as possible while still producing a match.

`*?` Matches the previous atom zero or more times, while consuming as little input as possible.

`+?` Matches the previous atom one or more times, while consuming as little input as possible.

`??` Matches the previous atom zero or one times, while consuming as little input as possible.

`{n,}?` Matches the previous atom n or more times, while consuming as little input as possible.

{n,m}? Matches the previous atom between n and m times, while consuming as little input as possible.

Possessive Repeats

By default when a repeated pattern does not match then the engine will backtrack until a match is found. However, this behaviour can sometime be undesireable so there are also "possessive" repeats: these match as much as possible and do not then allow backtracking if the rest of the expression fails to match.

`*+` Matches the previous atom zero or more times, while giving nothing back.

`++` Matches the previous atom one or more times, while giving nothing back.

`?+` Matches the previous atom zero or one times, while giving nothing back.

`{n,}+` Matches the previous atom n or more times, while giving nothing back.

`{n,m}+` Matches the previous atom between n and m times, while giving nothing back.

Back References

An escape character followed by a digit n, where n is in the range 1-9, matches the same string that was matched by sub-expression n. For example the expression:

```
^(a*).*\1$
```

Will match the string:

```
aaabbaaa
```

But not the string:

```
aaabba
```

You can also use the `\g` escape for the same function, for example:

Escape	Meaning
\g1	Match whatever matched sub-expression 1
\g{1}	Match whatever matched sub-expression 1: this form allows for safer parsing of the expression in cases like \g{1}2 or for indexes higher than 9 as in \g{1234}
\g-1	Match whatever matched the last opened sub-expression
\g{-2}	Match whatever matched the last but one opened sub-expression
\g{one}	Match whatever matched the sub-expression named "one"

Finally the `\k` escape can be used to refer to named subexpressions, for example `\k<two>` will match whatever matched the subexpression named "two".

Alternation

The | operator will match either of its arguments, so for example: abc|def will match either "abc" or "def".

Parenthesis can be used to group alternations, for example: ab(d|ef) will match either of "abd" or "abef".

Empty alternatives are not allowed (these are almost always a mistake), but if you really want an empty alternative use (?:) as a placeholder, for example:

|abc is not a valid expression, but

(?:)|abc is and is equivalent, also the expression:

(?:abc)?? has exactly the same effect.

Character Sets

A character set is a bracket-expression starting with [and ending with], it defines a set of characters, and matches any single character that is a member of that set.

A bracket expression may contain any combination of the following:

Single Characters

For example [abc], will match any of the characters 'a', 'b', or 'c'.

Character Ranges

For example [a-c] will match any single character in the range 'a' to 'c'. By default, for Perl regular expressions, a character x is within the range y to z, if the code point of the character lies within the codepoints of the endpoints of the range. Alternatively, if you set the collate flag when constructing the regular expression, then ranges are locale sensitive.

Negation

If the bracket-expression begins with the ^ character, then it matches the complement of the characters it contains, for example [^a-c] matches any character that is not in the range a-c.

Character Classes

An expression of the form [[:name:]] matches the named character class "name", for example [[:lower:]] matches any lower case character. See character class names.

Collating Elements

An expression of the form [[.col.]] matches the collating element col. A collating element is any single character, or any sequence of characters that collates as a single unit. Collating elements may also be used as the end point of a range, for example: [[.ae.]-c] matches the character sequence "ae", plus any single character in the range "ae"-c, assuming that "ae" is treated as a single collating element in the current locale.

As an extension, a collating element may also be specified via it's symbolic name, for example:

```
[[.NUL.]]
```

matches a \0 character.

Equivalence Classes

An expression of the form [[=col=]], matches any character or collating element whose primary sort key is the same as that for collating element col, as with collating elements the name col may be a symbolic name. A primary sort key is one that ignores case, accentation, or locale-specific tailorings; so for example [[=a=]] matches any of the characters: a, À, Á, Â, Ã, Ä, Å, A, à, á, â, ã, ä and å. Unfortunately implementation of this is reliant on the platform's collation and localisation support; this feature can not be relied upon to work portably across all platforms, or even all locales on one platform.

Escaped Characters

All the escape sequences that match a single character, or a single character class are permitted within a character class definition. For example [\[\]] would match either of [or] while [\W\d] would match any character that is either a "digit", or is not a "word" character.

Combinations

All of the above can be combined in one character set declaration, for example: [[:digit:]a-c[.NUL.]].

Escapes

Any special character preceded by an escape shall match itself.

The following escape sequences are all synonyms for single characters:

Escape	Character
\a	\a

\e	0x1B
\f	\f formfeed
\n	\n newline
\r	\r carriage return
\t	\t tab
\v	\v vertical tab
\b	\b (but only inside a character class declaration).
\cX	An ASCII escape sequence - the character whose code point is X % 32
\xdd	A hexadecimal escape sequence - matches the single character whose code point is 0xdd.
\x{dddd}	A hexadecimal escape sequence - matches the single character whose code point is 0xdddd.
\0ddd	An octal escape sequence - matches the single character whose code point is 0ddd.
\N{name}	Matches the single character which has the symbolic name name. For example \N{newline} matches the single character \n.

"Single Character" Character Classes

Any escaped character x, if x is the name of a character class shall match any character that is a member of that class, and any escaped character X, if x is the name of a character class, shall match any character not in that class.

The following are supported by default:

Escape sequence	Equivalent to
\d	[[:digit:]]
\l	[[:lower:]]
\s	[[:space:]]
\u	[[:upper:]]
\w	[[:word:]]
\h	Horizontal whitespace
\v	Vertical whitespace
\D	[^[:digit:]]
\L	[^[:lower:]]
\S	[^[:space:]]
\U	[^[:upper:]]
\W	[^[:word:]]
\H	Not Horizontal whitespace
\V	Not Vertical whitespace

Character Properties

The character property names in the following table are all equivalent to the names used in character classes.

Form	Description	Equivalent character set form
\pX	Matches any character that has the property X.	[[:X:]]
\p{Name}	Matches any character that has the property Name.	[[:Name:]]
\PX	Matches any character that does not have the property X.	[^[:X:]]
	Matches any character that does not have the property	

\P{Name}	Name.	[^[:Name:]]

For example \pd matches any "digit" character, as does \p{digit}.

Word Boundaries

The following escape sequences match the boundaries of words:

\< Matches the start of a word.

\> Matches the end of a word.

\b Matches a word boundary (the start or end of a word).

\B Matches only when not at a word boundary.

Buffer Boundaries

The following match only at buffer boundaries: a "buffer" in this context is the whole of the input text that is being matched against (note that ^ and $ may match embedded newlines within the text).

\` Matches at the start of a buffer only.

\' Matches at the end of a buffer only.

\A Matches at the start of a buffer only (the same as \\\`).

\z Matches at the end of a buffer only (the same as \\').

\Z Matches a zero-width assertion consisting of an optional sequence of newlines at the end of a buffer: equivalent to the regular expression (?=\v*\z). Note that this is subtly different from Perl which behaves as if matching (?=\n?\z).

Continuation Escape

The sequence \G matches only at the end of the last match found, or at the start of the text being matched if no previous match was found. This escape useful if you're iterating over the matches contained within a text, and you want each subsequence match to start where the last one ended.

Quoting Escape

The escape sequence \Q begins a "quoted sequence": all the subsequent characters are treated as literals, until either the end of the regular expression or \E is found. For example the expression: \Q*+\Ea+ would match either of: *+a *+aaa

Unicode Escape

\C Matches a single code point: this has exactly the same effect as a "." operator.

\X Matches a combining character sequence: that is any non-combining character followed by a sequence of zero or more combining characters.

Matching Line Endings

The escape sequence \R matches any line ending character sequence, specifically it is identical to the expression (?>\x0D\x0A?|[\x0A-\x0C\x85\x{2028}\x{2029}]).

Keeping Back Some Text

\K Resets the start location of $0 to the current text position: in other words everything to the left of \K is "kept back" and does not form part of the regular expression match. $` is updated accordingly.

For example foo\Kbar matched against the text "foobar" would return the match "bar" for $0 and "foo" for $`. This can be used to simulate variable width lookbehind assertions.

Any Other Escape

Any other escape sequence matches the character that is escaped, for example \@ matches a literal '@'.

Perl Extended Patterns

Perl-specific extensions to the regular expression syntax all start with (?.

Named Sub-expression

You can create a named subexpression using:

```
(?<NAME>expression)
```

Which can be then be refered to by the name NAME. Alternatively you can delimit the name using 'NAME' as in: (?'NAME'expression)

These named subexpressions can be refered to in a backreference using either \g{NAME} or \k<NAME> and can also be refered to by name in a Perl format string for search and replace operations, or in the match_results member functions.

Comments

(?# ...) is treated as a comment, it's contents are ignored.

Modifiers

(?imsx-imsx ...) alters which of the perl modifiers are in effect within the pattern, changes take effect from the point that the block is first seen and extend to any enclosing). Letters before a '-' turn that perl modifier on, letters afterward, turn it off.

(?imsx-imsx:pattern) applies the specified modifiers to pattern only.

Modifier	Description
i	Case sensitivity. When on regular expression is case insensitive
m	line boundaries. Make ^ and $ match line boundaries within a string
s	match anything. Make . match newlines
x	extended regular expressions allow whitespace and comments in regular expressions

Non-marking Groups

(?:pattern) lexically groups pattern, without generating an additional sub-expression.

Branch Reset

(?|pattern) resets the subexpression count at the start of each "|" alternative within pattern.

The sub-expression count following this construct is that of whichever branch had the largest number of sub-expressions. This construct is useful when you want to capture one of a number of alternative matches in a single sub-expression index.

In the following example the index of each sub-expression is shown below the expression:

```
# before  ---------------branch-reset----------- after
/ ( a )  (?| x ( y ) z | (p (q) r) | (t) u (v) ) ( z ) /x
# 1            2         2  3        2     3     4
```

Lookahead

(?=pattern) consumes zero characters, only if pattern matches.

(?!pattern) consumes zero characters, only if pattern does not match.

Lookahead is typically used to create the logical AND of two regular expressions, for example if a password must contain a lower case letter, an upper case letter, a punctuation symbol, and be at least 6 characters long, then the expression:

```
(?=.*[[:lower:]])(?=.*[[:upper:]])(?=.*[[:punct:]]).{6,}
```

could be used to validate the password.

Lookbehind

(?<=pattern) consumes zero characters, only if pattern could be matched against the characters preceding the current position (pattern must be of fixed length).

(?<!pattern) consumes zero characters, only if pattern could not be matched against the characters preceding the current position (pattern must be of fixed length).

Independent Sub-expressions

(?>pattern) pattern is matched independently of the surrounding patterns, the expression will never backtrack into pattern. Independent sub-expressions are typically used to improve performance; only the best possible match for pattern will be considered, if this doesn't allow the expression as a whole to match then no match is found at all.

Recursive Expressions

(?N) (?-N) (?+N) (?R) (?0)

(?R) and (?0) recurse to the start of the entire pattern.

(?N) executes sub-expression N recursively, for example (?2) will recurse to sub-expression 2.

(?-N) and (?+N) are relative recursions, so for example (?-1) recurses to the last sub-expression to be declared, and (?+1) recurses to the next sub-expression to be declared.

Conditional Expressions

(?(condition)yes-pattern|no-pattern) attempts to match yes-pattern if the condition is true, otherwise attempts to match no-pattern.

(?(condition)yes-pattern) attempts to match yes-pattern if the condition is true, otherwise fails.

condition may be either: a forward lookahead assert, the index of a marked sub-expression (the condition becomes true if the sub-expression has been matched), or an index of a recursion (the condition become true if we are executing directly inside the specified recursion).

Here is a summary of the possible predicates:

`(?(?=assert)yes-pattern|no-pattern)`
: Executes yes-pattern if the forward look-ahead assert matches, otherwise executes no-pattern.

`(?(?!assert)yes-pattern|no-pattern)`
: Executes yes-pattern if the forward look-ahead assert does not match, otherwise executes no-pattern.

`(?(R)yes-pattern|no-pattern)`
: Executes yes-pattern if we are executing inside a recursion, otherwise executes no-pattern.

`(?(RN)yes-pattern|no-pattern)`
: Executes yes-pattern if we are executing inside a recursion to sub-expression N, otherwise executes no-pattern.

`(?(DEFINE)never-exectuted-pattern)`
: Defines a block of code that is never executed and matches no characters: this is usually used to define one or more named sub-expressions which are refered to from elsewhere in the pattern.

Operator Precedence

The order of precedence for operators is as follows:

1. Collation-related bracket symbols [==] [::] [..]
2. Escaped characters \
3. Character set (bracket expression) []
4. Grouping ()
5. Single-character-ERE duplication * + ? {m,n}
6. Concatenation
7. Anchoring ^$
8. Alternation |

What Gets Matched

If you view the regular expression as a directed (possibly cyclic) graph, then the best match found is the first match found by a depth-first-search performed on that graph, while matching the input text.

Alternatively:

The best match found is the leftmost match, with individual elements matched as follows;

Construct	What gets matched
AtomA AtomB	Locates the best match for AtomA that has a following match for AtomB.
Expression1 | Expression2	If Expresion1 can be matched then returns that match, otherwise attempts to match Expression2.
S{N}	Matches S repeated exactly N times.
S{N,M}	Matches S repeated between N and M times, and as many times as possible.
S{N,M}?	Matches S repeated between N and M times, and as few times as possible.
S?, S*, S+	The same as S{0,1}, S{0,UINT_MAX}, S{1,UINT_MAX} respectively.
S??, S*?, S+?	The same as S{0,1}?, S{0,UINT_MAX}?, S{1,UINT_MAX}? respectively.
(?>S)	Matches the best match for S, and only that.
(?=S), (?<=S)	Matches only the best match for S (this is only visible if there are capturing parenthesis within S).
(?!S), (?<!S)	Considers only whether a match for S exists or not.
(?(condition)yes-pattern | no-pattern)	If condition is true, then only yes-pattern is considered, otherwise only no-pattern is considered.

SEE ALSO

Pin Functions

DESCRIPTION

This section describes the pin function definitions used for ESP automatically generated testbenches.

The pin function is set with the *-function* switch of the **set_testbench_pin_attributes** command. The pin function is use when writing the testbench. The following table defines the possible pin function values and their interpretation.

address	address pins, such as addr[10:0]
data	data pins, such as data[15:0]A
outen	output enable pin, such as oe, enabled for read operation
chipen	chip enable pin, such as ce, me, enabled for both read and write operation
write	write enable pins, such as we, we[10:0], enabled for write operation and disabled for read operation
read	read enable pins, such as re, enabled for read operation and disabled for write operation
reset	reset pin, such as rst, set, clear, enabled only for reset cycles, and disabled for the rest of the cycles
control	when it is hard to tell the function of an input, it can be specified as control. This pin would be tested symbolically to cover both the enabled and disabled situation. This is useful when the active low or active high is not clear, or for a write bit/byte select where both active low and active high are possible. During all binary cycles, a control signal is set to its inactive input value
binary	it is set to be a constant binary value all the time, and it holds the active value all the time
writemask	is set on input ports. Functionality can be active, inactive, or symbolic.
readmask	is set on input ports. Functionality can be active, inactive, or symbolic.
writereadmask	is set on input ports. Functionality can be active or symbolic.

Pins for test modes, power modes and redundancy should be specified as control which is the default function.

SEE ALSO

```
Commands:
set_testbench_pin_attributes
write_testbench

Variables:

Topics: output_checkers
```

Power Integrity Verification

DESCRIPTION

ESP can be used for power integrity verification. Power integrity verification uses a special mode in the ESP Shell called `inspector`. In inspector mode ESP can be used to:

- Check for electrical design rule violations
  - shorted power and ground nets
  - power domain isolation bugs
  - missing or incorrectly designed level shifters
  - sneak paths between supplies (including through virtual power ground)
  - static check for unbuffered inputs
  - floating outputs
- Check for types of power mode violations
  - check for specific supply voltages on nets at specific times
  - check for specific pull-up/pull-down paths on nets at specific times
  - check that pull-up/pull-down paths are disabled at specific times
  - check for incorrect output voltage

Enabling Power Integrity verification

To enable power integrity verification follow these steps.

1. set_verify_mode inspector
2. read Verilog source
3. read SPICE design
4. match ports
5. configure SPICE devices with **set_device_model** or **set_process** command
6. optionally configure power integrity rules using **set_inspector_rules** .
7. define power domains using **set_power_domain**
8. define any needed constraints using **set_constraint**
9. set testbench style with **set_app_var** **testbench_style**
10. run verify
11. look at results with **report_inspector_results** .
12. Debug with waveforms, **IST** or SPICE.

SEE ALSO

Commands:
remove_power_domain
report_inspector_results
report_inspector_rules
report_power_domains
report_testbench_pins
reset_inspector_rules
set_device_model
set_inspector_rules
set_power_domain
set_process
set_verify_mode

Topics:
device_model_simulation

quit

NAME

quit

Exits the shell.

SYNTAX

```
quit
```

ARGUMENTS

None.

DESCRIPTION

This command exits from the application. It is basically a synonym for **exit** with no arguments.

The shell will report the maximum memory used and the total CPU time used.

EXAMPLES

The following example exits the current session.

```
shell> quit
Maximum memory usage for this session: 12 MB
CPU usage for this session: 0.01 seconds
Thank you for using ESP (R)!
```

SEE ALSO

Commands: **exit**

Redundancy Verification

SUMMARY

In order to improve yield on arrayed blocks such as memories, designers may implement redundant cells which can be used instead of defective cells, based on control values determined after post-silicon testing. These control values are normally programmed using fuses external to the memory block, and just look like normal control inputs to the block being verified. Potentially, these fuses may also be part of the block being tested.

Two common forms of redundancy are row redundancy and column-redundancy, which may be used separately or together.

ESP can be used to verify that this redundancy logic functions correctly.

DESCRIPTION

Redundancy verification uses the standard **ESP shell flow** with some modifications. The flow modifications :

- Identify nets that are remapped by the redundancy controls.
- Identify the redundancy control inputs
- Specify the defect tolerance for the redundancy logic

Use the **set_net_group** command to define a set of nets that represent the defects to be remapped by redundancy logic. For a memory this would typically be the row or column selection nets.

Use the **set_constraint** command with the *-symbolic_static* option to define a set of static symbols on the inputs that are the redundancy mapping controls.

Use the **set_symbol_to_pass** to identify which inputs are used to actually map the redundant logic. These inputs will be tested to ensure some value set will map out all the individual defects as defined by the **set_net_group** and **set_stuck_fault_group** commands.

Use the **set_stuck_fault_group** command to define the remapping capabilities in terms of the number of defects that can be remapped within a given net group.

EXAMPLES

Verify redundancy on a memory with support for remapping of 3 rows and 2 columns.

```
# The following commands are added to the normal verification
#  to enable redundancy checking
set_net_group -i -design ARRAY -net RA*  ROWS     # row selection
set_net_group -i -design ARRAY -net CA*  COLUMNS  # column selection
set_constraint -symbolic_static {RRA1 RRA2 RRA3 RCA1 RCA2 REN}
set_symbol_to_pass {RRA1 RRA2 RRA3 RCA1 RCA2 REN}
set_stuck_fault_group -i -number 3 ROWS
set_stuck_fault_group -i -number 2 COLUMNS
```

SEE ALSO

Commands: **report_net_groups** , **report_stuck_fault_groups** , **set_constraint** , **set_net_group** , **set_stuck_f**

Scan Chain Verification

Overview

This topic discusses how to modify the ESP tool default symbolic testbench to fully verify the scan chain in a design. This test is only used when both designs have the full scan chain defined.

The general flow is to use the INIT cycle to initialize the entire scan chain to a known value. Next, two symbolic cycles are used to load the scan chain with symbols from all the inputs. Finally, the flush cycles perform a series of scan outs to flush the entire scan chain and check the results.

This style of scan testing will ensure that both the reference and the implementation have the same scan chain order.

Methodology

The scan test consists of 0 binary cycles, followed by 2 symbolic cycles and finally N flush cycles. Where N is 1 more than the length of the longest scan chain. This approach is appropriate for 1 or more scan chains. Define this set of cycles with these commands:

```
set maxScanChainDepth 42
set_app_var testbench_style symbolic
set_app_var testbench_binary_cycles 0
set_app_var testbench_symbolic_cycles 2
set_app_var testbench_flush_cycles [expr $maxScanChainDepth + 1]
```

Change the value **42** in the example above to be the scan chain depth of your specific design. If there is more than one scan chain, set maxScanChainDepth to the depth of the longest scan chain in the design to be verified.

Modifying the INIT cycle

The symbolic testbench has a simple INIT task that sets all the inputs to the inactive state and applies several clock cycles. Outputs are not checked during this INIT cycle.

The INIT task needs to be modified if a scan chain has internal scan flops that do not directly capture inputs values Designs with serial load of the repair address are one example where scan flops do not directly capture inputs values.

The ESP tool has the testbench_initialization_file variable to provide the name of a Verilog source code file that will modify the INIT cycle as defined by the INIT task. For scan verification, create Verilog code that loads 0 into every scan flop in every scan chain.

For example if the scan enable is named SE and the scan clock is named CLK. The following Tcl code will create a file INIT.v that has Verilog code to load all of the scan flops.

```
# Place this code after all set_testbench_pin_attribute statements
#  in ESP Tcl script.
# Use of this code requires the use of -sverilog in previous read_verilog commands
#
set tif INIT.v
file delete -force $tif
if {[catch {open $tif w} fid]} {
  puts stderr "Could not open file $tif for writing\n$fid"
  exit 1
}
#  now for the Verilog code
#  default clock already at first edge when $tif included
#  change CLK to proper clock for design
#  change SE to proper scan enable for design
#
set CLK CLK
set SE SE
puts $fid "apply_global_constraints;"
puts $fid "#(`CLOCKPHASE);"
puts $fid "$CLK = 1'b0;"
puts $fid "#(`CLOCKPHASE - `SETUPTIME);"
puts $fid {inno_cycle="SCANINIT";}
puts $fid "for (int i=0; i<$maxScanChainDepth; i++) begin"
puts $fid "  $SE=1'b1;"
puts $fid "  apply_global_constraints;"
puts $fid "  #(`SETUPTIME); // Advance to clock edge"
puts $fid "  $CLK = 1'b1;"
puts $fid "  #(`CLOCKPHASE);"
puts $fid "  $CLK = 1'b0;"
puts $fid "  #(`CHECKTIME);"
puts $fid "end"
puts $fid "$SE=1'b0;"
puts $fid {inno_cycle="INIT";}
puts $fid "#(`SETUPTIME);"
puts $fid "$CLK = 1'b1;"
close $fid
set_app_var testbench_initialization_file $tif
```

Modifying the other cycles

Add a set of constraints on the scan enable port so that the scan operation starts in symbolic cycle 2 and continues through all of the flush cycles. The following example shows how to constrain the scan enable port named SE.

```
set_constraint SE -set {1'b1}
# if you need binary cycles then following loop ensures scan disabled in binary cycles
for {set i=1} {$i<[get_app_var testbench_binary_cycles]} {incr i} {
  set_constraint SE -set {1'b0} -cycle "BINCYCLE$i"
}
set_constraint SE -set {1'b0} -cycle {INIT}
set_constraint SE -set {1'b0} -cycle {SYMCYCLE1}
```

The first symbolic cycle will parallel load the scan chain. Then the second symbolic cycle starts the scan operation.

Code Coverage

Code coverage for this testbench will show that symbolic cycle 2 symbols for all inputs other than scan in will be dropped. This is expected. These dropped symbols should be caught in other testbenches that are required for complete design verification.

source

NAME

source

SYNTAX

```
source  [-echo] [-verbose]  [-continue_on_error] <file_name>
```

ARGUMENTS

<file_name>	Specifies the pathname of a file containing commands to be executed.
-echo	Used during TCL Mode, echoes the commands in the file_name and displays them in the GUI transcript or shell xterm. Note that this option is a non-standard extension to Tcl.
-verbose	Displays the result of each command executed. Note that error messages are displayed regardless. Also note that this option is a non-standard extension to Tcl.
-continue_on_error	Specify this option to continue the script even if an error is encountered that is not otherwise handled. Use of this switch is preferred over the use of the **sh_continue_on_error** application variable

DESCRIPTION

Use this command for executing commands contained in an external file. The command history list will only contain this command and not include commands inside the *command_file* file.

EXAMPLES

```
> source noabort
> source my_setup.cmd
```

SEE ALSO

```
Topics: startup_command_files
```

SPICE to SPICE

Verifying SPICE to SPICE

The ESP tool can be used to verify that two SPICE designs have the same functionality. The flow is similar to the default Verilog to SPICE flow with these differences:

- SPICE to SPICE verification cannot use the legacy parser.
- The reference container can contain a SPICE design instead of a Verilog design. Use the -r option to the read_spice command to place SPICE design data into the reference container.
- The directions of all the ports will be inout. Use the -direction option of the set_testbench_pin_attributes command to define the correct port direction: input or output.
- The SPICE transistor technology can be different in the two containers. Use the -i or -r option to the set_process command with different ESP device model files to have different transistor models used for the two containers.

Note:
When both containers are SPICE, ESP generates a testbench that uses the strict compare checker for outputs instead of the comparex checker that are created for Verilog to SPICE verification.

Examples

Verify a RAM design in two different SPICE technologies. The technology files are tech1.edm and tech2.edm. The SPICE files are ram_tech1.sp and ram_tech2.sp.

```
esp -f run.tcl

Where run.tcl contains:

set_process -r -technology_file tech1.edm
read_spice -r ram_tech1.sp
set_top_design -r RAM
set_process -i -technology_file tech2.edm
read_spice -i ram_tech2.sp
set_top_design -i RAM
# define supplies for both designs
set_supply_net -r VDD -design RAM -type real -logic 1
set_supply_net -r VSS -design RAM -type real -logic 0
set_supply_net -i VDD -design RAM -type real -logic 1
set_supply_net -i VSS -design RAM -type real -logic 0
match_design_ports
set_testbench_pin_attributes -direction output -checker compare Q
set_testbench_pin_attributes -direction input -function read RE
set_testbench_pin_attributes -direction input -function address A
set_testbench_pin_attributes -direction input -function clock CLK
set_testbench_pin_attributes -direction input -function data DI
set_testbench_style sram
verify
exit
```

Startup Command Files

Description

ESP automatically executes *command file*'s from multiple locations upon startup.

The sequence of files executed is:

- `$SYNOPSYS/admin/setup/.synopsys_esp.setup` in the installation directory
- `$HOME/.synopsys_esp.setup` in the HOME directory
- `.synopsys_esp.setup` in the current working directory

Each startup command file is just like any other command file, except that it is executed automatically at startup. Any commands that are legal for command files may be used.

Startup command files are executed prior to any command file specified in the ESP invocation line. Specifying a command file containing ESP commands may be done in a number of ways:

```
esp_shell command_file [other_args]...
esp_shell  [other_args]... < command_file
```

Within a script as a "here" document:

```
  #!/bin/sh
  esp_shell [other_args] -file <<!
  source command_file
  exit -force
  !
  % esp_shell [other_args]
  source command_file
```

By default, commands in startup files are not echoed to the transcript. To see the commands as they are executed, place a set messages display command at the beginning of the startup file.

To invoke ESP without executing any startup command files, use the -no_init switch:

```
  % esp_shell -no_init
```

SEE ALSO

ESP Specific System Tasks

Overview

ESP defines a number of system tasks to extend Verilog to support symbolic simulation and for gathering coverage information. ESP also includes system tasks to support the **FSDB** waveform dumping format.

Simulation

The following Verilog system tasks are defined to support simulation, debug and modeling.

	Binary Testvector Creation
$esp_error	Generate a testvector and issue and error
$esp_context	Define simulation error context for shell
$esp_gen	Generate a testvector with comment
	Symbol Definition
$esp_var	Define a symbol
$esp_timevar	Define a time symbol
$esp_ctrlvar	Define a control symbol
$esp_addrvar	Define an address symbol
$esp_datavar	Define a data symbol
$esp_retire	Remove a symbol definition
	Constraints
$esp_exclusive	Constrain a set of buses to unique values
$esp_const_one	Returns true if value is binary 1
$esp_const_zero	Return true if value is binary 0
$esp_const	Returns true if value is binary 0,1 X or Z
$esp_contains	Returns true if symbolic variable in equation for 1st argument
	Miscellaneous
$esp_smminit	Efficiently load memory with binary value
$esp_multi_select	Locate cause of X in switch level design
$esp_onehotindex	Decode onehot bus to integer

ESP Symbolic Coverage System Tasks

A number of system tasks have been implemented to assist in gathering symbolic coverage information. For more information about how symbolic coverage works, see the **ESP Coverage User Guide**

Waveform Dumping

In addition to the standard Verilog **VCD** format, ESP supports the **FSDB** format.

SEE ALSO

Topics: **esp_addrvar** **esp_const** **esp_const_one** **esp_const_zero** **esp_contains** **esp_context** **esp_ctrlvar** **esp_da**

Testbench Customization

OVERVIEW

ESP generates testbenches as part of the normal verification flow. For many designs these automatic testbenches only require minor modifications for a successful verification.

The generated testbenches are structured so that individual parts of the testbench can be replaced by the verification specialist without hand editing the generated testbench. This is accomplished by providing for the inclusion of four different files in the generated testbench. Each of these files has a specific purpose. These files are:

setup
: This file should be used to specify all the testbench overrides.

declaration
: This file is used to define registers, wires, functions, tasks and any other module level Verilog code to be added to the testbench.

initialization
: This file is used to define the device under test (DUT) initialization sequence.

global constraints
: This file is used to define complex global constraints on inputs that can not be specified by the **set_constraint** command.

Each individually replaceable section of the testbench is wrapped by Verilog compiler directives. These sections use a naming convention to define the macro name that must be defined to remove the conditional section of the testbench. This macro is called a guard. The macro name is:

```
REMOVE_DEFAULT_<name of section>
```

For example the define statements section by default looks like:

```
`ifdef REMOVE_DEFAULT_define_statements
`else
`define MAX_PARAM_SIZE 255
`define CLOCKPERIOD 40
`define SETUPTIME   10
`define CLOCKPHASE  (`CLOCKPERIOD/2.0)
`define CHECKTIME   (`CLOCKPHASE - `SETUPTIME)
`endif
```

The guard is the macro *REMOVE_DEFAULT_defines_statements*.

The general structure of the testbench is:

```
`timescale 1ns/1ps
module inno_tb_top;
  `include setup_file
```

```
    defines_statements
    reg_wire_statements
    ref_instantiation
    imp_instantiation
    ESPsym
    `include declaration_file
    sim_cycle_initial_block
    check_<output>
    display_inputs
    display_inputs_<clock>
    force_<input>
    force_all_hierarchy_signals
    assign_buffer_value_<clock>
    apply_global_constraints
      `include  constraints_file
    apply_contention_check_<clock>
    apply_global_constraints_<clock>
      `include  constraints_file
    apply_symbol_masks
    apply_symbol_masks_<input>
    INIT
      `include initialization_file
    INIT_<clock>
      `include initialization_file
    apply_vector
    apply_vector_phase2
    apply_vector_<clock>
  endmodule
```

The Verilog variable `esp_testbench_style` is defined in all generated testbenches to allow testbench specific modifications. Several testbench styles like SRAM actually generate a suite of testbenches. This macro will have one of the following values:

symbolic	Symbolic testbench.
dataint	Data integrity
protocol	Protocol
dualphs	Two phase - assert twice, check twice
clkenum	Clock enumeration
clkwave	Clock wave
romhold	Dual phase ROM hold test
cammatch	CAM match integrity test
holdchk	Hold time X value test
portcov	Multiple data integrity tests for each clock
library	Library cell style test

DESCRIPTION

Many verifications can be completed by using only the Tcl shell commands to affect the generated testbench. Changing the style of testbench, changing the number of testbench cycles, applying input constraints and even the name of the testbench module itself can be done with Tcl commands. When Tcl shell commands are not sufficient to modify a testbench, then the user can write their own versions of sections of the testbench. These modifications can be done without actually editing the generated testbench.

To replace the INIT section of the testbench you can place the following code into the setup file:

```
`define REMOVE_DEFAULT_INIT
```

Next place the code to replace the INIT section into the declarations file. An example is:

```
task INIT;
begin
  // It is good practice to set inno_cycle
  inno_cycle = "INIT";
  // put your Verilog code here
end
endtask
```

The defined section names are:

- REMOVE_DEFAULT_INIT
- REMOVE_DEFAULT_INIT_<clock>
- REMOVE_DEFAULT_ESPsym
- REMOVE_DEFAULT_apply_contention_check_<clock>
- REMOVE_DEFAULT_apply_global_constraints
- REMOVE_DEFAULT_apply_global_constraints_<clock>
- REMOVE_DEFAULT_apply_symbol_masks
- REMOVE_DEFAULT_apply_symbol_masks_<input>
- REMOVE_DEFAULT_apply_vector
- REMOVE_DEFAULT_apply_vector_<clock>
- REMOVE_DEFAULT_apply_vector_phase2
- REMOVE_DEFAULT_assign_buffer_value_<clock>
- REMOVE_DEFAULT_check_<output>
- REMOVE_DEFAULT_define_statements
- REMOVE_DEFAULT_display_inputs
- REMOVE_DEFAULT_display_inputs_<clock>
- REMOVE_DEFAULT_force_<input>
- REMOVE_DEFAULT_force_all_hierarchy_signals
- REMOVE_DEFAULT_imp_instantiation
- REMOVE_DEFAULT_ref_instantiation
- REMOVE_DEFAULT_reg_wire_statements
- REMOVE_DEFAULT_sim_cycle_initial_block

Where:

- <clock> is a clock port (-function clock)
- <input> is an input or inout port
- <output> is an output or inout port

Not all sections appear in all styles of testbench. Sections with *<clock>* only will occur when more than one clock is defined in a design.

The following sections are defined outside of the main testbench files:

- ESPCOV
- VCDDUMP

These two macros must be undefined in order to remove their respective sections of the testbench.

Several of the sections are related. If one is replaced then the related sections also have to be replaced. These relationships are explained within the section descriptions.

Setup File

This file is used to define any macros used to replace any of the testbench sections. This file will typical consist of a set of `define statements.

Generally, no other Verilog code would appear in this file. Verilog code should be placed in one of the other include files.

Only place Verilog code in this file when one or more of the following is true:

- Replacing the defines section
- Replacing the signal definition section
- Replacing the instantiations of the reference or implementation designs
- Replacing the symbolic variables definition section

The setup file is defined by the Tcl variable **testbench_setup_file** .

Declarations File

This file is used to hold any Verilog code that is legal within a module scope. It will be included after the default symbolic variables definition section.

The declaration file is defined by the Tcl variable **testbench_declaration_file** .

Initialization File

This file is included in the default testbench in the INIT task just before the first call to *apply_global_constraints*.

This file can hold any Verilog code that is legal within a begin/end block. Typically, this file is used to specify complex initialization sequences.

The initialization file is defined by the Tcl variable **testbench_initialization_file** .

Constraint File

This file is included at the beginning of the *apply_global_constraints* task.

This file can contain any Verilog code that is legal within a Verilog task. Typically, this file is used to specify complex constraints that are difficult to express using the Tcl command **set_constraint** .

The constraint file is defined by the Tcl variable **testbench_constraint_file** .

INIT Task

The INIT task is guarded by the REMOVE_DEFAULT_INIT macro. If there is more than one clock then the clock name a appended to the name of the INIT task and these tasks are guarded by the REMOVE_DEFAULT_INIT_<clock> macro.

This INIT task is intended to be used to initialize the design into a known good starting state. Symbolic values are not generally used in the INIT task.

Remember to call *apply_global_constraints* after you set inputs in a replacement INIT task.

Symbolic Variable Declaration Block

The symbolic variable declaration block is guarded by the REMOVE_DEFAULT_ESPsym macro.

Within the block the actual creations of symbols if further guarded by the *ESPsym* macro. When ESP is running in symbolic mode then symbolic variables are created. When ESP is running in binary mode or another simulator is simulating the testbench then values for the symbolic variables are read from the *esp.TestVector* file.

Contention Checking

The apply contention checking task is guarded by the REMOVE_DEFAULT_apply_contention_check_<clock> macro.

This task only exists in the *mpt* testbench created by setting the testbench style to clkwave.

This task warns of a possible setup contention between the named clocks and other clocks.

Apply Global Constraints Task

The apply global constraints task is guarded by the REMOVE_DEFAULT_apply_global_constraints macro. If there is more than one clock defined, then the clock name is used as part of the task name and the REMOVE_DEFAULT_apply_global_constraints_<clock> macros are used as guards.

apply_global_constraints calls *apply_symbolic_mask* if the current testbench is using symbolic masks.

Apply Symbol Mask Task

The apply symbolic masking task is guarded by the REMOVE_DEFAULT_apply_symbol_masks macro. If there is more then one clock, then the clock name is used as part of the task name and the REMOVE_DEFAULT_apply_symbol_masks_<input> macros are used as guards.

Not all testbench styles use symbolic masks. The symbolic mask task is called from the *apply_global_constraints* task.

Apply Vector Task

The apply vector task is guarded by the REMOVE_DEFAULT_apply_vector macro.

In the two phase testbench (*.2ph) there is a second apply vector task that is guarded by the REMOVE_DEFAULT_apply_vector_phase2 macro.

If there is more than one clock defined then the apply vector task will have separated names for each of the clocks. These tasks are guard by the REMOVE_DEFAULT_apply_vector_<clock> macros. Where <clock> is the name of the clock.

The apply vector set of tasks actually apply the verification stimulus. These tasks call the global constraints task. These tasks also call the output checker routines.

Assign Buffer Value Task

The assign buffer task value task is guarded by the REMOVE_DEFAULT_assign_buffer_value_<clock> macro.

The actual inputs to the reference design and implementation design are buffered from the apply vector task and INIT task. This allows the global constraints task to modify the inputs without causing input glitches.

The assign buffer value task applies the possibly constrained testbench inputs to the actual reference and implementation inputs.

Output Checker Tasks

Each output will have it's own output checker task.

REMOVE_DEFAULT_check_<output>

General Testbench Settings

The section of code that controls various defaults in the testbench is guarded by the REMOVE_DEFAULT_define_statements macro.

The information in this section includes the clock setup time, clock period, cycle time information and the default size of parameters for the *inno* functions.

This section should general not be replaced. This section has impacts on the apply vectors task and the INIT task sections.

The only time you should have to override the default size for parameters to the *inno* functions is if there is a bus signal wider than 255. The macro to replace is *MAX_PARAM_SIZE*.

The clock period, setup time and cycle time can all be controlled by the **create_clock** command.

The default names for the clock macros are:

```
CLOCKPERIOD
SETUPTIME
CLOCKPHASE
CHECKTIME
PHASEDELAY
CLOCKPERIOD_<clock>
CLOCKPHASE_<clock>
PHASEDELAY_<clock>
SETUPTIME_<clock>
CHECKTIME_<clock>
```

The <clock> form of the clock macros are only used in the *mpt* testbench which is generated by setting the testbench style to clkwave.

The CHECKTIME and CHECKTIME_<clock> macros are calculated based upon the settings of other macros.

Changing the name of these macros requires that the apply vector task and INIT task sections also be replaced. These sections use the clock macros for vector timing.

Displaying Inputs

The displaying of input values is guarded by the REMOVE_DEFAULT_display_inputs macro. If there are multiple clocks then the clock name is appended to macro name. Inputs are displayed independently for each clock.

Each display input task can be replaced. These tasks are called from the default apply vector task and the default INIT task.

Forcing Inputs

When there are bidirectional ports ESP creates special FORCEINPUT signals to control the direction of these ports in the testbench. When the FORCEINPUT_<name> signal is 1 then the signal <name> is a testbench input. Otherwise, the signal <name> is an output and the associated output checker will compare the values between the reference and implementation designs.

Implementation Design Instantiation

The instantiation of the implementation design is guarded by the REMOVE_DEFAULT_imp_instantiation macro.

If the implementation design is removed and not replaced then all of the output checker tasks should also be replaced by empty task definitions to prevent testbench failure.

Reference Design Instantiation

The instantiation of the reference design is guarded by the REMOVE_DEFAULT_ref_instantiation macro.

If the reference design is removed and not replaced then all of the output checker tasks should also be replaced by empty task definitions to prevent testbench failure.

Register and Wire Declarations

The register and wire declaration section is guarded by the REMOVE_DEFAULT_reg_wire_statements macro.

This section declares all of the signals needed by the testbench. If you need to remove this section of code, you should consider using a fully custom testbench.

To add more signals to the testbench use the declarations file.

Changing this section has implications throughout the whole testbench. Exercise caution in modifying this section of the testbench.

This block is affected by the ports in the reference design, the number of clocks in the design and other testbench settings.

Simulation Cycle Initial Block

The simulation cycle initial block is guarded by REMOVE_DEFAULT_sim_cycle_initial_block macro.

This Verilog block is the main block that defines the actual verification process. This block normally invokes the INIT and apply_vector tasks.

This block is affected by the setting of:

- **testbench_style**
- **testbench_binary_cycles**
- **testbench_flush_cycles**
- **testbench_symbolic_cycles**

- **set_testbench_pin_attributes**

Custom Output Checkers

All testbenches created by the ESP tool change the value of the Verilog variable `esp_check_num` just before any set of output checks are performed. The number of these checks is controlled by the application variables **testbench_style** and **testbench_output_checks**.

Custom checkers can be added to the testbench using the declaration or setup files. The custom checker can be a new task that looks like other check task in the testbench or it can be freeform Verilog code. To have your checker invoked at the same time as other testbench checkers write a Verilog always block that is sensitive to changes in the `esp_check_num` variable.

```
// Code to add to a declaration or setup file to trigger custom output checker
always @(esp_check_num) begin
  // use case or if when checks are special for the edge
  case (esp_check_num)
    1: begin
       // first check in the test cycle
       // usually just before 1st clock edge
       // add code here if special 1st check
       end
    2: begin
       // second check
       //  usually before 2nd clock edge
       //  but if 4check style then before
       //  the input changes after the 1st clock
       end
    3: begin
       // 3rd check
       //  usually the last check at the end of test cycle
       //  but if 4check then this check right
       //    before the second clock edge
       end
    4: begin
       // 4th check
       //  last check of test cycle when 4check
       end
    default: begin
       // wait there are no other legal values
       //  safety check
       $display("%t: Warning illegal check number %d",$realtime,esp_check_num);
   endcase
 end
```

Custom checkers are not restricted to the design ports. They can use Verilog cross module references (XMRs) to reach inside a design to examine any net in the design.

Customer checkers do not have to check nets in both designs. They can check that a net in one design has a specific value. This is useful to check that a Verilog model that has power ports will produce all X outputs when the power ports are in an illegal state.

SEE ALSO

```
Commands:

Variables:

Topics:
```

Timescale

Summary

ESP uses a default timescale of 1ns/1ns for Verilog code. The SPICE design uses a timescale of 1ns/1ps.

Details

ESP uses a default timescale of 1n/1ns which is different from the VCS default of 1s/1s. It is recommended that all Verilog code use the `timescale command to explicitly set the timescale.

ESP behaves per the language specification in regards to the `timescale directive.

- If any source file has a `timescale directive then the first file that is parsed must have a `timescale directive before the first module, macromodule or primitive statement.
- A `timescale directive is in effect until the next `timescale directive is encountered. This can span multiple source files. **Note:**For designs with SPICE source files, any active `timescale directive is "suspended" while the ESP SPICE timescale of 1ns/1ps is used. Any Verilog file read after the SPICE files will resume the "suspended" timescale.
- Some ESP supplied Verilog library components (for example danmos/dapmos) are read as if they are SPICE source files. These components use the Verilog 2001 attribute of *(* esp_db_primitive *)*. This attribute is used by ESP to mark Verilog source as being from SPICE.

Transistor-only Simulation

What is it?

The ESP tool allows a transistor-only design to be simulated without requiring a Verilog reference design. To do this use simulate mode, instead of the default compare mode. Simulate mode can be used for Verilog-only designs by supplying your own testbench and this can also be done for transistor-only designs.

How do I do it?

The steps to perform the simulation include:

1. Enter simulate mode rather than compare mode **set_verify_mode**
2. Read the SPICE design into the implementation container with the **read_spice** command
3. Set the top level design with the **set_top_design** command
4. Set port directions using the -direction option of the **set_testbench_pin_attributes** command
5. Choose your custom testbench **set_testbench**
6. Simulate the design **verify**

In addition, if you want coverage information to be accumulated you will need to specify your top level design instance via **testbench_design_instance** , and if your custom testbench top level module name is not inno_tb_top you will need to specify that via **testbench_module_name** .

Example

```
set_verify_mode simulate
read_spice -i ram.sp
set_top_design -i RAM1R1W
set_testbench_pin_attributes CLK  -function Clock   -direction input
set_testbench_pin_attributes WE   -function Write   -direction input
set_testbench_pin_attributes RE   -function Read    -direction input
set_testbench_pin_attributes A    -function Address -direction input
set_testbench_pin_attributes DI   -function Data    -direction input
set_testbench_pin_attributes DOUT                   -direction output
write_esp_db -i ram.gv
set_app_var coverage true
set_app_var testbench_design_instance ximp
set_testbench "VT.sym.ximp"
verify
report_log
report_coverage -all -spec cov.spec -filterdetail
quit
```

This capability would normally be used to simulate binary input vectors but symbolic testbenches are also supported.

SEE ALSO

Commands: **set_verify_mode** , **set_testbench_pin_attributes** , **read_spice** , **set_testbench** , **verify**

Variables: **coverage** , **testbench_design_instance** , **testbench_module_name**

Topics:

Using Online Help

You can run ESP Online Help in most standard HTML browsers, such as Mozilla, Mozilla Firefox, SeaMonkey, Internet Explorer, and Safari.This section describes tips for setting up, launching, and using Online Help. The following topics are covered:

- #Setting Up Online Help
- #Launching Online Help
- #Searching in Online Help
- #Accessing User Guides and Release Notes
- #Running Online Help in Windows
- #Limitations

Setting Up Online Help

Note the following when configuring Online Help:

- If you are starting Online Help for the first time, and you have an existing Netscape profile, Mozilla will initially try to convert your profile. In this case, select "Do Not Convert."
- To set up a default browser for Help, do the following:
  1. If you want to use Firefox, specify the following:
     ```
     alias Firefox '</usr/bin>/firefox'


     (where </usr/bin> is a path on your network)
     ```
  2. If you want to use Mozilla, specify the following:
     ```
     alias mozilla '</usr/bin/>mozilla'
     ```
  3. Specify the following environment variable to use Firefox as your default browser for running Online Help:
     ```
     setenv USER_HELP_BROWSER /usr/bin/firefox
     ```
- If you are running ESP using Solaris/SparcOS VNC [TightVNC or RealVNC] versions 3.3 or older, the invocation of the web browser has been intentionally suppressed, as this configuration is known to crash vncserver. The workaround in this case is to initially invoke vncserver with a pixel depth of 8, as shown in the example below:
  ```
  vncserver -depth 8


  If you are sure your vncserver will not crash, you can force the web browser open by setti


  FORCE_BROWSER_OPEN 1


  Please be aware of the risk of crashing vncserver: if a crash occurs, all applications dis
  ```
- Your browser's preferences for page setup (ie: open page-links in a new window, most recent viewed window, or new tab/window), can affect the display of popup windows in Online Help. It is recommended that you use the browser's default preference settings for page setup.

Launching Online Help

You can launch Online Help using any of the following methods:

- From the menu bar in the GUI, choose Help → Table of Contents, or select a particular topic to open i.e., Command Summary, Getting Started, etc.).
- In the command-line text field of the GUI, use the following syntax to open a topic related to either a specific command (read_verilog) or a message (i.e., ESPUI-001):

  ```
  man <command> | <message id>
  ```

- In ESP shell mode, start Online Help as a stand-alone application by using the man command syntax described in the previous bullet. For example:

  ```
  Verify > man ESPUI-001
  ```

- Right-click on a particular command or message in the GUI console window, then select Help Topic. The Help topic for the command on message will appear.
- Click on the Help button within a dialog box in the GUI to bring up a Help topic that describes the active dialog box.

Searching in Online Help

There are several different methods you can use to search for subject matter in Online Help;

- Use the Index to search for a Topic:
  1. Click the Index tab at the bottom of the navigation pane on the left side of the Help browser.

     ```
     The navigation pane loads the Index.
     ```

  2. Either enter the name of the topic in the text field, or scroll down the navigation bar to the topic of interest.

     ```
     A list of topics is displayed that contains the word or words you typed or sel
     ```

  3. Click the topic you want to display in the main window.
- Use the Search Text Tab:
  1. Click the Search tab at the bottom of the navigation pane.
  2. In the text entry field at the top of the Search pane, type the word that you want to find, then click the Search button.

     ```
     A list of topics is displayed that contains the word or words you typed.
     The topics are displayed based upon the most ranking of keyword matches.
     ```

  3. Click on the name of the topic you want to display in the main window. **Note:** The following restrictions apply when searching for commands or options containing underscore characters:
     - If you are searching for a command or an option that contains an underscore character (such as read_verilog), you must enter a space in place of the underscore in the Search pane text field. You can further narrow the search by using quotations.
     - If you are searching for a command option that is preceded by a dash and contains an underscore character (such as -spice_model_file), you can enter the underscore in the Search pane text field as long as you also enter a dash in front of the option. When using this method, you must enter the complete name of the option.
- Quick Search for Text in a Topic
  1. Enter a word or phrase you want to search for in the Quick Search text field in the Help Toolbar. The occurrences of the word or phrase you entered highlight in yellow in the topic window.
  2. To remove the yellow highlighting, click the "Remove Search Highlighting" button in the Help

Toolbar.

Accessing User Guides and Release Notes

You can access the ESP User Guide and the ESP Release Notes by selecting the corresponding topic in the TOC.

Running Online Help in Windows

You can run a stand-alone version of Online Help in a Windows environment, as described in the following steps:

1. Copy the webhelp.tgz file from the following location in the ESP installation directory to a Windows machine:

   ```
   $SYNOPSYS_install_path/doc/esp/esp_olh/webhelp.tgz
   ```

2. Extract the contents of the webhelp.tgz file. (WinZip works well for this task)
3. To create a shortcut for Online Help, right-click the default.htm file in the esp_olh directory.
4. Drag and drop the shortcut to your desktop or application tool bars.
5. Double-click on the shortcut to launch ESP Online help.

Note: If you are running a 3.x version of a Mozilla-based browser (i.e,. Mozilla, Firefox, SeaMonkey) in Windows, the search function in ESP Online Help will hang. This is due to a security policy when running local content. To work-around this issue, you follow the suggestions described at the following URLs:

- **MadCap Forums: Re: Browser Support**
- **Mozillazine "Links to local pages do not work"**

1. Open Firefox 3.x
2. Type about:config into the address bar and hit enter
3. Click yes at the “Void Warranty” screen
4. Filter for security.fileuri.strict_origin_policy
5. Change it to False by double-clicking the option
6. WebHelp will now work in your local browser

Note: FireFox 3.08 does not seem to exhibit this issue.

Limitations

Note the following limitations when running Online Help:

- If you are running ESP Online Help using Solaris/SparcOS VNC [TightVNC or RealVNC] versions 3.3 or older, this configuration is known to crash vncserver. See the workaround described in "Setting Up Online Help."
- The search function in ESP Online Help is limited when searching for commands containing the underscore character. See the workaround described in "Searching in Online Help."
- Depending on the settings in your browser's page set-up preferences (i.e., open links in a new window, most recent viewed window, or new tab/window, multiple tabs, etc), popup windows may not work in ESP Online Help. To avoid this problem, it is recommended that you use your browser's default page set-up settings.
- When running a 3.x version of a Mozilla-based browser (i.e,. Mozilla, Firefox, SeaMonkey) in Windows, the search function in ESP Online Help will hang. See the workaround described in "Running Online Help in Windows."

Variables Listed by Functional Group

Code and Functional Coverage

coverage

Matching

match_ports_strict

Reading SPICE

bigendian

flattening_device_count

mos_transconductance_ratio

netlist_aggressive_net_compression

netlist_aggressive_port_compression

netlist_bus_extraction_style

netlist_case

netlist_inverter_chain_extraction

netlist_use_verilog_escape

Testbench

testbench_binary_cycles

testbench_constraint_file

testbench_dump_symbolic_waveform

testbench_flush_cycles

testbench_implementation_instance

testbench_initialization_file

testbench_module_name

testbench_output_checks

testbench_reference_instance

testbench_style

testbench_symbolic_cycles

Verification

spice_simulator

threshold_high_impedance_resistance

threshold_warn_big_rc_delay

verify_coverage_max_dropped_symbol_levels

verify_delay_round_multiple

verify_dynamic_reorder

verify_feedback_detection

verify_glitch_removal

verify_hierarchical_compression

verify_max_number_of_oscillations
verify_max_reported_oscillations
verify_randomize_variables
verify_stop_on_randomize
verify_use_specify
Waveform Debug
waveform_format
waveform_viewer

Variables

Description

ESP supports the use of variables in commands and command files. Variable usage is recognized by the leading dollar sign '$' followed by the variable name.

There are three types of variables supported:

- UNIX environment variables
- Application variables
- User-defined variables

Environment Variables

UNIX environment variables are defined in the standard method for the shell in use, typically with the setenv or set/export commands. These must be defined in the shell environment prior to invoking ESP shell.

Application Variables

Application variables are defined by the ESP shell. These are documented in the **variables** section of the on line help. Use the **set_app_var** and **get_app_var** commands to set and get the values of application variable in your scripts.

Using **set_app_var** reduces scripting errors. Variable names are checked by **set_app_var** . Example:

```
esp_shell (setup)> set cov true
true
esp_shell (setup)> set_app_var cov true
Error: Variable 'cov' is not an application variable.  Value will still be set in Tcl.  (CMD-104)
true
```

Using **set** for an application variable does not work inside a Tcl procedure.

kk

User Variables

User-defined variables are defined within the ESP Tcl commands using the standard Tcl **set** command to set variables.

```
  set FOO /global/path/to/file
```

user-defined variables can be used within any Tcl command.

Examples

```
set_app_var coverage true
set_app_var verify_delay_round_multiple 1
set mypath /home/user/alfred/project1
puts $mypath
puts [get_app_var coverage]
read_verilog -r $mypath/core1/ram.v
```

See Also

Variables

bigendian **collection_result_display_limit** **coverage** **flattening_device_count** **match_ports_strict** **mos_transconductance_ratio** **netlist_aggressive_net_compression** **netlist_aggressive_port_compression** **netlist_bus_extraction_style** **netlist_case** **netlist_inverter_chain_extraction** **netlist_process_estimation_version** **netlist_unwrap_transistors** **netlist_use_verilog_escape** **sh_allow_tcl_with_set_app_var** **sh_allow_tcl_with_set_app_var_no_message_list** **spice_simulator** **testbench_binary_cycles** **testbench_constraint_file** **testbench_declaration_file** **testbench_design_instance** **testbench_dump_symbolic_waveform** **testbench_flush_cycles** **testbench_implementation_instance** **testbench_initialization_file** **testbench_module_name** **testbench_output_checks** **testbench_reference_instance** **testbench_setup_file** **testbench_style** **testbench_symbolic_cycles** **threshold_high_impedance_resistance** **threshold_warn_big_rc_delay** **verify_auto_effort_selection** **verify_coverage_max_dropped_symbol_levels** **verify_delay_round_multiple** **verify_dynamic_reorder** **verify_feedback_detection** **verify_glitch_removal** **verify_hierarchical_compression** **verify_imitate_simulator** **verify_max_number_error_vectors** **verify_max_number_of_oscillations** **verify_max_number_oscillation_vectors** **verify_max_reported_oscillations** **verify_negative_timing_checks** **verify_partial_transition_sensitivity** **verify_partial_transition_threshold** **verify_randomize_variables** **verify_spice_simulation_mode** **verify_stop_on_nonzero_delay_oscillation** **verify_stop_on_randomize** **verify_stop_on_zero_delay_oscillation** **verify_suppress_nonzero_delay_oscillation** **verify_use_specify** **waveform_dump_control** **waveform_format** **waveform_viewer**

Value Change Dump (VCD)

Description

Waveform file format used by Verilog simulators. This is an ASCII file format defined by IEEE Standard 1364.

ESP will only dump binary values into a VCD file. ESP symbolic equations can not be dumped into a VCD file.

SEE ALSO

`Commands:` **debug_design** , **start_waveform_viewer**

`Variables:` **waveform_format** , **waveform_viewer**

`Topics:` **dve** , **esp_shell_flow** , **fsdb** ,

VCS Options

Compile and Runtime Options

The following table shows which VCS options are supported by -vcs switch to **read_verilog** and **read_spice_behavioral_model** commands. This list is based upon the VCS tool version F-2011.12.

In general, the ESP tool does not support VPD, NTB, Vera, Coverage, AMS or C compiler switches for the VCS tool. The ESP tool does not distinguish between runtime and compile time.

In the following tables **NO**means the switch is not supported. If the switch is given, the ESP tool may generate an error message. **IGNORED**means that the switch is ignored and will have no effect. **YES**means that the switch works in the ESP tool the same as in the VCS tool. Any specific behavior differences are noted in the table.

Compile-Time Options

Supported	switch	Description	
NO	-ams	Enables the use of Verilog-AMS code in VCS 2-step mode.	
NO	-ams_discipline <discipline_name>	Specifies the default discrete discipline in VerilogAMS in VCS 2-step mode.	
NO	-ams_iereport	Provides the auto-inserted connect modules (AICMs) information in VCS 2-step mode.	
NO	-as <assembler>	Specifies an alternative assembler. Only applicable in incremnetal compile mode, which is the default. Not supported on IBM RS/6000 AIX.	
NO	-ASFLAGS <options>	Passes options to assembler. Not supported on IBM RS/6000 AIX.	
NO	-assert <keyword_argument>	The keyword arguments and what they do are as follows:	
		disable_cover	Disables coverage for SVA cover statements.
		dumpoff	Disables the dumping of SVA information in the VPD file.
		dve	Enbables SystemVerilog assertions tracing in the VPD file that you load into DVE. This tracing enables you to see asertion attempts.
		enable_diag	Enables further control of SystemVerilog assertions result reporting with runtime options.
			Ignores SystemVerilog assertion

filter_past	subsequences containing past operators that have not yet eclipsed the history threshold.
vpiSeqBeginTime	Enables you to see the simulation time that a SystemVerilog assertion sequence starts when using Debussy.
vpiSeqFail	Enables you to see the simulation time that a SystemVerilog assertion sequence doesn't match when using Debussy.

NO	-C	Stops after generating the intermediate C or assembly code.
NO	-cc <compiler>	Specifies and alternative C compiler.
NO	-CC <options>	Works the same as -CFLAGS.
NO	-CFLAGS <options>	Pass options to C compiler. Multiple -CFLAGS are allowed. Allows passing of C compiler optimization levels.
NO	-cm line|cond|fsm|tgl|branch|assert	Specifies compiling for the specified type or types of coverage. The arguments specifies the types of coverage: If you want VCS to compile for more than one type of coverage, use the plus (+) character as a delimiter between arguments, for example: `-cm line+cond+fsm+tgl`

line	Compile for line or statement coverage
cond	Compile for condition coverage
fsm	Compile for FSM coverage
tgl	Compile for toggle coverage
branch	Compine for branch coverage
assert	Compile for SystemVerilog assertion coverage

NO	-cm_assert_hier <filename>	Limits SystemVerilog assertions coverage to the module instances listed in the specified file.
NO	-cm_cond <arguments>	Modifies condition coverage as specified by the argument or arguments:

basic	Only logical conditions and no multiple conditions
std	The default: only logical, multiple, sensitized conditions
full	Logical and non-logical, multiple conditions, no sensitized conditions
allops	Logical and non-logical conditions
event	Signals in event controls in the sensitivity list position are conditions
anywidth	Enables conditions that need more than 32 bits
for	Enables conditions if for loops
tf	Enables conditions in user-defined tasks and functions
sop	Condition SOP coverage instead of sensitized conditions also tells VCS that when it reads conditional expressions that contain the ^ bitwise XOR and ~^ bitwise XNOR operators, it reduces

		the expression to negation and logical AND or OR You can specify more than one argument. If you do use the + plus delimiter between arguments, for example: `-cm_cond basic+allops`
NO	-cm_constfile <filename>	Specifies a file listing signals and 0 or 1 values. VCS compiles for line and condition coverage as if these signals were permanently at the specified values and you included the -cm_noconst option.
NO	-cm_count	Enables cmView to do the following: • In toggle coverage, not just whether a signal toggled from 0 to 1 and 1 to 0, but also the number of times it so toggled. • In FSM coverage, not just whether an FSM reached a state, had such a transition, but also the number of times it did. • In condition coverage, not just whether a condition was met or not, but also the number of times the condition was met. • In Line Coverage, not just whether a line was executed, but how many times.
NO	-cm_dir <directory_path_name>	Specifies and alternative name and location for the coverage database directory.
NO	-cm_fsmcfg <filename>	Specifies an FSM coverage configuration file.
NO	-cm_fsmopt <keyword_argument>	The keyword arguments are as follows:

allowTemp	Allows FSM extraction when there is indirect assignment to the variable that holds the current state.
optimist	Specifies identifying illegal transitions when VCS extracts FSMs in FSM coverage. cmView then reports illegal transitions in report files.
report2StateFsms	By default VCS does not extract two state FSMs. This keyword tells VCS to extract them.
reportvalues	Specifies reporting the value transitions of the reg that holds the current state of a One Hot or Hot Bit FSM where there are parameters for the bit numbers of the signals that hold the current and next state.
reportWait	Enables VCS to monitor transitions when the signal holding the current state is assigned the same state value.
reportXassign	Enables the extraction of FSMs in which a state contains the X (unknown) value.

NO	-cm_fsmresetfiltser <filename>	Filters out transitions in assignment statements controlled by if statements where the conditional expression (following the keyword if) is a signal you specify in the file.

NO	-cm_hier <filename>	When compiling for line, condition, FSM or toggle coverage, specifies a configuration file that specifies module definitions, source files, or module instances and their subhierarchies, that you want VCS to exclude from coverage or be the only parts of the design compiled for coverage.
NO	-cm_ignorepragmas	Tells VCS to ignore pragmas for coverage metrics.
NO	-cm_libs yv|celldefine	Specifies compiling for coverage source files in Verilog libraries when you include the yv argument. Specifies compiling for coverage module definitions that are under the `` `celldefine ``compiler directive when you include the celldefine argument. You can specify both arguments using the plus (+) delimiter.
NO	-cm_line contassign	Enables line coverage for continuous assignments.
NO	-cm_name <filename>	As a compile-time or runtime option, specifies the name of the intermediate data files.
NO	-cm_noconst	Tells VCS not to monitor for conditions that can never be met or lines that can never execute because a signal is permanently at a 1 or 0 value.
NO	-cm_resetfilter	You can filter out of FSM coverage transitions in assignments controlled controlled by if statements where the conditional expression (following the if keyword) is a signal you specify in the file. This filtering out can be on the specified signal in any module or the module you specify in the file. You can also specify the FSM and whether the signal is true or false.
NO	-cm_tglfile <filename>	Specifies displaying at runtime a total toggle count for one or more subhierarchies specified by the top-level module instance entered in the file.
NO	-cm_tgl mda	Enables toggle coverage for Verilog 2001 and SystemVerilog unpacked multidimensional arrays.
NO	-cpp	Specifies a C++ compiler.
NO	-comp64	Compiles the design in 64 bit mode and creates a 32 bit executable for simulating in 32 bit mode.
NO	-debug	Enables the use of UCLI commands and DVE.
NO	-debug_all	Enables the use of UCLI and DVE. Also enables line stepping.
IGNORED	-doc	Starts browser to display the HTML files for the VCS/VCSi documentation.
NO	-dve_opt <dve_option>	You can use the argument called -dve_opt to pass DVE arguments from simv to DVE. Each DVE argument must be preceded by -dve_opt argument. In cases where the argument requires an additional option, the = sign needs to be used.(E.g. -dve_opt -session=file.tcl)
NO	-e <new_name_for_main>	Specifies the name of your main() routine in your PLI application.
YES	-f <filename>	Specifies a file that contains a list of pathnames to source files and compile-time options.
		Same as the -f option but allows you to specify a path to

YES	-F <filename>	the file and the source files listed in the file do not have to be absolute pathnames.

NO	-file filename	This option is for problems you might encounter with entries in files specified with the -f or -F options. This file can contain more compile-time options and different kinds of files. It can contain options for controlling compilation and PLI options and object files. You can also use escape characters and meta-characters in this file, like $, \`, and ! and they will expand, for example:
		`-CFLAGS '-I$VCS_HOME/include'`
		`/my/pli/code/$PROJECT/treewalker.o`
		`-P /my/pli/code/$PROJECT/treewalker.tab`
		You can comment out entries in this file with the Verilog `//`and `/  /` comment characters.

NO	-full64	Compiles the design in 64 bit mode and creates a 64 bit executable for simulating in 64 bit mode.
NO	-gen_asm	Specifies generating intermediate assembly code. Not supported on IBM RS/6000 AIX.
NO	-gen_c	Specifies generating intermediate C code. This is the default in IBM RS/6000 AIX.
NO	-gen_obj	Generate object code; default on Linux, Solaris, and HP platforms. Not supported on IBM RS/6000 AIX.
NO	-h or -help	Lists descriptions of the most commonly used compile-time and runtime options.
NO	-ID	Displays the hostid or dongle ID for your machine.
NO	-ignore <keyword_argument>	The keyword arguments are as follows: unique_checks Suppresses warning messages about SystemVerilog unique if and unique case statements. priority_checks Suppresses warning messages about SystemVerilog priority if and priority case statements. all Suppresses warning messages about SystemVerilog unique if, unique case, priority if and priority case statements.
NO	-j<number_of_processes>	Specifies the number of processes to use for parallel compilation. There is no space between the j character and the number.
NO	-l <filename>	(lower case L) Specifies a log file where VCS records compilation messages and runtime messages if you include the -R, -RI, or -RIG options.
NO	-ld <linker>	Specifies an alternative linker.
NO	-LDFLAGS <options>	Pass options to the linker. Only applicable in incremental compile mode.
NO	-line	Enables stepping through the code and source line breakpoints in DVE.
NO	-lmc-swift	Enables the LMC SWIFT interface.

NO	-lmc-swift-template <swift_model_name>	Generates a Verilog template for a SWIFT Model.
NO	-l<name>	Links the <name> library to the resulting executable.
NO	-load <shared_VPI_library>:<registration_routine>	Specifies the registration routine in a shared library for a VPI application.
NO	-Marchive=<number_of_module_definitions>	Tells the linker to create temporary object files that contain the specified number of module definitions. Use this option if there is a command line buffer overflow caused by too many object files on the linker command line.
NO	-Mdelete	Use this option for the rare occurrence when the chmod -x simv command in the make file can't change the permissions on an old simv executable. This option replaces this command with the `rm -f simv`command in the make file.
NO	-Mlib=<directory>	Specifies the directory where VCS looks for descriptor information to see if a module needs to be recompiled. Also specifies a central place for object files. You use this option for shared incremental compilation.
NO	-Mmakeprogram=<program>	Program used to make object (default is make).
NO	-Mupdate[=0]	By default VCS overwrites the Makefile between compilations. If you wish to preserve the Makefile between compilations, enter this option with the 0 argument. Entering this argument without the 0 argument, specifies the the default condition, incremental compilation and updating the Makefile.
NO	-negdelay	Enables the use of negative values in IOPATH and INTERCONNECT entries in SDF files.
NO	-noIncrComp	Disables incremental compilation.
NO	-notice	Enables verbose diagnostic messages.
NO	-ntb	Enables the use of the OpenVera Testbench language constructs described in the OpenVera Language Reference Manual: Native TestBench.
NO	-ntb_cmp	Compiles and generates the testbench shell (file.vshell) and shared object files. Use this option when compiling the .vr file separately from the design files.
NO	-ntb_define <macro>	Specifies any OpenVera macro name on the command line. You can specify multiple macro names using the + delimiter.
NO	-ntb_filext <.ext>	Specifies an OpenVera file extension. You can specify multiple filename extensions using the + delimiter.
NO	-ntb_incdir <directory_path>	Specifies the include directory path for OpenVera files. You can specify multiple include directories using the + delimiter.
NO	-ntb_noshell	Tells VCS not to generate the shell file.Use this option when you recompile a testbench.
		The keyword arguments are as follows: Reports error, during compilation or

NO	-ntb_opts <keyword_argument>	check	simulation, when there is an out-of-bound or illegal array access.
		dep_check	Enables dependency analysis and incremental compilation. Detects files with circular dependencies and issues an error message when VCS cannot determine which file to compile first.
		no_file_by_file_pp	By default, VCS does file by file preprocessing on each input file, feeding the concatenated result to the parser. This argument disables this behavior.
		print_deps[=<filename>]	Enter this argument with the dep_check argument. This argument tells VCS to display the dependencies for the source files on the screen or in the file that you specify.
		tb_timescale=<value>	Specifies an overriding timescale for the testbench. The timescale is in the Verilog format (for example, 10ns/10ns).
		use_sigprop	Enables the signal property access functions. (for example, vera_get_ifc_name()).
		vera_portname	Specifies the following: • The Vera shell module name is named vera_shell. • The interface ports are named ifc_signal. • Bind signals are named, for example, as: \if_signal[3:0].
		You can enter more than one keyword argument, using the + delimiter, for example: `-ntb_opts use_sigprop+vera_portname`	
NO	-ntb_shell_only	Generates only a .vshell file. Use this option when compiling a testbench separately from the design file.	
NO	-ntb_sfname <filename>	Specifies the filename of the testbench shell.	
NO	-ntb_sname <module_name>	Specifies the name and directory where VCS writes the testbench shell module.	
NO	-ntb_spath	Specifies the directory where VCS writes the testbench shell and shared object files. The default is the compilation directory.	
NO	-ntb_vipext <.ext>	Specifies an OpenVera encrypted-mode file extension to mark files for processing in OpenVera encrypted IP mode. Unlike the -ntb_filext option, the default encrypted-mode extensions .vrp, .vrhp are not overridden, and will always be in effect. You can pass multiple file extensions at the same time using the + delimiter.	
NO	-ntb_vl	Specifies the compilation of all Verilog files, including the design, the testbench shell file and the top-level Verilog	

		module.
NO	-o <name>	Specifies the name of the executable file that is the product of compilation. The default name is simv.
NO	-ovac	Starts the OVA compiler to check the syntax of OVA files on the vcs command line.
NO	-ova_cov	Enables functional coverage.
NO	-ova_cov_events	Enables functional coverage reporting of expressions.
NO	-ova_cov_hier <filename>	Limits functional coverage to the module instances listed in the specified file.
NO	-ova_debug or -ova_debug_vpd	Enables OVA attempt dumping into VPD.
NO	-ova_file <filename>	Specifies an OpenVera Assertions file. Not required if the filename has an .ova extension.
NO	-ova_filter_past	For assertions that are defined with the past operator, ignore these assertions where the past history buffer is empty. For instance, at the very beginning of the simulation the past history buffer is empty. So, a check/forbid at the first sampling point and subsequent sampling points should be ignored until the past buffer has been filled with respect to the sampling point.
NO	-ova_filter_last	Ignores assertion subsequences containing past operators that have not yet eclipsed the history threshold.
NO	-ova_enable_diag	Enables runtime options for controlling functional coverage reports.
NO	-ova_inline	Enables compiling OVA code that is written in Verilog source files.
NO	-ova_lint	Enables general rules for the OVA linter
NO	-ova_lint_magellan	Enables Magellan rules for the OVA linter.
NO	-override-cflags	Tells VCS not to pass its default options to the C compiler.
NO	-override_timescale=<time_unit>/<time_precision>	Overrides the time unit and precision unit of all the `timescale compiler directives in the source code and, like the -timescale option, privides a timescale for all module definitions that preceed the first `timescale compiler directive.
NO	-P <pli.tab>	Specifies a PLI table file.
YES (VCS paraser only)	-parameters <filename>	Changes parameters values to values specified in the file. The syntax for a line in the file is as follows: assign <value> <path_to_parameter> The path to the parameter is similar to a Verilog hierarchical name except that you use slash characters (/) instead of periods as delimiters.
NO	-platform	Returns the name of the platform directory in your VCS installation directory.
YES (Top level only. VCS parser only)	-pvalue+<parameter_hierarchical_name>=<value>	Changes the specified parameter to the specified value.

YES	-q	Suppresses VCS compiler messages.
NO	-R	Run the executable file immediately after VCS links together the executable file. You can add any runtime option to the vcs command line.
NO	-s	Stop simulation just as it begins. Use this option with the -R.
NO	-sim_res=<time_precision>	Defines simulation resolution. It also defines timescales for modules which don't have timescales after analysis.
NO	-sv_pragma	Tells VCS to compile the SystemVerilog assertions code that follows the sv_pragma keyword in a single line or multi-line comment.
NO	-sysc	Tells VCS to look in the ./csrc directory for the subdirectories containg the wrapper and interface files neded by the VCS/SystemC cosimulation interface to connect the Verilog and SystemC parts of a mixed Verilog and SystemC design.
NO	-syslib <libs>	Specifies system libraries to be linked with the runtime executable.
YES	-timescale=<time_unit>/<time_precision>	If only some source files contain the `timescale compiler directive and the ones that don't appear first on the vcs command line, use this option to specify the time scale for these source files.
YES	-u	Changes all characters in identifiers to uppercase.
NO	-ucli	Specifies UCLI mode at runtime.
NO	-V	Enables the verbose mode.
YES	-v <filename>	Specifies a Verilog library file to search for module definitions.
NO	-vera	Specifies the standard VERA PLI table file and object library.
NO	-vera_dbind	Specifies the VERA PLI table file and object library for dynamic binding.
NO	-Vt	Enables warning messages and displays the time used by each command.
YES	-y <directory_pathname>	Specifies a Verilog library directory to search for module definitions.
NO	+acc+1|2|3|4	Old style method to enable PLI ACC capabilities for the entire design. 1 enables all capabilities but breakpoints and delay annotation. 2 enables what 1 enables plus breakpoints on value changes of nets and registers. 3 enables what 2 enables plus module path delay annotation. 4 enables what 3 enables plus gate delay annotation.
NO	+ad=<partition_filename>	Specifies the partition files used in mixed signal simulation.
IGNORED	+allmtm	Allows you to specify at runtime which values in min:typ:max delay value triplets in compiled SDF files using the +mindelays, +maxdelays, or +typdelays runtime options.

NO	+applylearn[+<filename>]	Compiles your design to enable only the ACC capabilities that you needed for the debugging operations you did during a previous simulation of the design. The +vcs+learn+pli runtime option records where you used ACC capabilities in a file named pli_learn.tab. If you do not change the file's name or location, you can omit +<filename> from this option.
NO	+autoprotect[<file_suffix>]	Creates a protected source file; all modules are encrypted.
NO	+auto2protect[<file_suffix>]	Create a protected source file that does not encrypt the port connection list in the module header; all modules are encrypted.
NO	+auto3protect[<file_suffix>]	Creates a protected source file that does not encrypt the port connection list in the module header or any parameter declarations that precede the first port declaration; all modules are encrypted.
NO	+bidir+1	Tells VCS to finish compilation when it finds a bidirectional registered mixed-signal net.
NO	+charge_decay	Enables charge decay in trireg nets. Charge decay will not work if you connect the trireg to a transistor (bidirectional pass) switch such as tran, rtran, tranif1, or rtranif0.
NO	+csdf+precompile	Precompiles your SDF file into a format that is for VCS to parse when it is compiling your Verilog code.
NO	+csdf+precomp+dir+<directory>	Specifies the directory path where you want VCS to write the precompiled SDF file.
NO	+csdf+precomp+ext+<ext>	Specifies an alternative to the "_c" character string addition to the filename extension of the precompiled SDF file.
YES	+define+<macro_name>=<value>	Defines a text macro. Test for this definition in your Verilog source code using the `ifdef compiler directive.
YES	+delay_mode_distributed	Specifies ignoring the module path delays and use only the delay specifications on all gates, switches, and continuous assignments.
YES	+delay_mode_path	For modules with specify blocks, specifies ignoring the delay specifications on all gates and switches and use only the module path delays and the delay specifications on continuous assignments.
YES	+delay_mode_unit	Specifies ignoring the module path delays and change all the delay specifications on all gates, switches, and continuous assignments to the shortest time precision argument of all the `timescale compiler directives in the source code.
YES	+delay_mode_zero	Change all the delay specifications on all gates, switches, and continuous assignments to zero and change all module path delays to zero.
NO	+deleteprotected	Allows overwriting of existing files when doing source protection.
NO	+error+<n>	Enables you to increase the maximum number of NTB errors at compile-time to <n>.
		Specifies the directories that contain the files you specified

YES	+incdir+<directory>	with the `include compiler directive. You can specify more that one directory, separating each path name with the + character.
YES	+libext+<extension>	Specifies that VCS only search the source files in a Verilog library directory with the specified extension. You can specify more than one extension, separating each extension with the + character. For example, +libext++.v specifies searches library files with no extension and library files with the .v extension. Enter this option when you enter the -y option.
NO	+liborder	Specifies searching for module definitions in the libraries that follow, on the vcs command line, a library that contains an unresolved instance before searching the libraries that precede the library with the unresolved instance.
NO	+librescan	Specifies always starting the search for unresolved module definitions with the first library specified on the vcs command line.
NO	+libverbose	Tells VCS to display a message when it finds a module definition in a source file in a Verilog library directory that resolves a module instantiation statement that VCS read in your source files, a library file, or in another file in a library directory.
NO	+lint=[no]ID|none|all,...	Enables or disables Lint messages about your Verilog code.
YES	+maxdelays	Use maximum value when min:typ:max values are encountered in delay specifications SDF files.
NO	+memcbk	Enables callbacks for memories and multidimensional arrays (MDAs). Use this option if your design has memories or MDAs and you are doing any of the following: • Writing a VCD or VPD file during simulation. For VCD files, at runtime, you must also enter the +vcs+dumparrays runtime option. For VPD files you must enter the $vcdplusmemon system task. VCD and VPD files are used for post-processing with DVE or debugging using SmartDebug. • Using the VCS/SystemC Interface • Interactive debugging with DVE • Writing an FSDB file for Verdi • Using any debugging interface application - VCSD/PLI (acc/pli) that needs to use value change callbacks on memories or MDAs. APIs like acc_add_callback, vcsd_add_callback, and vpi_register_cb need this option if these APIs are used on memories or MDAs.
NO	+memopt	Applies optimizations to reduce memory.
YES	+mindelays	Use minimum value when min:typ:max values are encountered in delay specifications and SDF files.
NO	+multisource_int_delays	Enables multisource interconnect delays.

NO	+nbaopt	Removes the intra-assignment delays from all the nonblocking assignment statements in your design.
NO	+neg_tchk	Enables negative values in timing checks.
NO	+nocelldefinepli+0|1|2	For specifying what VCS records in the VPD file about nets and registers defined under the `celldefine compiler directive.  0: enables recording the transition times and values of nets and registers in all modules defined under the `celldefine compiler directive or defined in a library that you specify with the -v or -y compile-time options. 1: disables recording the transition times and values of nets and registers in all modules defined under the `celldefine compiler directive. 2: disables recording the transition times and values of nets and registers in all modules defined under the `celldefine compiler directive or defined in a library that you specify with the -v or -y compile-time options whether the modules in these libraries are defined under the `celldefine compiler directive or not.
NO	+noerrorIOPCWM	Changes the error condition, when a signal is wider or narrower than the inout port to which it is connected, to a warning condition, thus allowing VCS to create the simv executable after displaying the warning message.
NO	+nolibcell	Specifies not defining modules in libraries as cells unless they are under the `celldefine compiler directive.
YES	+nospecify	Suppresses module path delays and timing checks in specify blocks.
NO	+notimingcheck	Suppresses timing checks in specify blocks.
NO	+nowarnTFMPC	Suppress the "Too few module port connections" warning messages during Verilog Compilation.
NO	+no_notifier	Disables the toggling of the notifier register that you specify in some timing check system tasks.
NO	+no_tchk_msg	Disables the display of timing check warning messages but does not disable the toggling of notifier registers in timing checks. This is also a runtime option.
NO	+optconfigfile+<filename>	Specifies the VCS configuration file.
NO	+overlap	Enables accurate simulation of multiple non-overlapping violation windows for the same signals specified with negative delay values in timing checks. See the section on "Using Multiple Non-Overlapping Windows" in the VCS/VCSi User Guide.
NO	+pathpulse	Enables the search for the PATHPULSE$ specparam in specify blocks.
NO	+pli_unprotected	Enables PLI and CLI access to the modules in the protected source file being created (PLI and CLI access is normally disabled for protected modules).

NO	+plusarg_save	Enter this option in the file that you specify with the -f option so that VCS passes to the simv executable the options beginning with a plus + character that follow in the file.
NO	+plusarg_ignore	Also enter this option in the file that you specify with the -f option so that VCS does not pass to the simv executable the options that follow in the file. Use this option with the +plusarg_save option to specify that other options should not be passed.
NO	+print+bidir+warn	Tells VCS to display a list of bidirectional registered mixed-signal nets.
NO	+protect[<file_suffix>]	Creates a protected source file; only encrypting \`protect/\`endprotect regions.
NO	+pulse_e/<number>	Specifies flagging as error and drive X for any path pulse whose width is less than or equal to the percentage of the module path delay specified by the number argument.
NO	+pulse_int_e/<number>	Same as the +pulse_e option but only applies to interconnect delays.
NO	+pulse_int_r/<number>	Same as the +pulse_r option but only applies to interconnect delays.
NO	+pulse_on_event	Specifies that when VCS encounters a pulse shorter than the module path delay, VCS waits until the module path delay elapses and then drives an X value on the module output port and displays an error message.
NO	+pulse_on_detect	Specifies that when VCS encounters a pulse shorter than the module path delay, VCS immediately drives an X value on the module output port, and displays an error message. It does not wait until the module path delay elapses.
NO	+pulse_r/<number>	Reject any pulse whose width is less than number percent of module path delay.
NO	+putprotect+<target_dir>	Specifies the target directory for protected files.
NO	+race	Specifies that VCS generate a report, during simulation, of all the race conditions in the design and write this report in the race.out file.
NO	+race=all	Analyzes the source code during compilation to look for coding styles that cause race conditions.
NO	+racecd	Specifies that VCS generate a report, during simulation, of the race conditions in the design between the \`race and \`endrace compiler directives and write this report in the race.out file.
NO	+race_maxvecsize=<size>	Specifies the largest vector signal for which the dynamic race detection tool looks for race conditions.
NO	+rad	Performs Radiant Technology optimizations on your design.
NO	+sdfprotect[<file_suffix>]	Creates a protected SDF file.
NO	+sdf_nocheck_celltype	Tells VCs not to check to make sure that the CELLTYPE entry in the SDF file does not match the module identifier for a module instance before back annotating delay values from the SDF file to the module instance.
NO	+sdfverbose	Enables the display of more than ten warning and more than ten error messages about SDF back annotation.

NO	+spl_read	Tells VCS to treat output ports as inout ports in order to facilitate more accurate multi-driver contention analysis across module boundaries. This option can have an adverse impact on runtime performance.
NO	+systemverilogext+<ext>	Specifies a filename extension for source files containing SystemVerilog source code.
NO	+tetramax	Enables simulation of TetraMAX's testbench in zero delay mode.
NO	+timopt+<clock_period>	Enables Timing Check Optimizations the +<clock_period> argument specifies the clock period of the fastest clock in the design. Starts the Timopt timing optimizer. the +<clock_period> argument specifies the clock period of the fastest clock in the design. Timopt applies timing optimizations to your design. Timopt also writes a timopt.cfg file in the current directory. This file contains clock signals and module definitions of sequential devices it's not sure of. You edit this file and recompile without the +<clock_period> argument to obtain more Timopt optimizations.
NO	+transport_int_delays	Enables transport delays with full pulse control for single source nets.
NO	+transport_path_delays	Turns on the transport behavior for I/O paths.
NO	+typdelays	Use typical value when min:typ:max values are encountered in delay specifications and SDF files.
YES (on by default)	+v2k	Enables the use of new Verilog constructs in the 1364-2001 standard.
NO	+vc[+abstract][+allhdrs][+list]	Enables the direct call of C/C++ functions in your Verilog code using the DirectC interface. The optional suffixes specify the following: +abstract Specifies that you are using abstract access trough vc_handles to the data structures for the Verilog arguments. +allhdrs Writes the vc_hdrs.h file that contains external function declarations that you can use in your Verilog code. +list Displays on the screen all the functions that you called in your Verilog source code.
NO	+vcs+dumpvars	A substitute for entering $dumpvars, without arguments, in your Verilog code.
NO	+vcs+flush+log	Increases the frequency of flushing both the compilation and simulation log file buffers.
NO	+vcs+flush+all	Shortcut option for entering all three of the +vcs+flush+log, +vcs+flush+dump, and +vcs+flush+fopen options.
NO	+vcs+initmem+0|1|x|z	Initializes all bits of all memories in the design.
NO	+vcs+initreg+0|1|x|z	Initializes all bits of all regs in the design.

NO	+vcs+lic+vcsi	Checks out three VCSi licenses to run VCS.
NO	+vcsi+lic+vcs	Checks out a VCS license to run VCSi when all VCSi licenses are in use.
NO	+vcs+lic+wait	Tells VCS to wait for a network license if none is available.
NO	+vcsi+lic+wait	Tells VCSi to wait for a network license if none is available.
NO	+vcs+vcdpluson	A compile-time substitute for $vcdpluson option. The +vcs+vcdpluson switch enables dumping for the entire design. You would however need to use a debug switch (example -debug_pp) to dump the data.
NO	+verilog1995ext+<ext>	Specifies a filename extension for source files containing Verilog 1995 source code.
NO	+verilog2001ext+<ext>	Specifies a filename extension for source files containing Verilog 2001 source code.
NO	+vhdllib+<logical_libname>	This option specifies the VHDL logical library to use for VHDL design entity instances that you instantiate in your Verilog design.
NO	+vpi	Enables the use of VPI PLI access routines.
NO	+warn=[no]ID|none|all,...	Enables or disables warning messages.

Runtime Options

Supported	**switch**	**Description**
NO	-a <filename>	Specifies appending all messages from simulation to the bottom of the text in the specified file as well as displaying these messages to the standard output.
		The keyword arguments and what they do are as follows:

dumpoff	Disables the dumping of SVA information in the VPD file during simulation.
filter	Blocks reporting of trivial SystemVerilog assertion implication successes. These happen when an implication construct registers a success only because the precondition (antecedent) portion is false (and so the consequence portion is not checked). With this option, reporting only shows successes in which the whole expression matched.
finish_maxfail=<N>	Terminates the simulation if the number of SystemVerilog assertion failures for any assertion reaches N. N must be supplied, otherwise no limit is set.
global_finish_maxfail=<N>	Stops the simulation when the total number of failures, from all SystemVerilog Assertions,

			reaches N.
		maxcover=<N>	Disables the collection of coverage information for cover statements after the cover statements are covered N number of times. <N> must be a positive integer, it can't be 0.
		maxfail=<N>	Limits the number of SystemVerilog assertion failures for each assertion to N. When the limit is reached, the assertion is disabled. N must be supplied, otherwise no limit is set.
NO	-assert <keyword_argument>	maxsuccess=<N>	Limits the total number of reported SystemVerilog assertion successes to N. N must be supplied, otherwise no limit is set. The monitoring of assertions continues, even after the limit is reached.
		nocovdb	Tells VCS not to write the <program_name>.db file for assertion coverage.
		nopostproc	Disables the display of thw SVA coverage summary at the end of simulation.
		quiet0|1	quiet0: Disables messages, in standard output, about assertion failures. quiet1: Disables messages, in standard output, about assertion failures, but displays the summary of them at the end of simulation. The never triggered assertions are also reported.
		report[=<filename>]	Generates a SystemVerilog assertion report file in addition to displaying results on your screen. By default the file's name and location is ./simv.vdb/report/ova.report, but you can change this by entering the filename pathname argument.
		success	Enables reporting of successful SystemVerilog assertion matches in addition to failures. The default is to report only

		<table><tr><td></td><td>failures.</td></tr><tr><td>verbose</td><td>Adds more information to the report specified by the report=<filename> keyword, including assertions that never triggered and attemts that did not finish, and a summary with the number of assertions present, attempted, and failed.</td></tr></table> You can enter more than one keyword, using the plus + separator, for example: `-assert maxfail=10+maxsuccess=20+success+filter`
NO	-cm line|cond|fsm|tgl|branch|assert	Specifies monitoring for the specified type or types of coverage. The arguments specifies the types of coverage: line Monitor for line or statement coverage. cond Monitor for condition coverage. fsm Monitor for FSM coverage. tgl Monitor for toggle coverage. branch Monitor for branch coverage. assert Monitor for SystemVerilog assertions coverage. If you want VCS to monitor for more than one type of coverage, use the plus + character as a delimiter between arguments, for example: `-cm line+cond+fsm+tgl`
NO	-cm_dir <directory_path_name>	Specifies and alternative name and location for the coverage database directory.
NO	-cm_glitch <period>	Specifies a glitch period during which VCS does not monitor for coverage caused by value changes. The period is an interval of simulation time specified with a non-negative integer.
NO	-cm_log <filename>	Specifies a log file for monitoring for coverage during simulation.
NO	-cm_name <filename>	As a compile-time or runtime option, specifies the name of the intermediate data files. On the cmView command line, specifies the name of the report files.
NO	-cm_tglfile <filename>	Specifies displaying at runtime a total toggle count for one or more subhierarchies specified by the top-level module instance entered in the file.
NO	-E <program>	Starts the program that displays the compile-time options that were on the vcs command line when you created the simv (or simv.exe or some other name specified with the -o option) executable file.
NO	-grw <filename>	Sets the name of the $gr_waves output file to the specified file. The default filename is grw.dump.
NO	-gui	Starts the Discovery Visual Environment (DVE) graphical user interface.
NO	-i <filename>	Specifies a file containing CLI commands that VCS executes when simulation starts.
		Specifies an alternative name or location for the

NO	-k <filename> | off	vcs.key file into which VCS writes the CLI interactive commands that you enter during simulation. The off argument tells VCS not to write this file.
NO	-l <filename>	Specifies writing all messages from simulation to the specified file as well as displaying these messages in the standard output. This option begins with the letter "l" (lowercase "L") for log file.
NO	-ova_filter	Blocks reporting of trivial if-then successes. These happen when an if-then construct registers a success only because the if portion is false (and so the then portion is not checked). With this option, reporting only shows successes in which the whole expression matched. This option is enabled by the -ova_enable_diag compile-time option.
NO	-ova_max_fail <N>	Limits the number of reported failures for each assertion to N. The monitoring of assertions continues, even after this limit is reached. This option is enabled by the -ova_ebable_diag compile-time option.
NO	-ova_max_success <N>	Limits the number of successes for each assertion to N. The monitoring of assertions continues, even after the limit is reached. This option is enabled by the -ova_enable_diag compile-time option.
NO	-ova_name <name | /<pathname>/<name>	Specifies an alternative name or location and name for the ./simv.vdb/ scov/results.db and ./simv.vdb/reports/ova.report files. You use this option if you want data and reports from a series of simulation runs. It's a way of keeping VCS from overwriting these files from a prvious simulation. If you just specify a name the alternatively named files will be in the default directories. If you specify a pathname, with an argument containing the slash charcter /, you specify a different location and name for these files, for example: `-ova_name /net/design1/ova/run2` This example tells VCS to write run2.db and run2.report in the /net/design1/ova directory.
NO	-ova_simend_max_fail <N>	Terminates the simulation if the number of failures for any assertion is reached. This option is enabled by the -ova_enable_diag compile-time option.
NO	-ova_success	Enables the reporting of successful matches. This option is enabled by the -ova_enable_diag compile-time option.
NO	-ova_quiet [1]	Disables displaying functional coverage results on the screen. The optional 1 argument specifies displaying a summary of these results.
		Adds more information to the end of the report including assertions that never triggered and attempts

NO	-ova_verbose	that did not finish, and a summary with the number of assertions present, attempted, and failed.
YES	-q	Quiet mode. Suppress printing of VCS header and summary information, the proprietary message at the beginning of simulation, and the VCS Simulation Report at the end of simulation (time, CPU time, data structure size, and date)
YES (limited language support)	-sverilog	Enables the use of the Verilog language extensions in the Accellera SystemVerilog specification.
NO	-ucli	Enables the use of UCLI commands.
NO	-V	Verbose mode. Print VCS version and extended summary information. Prints VCS compile and run-time version numbers, and copyright information, at start of simulation.
NO	-vcd <filename>	Sets the output VCD file name to the specified file. The default filename is verilog.dump. A $dumpfile system task in the Verilog source code will override this option.
NO	+vcdfile+<filename>	Specifies the VCD file you want to use for post-processing.
NO	-xzcheck [nofalseneg]	Checks all the conditional expressions in the design and displays a warning message every time VCS evaluates a conditional expression to have an X or Z value. nofalseneg Suppresses the warning message when the value of a conditional expression transitions to X or Z and then to 0 or 1 in the same simulation time step.
NO	+maxdelays	Species using the compiled SDF file for maximum delays generated by the +allmtm compile-time option. Also specifies using maximum delays for SWIFT VMC or SmartModels or Synopsys hardware models if you also enter the +override_model_delays runtime option.
NO	+mindelays	Specifies using the compiled SDF file for minimum delays generated by the +allmtm compile-time option. Also specifies using minimum delays for SWIFT VMC or SmartModels or Synopsys hardware models if you also enter the +override_model_delays runtime option.
NO	+no_notifier	Suppresses the toggling of notifier registers that are optional arguments of timing check system tasks.
NO	+no_pulse_msg	Suppresses pulse error messages, but not the generation of StX values at module path outputs when a pulse error condition occurs.
NO	+no_tchk_msg	Disables the display of timing check warning messages but does not disable the toggling of notifier registers in timing checks. This is also a compile-time option.
NO	+notimingcheck	Suppress timing checks.
NO	+ntb_cache_dir=<path_name_to_directory>	Specifies the directory location of the cache that VCS maintains as an internal disk cache for randomization.
		Causes the simulation to stop immediately when a

NO	+ntb_debug_on_error	simulation error is encountered. In addition to normal verification errors, This option halts the simulation in case of runtime errors as well.
NO	+ntb_enable_solver_trace=<value>	Enables a debug mode that displays diagnostics when VCS executes a randomize() method call. Allowed values are: 0 Do not display (default). 1 Displays the constraints VCS is solving. 2 Displays the entire constraint set.
NO	+ntb_enable_solver_trace_on_failure=<value>	Enables a mode that displays trace information only when the VCS constraint solver fails to compute a solution, usually due to inconsistent constraints. When the value of the option is 2, the analysis narrows down to the smallest set of inconsistent constraints, thus aiding the debugging process. Allowed values are 0, 1, 2. The default value is 2.
NO	+ntb_exit_on_error[=<value>]	Causes VCS to exit when value is less than 0. The value can be: 0 continue. The simulation finishes regardless of the number of errors 1 exit on first error (default value) N exit on Nth error
NO	+ntb_load=path_name_to_libtb.so	Specifies loading the testbench shared object file libtb.so.
NO	+ntb_random_seed=<value>	Sets the seed value used by the top level random number generator at the start of simulation. The random(seed) system function call overrides this setting. The value can be any integer number.
NO	+ntb_solver_mode=<value>	Allows choosing between one of two constraint solver modes. When set to 1, the solver spends more pre-processing time in analyzing the constraints, during the first call to randomize() on each class. When set to 2, the solver does minimal pre-processing, and analyzes the constraint in each call to randomize(). Default value is 2.
NO	+ntb_stop_on_error	Causes the simulation to stop immediately when a simulation error is encountered, turning it into a cli debugging environment. In addition to normal verification errors, ntb_stop_on_error halts the simulation in case of run time errors. The default setting is to execute the remaining code within the present simulation time.
NO	+override_model_delays	Enables you to use the +mindelays, +typdelays, or +maxdelays runtime options to specify timing for SWIFT SmartModels or Synopsys hardware models.
NO	+sdfverbose	Enables the display of more than ten warning and ten error messages about SDF back annotation.
NO	+typdelays	Specifies using the compiled SDF file for typical delays generated by the +allmtm compile-time option. Also specifies using typical delays for SWIFT VMC or

		SmartModels or Synopsys hardware models if you also enter the +override_model_delays runtime option.
NO	+vcs+dumparrays	Enables dumping memory and multi-dimesional array values in the VCD file. You must also have use the +memcbk compile-time option.
NO	+vcs+dumpoff+<t>+<ht>	Turn off value change dumping ($dumpvars system task) at time <t>. <ht> is the high 32 bits of a time value greater than 32 bits.
NO	+vcs+dumpon+<t>+<ht>	Suppress $dumpvars system task until time <t>. <ht> is the high 32 bits of a time value greater than 32 bits.
NO	+vcs+dumpvarsoff	Suppress $dumpvars system tasks.
YES	+vcs+finish+<t>+<ht>	Finish simulation at time <t>. <ht> is the high 32 bits of a time value greater than 32 bits.
NO	+vcs+grwavesoff	Suppress $gr_waves system tasks.
NO	+vcs+ignorestop	Tells VCS to ignore the $stop system tasks in your source code.
NO	+vcs+flush+log	Increases the frequency of flushing both the compilation and simulation log file buffers.
NO	+vcs+flush+dump	Increases the frequency of flushing all the buffers for VCD files.
NO	+vcs+flush+fopen	Increases the frequency of flushing all the buffers for files opened by the $fopen system function.
NO	+vcs+flush+all	Shortcut option for entering all three of the +vcs+flush+log, +vcs+flush+dump, and +vcs+flush+fopen options.
NO	+vcs+learn+pli	Keeps track of where you use ACC capabilities for debugging operations so that you can recompile your design and in the next simulation enable them only where you need them. With this option VCS writes the pli_learn.tab secondary PLI table file. You input this file when you recompile your design with the +applylearn compile-time option.
NO	+vcs+lic+vcsi	Checks out three VCSi licenses to run VCS.
NO	+vcsi+lic+vcs	Checks out a VCS license to run VCSi when all VCSi licenses are in use.
NO	+vcs+lic+wait	Wait for network license if none is available when the job starts.
NO	+vcsi+lic+wait	Tells VCSi to wait for a network license if none is available.
NO	+vcs+mipd+noalias	If during a simulation run, acc_handle_simulated_net is called before MIPD annotation happens, a warning message is issued. When this happens you can use this option to disable such aliasing for all ports whenever mip, mipb capabilities have been specified. This option works for regular sdf annotation and not for compiled SDF.
NO	+vcs+nostdout	Disables all text output from VCS including messages and text from $monitor and $display and other system tasks. VCS still writes this output to the log file if you include the -l option.

NO	+vcs+stop+<t>+<ht>	Stop simulation at time <t>. <ht> is the high 32 bits of a time value greater than 32 bits (optional). See the section on "Specifying A Long Time Before Stopping Simulation" in the VCS/VCSi User Guide.
NO	+vera_load=<filename.vro>	Specifies the VERA object file.
NO	+vera_mload=<filename>	Specifies a text file that contains a list of VERA object files.

Options for Specifying How VCS Writes the VPD File

NO	+vpdfile+<filename>	At runtime, defines an alternative name of the VPD file that VCS writes instead of the default name vcdplus.vpd.
NO	+vpdfileswitchsize+<number_in_MB>	Specifies a size for the VPD file. When the VPD file reaches this size, VCS closes the VPD file and opens a new one with the same design hierarchy as the previous VPD file. There is a number suffix added to the VPD file name to differentiate them.
NO	+vpdbufsize+<MB>	VCS uses an internal buffer to store value changes before it writes them to the VPD file on disk. VCS makes this buffer size either 5 MB or large enough to record 15 value changes for all nets and registers in your design, which ever is larger. You can use this option to override the buffer size that VCS calculates for the buffer size. You specify a buffer size in megabytes.
NO	+vpddrivers	Tells VCS to record the values of all the drivers of all the nets.
NO	+vpdfilesize+<MB>	Specifies the maximum size of the VPD file. When VCS reaches this limit, VCS overwrites the oldest simulation history data in the file with the newest.
NO	+vpdignore	Tells VCS to ignore $vcdplus system tasks so VCS does not write a VPD file.
NO	+vpdports	Tells VCS to record, in the VPD file, the port direction of signals that are ports.
NO	+vpdnocompress	Disables the automatic compressing of the data in VPD files.
NO	+vpdupdate	If VCS is writing a VPD file during simulation, this option enables you to have VCS halt writing to the VPD file while the simulation is running and so that you can view the recorded results in DVE. This option enables you to use the update feature in DVE.
NO	+vpdnostrengths	Disables recording strength information in the VPD file.

The following VCS environment variables are supported as shown.

Environment Variables

Supported	**switch**	**Description**
NO	DISPLAY_VCS_HOME	Enables the display at compile time of the path to the directory specified with the VCS_HOME environment.
NO	VCS_HOME	Specifies the directory where you installed VCS.
NO	VCSI_HOME	Specifies the directory where you installed VCSi.
NO	TMPDIR	Specifies the directory for temporary compilation files.
NO	VCS_CC	Specifies the C compiler.
NO	VCS_COM	Specifies the path to the VCS compiler executable named vcs1 (or vcs1.exe).
NO	VCS_LOG	Specifies the runtime log file name.
NO	VCS_RUNTIME	Specifies which runtime library named libvcs.a VCS uses.

NO	VCS_SWIFT_NOTES	Enables the printf PCL command.
NO	VCS_WARNING_ATSTAR	Specifies the number of signals in a Verilog-2001 implicit sensitivity list that must be exceeded before VCS displays a warning. The default limit is 100 signals.

Verilog to Verilog

Verifying Verilog to Verilog

The ESP tool supports Verilog designs in both the reference and implementation containers.

ESP supports using different compiler directives for the two containers. This means you can have one source design with a `define macro that selects a different RTL/behavioral coding.

Do not define supplies for Verilog to Verilog verification. Treat power and ground ports as constants with the `-function binary` option to the **set_testbench_pin_attributes** command or treat them the same as any other input.

Note:
When both containers are Verilog, ESP generates a testbench that uses the strict compare checker for outputs instead of the comparex checker that are created for Verilog to SPICE verification.

Examples

Verify ram design with an FPGA (ram_fpga.v) and a behavioral (ram.v) description.

```
read_verilog -r ram.v
set_top_design -r RAM
read_verilog -i ram_fpga.v
set_top_design -i RAM
match_design_ports
set_testbench_pin_attributes -checker compare Q
set_testbench_pin_attributes -function read RE
set_testbench_pin_attributes -function address A
set_testbench_pin_attributes -function clock CLK
set_testbench_pin_attributes -function data DI
set_testbench_style sram
verify
```

Verify a ram design where one source file includes both versions of the design. The macros FPGA and RTL are used to select the appropriate model.

```
read_verilog -r ram.v -vcs {+define+RTL}
set_top_design -r RAM
read_verilog -i ram.v -vcs {+define+FPGA}
set_top_design -i RAM
match_design_ports
set_testbench_pin_attributes -checker compare Q
set_testbench_pin_attributes -function read RE
set_testbench_pin_attributes -function address A
set_testbench_pin_attributes -function clock CLK
set_testbench_pin_attributes -function data DI
set_testbench_style sram
verify
```