Module 21

Using System Tasks and System Functions

This module examines system tasks and system functions commonly used for stimulus generation and debugging. Later modules will revisit some of these in more detail and introduce some more.

Module Objective

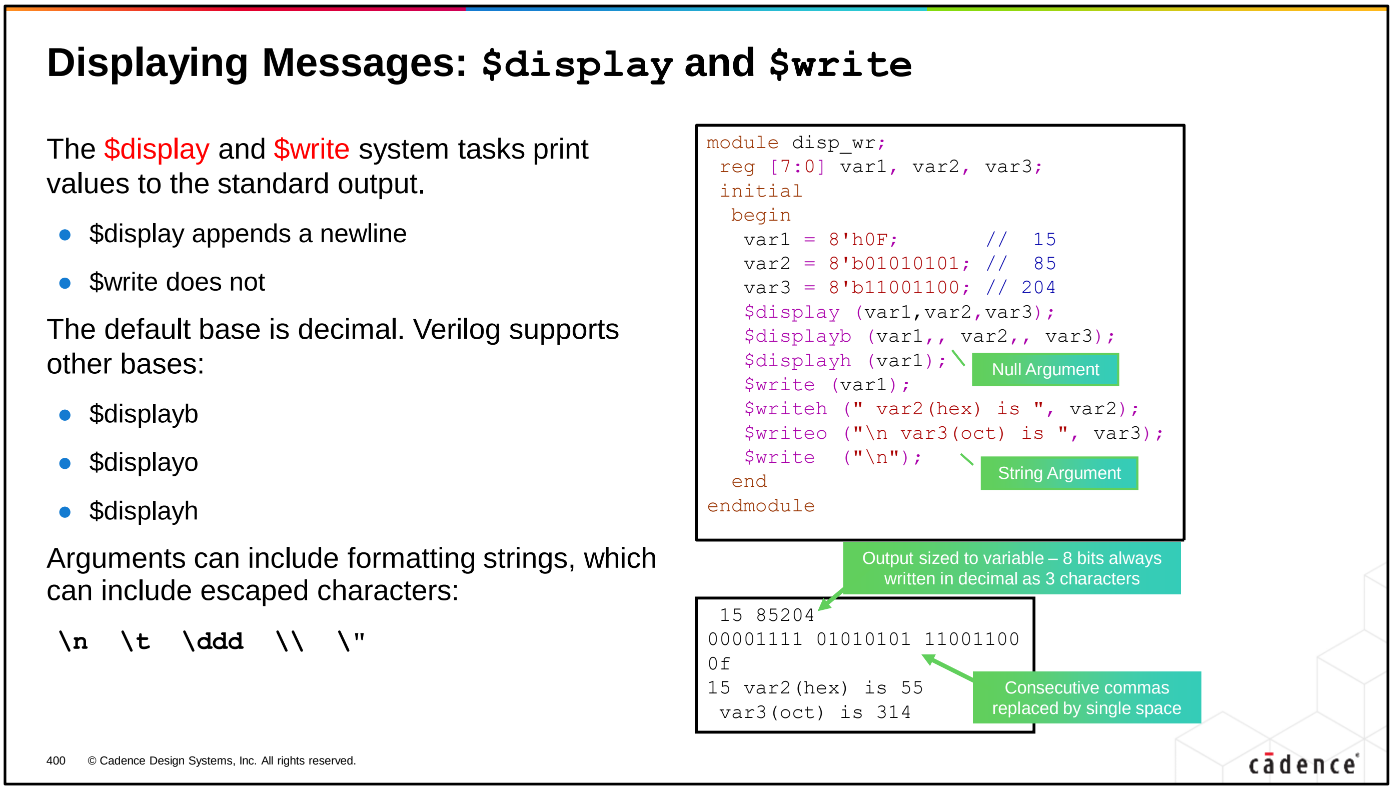
In this module, you:

- Use system tasks and functions for stimulus generation and debug

Topics

- Displaying messages
- Getting simulation time
- Formatting the time display
- File input and output
- Applying stimulus from a file
- Controlling the simulation
- Passing real values through ports
- Getting command-line values
- Dumping value-change data
- Checkpointing the simulation

This page does not contain notes.



Displaying Messages: `$display` and `$write`

The $display and $write system tasks print values to the standard output.

- $display appends a newline
- $write does not

The default base is decimal. Verilog supports other bases:

- $displayb
- $displayo
- $displayh

Arguments can include formatting strings, which can include escaped characters:

\n \t \ddd \\ \"

```
module disp_wr;
 reg [7:0] var1, var2, var3;
 initial
  begin
   var1 = 8'h0F;        //  15
   var2 = 8'b01010101;  //  85
   var3 = 8'b11001100;  // 204
   $display (var1,var2,var3);
   $displayb (var1,, var2,, var3);
   $displayh (var1);
   $write (var1);
   $writeh (" var2(hex) is ", var2);
   $writeo ("\n var3(oct) is ", var3);
   $write  ("\n");
  end
endmodule
```

Null Argument

String Argument

Output sized to variable – 8 bits always written in decimal as 3 characters

```
 15 85204
00001111 01010101 11001100
0f
15 var2(hex) is 55
 var3(oct) is 314
```

Consecutive commas replaced by single space

400 © Cadence Design Systems, Inc. All rights reserved.

cādence

The $display and $write system tasks are identical, except that $display adds a newline character to the end of every output, and $write does not.

The $display and $write system tasks support multiple default bases. The default base for $display and $write is decimal. Append the b character to make the default base binary, o for octal and h for hexadecimal. You can embed formatters in string arguments to override that default radix for individual subsequent arguments. Later pages demonstrate this.

The display width depends on the base and the maximum size of the variable. For example, the maximum value of an 8-bit vector is 256, which requires 3 decimal characters, so every 8-bit value written in decimal requires 3 characters. To display an 8-bit vector requires 8 characters in binary, 3 in octal and 2 in hexadecimal. The decimal radix replaces leading zeros by spaces, while the other radices display the leading zeros. When using a format string, you can override this automatic sizing. Later pages demonstrate this.

You can include string arguments to separate values and to make the output more meaningful. You can also separate values by including null arguments. A null argument is an extra comma between arguments. For a null argument, the simulator displays a single space.

Formatting Text Output

Each format specifier formats one argument value.

- Usually grouped into one first argument format string.
- Can be distributed among the task arguments.
- Matched to argument values in order – one to one match.
  - More values than format specifiers simply use default format.
  - More format specifiers than values is an error.

```
module disp_wr_fmt;
 reg [7:0] var1, var2, var3, var4;
 initial
  begin
   var1 = 8'h0F;        //  15
   var2 = 8'b01010101;  //  85
   var3 = 8'b11001100;  // 204
   $display ("%b", var1);
   $display ("var2 binary: %b \t", var2, "hex: %h", var2);
   $display ("%h \t %h \t %h", var1, var2, var3);
   $write ("var3 is hex %h \t octal %o \n", var3, var3);
  end
endmodule
```

```
00001111
var2 binary: 01010101     hex: 55
0f        55        cc
var3 is hex cc     octal 314
```

You can embed formatters in your string arguments to, among other things, override the default base. A formatter can apply to only one subsequent argument, but other than that, you can group the formatters any way you want. Many people put them all in one first argument string, and some people insert individual format strings just prior to the argument they apply to.

When using format strings, you must be very careful to not apply the formatters to null arguments. You might want to adopt the convention that you do not use both in the same statement.

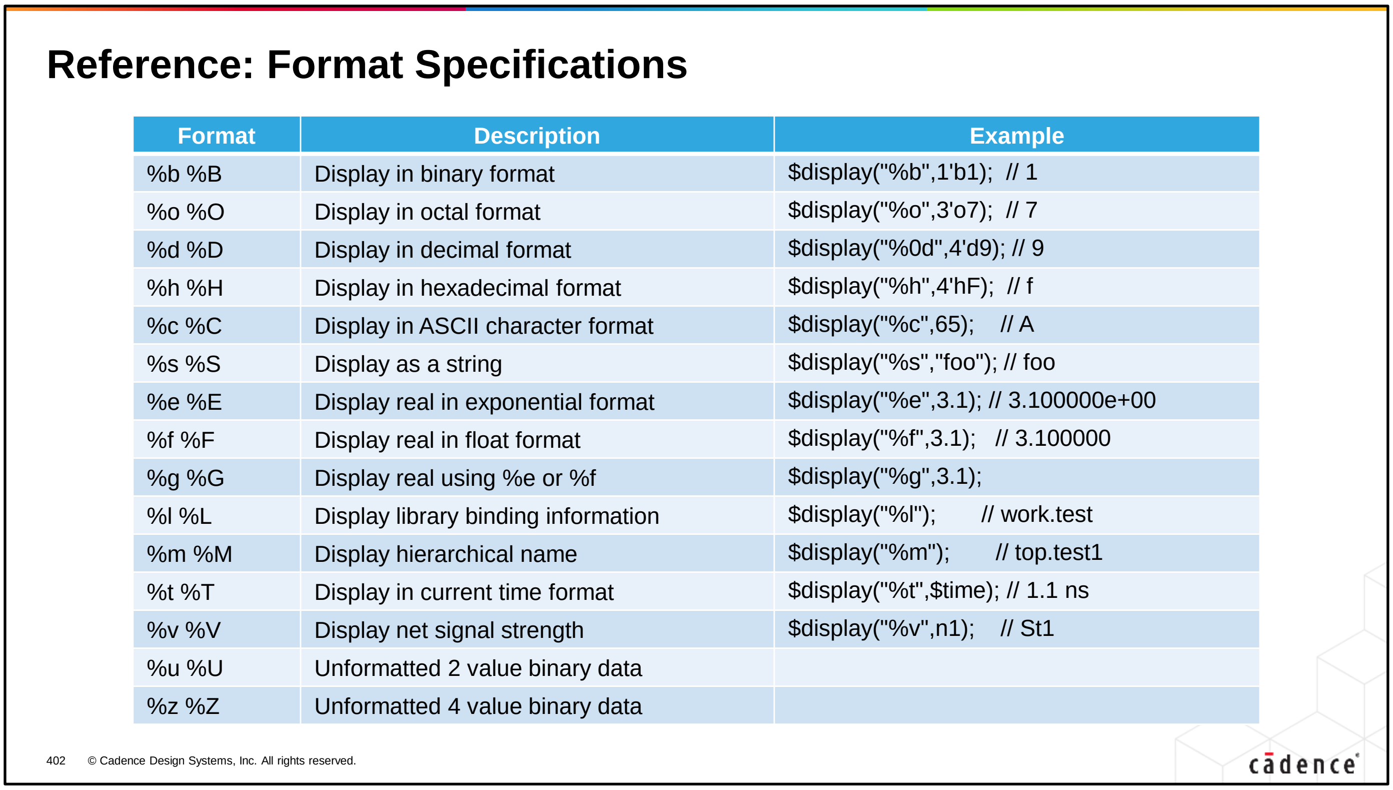
Within any character string literal, you can place escaped character sequences to represent special characters such as newline and tab characters.

The 1st display statement overrides the default base to display the value of the var1 argument in binary. Format characters can be in either case, but most people use the lower case as is done here. The var1 argument is an 8-bit vector, so displays in 8 characters.

The 2nd display statement displays the value of var2 in binary and hexadecimal, with a space and tab between them.

The 3rd display statement displays the value of var1, var2 and var3 in hexadecimal with a space and tab and an extra space between them.

The 4th display statement displays the value of var3 in hexadecimal and octal, with a space and tab and extra space between them.



Reference: Format Specifications

Format	Description	Example
%b %B	Display in binary format	$display("%b",1'b1); // 1
%o %O	Display in octal format	$display("%o",3'o7); // 7
%d %D	Display in decimal format	$display("%0d",4'd9); // 9
%h %H	Display in hexadecimal format	$display("%h",4'hF); // f
%c %C	Display in ASCII character format	$display("%c",65); // A
%s %S	Display as a string	$display("%s","foo"); // foo
%e %E	Display real in exponential format	$display("%e",3.1); // 3.100000e+00
%f %F	Display real in float format	$display("%f",3.1); // 3.100000
%g %G	Display real using %e or %f	$display("%g",3.1);
%l %L	Display library binding information	$display("%l"); // work.test
%m %M	Display hierarchical name	$display("%m"); // top.test1
%t %T	Display in current time format	$display("%t",$time); // 1.1 ns
%v %V	Display net signal strength	$display("%v",n1); // St1
%u %U	Unformatted 2 value binary data	
%z %Z	Unformatted 4 value binary data	

402 © Cadence Design Systems, Inc. All rights reserved.

cādence

The integer formatters by default display values in a width sufficient to contain the maximum value the expression size can represent. For the decimal format they display leading spaces and for the other formats they display leading zeros. This tabular format makes vertically long displays more readable, but often unnecessarily wastes horizontal display space. An integer, for example, always requires 10 spaces no matter how small the value. You can override this automatic sizing. Placing a 0 character between the % character and the radix character reduces the displayed width to only the width actually required to display the value. Unlike the C language, in Verilog, you cannot specify the display format more precisely.

The binary formatter always displays high-impedance or unknown bits with lower case z and x characters. The octal and hexadecimal formatters use upper case Z and X characters where the bits that the character represents contain at least one bit with a binary value and at least one bit with a high-impedance or unknown value. In this case, an unknown bit trumps any high-impedance bits to display an upper case X. The decimal formatter does not try to compute the bit range in this case and displays the entire number using uppercase Z or uppercase X.

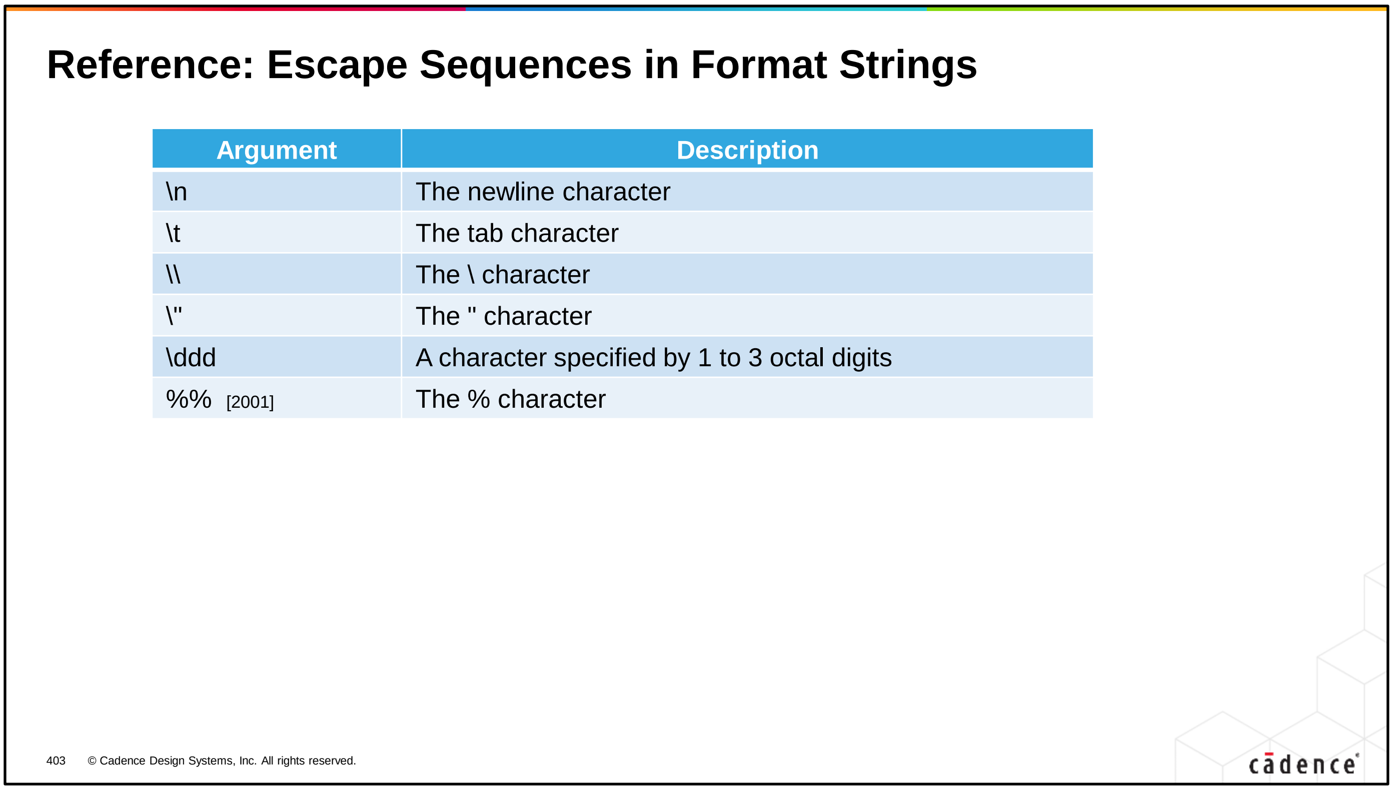
The %l formatter displays the library binding information – the module definition name preceded by a period (.) and the name of the library in which the elaborator found the compiled definition.

The %m formatter displays the hierarchical scope of the instance containing the display system task.

The %t formatter displays the argument according to the format you set with the $timeformat system task. With this task, you set the format width and precision for a time argument. You can use this formatter for any integer value if you want to thoroughly confuse your co-workers.

The %v formatter displays the strength and value of a net. Net drivers have associated with them one of eight strengths, that the simulator uses to resolve the strength and value of the net.

The %u and %z formatters write binary data, presumably to a file, for an application that reads binary data.



Reference: Escape Sequences in Format Strings

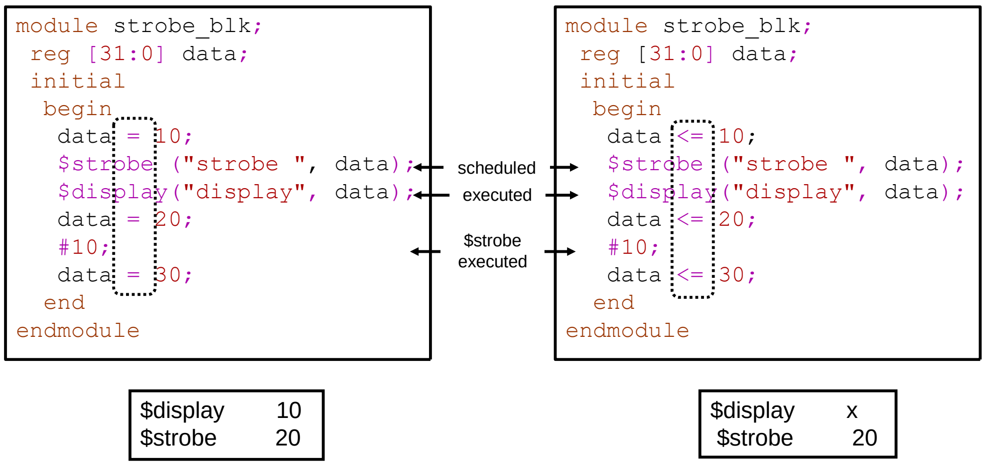
Argument	Description
\n	The newline character
\t	The tab character
\\	The \ character
\"	The " character
\ddd	A character specified by 1 to 3 octal digits
%% [2001]	The % character

Here is the full set of escape sequences provided by the IEEE Std 1364-2001 Verilog HDL. The Verilog-2001 update added the escaped percent (%) character. Note that it is escaped by another percent character and not by a backslash (\) character.

Displaying Messages: `$strobe`

The $strobe system task is similar to the $display system task

- Execution of $strobe is scheduled for the end of the current time "slice".



```
module strobe_blk;
 reg [31:0] data;
 initial
  begin
   data = 10;
   $strobe ("strobe ", data);
   $display("display", data);
   data = 20;
   #10;
   data = 30;
  end
endmodule
```

```
module strobe_blk;
 reg [31:0] data;
 initial
  begin
   data <= 10;
   $strobe ("strobe ", data);
   $display("display", data);
   data <= 20;
   #10;
   data <= 30;
  end
endmodule
```

```
$display     10
$strobe      20
```

```
$display      x
$strobe      20
```

404 © Cadence Design Systems, Inc. All rights reserved.

The $strobe system task evaluates and displays its arguments at the end of the current simulation time, and is otherwise identical to the $display system task. Like the $display system task, it is available in four versions with four different default bases.

The left example uses blocking assignments, so the $display system task displays the assigned value 10 and the $strobe system task displays the later assigned value 20.

The right example uses nonblocking assignments, so the $display system task displays the not-yet-assigned default unknown value (x) and the $strobe system task displays the later assigned value 20.

Getting Simulation Time: `$stime`, `$time`, `$realtime`

The $time, $stime and $realtime system functions return the simulation time:

- $time
  - Returns time as a 64-bit integer
- $stime
  - Returns time as a 32-bit integer
- $realtime
  - Returns time as a real number

The simulator scales returned time values to the timescale of the calling module.

```
`timescale 1 ns / 100 ps

module time_sysf;
 reg [5:0] data;
 initial
  begin
   #10;
   data = 6'd20;
   $display ($time,, data);
   #10.4;
   data = 6'd30;
   $write ("time: %0d", $time);
   $write (" realtime: %f", $realtime);
   $write (" data: ", data, "\n");
  end
endmodule
```

%0d removes leading spaces

```
                    10 20
time: 20 realtime: 20.4 data: 30
```

The `timescale directive specifies the timescale units for following modules.

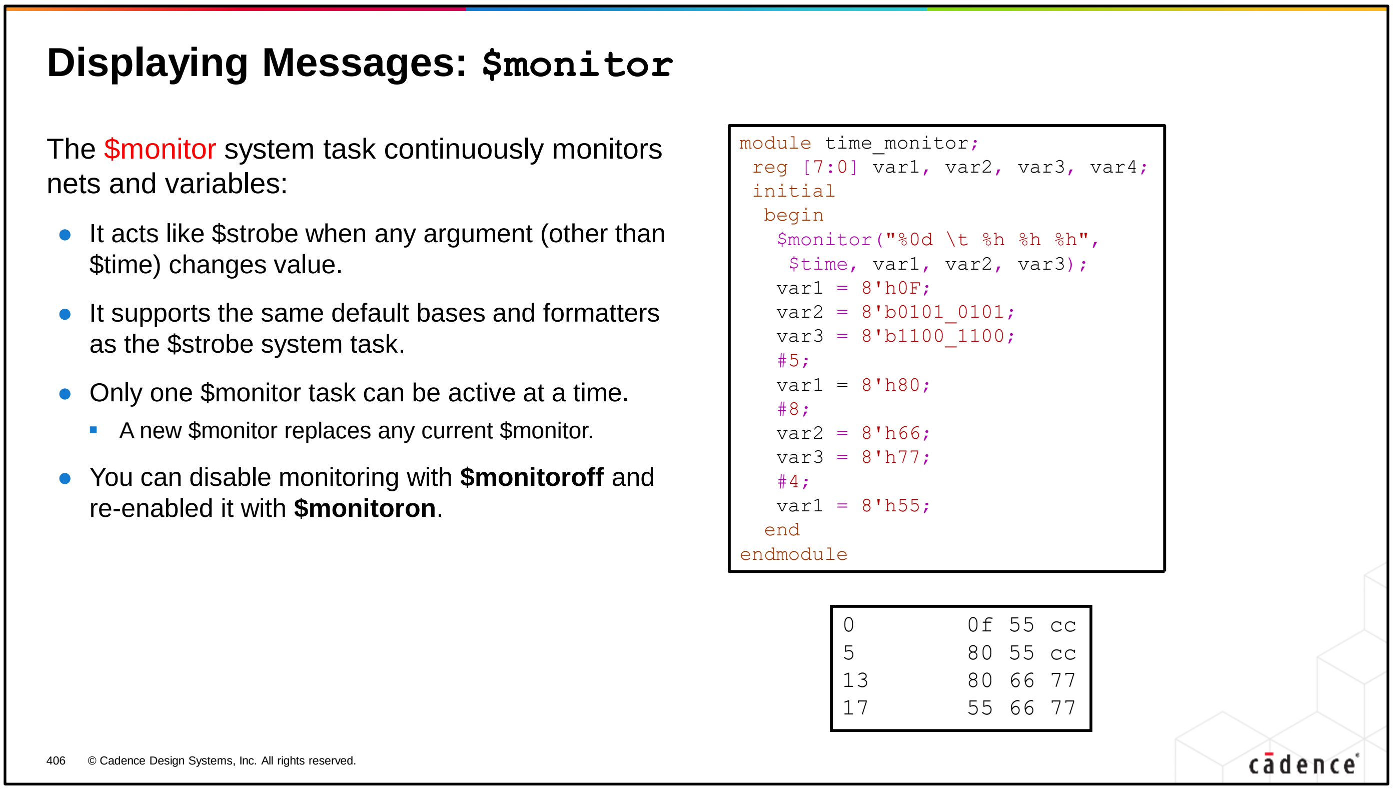
The $time system function returns 64-bit simulation time scaled to the time scale of the calling module and then if needed rounded to an integer value.

The $stime system function returns the lower 32 bits of the $time value. This might not for obvious reasons represent the actual simulation time. For short simulations, you can use $stime to display a value that takes ten spaces to display instead of twenty.

The $realtime system function returns 64-bit simulation time scaled to the time scale of the calling module and converted to a real value.

The $display statement displays 64-bit simulation time and a 6-bit data value separated with a space character.

The $write statements display formatted 64-bit simulation time followed by time as a real number followed by the data value, each separated by a space character.



Displaying Messages: `$monitor`

The $monitor system task continuously monitors nets and variables:

- It acts like $strobe when any argument (other than $time) changes value.
- It supports the same default bases and formatters as the $strobe system task.
- Only one $monitor task can be active at a time.
  - A new $monitor replaces any current $monitor.
- You can disable monitoring with **$monitoroff** and re-enabled it with **$monitoron**.

```
module time_monitor;
 reg [7:0] var1, var2, var3, var4;
 initial
  begin
   $monitor("%0d \t %h %h %h",
    $time, var1, var2, var3);
   var1 = 8'h0F;
   var2 = 8'b0101_0101;
   var3 = 8'b1100_1100;
   #5;
   var1 = 8'h80;
   #8;
   var2 = 8'h66;
   var3 = 8'h77;
   #4;
   var1 = 8'h55;
  end
endmodule
```

```
0          0f 55 cc
5          80 55 cc
13         80 66 77
17         55 66 77
```

406 © Cadence Design Systems, Inc. All rights reserved.

cadence

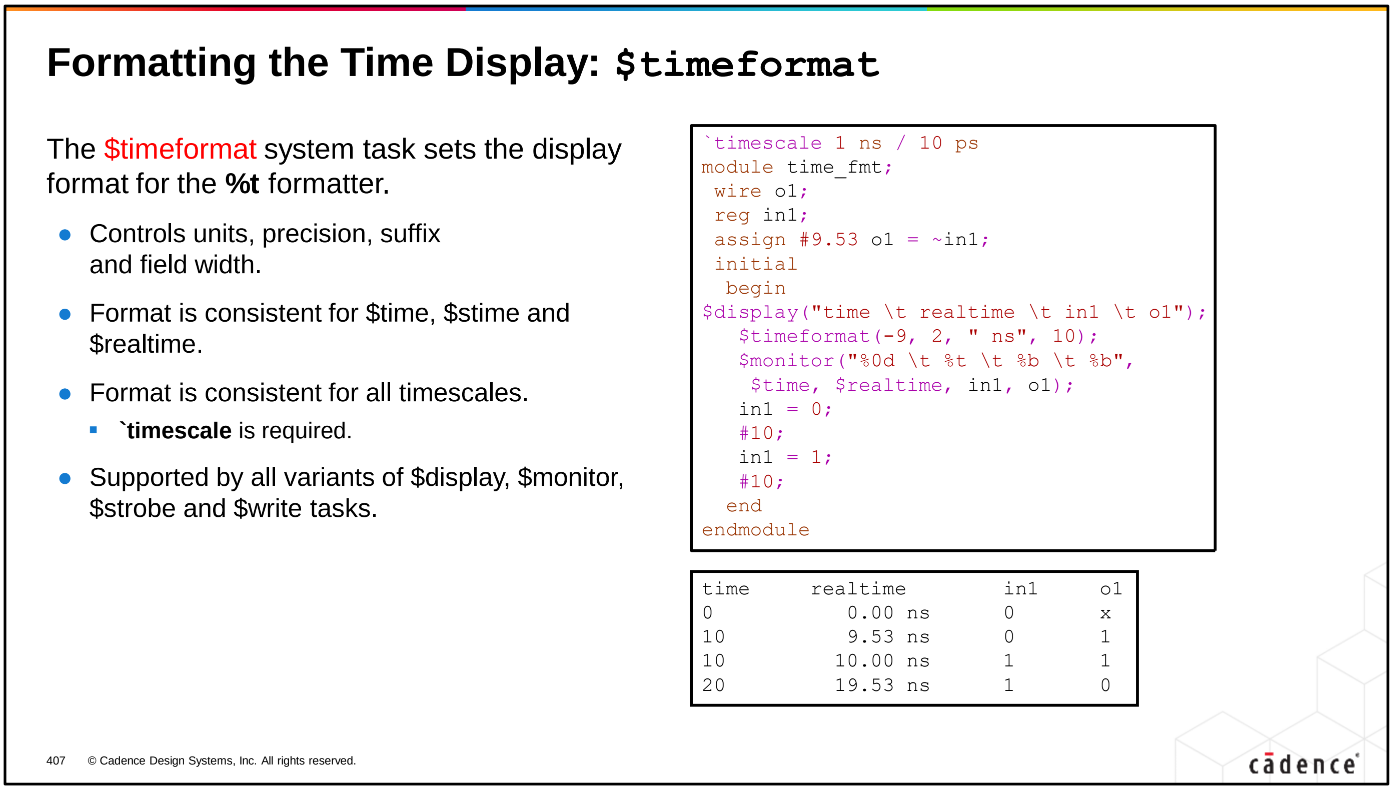
The $monitor system task, once invoked, continually monitors its arguments, and displays their values at the end of the time “slice” in which any argument changed.

Thus the $monitor system task, like the $strobe system task, involves the scheduler.

Like the $strobe system task, it is available in four versions with four different default bases.

At most one monitor is active at a time. A new call to $monitor replaces the list of monitored signals. The simulator ceases monitoring the previous list of signals and instead monitors the new list of signals.

You can control monitoring with the $monitoroff and $monitoron system tasks. You can use these system tasks to monitor signal values only between certain time intervals of the simulation.



Formatting the Time Display: `$timeformat`

The $timeformat system task sets the display format for the **%t** formatter.

- Controls units, precision, suffix and field width.
- Format is consistent for $time, $stime and $realtime.
- Format is consistent for all timescales.
  - **`timescale** is required.
- Supported by all variants of $display, $monitor, $strobe and $write tasks.

```
`timescale 1 ns / 10 ps
module time_fmt;
 wire o1;
 reg in1;
 assign #9.53 o1 = ~in1;
 initial
  begin
$display("time \t realtime \t in1 \t o1");
   $timeformat(-9, 2, " ns", 10);
   $monitor("%0d \t %t \t %b \t %b",
    $time, $realtime, in1, o1);
   in1 = 0;
   #10;
   in1 = 1;
   #10;
  end
endmodule
```

```
time     realtime         in1     o1
0           0.00 ns       0       x
10          9.53 ns       0       1
10         10.00 ns       1       1
20         19.53 ns       1       0
```

407 © Cadence Design Systems, Inc. All rights reserved.

cādence

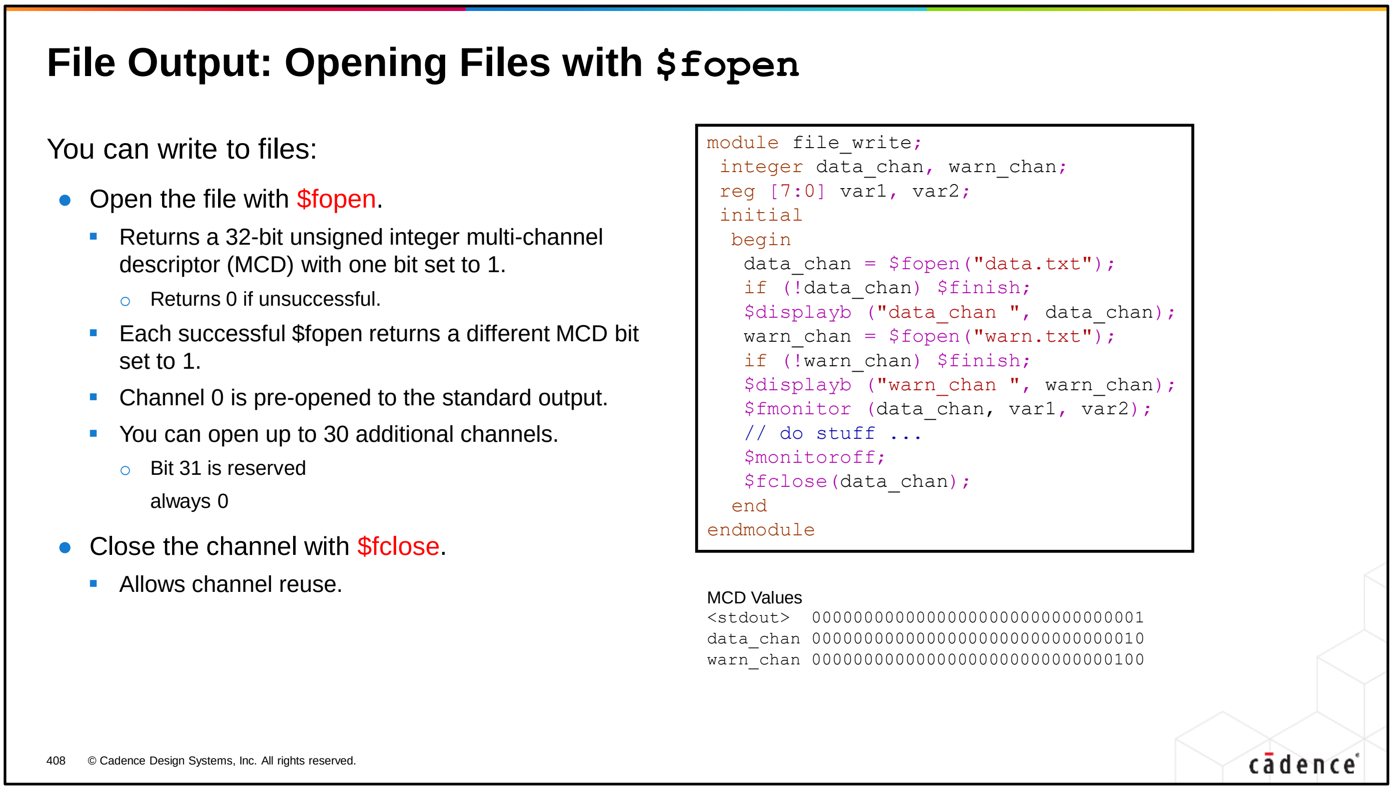
You use the $timeformat system task to specify how the %t formatter displays time values.

The $timeformat system task takes four arguments:

- The 1st argument is an integer giving the time unit to which to scale the displayed time. The argument value must be between 0, meaning seconds, and –15, meaning femtoseconds.
- The 2nd argument is an integer giving the precision for the display, that is, how many digits to display after the decimal point.
- The 3rd argument is a string to append to the display. People typically use this string to show the time units they scaled the time to.
- The 4th argument is an integer giving the minimum field width for the display. The display inserts leading spaces to force this minimum field width.

The $timeformat system task in this example scales simulation time to nanoseconds, displays it with two fractional digits, follows it with a string indicating its time units, and displays it with a ten-character minimum field width.

The IEEE Std 1364-2001 Verilog HDL Section 17.3.2 states that the $timeformat applies to “all %t formats specified in all modules that follow in the source description until another $timeformat system task is invoked”.



File Output: Opening Files with `$fopen`

You can write to files:

- Open the file with $fopen.
  - Returns a 32-bit unsigned integer multi-channel descriptor (MCD) with one bit set to 1.
    - Returns 0 if unsuccessful.
  - Each successful $fopen returns a different MCD bit set to 1.
  - Channel 0 is pre-opened to the standard output.
  - You can open up to 30 additional channels.
    - Bit 31 is reserved always 0
- Close the channel with $fclose.
  - Allows channel reuse.

```
module file_write;
 integer data_chan, warn_chan;
 reg [7:0] var1, var2;
 initial
  begin
   data_chan = $fopen("data.txt");
   if (!data_chan) $finish;
   $displayb ("data_chan ", data_chan);
   warn_chan = $fopen("warn.txt");
   if (!warn_chan) $finish;
   $displayb ("warn_chan ", warn_chan);
   $fmonitor (data_chan, var1, var2);
   // do stuff ...
   $monitoroff;
   $fclose(data_chan);
  end
endmodule
```

MCD Values

```
<stdout>  00000000000000000000000000000001
data_chan 00000000000000000000000000000010
warn_chan 00000000000000000000000000000100
```

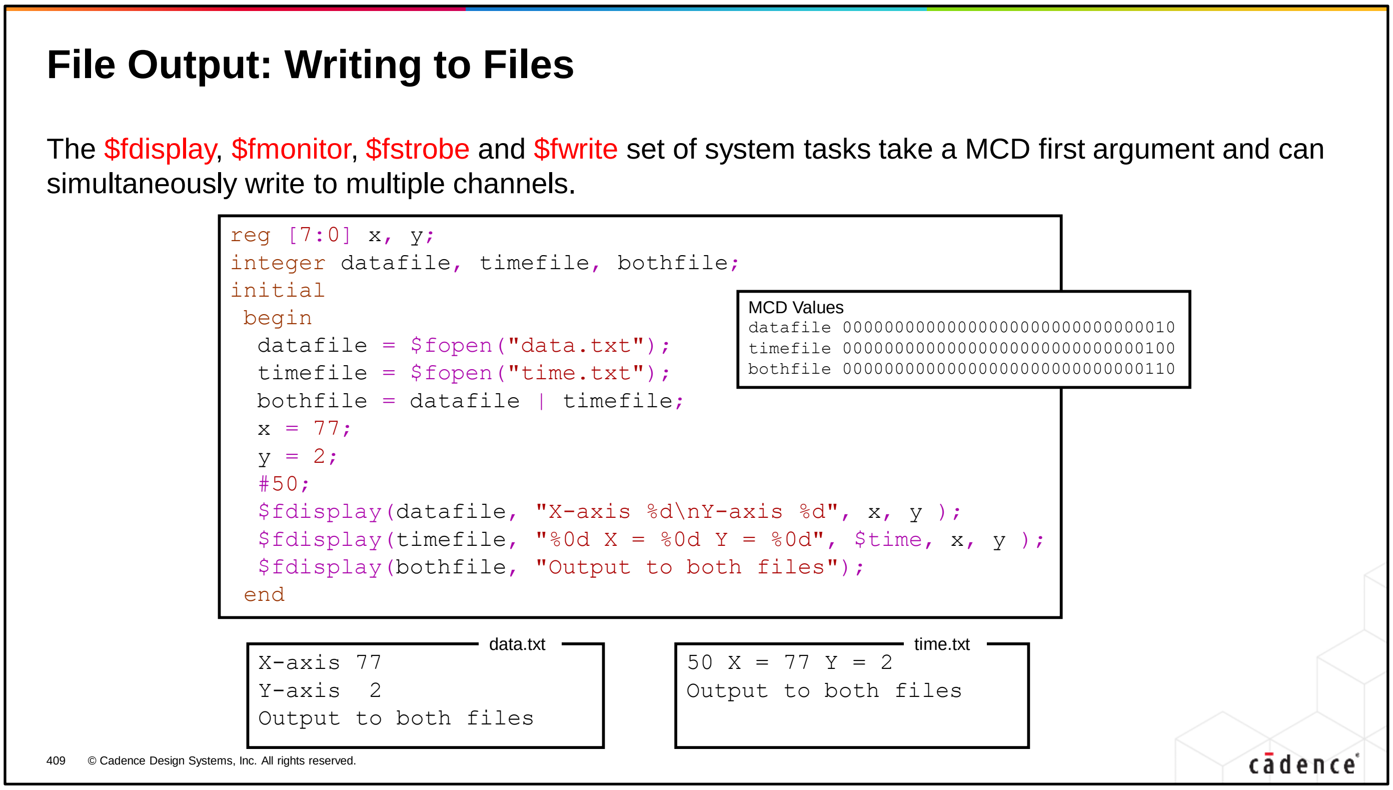
Use the $fopen system function to open a file. The $fopen system function returns a 32-bit unsigned multichannel descriptor (MCD) uniquely associated with the file. The system function returns 0 if it could not open the file for writing.

You must save the returned MCD as an integer or 32-bit reg vector.

Think of the MCD as a set of 32 flags, where each flag represents a single output channel:

- The most significant bit of an MCD is reserved for purposes we will discuss later and is always 0.
- The least significant bit of an MCD is reserved for the standard output. Most simulators also write this output to a log file.
- You can open up to 30 additional channels and simultaneously write to any combination of the open channels.

Use the $fclose system task to close the channels specified in its MCD argument. The $fclose system task closes the specified channels. Subsequent calls to the $fopen system function reuse those released channels.



File Output: Writing to Files

The $fdisplay, $fmonitor, $fstrobe and $fwrite set of system tasks take a MCD first argument and can simultaneously write to multiple channels.

```
reg [7:0] x, y;
integer datafile, timefile, bothfile;
initial
 begin
  datafile = $fopen("data.txt");
  timefile = $fopen("time.txt");
  bothfile = datafile | timefile;
  x = 77;
  y = 2;
  #50;
  $fdisplay(datafile, "X-axis %d\nY-axis %d", x, y );
  $fdisplay(timefile, "%0d X = %0d Y = %0d", $time, x, y );
  $fdisplay(bothfile, "Output to both files");
 end
```

MCD Values
```
datafile 00000000000000000000000000000010
timefile 00000000000000000000000000000100
bothfile 00000000000000000000000000000110
```

data.txt
```
X-axis 77
Y-axis   2
Output to both files
```

time.txt
```
50 X = 77 Y = 2
Output to both files
```

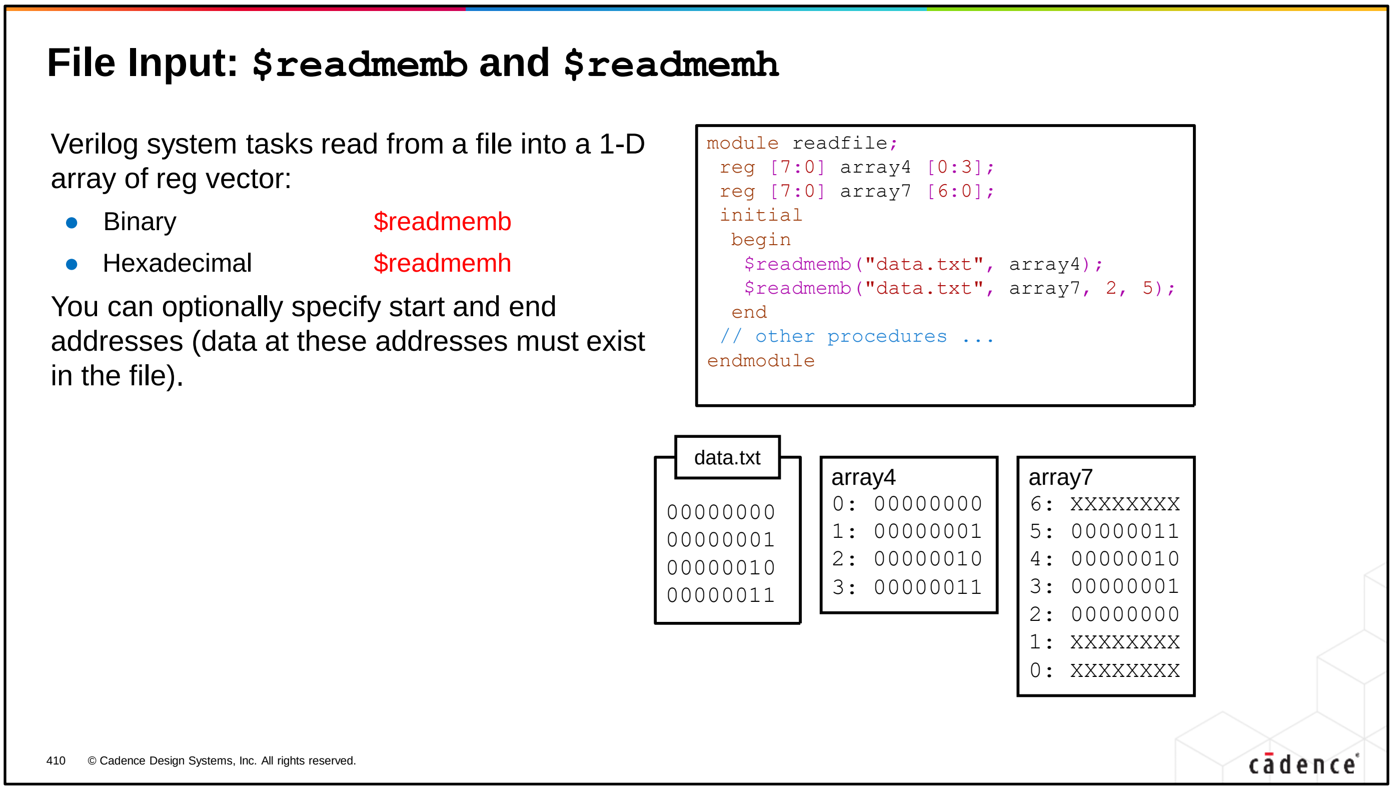
409 © Cadence Design Systems, Inc. All rights reserved.

cādence

The four formatted display tasks ($display, $write, $monitor, and $strobe) have counterparts that write to a specific set of channels rather than to just the standard output.

The counterparts have an f prefix character and take an extra MCD first argument. The MCD can be any arbitrary expression that results in a 32-bit unsigned integer value. This value determines which open files the system task writes to. You simultaneously write multiple channels by bitwise ORing existing MCDs. You can set bit 0 to also write to the standard output.

This example opens the data.txt file and the time.txt file. It then bitwise ORs the two MCDs to create a third MCD representing both channels. It later writes to the channels individually and simultaneously.



File Input: `$readmemb` and `$readmemh`

Verilog system tasks read from a file into a 1-D array of reg vector:

- Binary $readmemb
- Hexadecimal $readmemh

You can optionally specify start and end addresses (data at these addresses must exist in the file).

```
module readfile;
 reg [7:0] array4 [0:3];
 reg [7:0] array7 [6:0];
 initial
  begin
   $readmemb("data.txt", array4);
   $readmemb("data.txt", array7, 2, 5);
  end
 // other procedures ...
endmodule
```

data.txt

```
00000000
00000001
00000010
00000011
```

array4

```
0: 00000000
1: 00000001
2: 00000010
3: 00000011
```

array7

```
6: XXXXXXXX
5: 00000011
4: 00000010
3: 00000001
2: 00000000
1: XXXXXXXX
0: XXXXXXXX
```

410 © Cadence Design Systems, Inc. All rights reserved.

cadence

The $readmemb and $readmemh system tasks load data from a text file into an array. Depending on the task, you provide the data in a binary radix or a hexadecimal radix. The system task has no versions to accept octal data or decimal data.

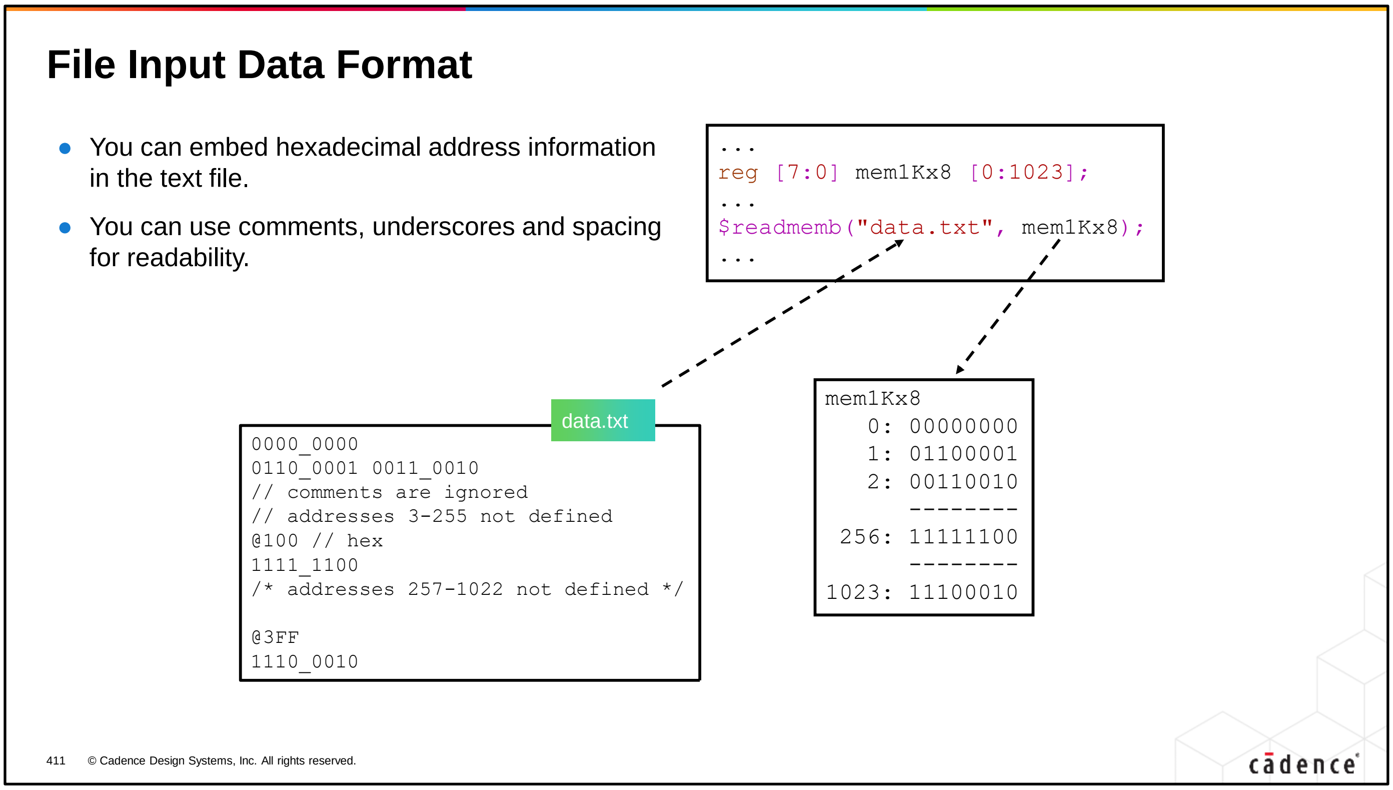
- The 1st argument is the data file name.
- The 2nd argument is the array to receive the data.
- The 3rd argument is an optional start address, and if you provide it, you can also provide –
- The 4th argument optional end address.

The data file itself can optionally specify to what addresses its data belongs.

If the memory addresses are not specified anywhere, then the system tasks load file data sequentially from the left memory address bound to the right memory address bound. The year 2005 standard changes this default loading order to be from the lowest address toward the highest address.

If the file does not specify addresses for its data and the system task specifies a starting address, then the system tasks load file data sequentially from that starting address toward the specified ending address, or if no ending address is specified, then toward the right memory address bound for Verilog 2001 tools and toward the high memory address bound for Verilog 2005 tools.

If the file does specify addresses for its data, then if the system task specifies an address range, that address range must encompass and exhaust the available data.



File Input Data Format

- You can embed hexadecimal address information in the text file.
- You can use comments, underscores and spacing for readability.

```
...
reg [7:0] mem1Kx8 [0:1023];
...
$readmemb("data.txt", mem1Kx8);
...
```

data.txt

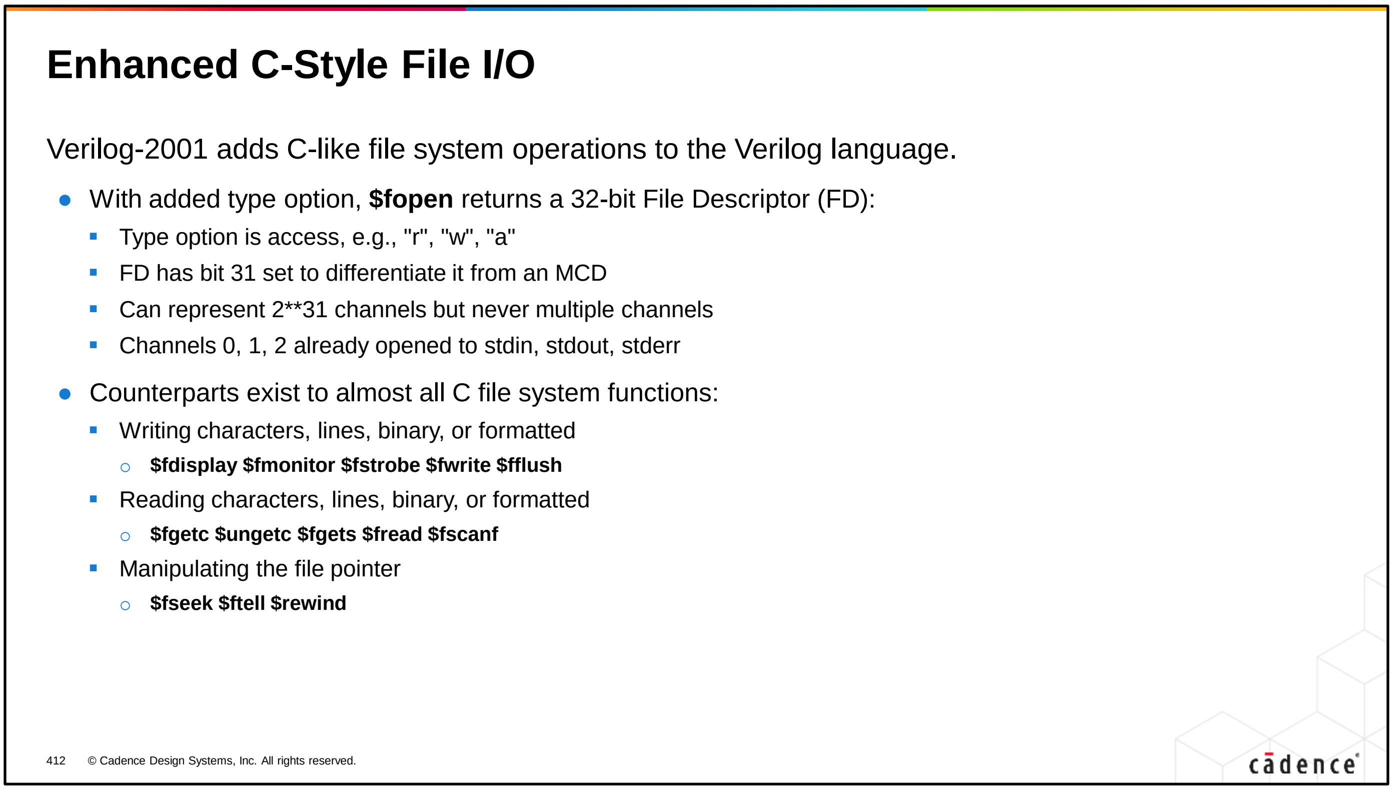
```
0000_0000
0110_0001 0011_0010
// comments are ignored
// addresses 3-255 not defined
@100 // hex
1111_1100
/* addresses 257-1022 not defined */

@3FF
1110_0010
```

```
mem1Kx8
     0: 00000000
     1: 01100001
     2: 00110010
        --------
   256: 11111100
        --------
  1023: 11100010
```

Here are the things that a data file can contain:

- White space, including spaces, newlines, tabs, and formfeeds.
- Comments, which can be single line or multiline.
- Binary numbers or case-insensitive hexadecimal numbers, which:
  - If address values, are hexadecimal digits following an at (@) character.
  - If data, are a sequence of binary or hexadecimal digits without base or size information but that might embed underscore (_) characters for readability.

This example data file contains 8-bit binary data for only the addresses 0, 1, 2, 256 and 1023. A system task call that specifies an address range must include these addresses in that range.



Enhanced C-Style File I/O

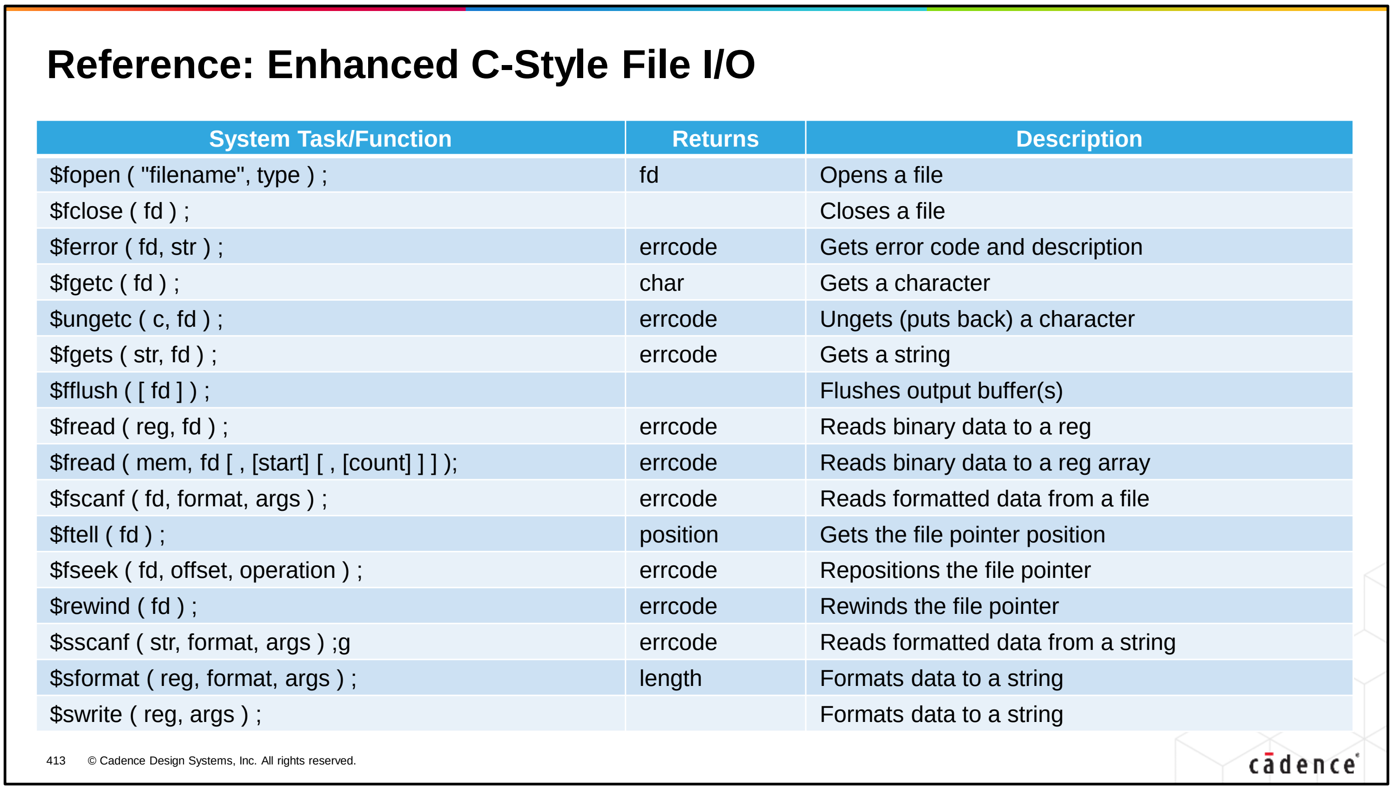
Verilog-2001 adds C-like file system operations to the Verilog language.

- With added type option, **$fopen** returns a 32-bit File Descriptor (FD):
  - Type option is access, e.g., "r", "w", "a"
  - FD has bit 31 set to differentiate it from an MCD
  - Can represent 2**31 channels but never multiple channels
  - Channels 0, 1, 2 already opened to stdin, stdout, stderr
- Counterparts exist to almost all C file system functions:
  - Writing characters, lines, binary, or formatted
    - **$fdisplay $fmonitor $fstrobe $fwrite $fflush**
  - Reading characters, lines, binary, or formatted
    - **$fgetc $ungetc $fgets $fread $fscanf**
  - Manipulating the file pointer
    - **$fseek $ftell $rewind**

412 © Cadence Design Systems, Inc. All rights reserved.

The Verilog 2001 update added support for C-style file I/O routines, thus providing the capability to input something more than just a memory data. These routines utilize a 32-bit unsigned integer file descriptor. A file descriptor represents a single channel. The file descriptor has the most significant bit set to 1 to differentiate it from a multichannel descriptor, which has the most significant bit set to 0. As in C, the first three channels are pre-opened to the standard output, input, and error files.

The enhanced C-style file I/O re-uses the existing output system tasks and provides new system tasks for input and for file pointer manipulation.



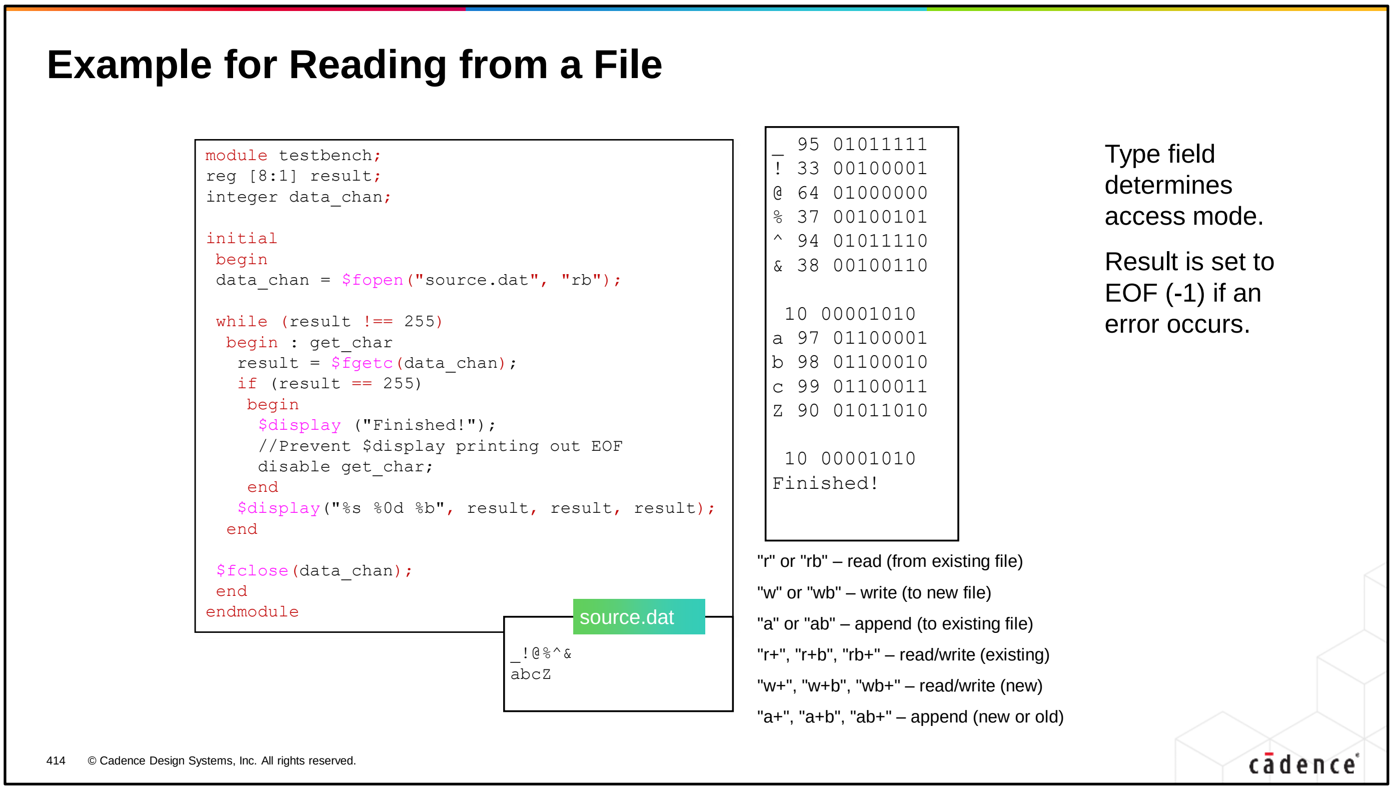
Reference: Enhanced C-Style File I/O

System Task/Function	Returns	Description
$fopen ("filename", type) ;	fd	Opens a file
$fclose (fd) ;		Closes a file
$ferror (fd, str) ;	errcode	Gets error code and description
$fgetc (fd) ;	char	Gets a character
$ungetc (c, fd) ;	errcode	Ungets (puts back) a character
$fgets (str, fd) ;	errcode	Gets a string
$fflush ([fd]) ;		Flushes output buffer(s)
$fread (reg, fd) ;	errcode	Reads binary data to a reg
$fread (mem, fd [, [start] [, [count]]]);	errcode	Reads binary data to a reg array
$fscanf (fd, format, args) ;	errcode	Reads formatted data from a file
$ftell (fd) ;	position	Gets the file pointer position
$fseek (fd, offset, operation) ;	errcode	Repositions the file pointer
$rewind (fd) ;	errcode	Rewinds the file pointer
$sscanf (str, format, args) ;g	errcode	Reads formatted data from a string
$sformat (reg, format, args) ;	length	Formats data to a string
$swrite (reg, args) ;		Formats data to a string

413 © Cadence Design Systems, Inc. All rights reserved.

cādence

This page intends to make you aware of what is available but encourages you to refer to the Verilog standard for full and complete documentation. The system tasks are similar, but not identical, to the C standard I/O library routines.



Example for Reading from a File

```
module testbench;
reg [8:1] result;
integer data_chan;

initial
 begin
 data_chan = $fopen("source.dat", "rb");

 while (result !== 255)
  begin : get_char
   result = $fgetc(data_chan);
   if (result == 255)
    begin
     $display ("Finished!");
     //Prevent $display printing out EOF
     disable get_char;
    end
   $display("%s %0d %b", result, result, result);
  end

 $fclose(data_chan);
 end
endmodule
```

source.dat

```
_!@%^&
abcZ
```

```
_  95  01011111
!  33  00100001
@  64  01000000
%  37  00100101
^  94  01011110
&  38  00100110

 10  00001010
a  97  01100001
b  98  01100010
c  99  01100011
Z  90  01011010

 10  00001010
Finished!
```

"r" or "rb" – read (from existing file)

"w" or "wb" – write (to new file)

"a" or "ab" – append (to existing file)

"r+", "r+b", "rb+" – read/write (existing)

"w+", "w+b", "wb+" – read/write (new)

"a+", "a+b", "ab+" – append (new or old)

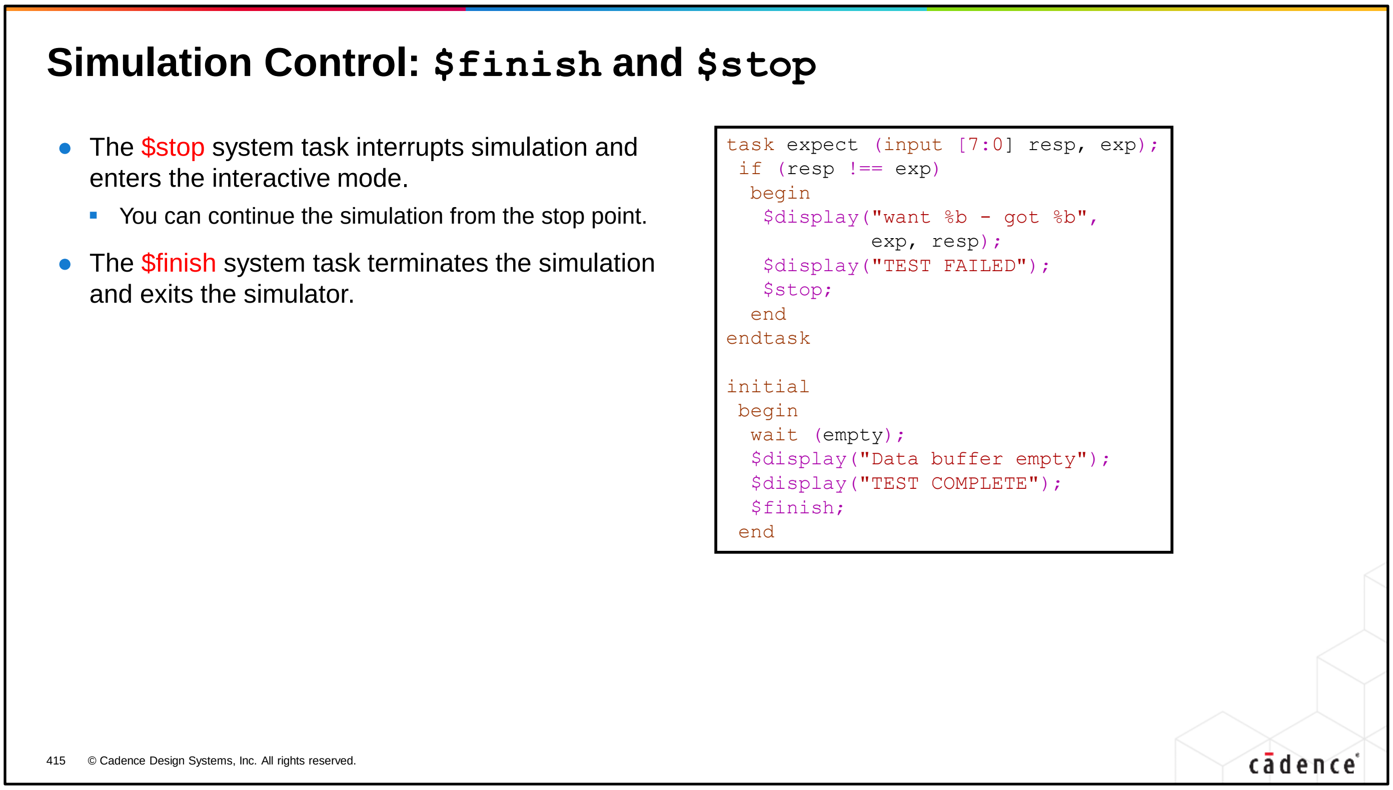
Type field determines access mode.

Result is set to EOF (-1) if an error occurs.

414 © Cadence Design Systems, Inc. All rights reserved.

cādence

An example for reading from a file using some of the C-style File I/O.



Simulation Control: `$finish` and `$stop`

- The $stop system task interrupts simulation and enters the interactive mode.
  - You can continue the simulation from the stop point.
- The $finish system task terminates the simulation and exits the simulator.

```
task expect (input [7:0] resp, exp);
 if (resp !== exp)
  begin
   $display("want %b - got %b",
            exp, resp);
   $display("TEST FAILED");
   $stop;
  end
endtask

initial
 begin
  wait (empty);
  $display("Data buffer empty");
  $display("TEST COMPLETE");
  $finish;
 end
```

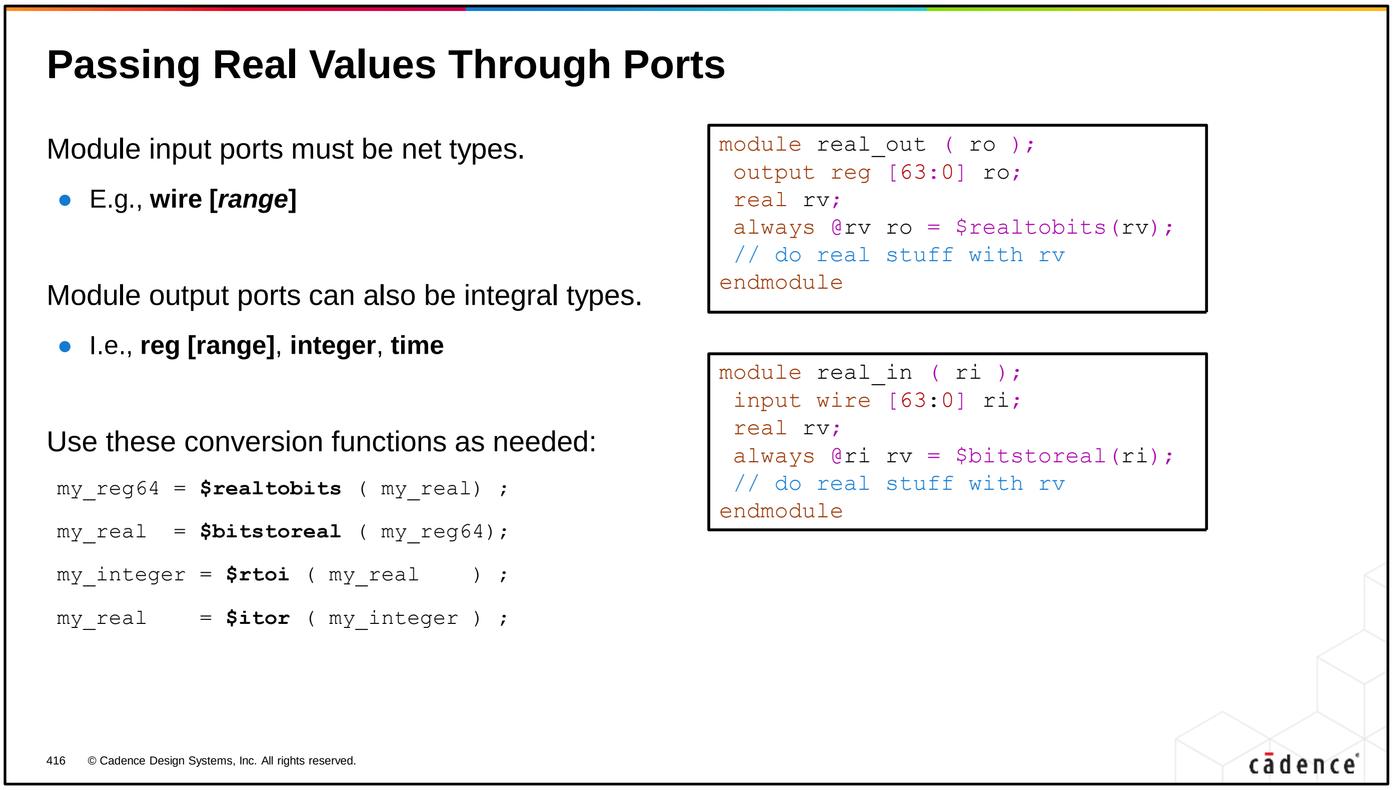
415 © Cadence Design Systems, Inc. All rights reserved.

cādence

The $stop system task interrupts the simulator and lets you enter commands interactively. You can continue the simulation from the stop point.

The $finish system task terminates the simulation.

This example defines a task to compare the actual response with the expected response and interrupt the simulation if they are not identical. If the test completes successfully then the initial block detects the completion and terminates the simulation.



Passing Real Values Through Ports

Module input ports must be net types.

- E.g., **wire [*range*]**

Module output ports can also be integral types.

- I.e., **reg [range]**, **integer**, **time**

Use these conversion functions as needed:

```
my_reg64 = $realtobits ( my_real) ;
my_real  = $bitstoreal ( my_reg64);
my_integer = $rtoi ( my_real    ) ;
my_real    = $itor ( my_integer ) ;
```

```
module real_out ( ro );
 output reg [63:0] ro;
 real rv;
 always @rv ro = $realtobits(rv);
 // do real stuff with rv
endmodule
```

```
module real_in ( ri );
 input wire [63:0] ri;
 real rv;
 always @ri rv = $bitstoreal(ri);
 // do real stuff with rv
endmodule
```

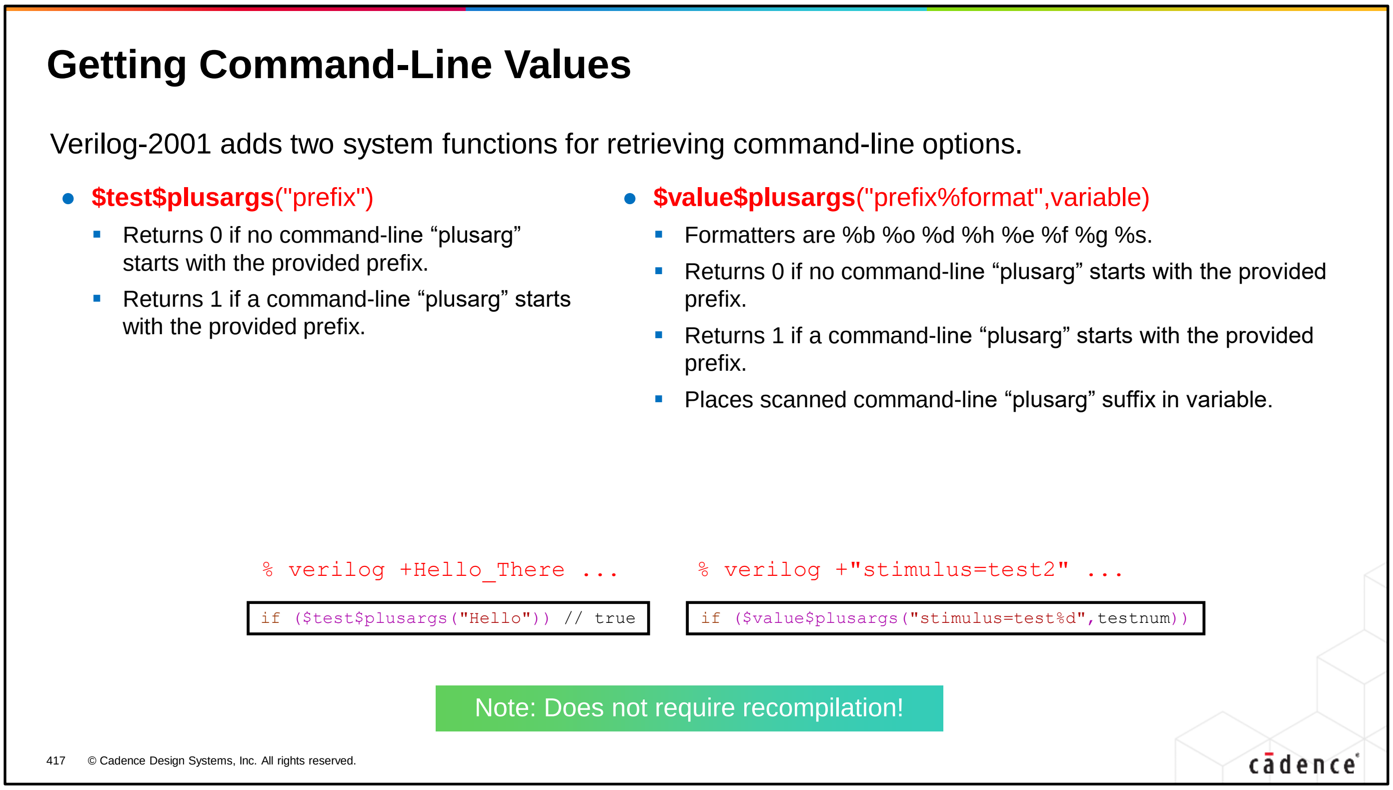
416 © Cadence Design Systems, Inc. All rights reserved.

cādence

You cannot directly pass a real value through a module port. You instead use the $realtobits and $bitstoreal system functions to convert a real value to and from a bit vector that you can pass through module ports. The Verilog bit vector representation of real values conforms to IEEE Std 754-1985 Binary Floating-Point Arithmetic. Verilog rounds the converted value to the nearest legal value.

The example module named real_out calls $realtobits in a procedure to convert an internal real variable value to a vector value for module output.

The example module named real_in calls $bitstoreal in a procedure to convert a vector module input value to a real value for the internal variable.



Getting Command-Line Values

Verilog-2001 adds two system functions for retrieving command-line options.

- **$test$plusargs**("prefix")
  - Returns 0 if no command-line “plusarg” starts with the provided prefix.
  - Returns 1 if a command-line “plusarg” starts with the provided prefix.
- **$value$plusargs**("prefix%format",variable)
  - Formatters are %b %o %d %h %e %f %g %s.
  - Returns 0 if no command-line “plusarg” starts with the provided prefix.
  - Returns 1 if a command-line “plusarg” starts with the provided prefix.
  - Places scanned command-line “plusarg” suffix in variable.

```
% verilog +Hello_There ...
```

```
if ($test$plusargs("Hello")) // true
```

```
% verilog +"stimulus=test2" ...
```

```
if ($value$plusargs("stimulus=test%d",testnum))
```

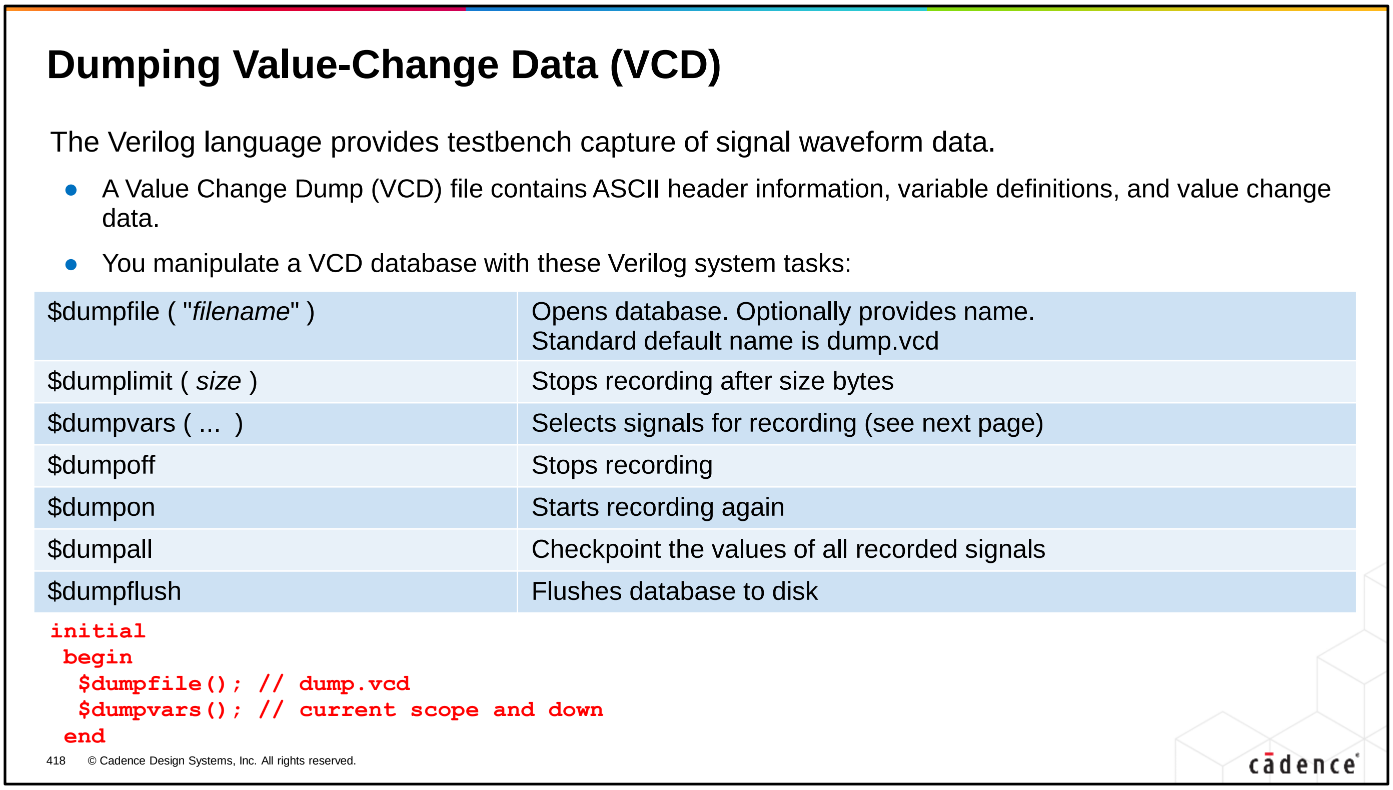
Note: Does not require recompilation!

The Verilog 2001 update added two system functions for retrieving command-line “plus” options:

- The $test$plusargs system function returns 0 if no command-line “plus” argument has the provided prefix, and 1 if at least one command-line “plus” argument does have the provided prefix.
- The $value$plusargs system function does that, and also retrieves the formatted value of the character substring following the prefix. You can use any of the integer, real or string formatters.

The first illustration is determining whether the invocation argument prefix “+Hello_There” was provided.

The second illustration is determining whether the invocation argument prefix “+stimulus=test” was provided, and if it was, interprets the immediately following character substring as a decimal number and places it in the variable named “testnum”.



Dumping Value-Change Data (VCD)

The Verilog language provides testbench capture of signal waveform data.

- A Value Change Dump (VCD) file contains ASCII header information, variable definitions, and value change data.
- You manipulate a VCD database with these Verilog system tasks:

$dumpfile ("*filename*")	Opens database. Optionally provides name. Standard default name is dump.vcd
$dumplimit (*size*)	Stops recording after size bytes
$dumpvars (…)	Selects signals for recording (see next page)
$dumpoff	Stops recording
$dumpon	Starts recording again
$dumpall	Checkpoint the values of all recorded signals
$dumpflush	Flushes database to disk

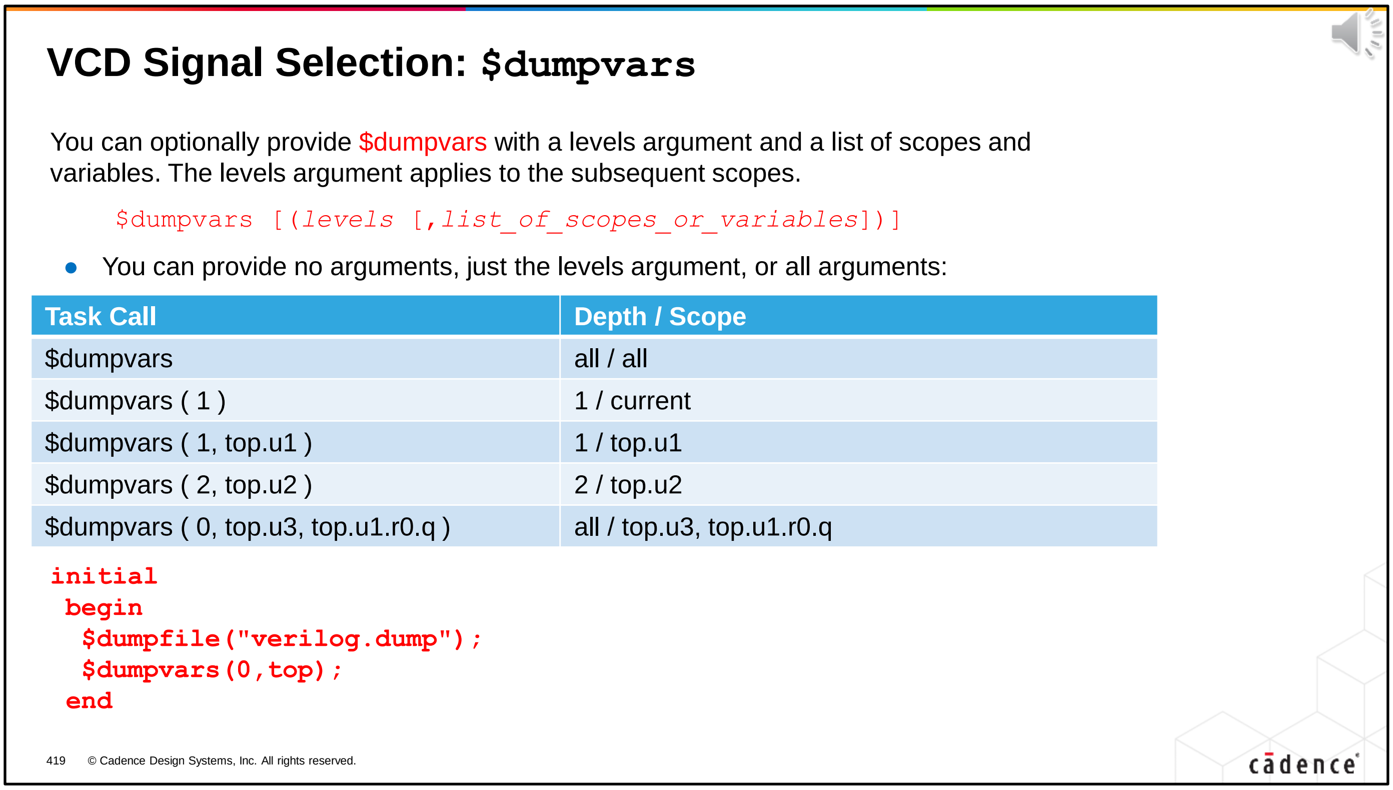
```
initial
 begin
  $dumpfile(); // dump.vcd
  $dumpvars(); // current scope and down
 end
```

418 © Cadence Design Systems, Inc. All rights reserved.

The IEEE Std. 1364-1995 Verilog LRM describes testbench capture of Verilog signal waveform data to a Value Change Dump (VCD) database. A VCD file contains ASCII header information, variable definitions, and value change data. The VCD facility records signal value changes that occur during the simulation of your design. It records changes on only the signals that you select.

You can manipulate a VCD database with these standard Verilog system tasks:

- Use the $dumpfile system task to open a VCD database. If you do not specify a filename argument, the simulator opens the default dump.vcd database. You can open only one database during simulation, and must open it before you invoke any other $dump* tasks.
- Use the $dumplimit system task to limit the size in bytes of the database file. The simulator stops dumping value change data when the database file reaches this limit.
- Use the $dumpvars system task to add signal names for the simulator to dump. You can invoke $dumpvars from several places in your simulation, but must execute them all at the same simulation time. You cannot add signals after the simulator starts dumping data. Following pages describe the levels and list_of_scopes_or_variables arguments you provide upon calling this task.
- Use the $dumpoff system task to suspend dumping of value change data. The simulator writes a record to the database that gives all dumped signals the unknown value, then suspends dumping.
- Use the $dumpon system task to resume dumping of value change data.
- Use the $dumpall system task to create a checkpoint of the dumped signals. The simulator normally dumps individual signal values only as they change. You can use this task to update the database just prior to ending simulation. Some waveform display tools may require this update.
- Use the $dumpflush system task during simulation to flush the dump file buffer to disk if you want some other application to be able to access all of the current data.



VCD Signal Selection: `$dumpvars`

You can optionally provide $dumpvars with a levels argument and a list of scopes and variables. The levels argument applies to the subsequent scopes.

```
    $dumpvars [(levels [,list_of_scopes_or_variables])]
```

- You can provide no arguments, just the levels argument, or all arguments:

Task Call	Depth / Scope
$dumpvars	all / all
$dumpvars (1)	1 / current
$dumpvars (1, top.u1)	1 / top.u1
$dumpvars (2, top.u2)	2 / top.u2
$dumpvars (0, top.u3, top.u1.r0.q)	all / top.u3, top.u1.r0.q

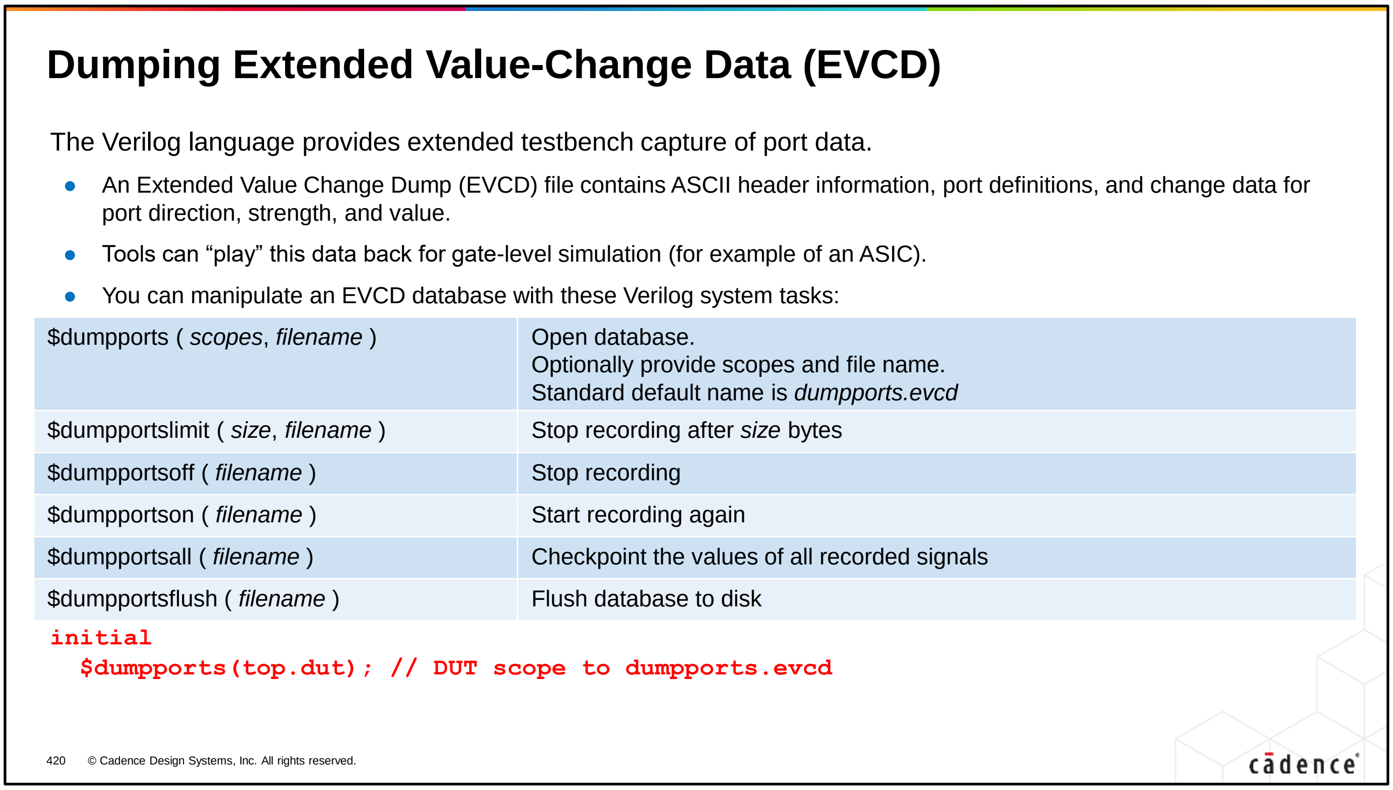
```
initial
 begin
  $dumpfile("verilog.dump");
  $dumpvars(0,top);
 end
```

419 © Cadence Design Systems, Inc. All rights reserved.

Use the $dumpvars system task to add signal names for the simulator to dump. You can invoke $dumpvars from several places in your simulation, but must execute them all at the same simulation time. You cannot add signals after the simulator starts dumping data. You can optionally provide a levels argument and a list_of_scopes_or_variables argument:

- The levels argument applies to subsequent scopes. The levels argument defaults to 0, which means to probe all levels of the current hierarchy.
- The list_of_scopes_or_variables argument defaults to the current scope. If you provide a levels argument, then you can also provide a list_of_scopes_or_variables argument to probe modules, instances, and signals in the specified hierarchies.

The example opens a VCD file called verilog.dump and dumps to it value change data for all nets and variables in and under the scope named top.



Dumping Extended Value-Change Data (EVCD)

The Verilog language provides extended testbench capture of port data.

- An Extended Value Change Dump (EVCD) file contains ASCII header information, port definitions, and change data for port direction, strength, and value.
- Tools can "play" this data back for gate-level simulation (for example of an ASIC).
- You can manipulate an EVCD database with these Verilog system tasks:

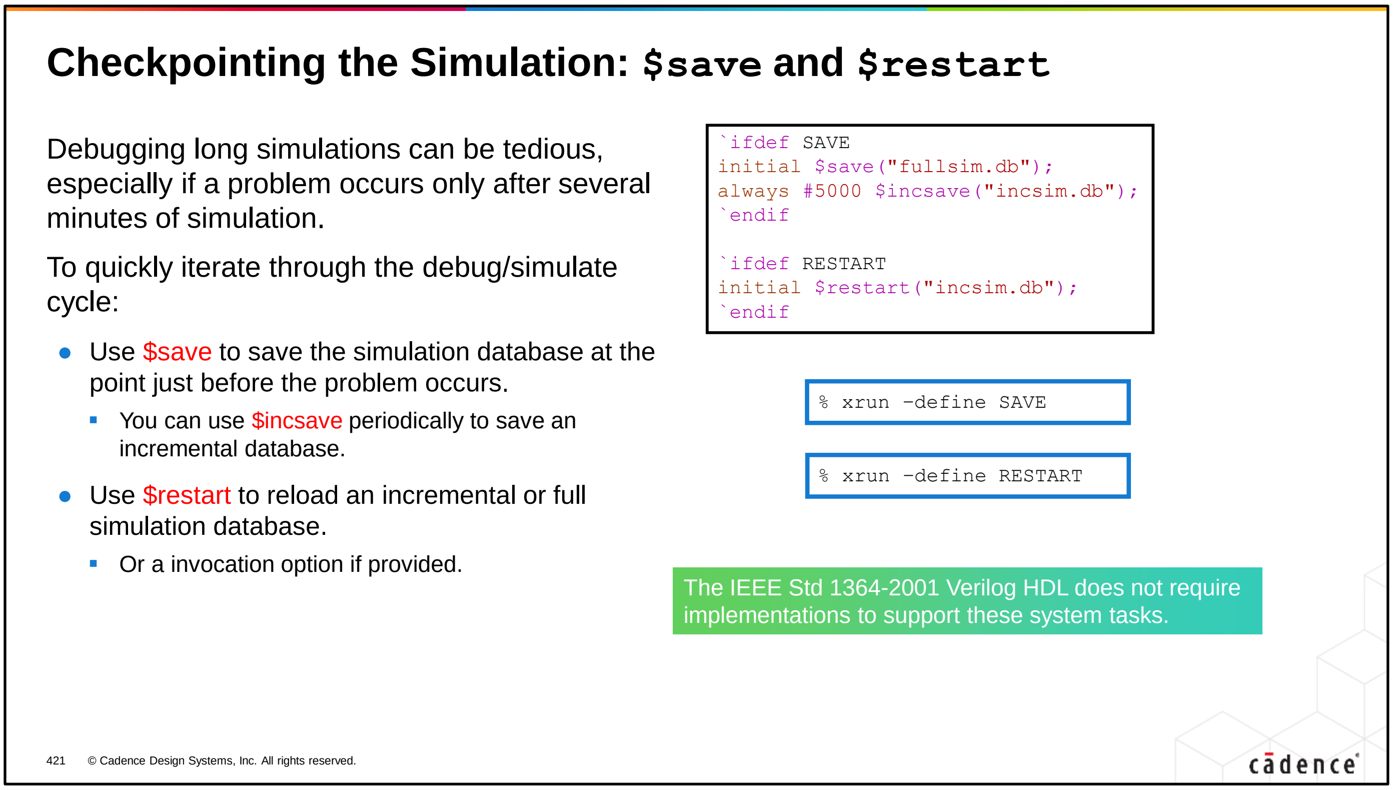
$dumpports (*scopes*, *filename*)	Open database. Optionally provide scopes and file name. Standard default name is *dumpports.evcd*
$dumpportslimit (*size*, *filename*)	Stop recording after *size* bytes
$dumpportsoff (*filename*)	Stop recording
$dumpportson (*filename*)	Start recording again
$dumpportsall (*filename*)	Checkpoint the values of all recorded signals
$dumpportsflush (*filename*)	Flush database to disk

```
initial
  $dumpports(top.dut); // DUT scope to dumpports.evcd
```

The Verilog-2001 update describes testbench capture of Verilog port data to an Extended Value Change Dump (EVCD) database. An EVCD database is a record of the port direction, strength, and value changes that occur during the simulation of your design. It records changes on only the ports that you select.

You can manipulate an EVCD database with these Verilog system tasks. You can provide each task with a filename argument. The filename argument can be a string literal, or a reg variable or expression representing the string's ASCII value. If you do not specify a filename argument, the $dumpports system task by default opens the "dumpports.evcd" database file, and the other system tasks apply to all open EVCD databases:

- Use the $dumpports system task to open an EVCD database and specify the scopes at which to dump port information. The default is the scope of the task call. You can invoke $dumpports from several places in your simulation, but must execute them all at the same simulation time, and may not reuse arguments. You cannot add scopes after the simulator starts dumping data.
- Use the $dumpportslimit system task to limit the size in bytes of the database file. The simulator stops dumping port change data when the database file reaches this limit.
- Use the $dumpportsoff system task to suspend dumping of port change data. The simulator writes a record to the database that gives all dumped ports the unknown value, then suspends dumping.
- Use the $dumpportson system task to resume dumping of port change data.
- Use the $dumpportsall system task to create a checkpoint of the dumped ports. The simulator normally dumps individual port data only as it changes. You can use this task to update the database just prior to ending simulation. Some waveform display tools may require this update.
- Use the $dumpportsflush system task during simulation to flush the dump file buffer to disk if you want some other application to be able to access all of the current data.



Checkpointing the Simulation: `$save` and `$restart`

Debugging long simulations can be tedious, especially if a problem occurs only after several minutes of simulation.

To quickly iterate through the debug/simulate cycle:

- Use $save to save the simulation database at the point just before the problem occurs.
  - You can use $incsave periodically to save an incremental database.
- Use $restart to reload an incremental or full simulation database.
  - Or a invocation option if provided.

```
`ifdef SAVE
initial $save("fullsim.db");
always #5000 $incsave("incsim.db");
`endif

`ifdef RESTART
initial $restart("incsim.db");
`endif
```

```
% xrun -define SAVE
```

```
% xrun -define RESTART
```

The IEEE Std 1364-2001 Verilog HDL does not require implementations to support these system tasks.

421 © Cadence Design Systems, Inc. All rights reserved.

cādence

These constructs are very useful for long simulations.

While debugging a problem, every time you need to add more signal information, you need to reset the simulation and simulate again up to the point of the problem.

If you save the state of the simulation just prior to the point of the problem, then you can quickly restart simulation at the point of the problem.

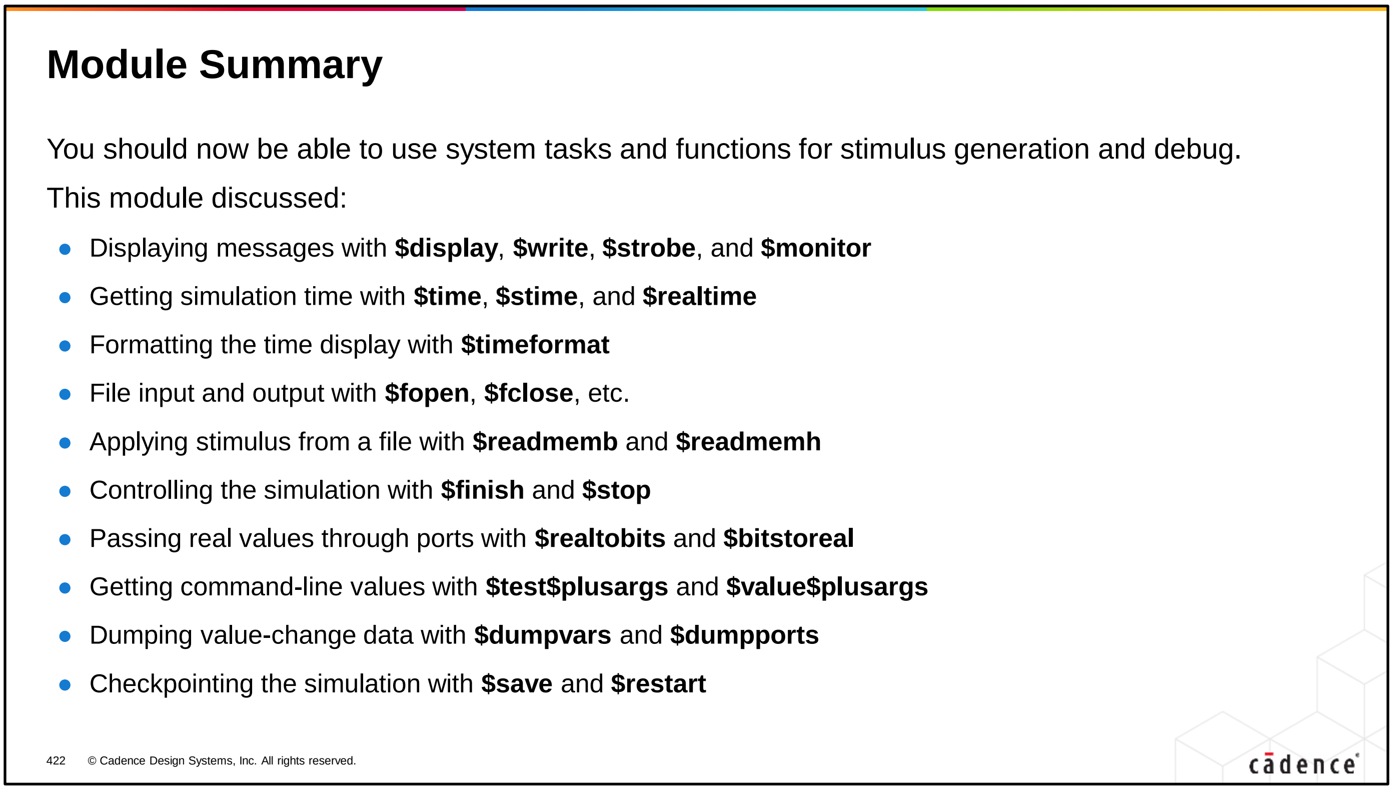
The $save system task saves everything needed to restart the simulation.

The $incsave system task saves only the current states of the nets, variables, and event queues. The incremental database is associated with the full database, so the full database must still exist to restart from an incremental database.

The IEEE Std 1364-2001 Verilog HDL does not require implementations to support these system tasks. Most simulator vendors provide alternative or additional proprietary ways to checkpoint the simulation that may have additional features. Check the vendor’s documentation of their simulator. For example, Cadence® Xcellium™ simulation does not support these system tasks, but instead provides the save and restart interactive commands.

The first illustrated simulation session saves a full simulation database (fullsim.db) and then every 5000 time units saves an incremental simulation database (incsim.db).

The second illustrated simulation session restarts simulation using the incremental simulation database (incsim.db), which first loads the full simulation database (fullsim.db) and then overlays it with the incremental data.



Module Summary

You should now be able to use system tasks and functions for stimulus generation and debug.

This module discussed:

- Displaying messages with **$display**, **$write**, **$strobe**, and **$monitor**
- Getting simulation time with **$time**, **$stime**, and **$realtime**
- Formatting the time display with **$timeformat**
- File input and output with **$fopen**, **$fclose**, etc.
- Applying stimulus from a file with **$readmemb** and **$readmemh**
- Controlling the simulation with **$finish** and **$stop**
- Passing real values through ports with **$realtobits** and **$bitstoreal**
- Getting command-line values with **$test$plusargs** and **$value$plusargs**
- Dumping value-change data with **$dumpvars** and **$dumpports**
- Checkpointing the simulation with **$save** and **$restart**

422 © Cadence Design Systems, Inc. All rights reserved.

You can now use system tasks and system functions for stimulus generation and debug.

This module discussed a variety of system calls your testbench can make to support stimulus generation and debug.

Module Review

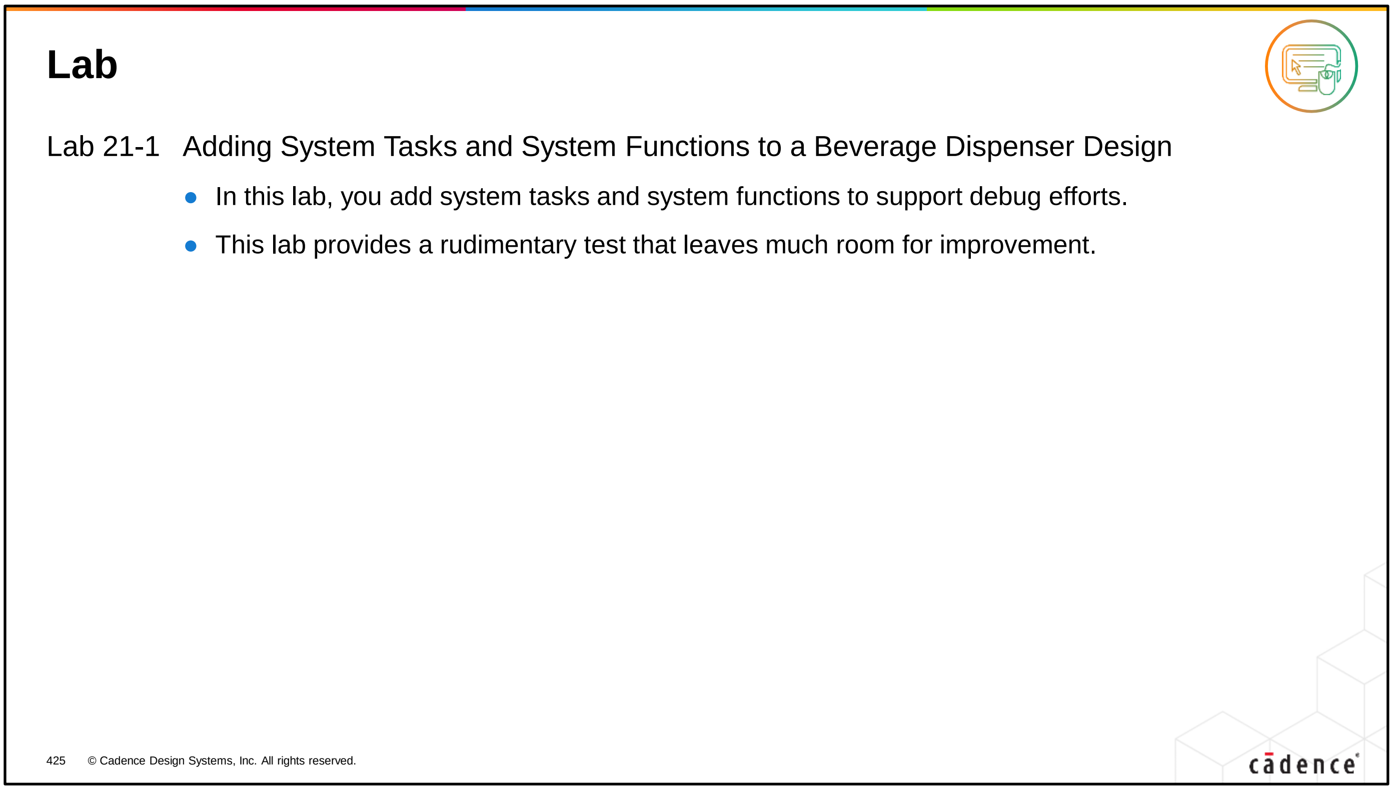
1. What output system task would you use to capture the "steady-state" settled values of a collection of nets and variables whenever any one or more of them transition?
2. Explain briefly the difference between **$stop** and **$finish**.
3. What is the decimal value of the integer "testnum" upon execution of
   `if ($value$plusargs("stimulus=test%d",testnum))`
   if the simulator invocation includes the **`+stimulus=test2`** option?
4. Which VCD system task would you use to dump value change data for the ports of an RTL ASIC you are developing, assuming you might want to use that data to test the gate-level synthesis netlist?

This page does not contain notes.

Module Review Solutions

1. What output system task would you use to capture the “steady-state” settled values of a collection of nets and variables whenever any one or more of them transition?
   - This is probably most easily done with the **$monitor** system task.
2. Explain briefly the difference between **$stop** and **$finish**.
   - The **$stop** system task interrupts simulation, from whence you or a script can continue it. The **$finish** system task irrevocably terminates simulation.
3. What is the decimal value of the integer “testnum” upon execution of:
   `if ($value$plusargs("stimulus=test%d",testnum))`
   if the simulator invocation includes the `+stimulus=test2` option?
   - The **$value$plusargs** system function interprets the subsequent “2” substring using the specified decimal formatter.
4. Which VCD system task would you use to dump value change data for the ports of an RTL ASIC you are developing, assuming you might want to use that data to test the gate-level synthesis netlist?
   - You would probably use **$dumpports** instead of **$dumpvars** because you need direction information and you do not need the values of internal signals.

This page does not contain notes.



Lab

Lab 21-1 Adding System Tasks and System Functions to a Beverage Dispenser Design

- In this lab, you add system tasks and system functions to support debug efforts.
- This lab provides a rudimentary test that leaves much room for improvement.

425 © Cadence Design Systems, Inc. All rights reserved.

cadence

Your objective for this lab is to add system tasks and system functions to support debug efforts.

The lab model is a beverage dispenser (drink machine). After resetting the machine, you load it with coins and cans. Then, as you insert nickels, dimes, and quarters, at 50 cents or above it dispenses a can and attempts to issue your change (if any). This drink machine has no coin return feature and inserted coins are not available later as change. If the EMPTY signal is true than you lose any coins you insert. If the USE_EXACT signal is true then you may find yourself short-changed. As the machine is currently defined, it does not attempt to issue lower-denomination coins if higher-denomination coins are not available.

The lab provides a rudimentary test that leaves much room for improvement. For this lab, you add system tasks and system functions to improve the test.