Using the New Verilog-2001 Standard
Part Two: Verifying Designs

by Stuart Sutherland
Sutherland HDL, Inc.
Portland, Oregon

Part 2-1

Using the New Verilog-2001 Standard
Part 2: Verifying Hardware
by Sutherland HDL, Inc., Portland, Oregon, © 2001

Part 2-2

copyright notice
©2001

All material in this presentation is copyrighted by Sutherland HDL, Inc., Portland, Oregon. All rights reserved. No material from this presentation may be duplicated or transmitted by any means or in any form without the express written permission of Sutherland HDL, Inc.

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

phone: (503) 692-0898
fax: (503) 692-1512
e-mail: info@sutherland-hdl.com
web: www.sutherland-hdl.com

Verilog® is a registered trademark of Cadence Design Systems, San Jose, CA

Part 2-1
About Stuart Sutherland and Sutherland HDL, Inc.

- Sutherland HDL, Inc. (founded 1992)
  - Provides expert Verilog HDL and PLI design services
  - Provides Verilog HDL and PLI Training
  - Located near Portland Oregon, World-wide services
- Mr. Stuart Sutherland
  - Over 13 years experience with Verilog
  - Worked as a design engineer on military flight simulators
  - Senior Applications Engineer for Gateway Design Automation, the founding company of Verilog
  - Author of the popular “Verilog HDL Quick Reference Guide” and “The Verilog PLI Handbook”
  - Involved in the IEEE 1364 Verilog standardization

Seminar Objectives

- The focus of this seminar is on understanding what is new in the Verilog-2001 standard
  - An overview of the Verilog HDL
  - Details on the major enhancements in Verilog-2001
  - Ideas on how you can use these enhancements, today
- Assumptions:
  - You have a background in hardware engineering
  - You are at least familiar with using Verilog-1995
Using the New Verilog-2001 Standard
Part 2: Verifying Hardware
by Sutherland HDL, Inc., Portland, Oregon, © 2001

Seminar Flow
Part 1 covers Verilog-2001 enhancements that primarily affect modeling hardware
- ANSI C style port lists
- Sensitivity list enhancements
- Model attributes
- Signed data types and signed arithmetic
- Multidimensional arrays
Part 2 covers Verilog-2001 enhancements that primarily affect verifying hardware
- New compiler directives
- Enhanced File I/O
- Re-entrant tasks and recursive functions
- Generate blocks
- Configuration blocks
- Deep submicron timing accuracy enhancements

Verilog-2001 Update
- The IEEE Std. 1364-2001 Verilog standard is official
  - Work on the standard was finished in March, 2000
  - IEEE balloting on the standard was completed in July, 2000
  - Clarifications to the standard as a result of ballot comments were approved in December, 2000
  - The IEEE officially ratified the standard in March, 2001
Why a New Standard?

- Add enhancements to Verilog
  - Design methodologies are evolving
    - System level design, intellectual property models, design re-use, very deep submicron, etc.
  - Cliff Cummings’ “Top Five Enhancement Requests” from a survey at the HDLCon 1996 conference

- Clarify ambiguities in Verilog 1364-1995
  - Verilog-2001 more clearly defines Verilog syntax and semantics

Goals for Verilog-2001

- Enhance Verilog for
  - Higher level, abstract system level modeling
  - Intellectual Property (IP) modeling
  - Greater timing accuracy for very deep submicron

- Make Verilog even easier to use
  - Eliminate redundancies in declarations
  - Simplify syntax of some verbose constructs

- Correct errata and ambiguities

- Maintain backward compatibility

- Ensure that EDA vendors will implement all enhancements!
Overview of HDL Enhancements

- 30+ major enhancements were added to the Verilog HDL
  - Brief description and examples
  - New reserved words

- Errata and clarifications
  - Dozens of corrections were made to the 1364-1995 standard
  - Do not affect Verilog users
  - Very important to Verilog tool implementors
  - *Not listed in this paper* — refer to the 1364-2001 Verilog Language Reference Manual (LRM)

Support For Verilog-2001

- Several simulator and synthesis companies are working on adding support for the Verilog-2001 enhancements

- Simulators:
  - Model Technology ModelSim — currently supports most new features
  - Co-Design SystemSim — currently supports most new features
  - Synopsys VCS — planned Q3-2001 support for several new features
  - Cadence NC-Verilog and Verilog-XL — no announced release date

- Synthesis:
  - Synopsys Presto (replaces DC compiler) — currently supports a synthesizable subset of Verilog-2001 enhancements
  - Cadence BuildGates — no announced release date
  - Exemplar Leonardo Spectrum — no announced release date

Information last updated July, 2001
Quick Review: Modeling a Test Bench

- The Verilog HDL is used to model a simulation test bench
  - The test bench is a module, which contains:
    - An instance of the top level of the design
    - Procedures to describe the input stimulus
    - Procedures to describe output verification

Quick Review: Verilog HDL Simulation Commands

- The Verilog HDL includes compiler directives and system tasks to control the simulation of Verilog models
  - `#<directive>` compiler directives
    - Executed prior to simulation time zero
    - Instructions to simulators on how to compile models
    - Always start with a `#` accent grave (the "back tic")
  - `$<task>` system tasks
    - Executed during simulation (i.e.: to display values)
    - Used as programming statements
    - Always start with a `$` dollar sign

This chapter only lists some of these simulator commands; Others are presented later in the course
Verilog-2001 Adds Enhanced Conditional Compilation

- Verilog-1995 provides compiler directives that allow Verilog source to be conditional compiled or excluded

```
ifdef <macro_name>
    <verilog_source_code>
else
    <verilog_source_code>
endif
```

- Verilog-2001 adds new directives `ifndef and `elsif for more extensive conditional compilation control

```
...`ifdef RTL_TEST
    alu_rtl ul (...);
elsif GATE
    alu_gate ul (...);
else
    initial $display("ERROR: neither RTL or GATE model instantiated");
endif
```

Verilog-2001 Adds File and Line Compiler Directives

- Verilog-2001 adds file and line compiler directives
  - New directives: `file and `line
  - Document the original location of Verilog source code
    - Verilog tools often include file name and line number information in error and warning messages
    - If a pre-process utility program modifies the Verilog source code, the original file and line information could be lost if not preserved by the `file and `line directives
Quick Review: Text Output System Tasks

- `$display` — displays a formatted message when executed

```verilog
always @(posedge clock)
  $display("At %d: a=%b b=%b sum=%b", $time, a, b, sum);
```

- `$strobe` — similar to `$display`, but postpones execution until the end of the current simulation time step

- `$write` — similar to `$display`, but does not add a new line

- `$monitor` — displays a formatted message when a signal changes value (a background task that is only invoked once!)

```verilog
initial
  $monitor("At %d: a=%b b=%b sum=%b", $time, a, b, sum);
```

Quick Review: Opening and Writing to Files

- `$fopen` opens a disk file for writing

  - Returns an integer that points to the open file
  - `$fmonitor`, `$fdisplay`, `$fstrobe` and `$fwrite` print to files
  - Up to 30 other files may be opened
  - `$fclose` will close a file and free the file pointer

```verilog
integer f1;
initial
begin
  f1 = $fopen("my_chip_outputs");
  $fmonitor(f1, "time=%t out_bus=%h", $realtime, out_bus);
end
```

Note: In Verilog-1995, a file is always opened as a new file
Quick Review: Writing to Multiple Files

- $fopen returns a 32-bit “multi-channel descriptor” (mcd)
- Each mcd has a single bit set
- Bit zero is reserved
- Represents standard out and the simulation log file
- Multiple files may be written to by OR-ing mcd's together

```verilog
integer f1, f2;
initial
begin
    f1 = $fopen("output.dat");
    f2 = $fopen("errors.dat");
    // check for errors on output
    // write error messages to all files and standard out
    forever @(out_bus)
        if (~out_bus === 1'bx) // xor all bits of bus to compare
            $fstrobe(f1|f2|1,"OUT_BUS ERROR AT %t", %realtime);
end
```

Quick Review: Reading Files

- Design Verification often needs to read from files, for example
  - To load patterns into RAM and ROM models
  - To read test vector files for stimulus
- Verilog has commands to read pattern files into memories
  - $readmemb reads ASCII files with binary patterns
  - $readmemh reads ASCII files with hexadecimal patterns
- To read any other file type requires customizing Verilog simulators using the Programming Language Interface (PLI)
  - Part of the IEEE Verilog standard
  - Provides access to the C file I/O routines
  - Very powerful, but requires writing file readers in C
Verilog-2001 Adds Enhanced File I/O System Tasks

- Verilog-2001 adds the ability to open up to $2^{30}$ files
  - Uses a “file descriptor” (fd) that represents a single file
  - Sets bit 31 and 1 or more additional bits
  - Cannot be OR-ed to represent multiple files
- Adds an optional \texttt{type} argument to $\texttt{fopen}$ to indicate if the file is opened for reading, writing, update (read/write), append, etc.

```verilog
cmd = $fopen("file_name"); // opens an mcd file for writing, only
fd = $fopen("file_name", <type>); // opens fd file for reading/writing
```

- Verilog-2001 adds several system task that can both read from and write to fd files, in ASCII or binary
  - $\texttt{ferror}$, $\texttt{fgetc}$, $\texttt{fgets}$, $\texttt{fflush}$, $\texttt{fread}$, $\texttt{fscanf}$, $\texttt{fseek}$, $\texttt{fsscanf}$, $\texttt{ftel}$, $\texttt{rewind}$, $\texttt{sformat}$, $\texttt{swrite}$, $\texttt{swriteb}$, $\texttt{swriteh}$, $\texttt{swriteo}$, $\texttt{ungetc}$

Quick Review: User Defined Invocation Options

- Verilog allows users to create new simulation invocation options
- The $\texttt{test$plusargs}$ system function checks to see if a “plus” option was used when simulation was invoked

```verilog
example invocation: verilog test.v chip.v +test2
```

```verilog
initial
begin
  if ($test$plusargs("test1"))
    $readmemh("test1.dat", vectors);
  else if ($test$plusargs("test2"))
    $readmemh("test2.dat", vectors);
  else if ($test$plusargs("test3"))
    $readmemh("test3.dat", vectors);
  else
    $display("Error: no test option specified");
end
```
Using the New Verilog-2001 Standard
Part 2: Verifying Hardware
by Sutherland HDL, Inc., Portland, Oregon, © 2001

Verilog-2001 Adds Enhanced Invocation Option Tests

- Verilog-2001 adds the ability to read arguments of invocation options, as well as doing a true/false test
- Returns true if option exists, and retrieves the value of any text after the option tested up to a white space, in the format specified

```
$value$plusargs("option%<format>", <variable>);
```

- %b read as binary value
- %o read as octal value
- %d read as decimal value
- %h read as hex value
- %e read as real value, exponential format
- %f read as real value, decimal format
- %g read as real value, shortest format
- %s read as character string

Example invocation: `verilog test.v chip.v +testfile=test2.dat`

```
reg [1023:0] file_string;
initial
if ($value$plusargs("testfile=%s", file_string))
    $readmemh(file_string, vectors);
else
    $display("Error: no test file option specified");
```

Quick Review: Generating Random Values

- The `$random` system function returns a random value
- Returns a 32-bit signed integer value

```
parameter seed = 1;
initial
for (i=0; i<=`num_tests; i=i+1)
    @(posedge test_clk)
    vector = $random(seed);
```

- Specifying a seed is optional
- Given the same seed value, the same random sequence will be generated each simulation

In Verilog-1995, the random number generator was not standardized, making it impossible to compare results from two different simulators
Verilog-2001 Adds a Standard Random Number Generator

- Verilog-2001 defines the C source code for the generator used by $random
- All simulators can generate the same random number sequence when given the same seed value
- Simulation results from different simulators can be compared
- New products do not need to re-invent number generators
- Uses the random number generator from Verilog-XL

Quick Review: Tasks and Functions in a Test Bench

- Verilog tasks and functions allow structured programming techniques to be used in the test bench
- A main control routine invokes various tests
- Tests can be written once and called many times

```
initial //Main test control
begin
  start_test("test1.data");
  verify;
  start_test("test2.data");
  verify;
  ...

end

task start_test;
  input [255:0] file_name;
  begin
    $readmemh(file_name, dsp.ram.core);
    sys_reset;
    end
endtask

task verify;
  ...
endtask

task sys_reset;
  ...
endtask
```
Quick Review: Verilog Tasks

- A **task** is a special type of procedure
  - Defined in the module in which it is used
  - Called from another procedure, like a subroutine
  - May delay the execution of statements
  - May have any number of outputs, inputs, and inouts

```verilog
module my_chip (...);
  ...
  always @(posedge clock)
  begin
    SendData(data, dout);
    ...
  end
  task SendData;
  input [31:0] data_bus;
  output [31:0] data_out;
  begin
    #3.2 data_out = data_bus;
  end
  endtask
endmodule
```

Arguments are passed in and out of the task in the order of declaration within the task.

A task can take time to execute

Quick Review: Verilog Functions

- A **function** is a special type of procedure
  - Defined in the module in which it is used
  - Called from any place a value can be used
  - Returns a scalar, vector or real-number value
  - Must execute in zero time
  - May only have inputs
  - Must have at least one input

```verilog
module my_chip (...);
  ...
  always @(posedge clock)
  begin
    result = Adder(a, b);
    ...
  end
  function [31:0] Adder;
  input [31:0] in1, in2;
  begin
    Adder = in1 + in2;
  end
  endfunction
endmodule
```

Arguments are passed in to the function in the order of declaration within the function.

The value assigned to the function name is the return value of the function.
Verilog-2001 Adds ANSI C Style Task/Function Declarations

- Using the declaration order to pass values is confusing
  - Different syntax than module declarations
  - Different syntax than C
- Verilog-2001 allows tasks and functions to define an argument list
  - Documents the order that values will be passed
  - More consistent with module declarations and ANSI C

```verilog
module my_chip (...);
  ...
  always @ (posedge clock)
  begin
    SendData(data, dout);
    ...
  end
  task SendData (input [31:0] data_bus,
                 output [31:0] data_out);
  begin
    #3.2 data_out = data_bus;
    end
  endtask
endmodule
```

Arguments are passed in and out of the task in the order of the argument list.

Verilog-2001 Adds Signed Functions

- With Verilog-1995, functions could return:
  - A 1-bit value
    ```verilog
    function f1 (input [63:0] a, b);
    ...
    ```
  - An unsigned vector of any vector width
    ```verilog
    function [7:0] f2 (input [63:0] a, b);
    ```
  - A signed 32-bit integer
    ```verilog
    function integer f3 (input [63:0] a, b);
    ```
  - A double-precision real value
    ```verilog
    function real f4 (input [63:0] a, b);
    ```
  - Verilog-2001 adds the ability to have a signed return of any vector size
    ```verilog
    function signed [63:0] f5 (input [63:0] a, b);
    ```

"signed" was a reserved word in the Verilog-1995 standard, but was not used.
Verilog-2001 Adds Constant Functions

- In Verilog-1995, vector widths had to be declared using constant expressions
- Verilog-2001 adds constant functions
  - Same syntax as standard Verilog functions
  - Limited to statements that can be evaluated at compile time
  - Can be called anywhere a constant expression is required
- Provides for more scalable, re-usable models

Example on next page

A Constant Function Example

```verilog
module ram (...);
  parameter RAM_SIZE = 1024;
  parameter ADDRESS = 12;
  input [ADDRESS-1:0] address_bus;

module ram (...);
  parameter RAM_SIZE = 1024;
  input [clogb2(RAM_SIZE)-1:0] address_bus;
  ...

function integer clogb2 (input integer depth);
  begin
    for(clogb2=0; depth>0; clogb2=clogb2+1)
      depth = depth >> 1;
  end
endfunction
  ...
```
Verilog-2001 Adds Re-entrant Tasks and Recursive Functions

- In Verilog-1995, tasks and functions use static storage
  - If a task is called while a concurrent call to the task is still executing, the second call will destroy the storage of the first call
  - If a function calls itself, the recursive call will overwrite the storage of its parent
- Verilog-2001 adds **automatic** tasks and functions
  - Each call to the task/function allocates unique storage
  - Concurrent task calls will not interfere with each other
  - Recursive calls to a function are stacked

```verilog
function automatic [63:0] factorial;
  input [31:0] n;
  if (n == 1)
    factorial = 1;
  else
    factorial = n * factorial(n-1);
endfunction
```

“automatic” is a new reserved word in Verilog-2001

Quick Review: Structural Modules

- Structural models are represented as a **netlist**
  - A netlist is a list of models and the interconnecting nets

```verilog
module chip (q1, q0, a, clk, rst);
  output q1, q1b, q0;
  input [3:0] a;
  input clk, rst;
  reg4  reg4 ();
  and2  and2 ();
  or2   or2 ();
  dff   dff1 ();
  dff   dff2 ();
endmodule
```

This is the chip.dff1 instance

All instances execute in parallel
(events produced by each instance are scheduled for sequential execution by a Verilog simulator)
Quick Review: Delays and Module Instances

- A module instance cannot be specified with delays
- Delays are specified inside component modules, not on the instance

```verilog
module dff (q, qb, ck, d, rst, pre);
output q, qb;
input clk, d, rst, pre;
...  
nand #2.15 p1 (q, n1, n2, rst);
...  
endmodule
```

no delays on the module instance

```verilog
module register (r, d, ld, clr);
output [31:0] r;
input [31:0] d;
input ld, clr;
...  
pullup (nc);
dff d0 (r[0], ,ld,d[0],clr,nc);
dff d1 (r[1], ,ld,d[1],clr,nc);
...  
endmodule
```

delays can only be specified on a primitive instance

Quick Review: Module Parameters

- Parameters are **run-time** constants
- Parameters can store:
  - integer numbers
  - real numbers
  - ASCII text strings
- `min:typ:max` delay expressions

```verilog
module RAM (data, address, ...);
parameter WORD = 8;  //default word width
parameter ADDR = 10;  //default address width
parameter SIZE = 1024;  //default RAM size
inout [WORD-1:0] data;
inout [ADDR-1:0] address;
reg [WORD-1:0] core [0:SIZE-1];
...  
endmodule
```

Part 2-33

Part 2-34
Verilog-2001 Adds Sized Parameter Constants

- A Verilog constant is a parameter
  parameter <identifier> = <value>;
  parameter [msb:lsb] <identifier> = <value>;
- If no size is specified, the parameter defaults to the size of the expression on the right-hand side

```verilog
module test;
    parameter CYCLE = 10;
    ...
    always #(CYCLE/2)
        clock = ~ clock
module state_machine(...);
    ...
    parameter [1:0] GO = 2'b01,
                        STOP = 2'b10;
    ...
    if (state == GO)
        //do something
    else if (state == STOP)
        //do something else
```

Most Verilog simulators and synthesis tools already supported sized parameters as a de facto standard.

Quick Review: Explicit Parameter Redefinition

- Parameters may be redefined for each instance of a module
- Redefinition occurs before simulation time 0
- Parameters become constant after loading is completed
- The `defparam` keyword explicitly redefines parameters
  - Uses the hierarchical name of the parameter

```verilog
module chip (...);
    ...
    RAM u1 (data1, ...);
    RAM u2 (data2, ...);
    defparam u2.SIZE = 256;
    defparam u2.ADDR = 8;
    defparam u2.WORD = 4;
    RAM u3 (data3, ...);
    defparam u3.WORD = 64;
    ...
module RAM (data, address, ...);
    parameter WORD = 8;
    parameter ADDR = 10;
    parameter SIZE = 1024;
    inout [WORD-1:0] data;
    input [ADDR-1:0] address;
    reg [WORD-1:0] core [0:SIZE-1];
    ...
```

The relative hierarchical name of the parameter to be redefined.
Quick Review: Implicit In-line Parameter Redefinition

- Module instantiation and parameter redefinition can be combined into a single statement
- The # token may be used as part of the module instantiation to implicitly redefine the module parameters
- Parameters must be redefined in the same order in which the parameters are declared

```
module chip ( . . . );
...
RAM u1 (data1, ...);
RAM u2 #(4, 8, 256) (...);
RAM u3 #(64) (...);
...
```

Parameter values must be passed in the order defined; a parameter cannot be skipped.

Verilog-2001 Adds Explicit In-line Parameter Passing

- Verilog-2001 adds the ability to explicitly name parameters when passing parameter values
- Provides better self-documenting code
- Parameter values can be passed in any order

```
module my_chip (...);
...
RAM #(8, 1023) ram2 (...);
endmodule
```

Verilog-1995         Verilog-2001

```
module ram (...);
parameter WIDTH = 8;
parameter SIZE = 256;
...
endmodule
```

```
module my_chip (...);
...
RAM #(8, .SIZE(1023)) ram2 (...);
endmodule
```
Verilog-2001 Adds
Fixed Constants

- Verilog-2001 adds a fixed constant, called a `localparam`

  \[
  \text{localparam}<\text{identifier}> = <\text{value}>; \\
  \text{localparam}[\text{msb} : \text{lsb}]<\text{identifier}> = <\text{value}>; \\
  \]

- If no size is specified, the parameter defaults to the size of the expression on the right-hand side
- A localparam cannot be redefined using defparam or SDF

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

Quick Review:
Multiple Module Instances

- The original Verilog HDL provided 2 ways to specify multiple instances of the same module

```verilog
module Reg4 (out, in, clk, rst);
  output [3:0] out;
  input [3:0] in;
  input clk, rst;
  dff i0 (out[0], in[0], clk, rst);
  dff i1 (out[1], in[1], clk, rst);
  dff i2 (out[2], in[2], clk, rst);
  dff i3 (out[3], in[3], clk, rst);
endmodule
```

- Verilog-1995 added a more simple form, `arrays of instances`

```verilog
module Reg4 (out, in, clk, rst);
  output [3:0] out;
  input [3:0] in;
  input clk, rst;
  dff i[3:0] (out, in, clk, rst);
endmodule
```
Verilog-2001 Adds Verilog Generate Blocks

- Verilog-2001 adds true generate capability
- Use for loops to generate any number of instances of:
  - Modules, primitives, procedures, continuous assignments, tasks, functions, variables, nets
- Use if-else and case decisions to control what instances are generated
- Provides greater control than the VHDL generate
- New reserved words added:
  - generate, endgenerate, genvar, localparam

Example on next page

Verilog Generate Example

```verilog
module multiplier (a, b, product);
    parameter a_width = 8, b_width = 8;
    localparam product_width = a_width + b_width;
    input [a_width-1:0] a;
    input [b_width-1:0] b;
    output [product_width-1:0] product;
    generate
        if ((a_width < 8) || (b_width < 8))
            CLA_multiplier #(a_width, b_width) u1 (a, b, product);
        else
            WALLACE_multiplier #(a_width, b_width) u1 (a, b, product);
    endgenerate
endmodule
```
Quick Review: Verilog Models After Synthesis

- RTL models are used before synthesis
- Structural models (netlists) are generated by synthesis
- Component models are created by library developers

Quick Review: Accurate Model Libraries

- Each component in a library is a separate Verilog module
- Each component model contains:
  - **Functionality**
    - How input values affect output values
  - **Technology** (CMOS, TTL, ECL, etc.)
    - How process technologies affect the device behavior
  - **Timing**
    - The time it takes input changes to propagate to the outputs
    - Restrictions on input changes in order for the models to function properly
Quick Review:
Specify Blocks

- Specify blocks specify timing information about a module
  - Must be inside the module boundary
  - May be defined anywhere within the module
- Specify blocks contain:
  - Pin-to-pin path delays for propagation times from module inputs to module outputs
  - Timing constraint checks such as setup and hold times
  - specparam constants to store data about the model

Verilog-2001 Adds
On-detect Pulse Error Propagation

- Verilog-1995 has on-event pulse error propagation
  - A pulse is a glitch on the inputs of a module path that is less than the delay of the path
  - An input pulse propagates to a path output as an X, with the same delay as if a valid input change had propagated to the output
- Verilog-2001 adds on-detect pulse error propagation
  - As soon as an input pulse is detected, a logic X is propagated to a path output, without the path delay
  - New reserved words added:
    - pulsestyle_onevent, pulsestyle_ondetect

```verilog
specify
  (in => out) = 3;
  $setup(in, posedge clk, 1.2);
  specparam drive = 0.4; specparam load = 1.8;
endspecify
```
Verilog-2001 Adds Negative Pulse Detection

- Due to different rising-transition and falling-transition delays, it is possible for the trailing edge of a glitch to propagate before the leading edge has propagated
- In Verilog-1995, a negative pulse is cancelled
- Verilog-2001 adds negative pulse detections
  - Negative pulse detection will propagate a logic X for the duration of the negative pulse
  - New reserved words added:
    - `showcancelled`, `noshowcancelled`

Quick Review: Input Timing Constraints

- Input timing constraints can be monitored using system tasks
  ```verilog
  $setup (data_event, reference_event, setup_limit, notifier);
  $hold (reference_event, data_event, hold_limit, notifier);
  $setuphold (reference_event, data_event, setup_limit, hold_limit, notifier);
  $skew (reference_event, data_event, skew_limit, notifier);
  $recovery (reference_event, data_event, recovery_limit, notifier);
  $period (reference_event, data_event, period_limit, notifier);
  $width (reference_event, width_limit, max_threshold, notifier);
  ```
- **Reference event** is the input that establishes a reference point for changes on the data event
- **Data event** is the input that is monitored for changes
- **Limit** is the timing required between reference and data
- **Notifier** (optional) is a scalar reg that is toggled on violations
Verilog-2001 Adds New Timing Constraint Checks

- Verilog-2001 adds new timing constraint checks to more accurately model very deep submicron input constraints:
  - $removal
  - $recrem
  - $timeskew
  - $fullskew
  - Refer to the IEEE 1364-2001 Verilog standard for details on these tasks.

Verilog-2001 Adds Negative Timing Constraints

- Verilog-2001 adds the ability to specify negative values for:
  - $setuphold setup and hold times
    - Adds new, optional arguments to the Verilog-1995 $setuphold task
  - $recrem recovery and removal times
    - A new timing check task in Verilog-2001
Verilog-2001 Adds Enhanced SDF support

- Verilog-2001 defines:
  - How timing objects in SDF map to objects in Verilog
  - Based on the latest SDF standard, IEEE 1497-1999

- Verilog-2001 adds a standard $sdf_annotate system task
  - Already a de-facto standard in all simulators

- Verilog-2001 changes the syntax of the specparam constant
  - Can now be declared at the module level as well as within a specify block (this change was required to support SDF labels)

Quick Review: Managing Verilog Designs

- Designs are often made up of many modules
  - Typically, each module is in a separate file x a complete design might be made up of hundreds of files
  - In a top-down design flow, there will be RTL versions and synthesized gate-level versions of the same module

- Verilog-1995 and earlier generations of Verilog left design management up to software tools—managing the location and version of Verilog models was not part of the Verilog language

- Most Verilog simulators provide:
  - A -f invocation option to manage design files
  - -v and -y invocation options to manage design libraries
  - The `uselib compiler directive to manage model versions
Quick Review:  Using `uselib With Model Libraries

- Most Verilog simulators have a `uselib compiler directive to scan libraries
  - `uselib is **not** in the IEEE Verilog standard (1995 or 2001)
  - The syntax is similar to the -v and -y options
- With `uselib, library information can be specified within the Verilog source code instead of at the invocation command
- Different instances of a module can come from different libraries

```verilog
module system (...);
...
`uselib file=../moto_lib.v
    moto_asic i1 (...);
`uselib dir=../lsi500k.dir libext=.v
    lsi_asic i2 (...);
endmodule
```

Verilog-2001 Adds Configurations

- Verilog-2001 adds configuration blocks
  - All software tools will have a consistent method
  - The version for each module instance can be specified
    - Virtual libraries specified within Verilog source code
    - Physical file locations specified in a “map” file
- New reserved words added: `config, endconfig, design, instance, cell, use, liblist`

Configuration rules and an example are on the next 2 pages
Verilog Configuration Notes

- Verilog design hierarchy is modeled the same as always.
- Configurations specify which module source code should be used for each instance of a module.
  - With Verilog-1995, it is up to the simulator on how to specify which model version should be used for each instance (if the simulator can do it at all).
- The configuration block is specified outside of all modules.
  - Can be in the same file as the Verilog source code.
  - Can be in a separate file.
  - Verilog model source code does not need to be modified in order to change the design configuration!
- A separate file maps logical library names to physical file locations.
  - Verilog source code does not need to be modified when a design is moved to a different physical source location!

Verilog Configuration Example

Verilog Design

```
module test;
  ...
  myChip dut (...);
  ...
endmodule

module myChip(...);
  ...
  adder a1 (...);
  adder a2 (...);
  ...
endmodule
```

Configuration Block (part of Verilog source code)

```
/* define a name for this configuration */
config cfg4

/* specify where to find top level modules */
design rtlLib.test

/* set the default search order for finding instantiated modules */
default liblist rtlLib gateLib;

/* explicitly specify which library to use for the following module instance */
instance test.dut.a2 liblist gateLib;
endconfig
```

Library Map File

```
/* location of RTL models (current directory) */
library rtlLib "./*.v";

/* Location of synthesized models */
library gateLib "./synth_out/*.v";
```
Verilog-2001 Adds Enhanced PLA Modeling

- Verilog-2001 extends the capability of the PLA system tasks ($async$or$array, $async$and$array, etc.)
  - In Verilog-1995, arguments had to be scalar
  - In Verilog-2001, arguments can be vectors

Verilog-2001 Adds Extended VCD Files

- Verilog-2001 adds new Value Change Dump (VCD) capabilities
  - Dump port change values
  - Dump strength level changes
  - Dump the time at which simulation finishes
  - New system tasks added:
    - $dumpports, $dumpportsall, $dumpportsoff, $dumpportson, $dumpportslimit and $dumpportsflush
Verilog-2001 Adds
PLI Enhancements

- Several enhancements added to the VPI library
  - Simulation control
    - Stop, finish, save, restart, etc.
  - Support for all the new Verilog-2001 HDL constructs
    - Multidimensional arrays, attributes, signed arithmetic, recursive functions, enhanced file I/O, etc.

- Maintenance updates to TF and ACC libraries
  - Corrected errata
  - Clarified ambiguities

The VPI Library Is The Future!

- All enhancements to the Verilog language are only supported in the VPI library of the PLI
  - The TF and ACC libraries ("PLI 1.0") are not being maintained, but not enhanced

The TF and ACC libraries are prehistoric!
You should be using the VPI PLI library…
Congratulations!

this concludes Part 2 of the workshop

“Using the New Verilog-2001 Standard”

Where can I learn even more?

- If you are a design engineer, we recommend:
  “Comprehensive Verilog HDL for Design Engineers”
  By Sutherland HDL, Inc. — www.sutherland-hdl.com
  Our 4-day workshop covers the entire Verilog language, including the new Verilog-2001 features, with lots of labs

- If you are a verification engineer, we recommend:
  “Advanced Verilog PLI Training”
  By Sutherland HDL, Inc. — www.sutherland-hdl.com
  A 4-day workshop on customizing and extending Verilog simulators by linking in C-based models, test routines, etc.

Additional Resources:
Verilog & Synthesis Books

- www.verilog-2001.com
  Information about the Verilog-2001 standard
  Verilog HDL Quick Reference Guide, Verilog-2001 version
  Stuart Sutherland—easy place for keywords, syntax, etc.
- IEEE Std 1364-2001
  IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language
- The Verilog Hardware Description Language
  Donald Thomas & Phil Moorby—good Verilog introduction
- The Verilog PLI Handbook
  Stuart Sutherland—using the PLI to extend the Verilog HDL
  check www.sutherland-hdl.com for a list of over 30 books
Using the New Verilog-2001 Standard
Part 2: Verifying Hardware
by Sutherland HDL, Inc., Portland, Oregon, © 2001

Additional Resources:
Verilog & Synthesis Resources

- [www.sutherland-hdl.com](http://www.sutherland-hdl.com) — Stuart Sutherland's web site — lots of Verilog web links
- [comp.lang.verilog](http://comp.lang.verilog) newsgroup — Great place to get quick answers to Verilog questions
- Other newsgroups: comp.lang.vhdl, comp.cad.synthesis, comp.arch.fpga
- [ESNUG](http://www.esnug.org) - E-mail Synopsys Users Group — John Cooley — jcooley@world.std.com
- [Verification Guild](http://www.verificationguild.org) — Verilog/VHDL verification newsletter — Janick Bergeron's newsletter on design verification — [www.janick.bergeron.com](http://www.janick.bergeron.com)

Additional Resources:
Verilog & Synthesis Conferences

- [HDLCon](http://www.hdlcon.org) — International HDL Conference — Formerly IVC/VIUF (International Verilog Conference / VHDL International Users Forum) — Good conference for information about Verilog/VHDL software tools — [www.hdlcon.org](http://www.hdlcon.org)
- [SNUG](http://www.synopsys.com) — Synopsys Users Group Conference — The best technical conference on Verilog/VHDL design methodologies and synthesis — [www.synopsys.com](http://www.synopsys.com) (click on the SNUG button)