The IEEE Verilog 1364-2001 Standard
What's New, and Why You Need It

Stuart Sutherland

Sutherland HDL, Inc.

(presented at HDLCon in March 2000 — minor updates made October, 2001)

Abstract

At the time of this conference, the proposed IEEE 1364-2000 Verilog standard is complete, and in the balloting process for final IEEE approval [update: official IEEE ratification was not completed until March 2001, making the official name IEEE 1364-2001, and the nickname Verilog-2001]. Verilog-2001 adds many significant enhancements to the Verilog language, which add greater support for configurable IP modeling, deep-submicron accuracy, and design management. Other enhancements make Verilog easier to use. These changes will affect everyone who uses the Verilog language, as well as those who implement Verilog software tools. This paper presents a summary of several of the enhancements in Verilog-2001.

1. History of the IEEE 1364 Verilog standard

The Verilog Hardware Description Language was first introduced in 1984, as a proprietary language from Gateway Design Automation. The original Verilog language was designed to be used with a single product, the Gateway Verilog-XL digital logic simulator.

In 1989, Gateway Design Automation was acquired by Cadence Design Systems. In 1990, Cadence released the Verilog Hardware Description Language and the Verilog Programming Language Interface (PLI) to the public domain. Open Verilog International (OVI) was formed to control the public domain Verilog, and to promote its usage. Cadence turned over to OVI the FrameMaker source files of the Cadence Verilog-XL user's manual. This document became OVI's Verilog 1.0 Reference Manual.

In 1993, OVI released its Verilog 2.0 Reference Manual, which contained a few enhancements to the Verilog language, such as array of instances. OVI then submitted a request to the IEEE to formally standardize Verilog 2.0. The IEEE formed a standards working group to create the standard, and, in 1995, IEEE 1364-1995 became the official Verilog standard.

It is important to note that for Verilog-1995, the IEEE standards working group did not consider any enhancements to the Verilog language. The goal was to standardize the Verilog language the way it was being used at that time. The IEEE working group also decided not to create an entirely new document for the IEEE 1364 standard. Instead, the OVI FrameMaker files were used to create the IEEE standard. Since the origin of the OVI manual was a user's manual, the IEEE 1364-1995 and IEEE 1364-2001 Verilog language reference manuals [1][2] are still organized somewhat like a user's guide.

2. Goals for IEEE 1364-2001 Verilog standard

Work on the IEEE 1364-2001 Verilog standard began in January 1997. Three major goals were established:

- Enhance the Verilog language to help with today's deep submicron and intellectual property modeling issues.
- Ensure that all enhancements were both useful and practical, and that simulator and synthesis vendors would implement Verilog-2001 in their products.
- Correct any errata or ambiguities in the IEEE 1364-1995 Verilog Language Reference Manual.

The Verilog-2001 standard working group was comprised of about 20 participants, representing a diversified mix of Verilog users, simulation vendors and synthesis vendors. The working group was divided into three task forces: The ASIC Task Force developed enhancements to meet the needs of very deep submicron timing accuracy. The Behavioral Task Force developed enhancements for Behavioral and RTL modeling. The PLI Task Force enhanced the Verilog Programming Language Interface to support changes from the other task forces, as well as adding new capabilities to the PLI.

3. Modeling enhancements

The 21 enhancements listed in this section give Verilog designers more capability for creating Verilog models. Many enhancements improve the ease and accuracy of writing synthesizable RTL models. Other enhancements allow models to be more scalable and re-usable. Only changes which add new functionality or syntax are listed here. Clarifications to Verilog-1995 are not listed.

This paper was first presented at the 9th Annual International HDL Conference and Exhibition, March 2000, Santa Clara, CA.. Minor updates and clarifications were made by the author in October, 2001.

3.1 Design management—Verilog configurations

The Verilog-1995 standard leaves design management to software tools, rather than making it part of the language. Each simulator vendor has devised ways to handle different versions of Verilog models, but these tool-specific methods are not portable across all Verilog software tools.

Verilog-2001 adds ***configuration blocks***, which allow the exact version and source location of each Verilog module to be specified as part of the Verilog language. For portability, virtual model libraries are used in configuration blocks, and separate ***library map files*** associate virtual libraries with physical locations. Configuration blocks are specified outside of module definitions. The names of configurations exist in the same name space as module names and primitive names. New keywords **config** and **endconfig** are reserved in Verilog-2001. Additional keywords are reserved for use within a configuration block: **design**, **instance**, **cell**, **use** and **liblist**.

The full syntax and usage of Verilog configuration blocks is beyond the scope of this paper. The following example illustrates a simple design configuration. The Verilog source code is typical; a test bench module contains an instance of the top-level of a design hierarchy, and the top level of the design includes instances of other modules.

```
module test;
  ...
  myChip dut (...); /* instance of design */
  ...
endmodule

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

The configuration block specifies the source code location of all, or specific, module instances. Because the configuration is specified outside of Verilog modules, the Verilog model source code does not need to be modified to reconfigure a design. In this configuration example, instance a1 of the adder will be compiled from the RTL library, and instance a2 from a specific gate-level library.

```
/* define a name for this configuration */
config cfg4
  /* specify where to find top level modules */
  design rtlLib.top
  /* 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
```

The configuration block uses virtual libraries to specify the location of the Verilog model sources. A library map file is used to associate the virtual library names with physical file locations. For example:

```
/* location of RTL models (current directory) */
library rtlLib ./*.v;
/* Location of synthesized models */
library gateLib ./synth_out/*.v;
```

3.2 Scalable models—Verilog generate

The Verilog-1995 standard has limitations on defining Verilog models that are scalable and easy to re-use in other designs. Verilog-1995 has the array of instances construct, which, though powerful, does not provide the flexibility needed for truly scalable, complex design structures.

Verilog-2001 adds ***generate loops***, which permit generating multiple instances of modules and primitives, as well as generating multiple occurrences of variables, nets, tasks, functions, continuous assignments, initial procedures, and always procedures. Generated declarations and instantiations can be conditionally created, using if–else decisions and case statements.

Four new keywords have been added in Verilog-2001: **generate**, **endgenerate**, **genvar** and **localparam**. The genvar keyword is a new data type, which stores positive integer values. It differs from other Verilog variables in that it can be assigned values and changed during compile or elaboration time. The index variable used in a generate loop must be declared as a genvar. A localparam is a constant that is similar to a parameter, but which cannot be directly changed using parameter redefinition. A generate block can also use certain Verilog programming statements to control what objects are generated. These are: **for** loops, **if**–**else** decisions and **case** decisions.

The following example illustrates using generate to create scalable module instances for a multiplier. If either of the multiplier's a_width or b_width parameters are less than 8, a CLA multiplier is instantiated. If a_width and b_width are 8 bits or more, a Wallace tree multiplier is instantiated.

```
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
```

The next example illustrates a multi-bit wide adder which uses a generate for-loop to instantiate both the primitive instances and the internal nets connecting the primitives. A re-definable parameter constant is used to set the width of the multi-bit adder and the number of instances generated.

```
module Nbit_adder (co, sum, a, b, ci);
  parameter SIZE = 4;
  output  [SIZE-1:0] sum;
  output             co;
  input   [SIZE-1:0] a, b;
  input              ci;
  wire    [SIZE:0]   c;

  genvar i;

  assign c[0] = ci;
  assign co = c[SIZE];

  generate
    for(i=0; i<SIZE; i=i+1)
      begin:addbit
        wire n1,n2,n3; //internal nets
        xor g1 (   n1, a[i], b[i]);
        xor g2 (sum[i],n1,   c[i]);
        and g3 (   n2, a[i], b[i]);
        and g4 (   n3, n1,   c[i]);
        or  g5 (c[i+1],n2,   n3);
      end
  endgenerate
endmodule
```

In the preceding example, each generated net will have a unique name, and each generated primitive instance will have a unique instance name. The name comprises the name of the block within the for-loop, plus the value of the genvar variable used as the loop index. The names of the generated n1 nets are:

```
addbit[0].n1
addbit[1].n1
addbit[2].n1
addbit[3].n1
```

The instance names generated for the first xor primitive are:

```
addbit[0].g1
addbit[1].g1
addbit[2].g1
addbit[3].g1
```

Note that these generated names use square brackets in the name. These are illegal characters in user-specified identifier names, but are permitted in generated names.

3.3 Constant functions

Verilog syntax requires that the declaration of vector widths and array sizes be based on literal values or constant expressions. For example:

```
parameter WIDTH = 8;
wire [WIDTH-1:0] data;
```

A limitation in the Verilog-1995 standard is that the constant expression can only be based on arithmetic operations. It is not possible to use programming statements to determine the value of a constant expression.

Verilog-2001 adds a new usage of Verilog functions, referred to as a ***constant function***. The definition of a constant function is the same as for any Verilog function. However, a constant function is restricted to only using constructs whose values can be determined at compile or elaboration time. Constant functions help to create re-usable models which can be scaled to different sizes.

The following example defines a function called clogb2 that returns an integer which has the value of the ceiling of the log base 2. This constant function is used to determine how wide a RAM address bus must be, based on the number of addresses in the RAM.

```
module ram (address_bus, write, select, data);
  parameter SIZE = 1024;
  input [clogb2(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
...
endmodule
```

3.4 Indexed vector part selects

In the Verilog-1995 standard, variable bit selects of a vector are permitted, but part-selects must be constant. Thus, it is illegal to use a variable to select a specific byte out of a word. The Verilog-2001 standard adds a new syntax, called ***indexed part selects***. With an indexed part select, a base expression, a width expression, and an offset direction are provided, in the form of:

```
[base_expr +: width_expr] //positive offset
[base_expr -: width_expr] //negative offset
```

The base expression can vary during simulation run-time. The width expression must be constant. The offset direction indicates if the width expression is added to or subtracted from the base expression. For example,:

```
reg [63:0] word;
reg [3:0] byte_num; //a value from 0 to 7
wire [7:0] byteN = word[byte_num*8 +: 8];
```

In the preceding example, if byte_num has a value of 4, then the value of word[39:32] is assigned to byteN. Bit 32 of the part select is derived from the base expression, and bit 39 from the positive offset and width expression.

3.5 Multidimensional arrays

The Verilog-1995 standard allows one-dimensional arrays of variables. Verilog-2001 extends this by permitting:

- Multi-dimensional arrays
- Arrays of both variable and net data types

This enhancement requires a change to both the syntax of array declarations, as well as the syntax for array indexing. Examples of declaring and indexing a one-dimensional and a three-dimensional array are shown below.

```
//1-dimensional array of 8-bit reg variables
//(allowed in Verilog-1995 and Verilog-2001)
reg [7:0] array1 [0:255];
wire [7:0] out1 = array1[address];

//3-dimensional array of 8-bit wire nets
//(new for Verilog-2001)
wire [7:0] array3 [0:255][0:255][0:15];
wire [7:0] out3 = array3[addr1][addr2][addr3];
```

3.6 Bit and part selects within arrays

The Verilog-1995 standard does not permit directly accessing a bit or part select of an array word. A full array word has to copied to a temporary variable, and the bit or part selected from the temporary variable. Verilog-2001 removes this restriction, and allows bit selects and part selects of array words to be directly accessed. For example:

```
//select the high-order byte of one word in a
//2-dimensional array of 32-bit reg variables
reg [31:0] array2 [0:255][0:15];
wire [7:0] out2 = array2[100][7][31:24];
```

3.7 Signed arithmetic extensions

For integer math operations, Verilog uses the data types of the operands to determine if signed or unsigned arithmetic should be performed. As a general rule (there are exceptions), if any operand in an expression is unsigned, unsigned operations are performed. To perform signed arithmetic, all operands in the expression must be signed. In Verilog-1995, the integer data type is signed, and the reg and net data types are unsigned. A limitation in Verilog-1995 is that the integer data type has a fixed vector size, which is 32-bits in most Verilog simulators. Thus, signed integer math in Verilog-1995 is limited to 32-bit vectors. The Verilog-2001 standard adds five enhancements to provide greater signed arithmetic capability:

- Reg and net data types can be declared as signed
- Function return values can be declared as signed
- Integer numbers in any radix can be declared as signed
- Operands can be converted from unsigned to signed
- Arithmetic shift operators have been added

The Verilog-1995 standard has a reserved keyword, **signed**, but this keyword was not used in Verilog-1995. Verilog-2001 uses this keyword to allow reg data types, net data types, ports and functions to be declared as signed types. Some example declarations are:

```
reg signed [63:0] data;
wire signed [7:0] vector;
input signed [31:0] a;
function signed [128:0] alu;
```

In Verilog-1995, a literal integer number with no radix specified is considered a signed value, but a literal integer with a radix specified is considered an unsigned value. Verilog-2001 adds an additional specifier, the letter '**s**', which can be combined with the radix specifier, to indicate that the literal number is a signed value.

```
16'hC501   //an unsigned 16-bit hex value
16'shC501  //a signed 16-bit hex value
```

In addition to being able to declare signed data types and values, Verilog-2001 adds two new system functions, **$signed** and **$unsigned**. These system functions are used to convert an unsigned value to signed, or vice-versa.

```
reg [63:0] a; //unsigned data type
always @(a) begin
  result1 = a / 2;         //unsigned arithmetic
  result2 = $signed(a) / 2;//signed arithmetic
end
```

One more signed arithmetic enhancement in Verilog-2001 is arithmetic shift operators, represented by >>> and <<< tokens. An arithmetic right-shift operation maintains the sign of a value, by filling with the sign-bit value as it shifts. For example, if the 8-bit signed variable D contained 8'b10100011, a logical right shift and an arithmetic right shift by 3 bits would yield the following:

```
D >> 3  //logical shift yields  8'b00010100
D >>> 3 //arithmetic shift yields 8'b11110100
```

3.8 Power operator

Verilog-2001 adds a power operator, represented by an ****** token. This operator preforms similar functionality as the C pow() function. It will return a real number if either operand is a real value, and an integer value if both operands are integer values. One practical application of the power operator is to calculate values such as 2^n. For example:

```
always @(posedge clock)
  result = base ** exponent;
```

3.9 Re-entrant tasks and recursive functions

Verilog-2001 adds a new keyword, **automatic**. This keyword can be used to declare an automatic task that is re-entrant. All task declarations within an automatic task are allocated dynamically for each concurrent task entry. A function can also be declared as automatic, which allows the function to be called recursively (declarations within the function will be allocated dynamically for each

recursive call). Declarations within an automatic task or function can not be accessed by hierarchical references.

A task or function that is declared without the automatic keyword behaves as Verilog-1995 tasks and functions, which are static. All declared items in a static task or function are statically allocated, and are shared by all calls to the task or function.

The following example illustrates a function which recursively calls itself in order to find the factorial (n!) of a 32-bit unsigned integer operand.

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

recursive call

3.10 Combinational logic sensitivity token

To properly model combinational logic using a Verilog always procedure, the sensitivity list must include all input signals used by that block of logic. In large, complex blocks of combinational logic, it is easy to inadvertently omit an input from the sensitivity list, which can lead to simulation and synthesis mismatches.

Verilog-2001 adds a new wild card token, **@***, which can be used to represent a combinational logic sensitivity list. The @* token indicates that the simulator or synthesis tool should automatically be sensitive changes on any values which are read in the following statement or statement group. In the following example, the @* token will cause the procedure to automatically be sensitive to changes on sel, a or b.

```
always @*  //combinational logic sensitivity
  if (sel)
    y = a;
  else
    y = b;
```

3.11 Comma-separated sensitivity lists

Verilog-2001 adds a second way to list signals in a sensitivity list, by separating the signal names with commas instead of the **or** keyword. The following two sensitivity lists are functionally identical:

```
always @(a or b or c or d or sel)

always @(a, b, c, d, sel)
```

The new, comma-separated sensitivity list does not add any new functionality. It does, however, make Verilog syntax more intuitive, and more consistent with other signal lists in Verilog.

3.12 Enhanced file I/O

Verilog-1995 has very limited file I/O capability built into the Verilog language. Instead, file operations are handled through the Verilog Programming Language Interface (PLI), which gives access to the file I/O libraries in the C language. Verilog-1995 file I/O also limits the number files it can open at the same time to, at most, 31.

Verilog-2001 adds several new system tasks and system functions, which provide extensive file I/O capability directly in the Verilog language, without having to create custom PLI applications. In addition, Verilog-2001 increases the limit of the number of files that can be open at the same time to 2^{30}. The new file I/O system tasks and system functions in Verilog-2001, listed alphabetically, are: **$ferror**, **$fgetc**, **$fgets**, **$fflush**, **$fread**, **$fscanf**, **$fseek**, **$fscanf**, **$ftel**, **$rewind** and **$ungetc**. Verilog-2001 also adds string versions of these commands, which allow creating formatted strings, or reading information from a string. These new system tasks are: **$sformat**, **$swrite**, **$swriteb**, **$swriteh**, **$swriteo** and **$sscanf**.

3.13 Automatic width extension beyond 32 bits

With Verilog-1995, assigning an unsized high-impedance value (e.g.: ‘bz) to a bus that is greater than 32 bits would only set the lower 32 bits to high-impedance. The upper bits would be set to 0. To set the entire bus to high-impedance requires explicitly specifying the number of high impedance bits. For example:

```
Verilog-1995:
  parameter WIDTH = 64;
  reg [WIDTH-1:0] data;
  data = 'bz;  //fills with 'h00000000zzzzzzzz
  data = 64'bz; //fills with 'hzzzzzzzzzzzzzzzz
```

The fill rules in Verilog-1995 make it difficult to write models that are easily scaled to new vector sizes. Redefinable parameters can be used to scale vector widths, but the Verilog source code must still be modified to alter the literal value widths used in assignment statements.

Verilog-2001 changes the rule for assignment expansion so that an unsized value of Z or X will automatically expand to fill the full width of the vector on the left-hand side of the assignment.

```
Verilog-2001:
  parameter WIDTH = 64;
  reg [WIDTH-1:0] data;
  data = 'bz;  //fills with 'hzzzzzzzzzzzzzzzz
```

3.14 In-line parameter passing by name

Verilog-1995 has two methods of redefining parameters within a module instance: *explicit redefinition* using defparam statements, and *in-line implicit redefinition* using the # token as part of the module instance. The latter

method is more concise, but because it redefines parameter by their declaration position, it is error-prone and is not self-documenting. The following example illustrates the two Verilog-1995 methods for parameter redefinition.

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

module my_chip (...);
  ...
  //Explicit parameter redefinition by name
  RAM ram1 (...);
  defparam ram1.SIZE = 1023;

  //Implicit parameter redefintion by position
  RAM #(8,1023) ram2 (...);
endmodule
```

Verilog-2001 adds a third method to redefine parameters, ***in-line explicit redefinition***. This new method allows in-line parameter values to be listed in any order, and document the parameters being redefined.

```
  //In-line explicit parameter redefintion
  RAM #(.SIZE(1023)) ram2 (...);
```

3.15 Combined port and data type declarations

Verilog requires that signals connected to the input or outputs of a module have two declarations: the direction of the port, and the data type of the signal. In Verilog-1995, these two declarations had to be done as two separate statements. Verilog-2001 adds a simpler syntax, by combining the declarations into one statement.

```
module mux8 (y, a, b, en);
  output reg   [7:0] y;
  input  wire  [7:0] a, b;
  input  wire        en;
```

3.16 ANSI-style input and output declarations

Verilog-1995 uses the older Kernighan and Ritchie C language syntax to declare module ports, where the order of the ports is defined within parentheses, and the declarations of the ports are listed after the parentheses. Verilog-1995 tasks and functions omit the parentheses list, and use the order of the input and output declarations to define the input/output order.

Verilog-2001 updates the syntax for declaring inputs and outputs of modules, tasks and functions to be more like the ANSI C language. That is, the declarations can be contained in the parentheses that show the order of inputs and outputs.

```
module mux8 (output reg   [7:0] y,
             input  wire  [7:0] a,
             input  wire  [7:0] b,
             input  wire        en );

function [63:0] alu (
    input [63:0]  a, b,
    input [7:0]   opcode );
```

3.17 Reg declaration initial assignments

Verilog-2001 adds the ability to initialize variables at the time they are declared, instead of requiring a separate initial procedure to initialize variables. The initial value assigned to the variable will take place within simulation time zero, just as if the value had been assigned within an initial procedure.

```
Verilog-1995:
  reg clock;
  initial
    clk = 0;

Verilog-2001:
  reg clock = 0;
```

3.18 "Register" changed to "variable"

Since the inception of Verilog in 1984, the term "register" has been used to describe the group of variable data types in the Verilog language. "Register" is not a keyword, it is simply a name for a class of data types, namely: reg, integer, time, real and realtime. The use of term "register" is often a source of confusion for new users of Verilog, who sometimes assume that the term implies a hardware register (flip-flops). The IEEE 1364-2001 Verilog Language Reference Manual replaces the term "register" with the more intuitive term "variable".

3.19 Enhanced conditional compilation

Verilog-1995 supports conditional compilation, using the `ifdef, `else and `endif compiler directives. Verilog-2001 adds more extensive conditional compilation control, with **`ifndef** and **`elsif** compiler directives.

3.20 File and line compiler directive

Verilog tools need to keep track of the line number and the file name of Verilog source code. This information can be used for error messages, and can be accessed by the Verilog PLI. If Verilog source is pre-processed by some other tool, however, the line and file information of the original source code can be lost. Verilog-2001 adds a **`line** compiler directive, which can be used to specify the original source code line number and file name. This allows the location in an original file to be maintained if another process modifies the source, such as by adding or removing lines of source text.

3.21 Attributes

The Verilog language was originally created as a hardware description language for digital simulation. As tools other than simulation have adopted Verilog as a source input,

there has been a need for these tools to be able add tool-specific information to the Verilog language. In Verilog-1995, there was no mechanism for adding tool-specific information, which led to non-standard methods, such as hiding synthesis commands in Verilog comments.

Verilog-2001 adds a mechanism for specifying properties about objects, statements and groups of statements in the HDL source. These properties are referred to as ***attributes***. Attributes may be used by various tools, including simulators, to control the operation or behavior of the tool. An attribute is contained within the tokens **(*** and ***)**. Attributes can be associated with all instances of an object, or with a specific instance of an object. Attributes can be assigned values, including strings, and attribute values can be re-defined for each instance of an object.

Verilog-2001 does not define any standard attributes. The names and values of attributes will be defined by tool vendors or other standards. An example of how a synthesis tool might use attributes is shown below:

```
(* parallel case *) case (1'b1) //1-hot FSM
  state[0]: ...
  state[1]: ...
  state[2]: ...
endcase
```

4. ASIC/FPGA accuracy enhancements

The original Verilog language was created in a time when 2 to 5 micron designs were common. As silicon technologies and design methodologies have changed, the Verilog language has evolved as well. Verilog-2001 continues this evolution, with enhancements specific for today's—and tomorrow's—deep submicron designs.

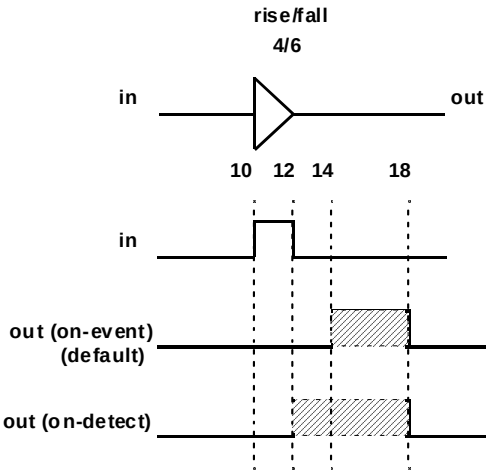
4.1 On-detect pulse error propagation

Verilog-1995 provides *on-event* pulse error propagation for pin-to-pin path delays. A pulse is a glitch on the inputs of a model path that is less than the delay of the path. With on-event propagation, the leading and trailing edge of an input pulse propagate to the path outputs as a logic X. The timing of the X on the output is the same as if the input changes had propagated to the output.

Verilog-2001 adds ***on-detect*** pulse error propagation. On-detect is a more pessimistic method of setting the output to an X when an input glitch occurs. As with on-event, on-detect changes the leading edge of the pulse into a transition to X, and the trailing edge to a transition from X, but the time of the leading edge is changed to occur immediately upon detection of the pulse.

On-event or on-detect can be explicitly specified using two new keywords within a Verilog specify block: **pulsestyle_onevent** and **pulsestyle_ondetect**. On-event pulse error propagation is the default pulse style.

```
specify
  pulsestyle_ondetect out;
  (in => out) = (4,6);
endspecify
```



4.2 Negative pulse detection

It is possible for an output logic X pulse to be calculated where the trailing edge of the pulse would occur prior to the leading edge of the pulse. In Verilog-1995, a negative pulse would be cancelled. Verilog-2001 provides a mechanism to have a logic X pulse propagate to the output to indicate that a negative pulse had occurred. Two more keywords are added to the Verilog specify block to explicitly enable or disable negative pulse propagation, **showcancelled** and **noshowcancelled**.

```
specify
  showcancelled out;
  (a => out) = (2,3);
  (b => out) = (4,5);
endspecify
```

4.3 New timing constraint checks

Verilog-2001 adds several new timing constraint checks, to more accurately model deep submicron timing. The new timing checks are: **$removal**, **$recrem**, **$timeskew** and **$fullskew**. A full description of the timing checks is beyond the scope of this paper. Refer to the IEEE 1364-2001 Verilog standard for details on these checks.

4.4 Negative timing constraints

Verilog-2001 changes the **$setuphold** timing constraint by adding four additional arguments. These new arguments allow accurate specification of negative setup or hold times. The setup and hold timing check values define a timing violation window with respect to a reference signal edge, during which the data must remain constant. A positive value for both setup and hold times indicates that

this violation window straddles the reference signal. A negative hold or setup time means the violation window is shifted to either before or after the reference edge. This can happen in a real device because of disparate internal device delays between the internal clock and data signal paths.

The new $setuphold arguments are added to the end of the Verilog-1995 $setuphold argument list. The new arguments are optional. If they are not specified, the syntax for $setuphold is the same as with Verilog-1995, thus maintaining backward compatibility.

The new **$recrem** timing check, which combines $recovery and $removal, can also accept negative values, using a syntax similar to $setuphold. Refer to the IEEE 1364-2001 Verilog standard for details on specifying negative timing constraints using these timing checks.

4.5 Enhanced SDF support

The IEEE 1364-2001 Verilog language reference manual adds a section which details how delays in an SDF file map to delays within the Verilog language. The discussion is based on the latest SDF standard, *IEEE Std 1497-1999* [3].

This latest SDF standard includes labels, which provide a means to annotate delays to Verilog procedures. In order to support SDF labels, one change has been made to the Verilog syntax. In Verilog-1995, specparam constants could only be declared within a specify block. Verilog-2001 allows specparam constants to be declared and used at the module level.

4.6 Extended VCD files

The Verilog-1995 standard defines a standard 4-state logic Value Change Dump (VCD) file format. The $dumpvars and related system tasks are used to create and control a VCD file.

Verilog-2001 adds extensions to the Value Change Dump (VCD) file format. These extensions add more detail on Verilog port changes, on net strength changes, and on the time at which simulation finished. In Verilog-2001, several new system tasks have been defined, which create and control an extended VCD file. These new system tasks are: **$dumpports**, **$dumpportsall**, **$dumpportsoff**, **$dumpportson**, **$dumpportslimit** and **$dumpportsflush**.

5. PLI enhancements

Verilog-2001 includes numerous updates to the Verilog Programming Language Interface portion of the Verilog standard. These changes fall into three primary groups:

- New features added to within the PLI
- Implementation of PLI support for all enhancements added to the Verilog language for Verilog-2001
- Clarifications to the Verilog-1995 PLI standard

The Verilog PLI standard includes three libraries of C functions, the TF, ACC and VPI libraries. The TF and ACC libraries are older versions of the Verilog PLI, and are maintained in the IEEE 1364 Verilog standard for backward compatibility. The VPI library is the most current version of the PLI standard, and offers many advantages over the older libraries.

The Verilog-2001 contains many clarifications and errata corrections to the definitions of the older TF and ACC libraries. However, no new features or capabilities have been added to the TF and ACC libraries. ***All enhancements to the Verilog PLI have been incorporated in the VPI library.*** These include support for the many new features in the Verilog language, as well as the addition of six new VPI routines: **vpi_control()**, **vpi_get_data()**, **vpi_put_data()**, **vpi_get_userdata()**, **vpi_put_userdata()** and **vpi_flush()**. Refer to the IEEE 1364-2001 Verilog standard for a full description of these new VPI routines.

6. References

1. *IEEE Std 1364-1995, IEEE Standard Hardware Description Language Based on the Verilog® Hardware Description Language*. The Institute of Electrical and Electronics Engineers, Inc. 345 East 47th Street, New York, NY 10017-2394, USA. ISBN 1-55937-727-5.
2. *IEEE Std p1364-2001, IEEE Standard Hardware Description Language Based on the Verilog® Hardware Description Language*. The Institute of Electrical and Electronics Engineers, Inc. 345 East 47th Street, New York, NY 10017-2394, USA. (ISBN not yet assigned).
3. *IEEE Std p1497-1999*, Standard for Standard Delay Format (SDF) for the Electronic Design Process. The Institute of Electrical and Electronics Engineers, Inc. 345 East 47th Street, New York, NY 10017-2394, USA. (ISBN not yet assigned).

7. Summary

The IEEE 1364-2001 Verilog standard is complete, and received official ratification by the IEEE in March, 2001. Verilog-2001 adds many important enhancements to the Verilog language, which provide powerful constructs for writing re-usable, scalable models, Intellectual Property modeling, and very deep submicron timing accuracy. Engineers who design with Verilog will receive significant benefit from these enhancements.

8. About the Author:

Stuart Sutherland is the founder and president of Sutherland HDL Inc., a company that specializes in Verilog training and design consulting. Mr. Sutherland is co-chairman of the IEEE 1364 PLI task force for the Verilog-2001 standard and editor of the PLI sections of the 1364 Verilog language reference manual. You can contact Mr. Sutherland at *stuart@sutherland-hdl.com*.