Getting the Most out of the New Verilog-2000 Standard

Stuart Sutherland
Sutherland HDL, Inc.

Don Mills
LCDM Engineering

stuart@sutherland.com
mills@lcdm-eng.com

ABSTRACT

Verilog-2000 (released as IEEE 1364-2001) adds many significant enhancements to the Verilog language, which add greater support for configurable IP modeling and deep-submicron accuracy, and development of 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 will review and highlight the main features added to the Verilog standard for the IEEE 1364-2001 update. The focus will be on new simulation and synthesis constructs. Where possible, status regarding Synopsys support for the new features will also be noted.

1.00 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 containing most, but not all, 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, and major enhancements to the Verilog PLI. 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 Gateway's Verilog-XL 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.

Originally scheduled for release in late 2000, the updated standard is commonly called Verilog-2000. Due to the balloting process, the approval of the standard actually will not be completed until spring 2001, at which time IEEE will designate the update name as IEEE 1364-2001. Both names are used interchangeably in this paper, the common name **Verilog-2000,** and the final official name **IEEE 1364-2001.**

2.00 Goals for Verilog-2000 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-2000 in their products.
- Correct any errata or ambiguities in the IEEE 1364-1995 Verilog Language Reference Manual.

The Verilog-2000 standard was submitted for ballot by IEEE members in June of 2000. The standard was approved by the ballot group by an overwhelming majority, approximately 90%. However, a number of comments received from the ballot group resulted in the IEEE standard

working group making several clarifications on the new enhancements in Verilog-2000. These changes were re-submitted to the ballot group in December of 2000, and were again approved by an overwhelming majority (92%). Final approval by the IEEE for the Verilog-2000 standard is expected to be received in March 2001.
The Verilog-2000 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.

3.00 Modeling enhancements

The 23 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. With the exception of the following paragraph, only changes which add new functionality or syntax are listed here. Verilog-2000 also contains many clarifications to Verilog-1995, which do not add new functionality. Notes are added to the sub-sections indicating Synopsys support with Presto and VCS at the time this paper was completed.

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”. This is a terminology change in Verilog reference documentation only, and does not affect any simulator or synthesis tool.

3.01 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-2000 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-2000. 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 how a simple Verilog design configuration might be used. 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";
```

Synopsys Support:

VCS 6.0	VCS 6.1	PRESTO
no	no	no

3.02 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-2000 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-2000: **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 can be 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 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 both of the a_width and b_width parameters are 8 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.

Synopsys Support:

VCS 6.0	VCS 6.1	PRESTO
no	no	no

3.03 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-2000 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. (This example uses another Verilog-2000 enhancement, a power operator.).

```
module ram (address, write, chip_select, data);
  parameter WIDTH = 8;
  parameter SIZE  = 256;
  localparam ADDRESS_SIZE = clogb2(SIZE);
  input [ADDRESS_SIZE-1:0] address;
  input write, chip_select;
  inout [WIDTH-1:0] data;

  reg [WIDTH-1:0] ram_data [0:SIZE-1];

  //define the clogb2 constant function
  function integer clogb2;
   input depth;
   integer i;
```

```
    begin
      clogb2 = 1;
      for (i = 0; 2**i < depth; i = i + 1)
        clogb2 = i + 1;
    end
  endfunction
  ...
endmodule
```

Synopsys Support:

VCS 6.0	VCS 6.1	PRESTO
no	no	no

3.04 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-2000 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.

Synopsys Support:

VCS 6.0	VCS 6.1	PRESTO
no	yes	yes

3.05 Multidimensional arrays

The Verilog-1995 standard allows 1-dimensional arrays of variables. Verilog-2000 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 1-dimensional and a 3-dimensional array are shown below.

```
//1-dimensional array of 8-bit reg variables
//(allowed in Verilog-1995 and Verilog-2000)
reg  [7:0] array1 [0:255];
wire [7:0] out1 = array1[address];

//3-dimensional array of 8-bit wire nets
//(new for Verilog-2000)
wire [7:0] array3 [0:255][0:255][0:15];
wire [7:0] out3 = array3[addr1][addr2][addr3];
```

Synopsys Support:

VCS 6.0	VCS 6.1	PRESTO
no	yes	yes

3.06 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 be copied to a temporary variable, and the bit or part selected from the temporary variable. Verilog-2000 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];
```

Synopsys Support:

VCS 6.0	VCS 6.1	PRESTO
no	no	yes

3.07 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. If either operand is unsigned, unsigned operations are performed. To perform signed arithmetic, both operands 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-2000 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-2000 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-2000 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-2000 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-2000 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 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
```

Synopsys Support:

VCS 6.0	VCS 6.1	PRESTO 2000.11	planned for PRESTO 2001.08
no	yes	no	yes

3.08 Power operator

Verilog-2000 adds a power operator, represented by an ****** token. This operator performs the same 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. The example for constant functions, in section 3.03, uses the power operator to determine the address vector width required to address a given memory size.

Synopsys Support:

	planned for	
VCS 6.0	VCS 6.1	PRESTO
no	yes	yes

3.09 Re-entrant tasks and recursive functions

Verilog-2000 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
```

Synopsys Support:

	planned for	
VCS 6.0	VCS 6.1	PRESTO
no	maybe	no

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-2000 adds a new wild card token, **@***, which represents a combinational logic sensitivity list. The @* token indicates that the simulator or synthesis tool should automatically be sensitive to any values used by the procedure in decisions or in expressions on the right-hand side of assignment statements. 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;
```

Synopsys Support:

	planned for	
VCS 6.0	VCS 6.1	PRESTO
no	yes	yes

3.11 Comma-separated sensitivity lists

Verilog-2000 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.

Synopsys Support:

	planned for	
VCS 6.0	VCS 6.1	PRESTO
no	yes	yes

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-2000 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-2000 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-2000, listed alphabetically, are: **$ferror**, **$fgetc**, **$fgets**, **$fflush**, **$fread**, **$fscanf**, **$fseek**, **$fsscanf**, **$ftel**, **$rewind**, **$sformat**, **$swrite**, **$swriteb**, **$swriteh**, **$swriteo,** and **$ungetc**. Note that Verilog-2000 will limit the number of files that can be opened using the original (default) $fopen syntax to, at most, 30 files, one less than the Verilog-1995 standard. This may pose a minor backward compatibility problem, but the IEEE standards group felt the new syntax for $fopen which can open an almost unlimited number of files was worth the trade-off.

Synopsys Support:

VCS 6.0	VCS 6.1	PRESTO
no	yes	no

Note that VCS 5.2 and earlier versions of VCS could implement much of the new file I/O system tasks and functions using the Verilog PLI. In fact, a Synopsys applications engineer has made sample file I/O PLI applications available on his personal web site for many years at www.chris.spear.net/pli .

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-2000 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-2000:
  parameter WIDTH = 64;
  reg [WIDTH-1:0] data;
  data = 'bz;        //fills with 'hzzzzzzzzzzzzzzzz
```

This Verilog-2000 enhancement that is not backward compatible with Verilog-1995. However, the IEEE standards group felt the Verilog-1995 behavior was a bug in the standard that needed to be corrected. It is expected that all existing models with greater than 32-bit busses have avoided this bug by explicitly specifying the vector sizes. Therefore, there should not be any compatibility problems with existing models.

Synopsys Support:

VCS 6.0	VCS 6.1	PRESTO
no	no	no

3.14 Explicit 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 parameters 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 redefinition by position
  RAM #(8,1023) ram2 (...);
endmodule
```

Verilog-2000 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 re-defined.

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

Synopsys Support:

VCS 6.0	VCS 6.1	PRESTO
yes	yes	yes

3.15 Sized Parameters

In the Verilog-1995 standard, the size of parameters is not specified. Instead, the parameter defaults to the size of the original value assigned to it. Verilog-1995 does not require or specify a syntax for sized parameter definitions, but most Verilog simulators support the syntax as a defacto standard. Verilog-2000 specifies syntax for sized parameters, but does not require it to be used. The following example illustrates the usage of sized parameter definition.

```
parameter [2:0]
  IDLE  = 3'd0,
  READ  = 3'd1,
  LOAD  = 3'd2,
  SYNC  = 3'd3,
  ERROR = 3'd4;
```

Synopsys Support:

VCS 6.0	planned for VCS 6.1	PRESTO
no	yes	yes

Some Synopsys tools have made use of sized parameters for years. Synopsys finite state machine (FSM) compiler has required that state vectors defined in a parameter list be made with sized parameters, as illustrated in the example above. The Verilog-2000 syntax for sized parameters is the same as that of Synopsys FSM.

3.16 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-2000 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;
```

Synopsys Support:

VCS 6.0	VCS 6.1	PRESTO 2000.11	planned for PRESTO 2001.08
yes	yes	no	yes

3.17 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-2000 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,
   input    [63:0]    b,
   input    [7:0]     opcode );
```

Synopsys Support:

VCS 6.0	VCS 6.1	PRESTO 2000.11	planned for PRESTO 2001.08
yes	yes	no	yes

3.18 Reg declaration initial assignments

Verilog-2000 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-2000:
 reg clock = 0;
```

Synopsys Support:

VCS 6.0	VCS 6.1	PRESTO
yes	yes	no

Presto does not support this construct, but this is not an issue, since this construct is not intended as a synthesis construct anyway. Its primary usage is for test benches and behavioral modeling.

3.19 Implicit nets with continuous assignments

Verilog-1995 will infer a net data type for an undeclared signal on the left-hand side of a continuous assignment only if the signal name is also connected to a port of that module. If the signal is not connected to a port, the Verilog-1995 considers the signal as undeclared. Verilog-2000 extends the implicit net declaration to include any signal on the left-hand side of a continuous assignment. If the signal is connected to a module port of the module, then the implicit net will default to the vector width of the port. If the signal is not connected to a port, the it will default to the vector width of the expression on the right-hand side of the assignment.

Synopsys Support:

VCS 6.0	VCS 6.1	PRESTO
unkown	unkown	unkown

3.20 Disabling implicit net declarations

Verilog-1995 will infer a net data type any time an undeclared signal is connected to a module port, a module instance, or a primitive instance. Implicit nets can be convenient, in that it saves having to declare every signal used within a module. However, implicit nets can also lead to unintentional bugs in a design, when a signal name is spelled incorrectly. The incorrect name is not a syntax error, it just infers a new signal has been created. Verilog-2000 adds a means to disable implicit net declarations, so that all signals must be explicitly declared. Disabling implicit nets is done using a new argument of **none** for the already existing compiler directive, **`default_nettype**. The word “none” is not a reserved word, it is simply an argument value.

Synopsys Support:

VCS 6.0	VCS 6.1	PRESTO
unknown	unknown	unknown

3.21 Enhanced conditional compilation

Verilog-1995 supports conditional compilation, using the `ifdef, `else, and `endif compiler directives. Verilog-2000 adds more extensive conditional compilation control, with **`ifndef**, **`elsif** and **`undef** compiler directives.

Synopsys Support:

VCS 6.0	VCS 6.1	PRESTO
yes	yes	yes

3.22 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-2000 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.

Synopsys Support:

VCS 6.0	VCS 6.1	PRESTO
no	no	no

3.23 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 to 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-2000 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. The attribute can appear as a prefix to a declaration, a module, a statement, or a port connection. It can also appear as a suffix to an operator or a function name in an expression. Attributes can be assigned values, including strings, and attribute values can be re-defined for each instance of an object.

Verilog-2000 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
```

Synopsys Support:

VCS 6.0	VCS 6.1	PRESTO
no	no	no

3.24 Array of Instances

The feature, Array of Instances, is part of the Verilog-1995 standard, and is not new with the Verilog-2000 update. It is noted here because it has not been supported by either VCS or the Synopsys synthesis Verilog reader in the past. The following example creates an array of instances of dff modules.

```
module RegN (out, in, clk, rst);
  parameter N = 16;
  output  [N-1:0] out;
  input   [N-1:0] in;
  input           clk, rst;

  dff i[N-1:0] (out, in, clk, rst);
endmodule
```

Synopsys Support:

VCS 5.2	VCS 6.0	VCS 6.1	PRESTO
yes	yes	yes	yes

4.00 ASIC/FPGA accuracy enhancements

The original Verilog language was created at 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-2000 continues this evolution, with enhancements specific for today's—and tomorrow's—deep-submicron designs.

These enhancements are directed toward gate level modeling, as used by ASIC and FPGA vendors to model their libraries. This paper focuses on the behavioral and RTL enhancements of Verilog-2000. The ASIC/FPGA enhancements are beyond the scope of this paper.

5.00 PLI enhancements

Verilog-2000 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 the PLI;
- Implementation of PLI support for all enhancements added to the Verilog language for Verilog-2000;
- Clarifications to the Verilog-1995 PLI standard.

The focus of this paper is on the behavioral and RTL enhancements of Verilog-2000. Details on the enhancements to the PLI are beyond the scope of this paper.

6.00 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).

7.00 Summary

The Verilog-2000 standard is finished, and the IEEE balloting process has been completed. The final step for the standard is approval by the IEEE Revcom which is expected to take place in March 2001. The official title for the Verilog-2000 standard will be "IEEE Std. 1364-2001". Verilog-2000 adds many important enhancements to the Verilog language. These enhancements, 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. Many of these new features are now available for use with the Synopsys VCS 6.0 simulator and Presto synthesis reader. VCS 6.1 will implement even more of the Verilog-2000 enhancements.

8.00 About the Authors:

Stuart Sutherland is the founder and president of Sutherland HDL Inc., a company that specializes in Verilog HDL and Verilog PLI training and design consulting. Mr. Sutherland is chairman of the IEEE 1364 PLI task force for the Verilog-2000 standard and editor of the PLI sections of the 1364-2001 Verilog language reference manual.

Don Mills is the founder and president of LCDM Engineering, a company that specializes in Verilog training, VHDL training, and design consulting. Mr. Mills has 15 years experience with ASIC design, and has worked with Synopsys tools since version 1.2.