**Revised - April 2002**
Important correction to ANSI style parameter lists added to this revision

# Verilog-2001 Behavioral and Synthesis Enhancements

Clifford E. Cummings
cliffc@sunburst-design.com / www.sunburst-design.com

Sunburst Design, Inc.
14314 SW Allen Blvd.
PMB 501
Beaverton, OR 97005

**ABSTRACT**

The Verilog-2001 Standard includes a number of enhancements that are targeted at simplifying designs, improving designs and reducing design errors.

This paper details important enhancements that were added to the Verilog-2001 Standard that are intended to simplify behavioral modeling and to improve synthesis accuracy and efficiency. Information is provided to explain the reasons behind the Verilog-2001 Standard enhancement implementations..

# 1.0 Introduction

For the past five years, experienced engineers and representatives of EDA vendors have wrestled to define enhancements to the Verilog language that will offer increased design productivity, enhanced synthesis capability and improved verification efficiency.

The guiding principles behind proposed enhancements included:
1. do not break existing designs,
2. do not impact simulator performance,
3. make the language more powerful and easier to use.

This paper details many of the behavioral and synthesis enhancements that were added to the Verilog-2001 Standard[1], including some of the rational that went into defining the added enhancements. This paper will also discuss a few errata and corrections to the yet unpublished 2001 Verilog Standard.

Immediately after the header for each enhancement, I make predictions on when you will likely see each enhancement actually implemented by EDA vendors.

## 1.1 Glossary of Terms

The Verilog Standards Group used a set of terms and abbreviations to help concisely describe current and proposed Verilog functionality. Many of those terms are used in this paper and are therefore defined below:

- ASIC - Application Specific Integrated Circuit
- EDA - Electronic Design Automation.
- HDLCON - International HDL Conference.
- IP - Intellectual Property (not internet protocol).
- IVC - International Verilog Conference - precursor to HDLCON when the Spring VIUF and IVC conferences merged.
- LHS - Left Hand Side of an assignment.
- LSB - Least Significant Bit.
- MSB - Most Significant Bit.
- PLI - the standard Verilog Programming Language Interface
- RHS - Right Hand Side of an assignment.
- RTL - Register Transfer Level or the synthesizable subset of the Verilog language.
- VHDL - VHSIC Hardware Description Language.
- VHSIC - Very High Speed Integrated Circuits program, funded by the Department of Defense in the late 1970's and early 1980's [2].
- VIUF - VHDL International Users Forum - the Spring VIUF conference was a precursor to HDLCON when the Spring VIUF and IVC conferences merged.
- VSG - Verilog Standards Group.

## 2.0 What Broke in Verilog-2001?

While proposing enhancements to the Verilog language, the prime directive of the Verilog Standards Group was to not break any existing code. There are only two Verilog-2001 behavioral enhancement proposals that potentially break existing designs. These two enhancements are described below.

### 2.1 31 open files

Verilog-1995[3] permitted users to open up to 31 files for writing. The file handle for Verilog-1995-style files is called an MCD (Multi-Channel Descriptor) where each open file is represented by one bit set in an integer. Only the 31 MSBs of the integer could be set for open files since bit 0 represented the standard output (STDOUT) terminal. The integer identifier-name was the file handle used in the Verilog code.

MCDs could be bit-wise or'ed together into another integer with multiple bits set to represent multiple open files. Using an MCD with multiple valid bits set, a designer can access multiple open files with a single command.

In recent years, engineers have found reasons to access more than 31 files while doing design verification. The 31 open-file limit was too restrictive.

At the same time, engineers were demanding better file I/O capabilities, so both problems were addressed in a single enhancement. The file I/O enhancement requires the use of the integer-MSB to indicate that the new file I/O enhancement is in use. When the integer-MSB is a "0", the file in use is a Verilog-1995-style file with multi-channel descriptor capability. When the integer-MSB is a "1", the file in use is a Verilog-2001-style file where it is now possible to have 2**31 open files at a time, each with a unique binary number file-handle representation (multi-channel descriptors are not possible with the new file I/O-style files.

Any existing design that currently uses exactly 31 open files will break using Verilog-2001. The fix is to use the new file I/O capability for at least one of the current 31 open files. It was necessary to steal the integer MSB to enhance the file I/O capabilities of Verilog.

### 2.2 `bz assignment

Verilog-1995 and earlier has a peculiar, not widely known "feature" (documented-bug!) that permits assignments like the one shown below in Example 1 to assign up to 32 bits of "Z" with all remaining MSBs being set to "0".

```
assign databus = en ? dout : 'bz;
```

Example 1 - Simple continuous assignment using 'bz to do z-expansion

If the databus in Example 1 is 32 bits wide or smaller, this coding style works fine. If the databus is larger than 32 bits wide, the lower bits are set to "Z" while the upper bits are all set to "0". All synthesis tools synthesize this code to 32 tri-state drivers and all upper bits are replaced with and-gates so that if the en input is low, the and-gate outputs also drive "0"s.

The correct Verilog-1995 parameterized model for a tri-state driver of any size is shown Example 2:

```
module tribuf (y, a, en);
  parameter SIZE = 64;
  output [SIZE-1:0] y;
  input  [SIZE-1:0] a;
  input             en;

  assign y = en ? a : {SIZE{1'bz}};
endmodule
```

Example 2 - Synthesizble and parameterizable Verilog-1995 three-state buffer model

In Verilog-2001, making assignments of 'bz or 'bx will respectively z-extend or x-extend the full width of the LHS variable.

The VSG determined that any engineer that intentionally made 'bz assignments, intending to drive 32 bits of "Z" and all remaining MSBs to "0" deserved to have their code broken! An engineer could easily make an assignment of 32'bz wherever the existing behavior is desired and the assignment will either truncate unused Z-bits or add leading zeros to the MSB positions to fill a larger LHS value.

### 2.3 Minimal risk

The VSG decided that there would be minimal impact from the file I/O enhancement that could not be easily solved using the new Verilog-2001 file I/O enhancement, and the 'bz assignment enhancement is not likely to appear in the code of any reasonably proficient Verilog designer, plus there is an easy work-around for the 'bz functionality if the existing silly behavior is actually desired!

## 3.0 LRM Errors

Unfortunately, adding new functionality to the Verilog language also required the addition of new and untested descriptions to the IEEE Verilog Standard documentation. Until the enhanced functionality is implemented, the added descriptions are unproven and might be short on intended enhancement functionality detail. What corner cases are not accurately described? The VSG could not compile the examples so there might be syntax errors in the newer examples.

One example of an error that went unnoticed in the new IEEE Verilog-2001 Standard is the Verilog code for a function that calculates the "ceiling of the log-base 2" of a number. This

example, given in section 10.3.5, makes use of constant functions. The clogb2 function described in the example from the IEEE Verilog Standard, duplicated below, has a few notable errors:

```
//define the clogb2 function
function integer clogb2;
  input depth;
  integer i,result;
  begin
    for (i = 0; 2 ** i < depth; i = i + 1)
      result = i + 1;
    clogb2 = result;
  end
endfunction
```

Example 3 - Verilog-2001 Standard constant function example from section 10 with errors

Errors in this model include:

(1) the input "depth" to the function in this example is only one bit wide and should have included a multi-bit declaration.

(2) the result is not initialized. If the depth is set to "1", the for-loop will not execute and the function will return an unknown value.

A simple and working replacement for this module that even works with Verilog-1995 is shown in Example 4:

```
function integer clogb2;
  input [31:0] value;
  for (clogb2=0; value>0; clogb2=clogb2+1)
    value = value>>1;
endfunction
```

Example 4 - Working function to calculate the ceiling of the log-base-2 of a number

## 4.0 Top Five Enhancements

At a "Birds Of a Feather" session at the International Verilog Conference (IVC) in 1996, Independent Consultant Kurt Baty moderated an after-hours panel to solicit enhancement ideas for future enhancements to the Verilog standard.

Panelists and audience members submitted enhancement ideas and the entire group voted for the top-five enhancements that they would like to see added to the Verilog language. These top-five enhancements gave focus to the VSG to enhance the Verilog language.

Although numerous enhancements were considered and many enhancements added to the Verilog 2001 Standard, the top-five received the most attention from the standards group and all five were added in one form or another. The top-five enhancements agreed to by the audience and panel were:

#1 - Verilog generate statement
#2 - Multi-dimensional arrays
#3 - Better Verilog file I/O
#4 - Re-entrant tasks
#5 - Better configuration control

Many enhancements to the Verilog language were inspired by similar or equivalent capabilities that already existed in VHDL. Many Verilog designers have at one time or another done VHDL design. Any VHDL capability that we personally liked, we tried adding to Verilog. Anything that we did not like about VHDL we chose not to add to Verilog.

### 4.1 Multi-Dimensional Arrays

**Expected to be synthesizable? Yes. This capability is already synthesizable in VHDL and is needed for Verilog IP development.**
**When? Soon!**

Before describing the generate statement, it is logical to describe the multi-dimensional array enhancement, that is essentially required to enable the power of generate statements.

Multidimensional arrays are intended to be synthesizable and most vendors will likely have this capability implemented around the time that the Verilog 2001 LRM becomes an official IEEE Standard.

In Verilog-1995, it was possible to declare register variable arrays with two dimensions. Two noteworthy restrictions were that net types could not be declared as arrays and only one full array-word could be referenced, not the individual bits within the word.

In Verilog-2001, net and register-variable data types can be used to declare arrays and the arrays can be multidimensional. Access will also be possible to either full array words or to bit or part selects of a single word.

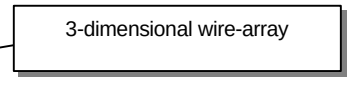
In Verilog-2001, it shall still be illegal to reference a group of array elements greater than a single word; hence, one still cannot initialize a partial or entire array by referencing the array by the array name or by a subset of the index ranges. Two-dimensional array elements must be accessed by one or two index variables, Three dimensional array elements must be accessed by two or three index variables, etc.

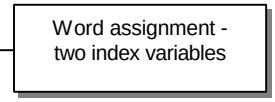
In Example 5, a structural model of a dual-pipeline model with one 2-bit data input is fanned out into two 2-bit by 3-deep pipeline stages and two 2-bit data outputs are driven by the two respective pipeline outputs. The flip-flops in the model have been wired together using a 3-dimensional net array called data. The data-word-width is listed before the identifier data, and the other two dimensions are placed after the identifier data.

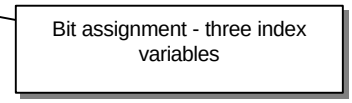
The connections between flip-flops are made using all three dimensions to indicate which individual nets are attached to the flip-flop data input and output, while connections to the ports

are done using only two dimensions to tie the 2-bit buses to the 2-bit data input and output ports of the model.

```
module dualpipe_v2k (dout1, dout0, din, en, clk, rst_n);
  output [1:0] dout1, dout0;
  input  [1:0] din, en;
  input        clk, rst_n;

  wire   [1:0] data [1:0] [3:0];

  assign data[1][0] = din;
  assign data[0][0] = din;

  dff u000 (.q(data[0][1][0]), .d(data[0][0][0]), .clk(clk), .en(en[0]), .rst_n(rst_n));
  dff u010 (.q(data[0][2][0]), .d(data[0][1][0]), .clk(clk), .en(en[0]), .rst_n(rst_n));
  dff u020 (.q(data[0][3][0]), .d(data[0][2][0]), .clk(clk), .en(en[0]), .rst_n(rst_n));

  dff u001 (.q(data[0][1][1]), .d(data[0][0][1]), .clk(clk), .en(en[0]), .rst_n(rst_n));
  dff u011 (.q(data[0][2][1]), .d(data[0][1][1]), .clk(clk), .en(en[0]), .rst_n(rst_n));
  dff u021 (.q(data[0][3][1]), .d(data[0][2][1]), .clk(clk), .en(en[0]), .rst_n(rst_n));

  dff u100 (.q(data[1][1][0]), .d(data[1][0][0]), .clk(clk), .en(en[1]), .rst_n(rst_n));
  dff u110 (.q(data[1][2][0]), .d(data[1][1][0]), .clk(clk), .en(en[1]), .rst_n(rst_n));
  dff u120 (.q(data[1][3][0]), .d(data[1][2][0]), .clk(clk), .en(en[1]), .rst_n(rst_n));

  dff u101 (.q(data[1][1][1]), .d(data[1][0][1]), .clk(clk), .en(en[1]), .rst_n(rst_n));
  dff u111 (.q(data[1][2][1]), .d(data[1][1][1]), .clk(clk), .en(en[1]), .rst_n(rst_n));
  dff u121 (.q(data[1][3][1]), .d(data[1][2][1]), .clk(clk), .en(en[1]), .rst_n(rst_n));

  assign dout1 = data[1][3];
  assign dout0 = data[0][3];
endmodule
```




Example 5 - Verilog-2001 structural dual-pipeline model using multidimensional wire arrays for connections

### 4.2 The Verilog Generate Statement
**Expected to be synthesizable? Yes**
**When? Soon.**

Inspired by the VHDL generate statement, the Verilog generate statement extends generate-statement capabilities beyond those of the VHDL-1993 generate statement.

In VHDL there is a for-generate (for-loop generate) and an if-generate statement. In Verilog-2001 there will be a for-loop generate statement, an if-else generate statement and a case generate statement.

### 4.3 The genvar index variable

After much debate, the VSG decided to implement a new index variable data type that can only be used with generate statements. The keyword for the generate-index variable is "genvar." This variable type is only used during the evaluation of generated instantiations and shall not be referenced by other statements during simulation. The VSG felt it was safest to define a new variable type with restrictive usage requirements as opposed to imposing rules on integers when used in the context of a generate statement.

Per the IEEE Verilog-2001 Draft Standard, a Verilog genvar must adhere to the following restrictions:

- Genvars shall be declared within the module where the genvars are used.
- Genvars can be declared either inside or outside of a generate scope.
- Genvars are positive integers that are local to, and shall only be used within a generate loop that uses them as index variables.
- Genvars are only defined during the evaluation of the generate blocks.
- Genvars do not exist during simulation of a Verilog design.
- Genvar values shall only be defined by generate loops.
- Two generate loops using the same genvar as an index variable shall not be nested.
- The value of a genvar can be referenced in any context where the value of a parameter could be referenced.

The Verilog generate for-loop, like the Verilog procedural for-loop, does not require a contiguous loop-range and can therefore be used to generate sparse matrices of instances that might prove useful to DSP related designs.

The Verilog if-else generate statement can be used to conditionally instantiate modules, procedural blocks, continuous assignments or primitives.

The Verilog case generate statement was added to enhance the development of IP. Perhaps a model could be written for a multiplier IP that chooses an implementation based on the width of the multiplier operands. Small multipliers might be implemented best one or two different ways but large multipliers might be implemented better another way. Perhaps the multiplier model could chose a different implementation based on power_usage parameters passed to the model.

A FIFO model might be created that infers a different implementations based on whether the model uses synchronous or asynchronous clocks.

### 4.4 Enhanced File I/O

**Expected to be synthesizable? No**
**When? Soon.**

Verilog has always had reasonable file-writing capabilities but it only has very limited built-in file-reading capabilities.

Standard Verilog-1995 file reading capabilities were limited to reading binary or hex data from a file into a pre-declared Verilog array and then extracting the data from the array using Verilog commands to make assignments elsewhere in the design or testbench.

Verilog-1995 file I/O can be enhanced through the PLI and the most popular package used to enhance Verilog file I/O is the package maintained by Chris Spear on his web site[4]. Any Verilog

simulator with built-in standard PLI can be compiled to take advantage of most of the Verilog-2001 file I/O enhancements today.

Chris' file I/O PLI code was the starting point for Verilog-2001 file I/O enhancements, and since Chris has already done most of the work of enhancing file I/O, it is likely that most Verilog vendors will leverage off of Chris' work to implement the new file I/O enhancements.

### 4.5 Re-entrant Tasks and Functions

**Expected to be synthesizable? Maybe?**
**When? Probably not soon.**

Verilog functions are synthesizable today and Verilog tasks are synthesizable as long as there are no timing controls in the body of the task, such as @(posedge clk). The #delay construct is ignored by synthesis tools.

This enhancement might be one of the last enhancements to be implemented by most Verilog vendors. Most existing Verilog vendors have complained that this enhancement is a departure from the all-static variables that currently are implemented in the Verilog language. Automatic tasks and functions will require that vendors push the current values of task variables onto a stack and pop them off when the a recursively executing task invocation completes. Vendors are wrestling with how they intend to implement this functionality.

This enhancement is especially important to verification engineers who use tasks with timing controls to apply stimulus to a design. Unknown to many Verilog users, Verilog-1995 tasks use static variables, which means that if a verification task is called a second time before the first task call is still running, they will use the same static variables, most likely causing problems in the testbench. The current work-around is to place the task into a separate verification module and instantiate the module multiple times in the testbench, each with a unique instance name, so that the task can be called multiple times using hierarchical references to the instantiated tasks.

By adding the keyword "automatic" after the keyword "task," Verilog compilers will treat the variables inside of the task as unique stacked variables.

What about synthesis? Tektronix, Inc. of Beaverton Oregon has had an in-house synthesis tool that was first used to design ASICs starting in the late 1980's, and that tool has had the capability to synthesize recursive blocks of code also since the late 1980s. The recursive capabilities made certain DSP blocks very easy to code. Some creative synthesis vendor might find some very useful abilities by permitting recursive RTL coding; however, it is not likely that recursive tasks will be synthesizable in the near future.

### 4.6 Configurations

**Expected to be synthesizable? Yes, Synthesis tools should be capable of reading configuration files to extract the files need to be included into a synthesized design.**
**When? This could be implemented soon.**

Configuration files will make it possible to create a separate file that can map the instances of a source file to specific files as long as the files can be accessed with a UNIX-like path name. This enhancement should remove the need to employ `uselib directives in the source model to change the source files that are used to simulate specific instances within a design.

The `uselib directive has never been standardized because it requires a designer to modify the source models to add directives to call specific files to be compiled for specific instances. Modifying the source files to satisfy the file mapping requirements of a simulation run is a bad idea and the VSG hopes that usage of the `uselib directives will eventually cease. The configuration file also offers an elegant replacement for the common command line switches: -y, -v and +libext+.v, etc. These non-standard command line switches should also slowly be replaced with the more powerful Verilog-2001configuration files.

## 5.0 More Verilog Enhancements

In addition to the top-five enhancement requests, the VSG considered and added other powerful and useful enhancements to the Verilog language. Many of these enhancements are described below.

### 5.1 ANSI-C style port declarations

**Expected to be synthesizable? Yes.**
**When? Almost immediately.**

Verilog-1995 requires all module header ports to be declared two or three times, depending on the data type used for the port. Consider the simple Verilog-1995 compliant example of a simple flip-flop with asynchronous low-true reset, as shown in Example 6.

```
module dffarn (q, d, clk, rst_n);
  output q;
  input  d, clk, rst_n;
  reg    q;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= d;
endmodule
```

Example 6 - Verilog-1995 D-flip-flop model with verbose port declarations

The Verilog-1995 model requires that the "q" output be declared three times, once in the module header port list, once in an output port declaration and once in a reg data-type declaration. The Verilog-2001 Standard combines the header port list declaration, port direction declaration and data-type declaration into a single declaration as shown in Example 7, patterned after ANSI-C style ports. Declaring all 1-bit inputs as wires is still optional.

```
module dffarn (
  output reg q,
  input      d, clk, rst_n);

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= d;
endmodule
```

Example 7 - Verilog 2001 D-flip-flop model with new-style port declarations

This enhancement is a more compact way of making port declarations and should be easy to implement for simulation and synthesis soon.

### 5.2 Parameter passing by name (explicit & implicit)

**Expected to be synthesizable? Yes.**
**When? Almost immediately.**

Verilog-1995 standardized two ways to change parameters for instantiated modules, (1) parameter redefinition and (2) defparam statements.

(1) Parameter redefinition is accomplished by instantiating a module and adding #( new_value1 , new_value2 , ...) immediately after the module name.

Advantage: this technique insures that all parameters are passed to a module at the same time that the module is referenced.

Disadvantage: all parameters must be explicitly listed, in the correct order, up to and including the parameter(s) that are changed. For example, if a module contains 10 parameter definitions, and if the module is to be instantiated requires that the seventh parameter be changed, the instantiation must include seven parameters within the parentheses, listed in the correct order and including the first six values even though they did not change for this instantiation. It is not permitted to simply list six commas followed by the new seventh parameter value.

(2) Using defparam redefinition is accomplished by instantiating a module and including a separate defparam statement to change the instance_name.parameter_name value to its new value.

Advantage: this technique gives a simple and direct correspondence between the instance-name, parameter-name pair and the new value.

Disadvantage: defparam statements can appear anywhere in the Verilog source code and can change any parameter on any module. Translation - when compiling a Verilog design, none of the parameters in any module are fixed until the last Verilog source file is read, because the last file might hierarchically change every single parameter in the design! A "grand-child" module might change all of the parameters of the "grand-parent" module, which might pass new parameter values to the "parent/child" module. It gets ugly and probably slows the compilation of a Verilog design.

Verilog-2001 adds a superior way of passing parameters to instantiated modules, using named parameter passing, using the same technique as named port instantiation.

Advantage #1: Only the parameters that change need to be referenced in named port instantiations. The same advantage that exists when using defparam statements.

Advantage #2: All parameter information is available when the module instantiation is parsed and parameters are passed down the hierarchy; they do not cause side-effects up the hierarchy.

This is the best solution for IP development and usage.

The current defparam statement will not be fully usable in some Verilog-2001 enhancements and the VSG hopes that the addition of named parameter redefinition will eventually cause defparam statement usage to die.

Vendors might want to flag defparam statements as Verilog-2001 compiler errors with the following message:

```
   "The Verilog compiler found a defparam statement in the source code at (file-
line#). To use defparam statements in the Verilog source code, you must include the
switch +Iamstupid on the command line which will degrade compiler performance.
Defparam statements can be replaced with named parameter redefinition as define by
the Verilog-2001 standard"
```

### 5.3 Signed Arithmetic

**Expected to be synthesizable? Could be(?)**
**When? Synthesis vendor dependent.**

The signed arithmetic enhancement removes a frequent complaint about Verilog, that the design has to explicitly code signed arithmetic functionality into the model.

Any vendor that already handles synthesis of signed arithmetic operations should be able to take advantage of this enhancement to facilitate signed arithmetic design tasks.

### 5.4 `ifndef & `elsif

**Expected to be synthesizable? Yes.**
**When? This could be implemented soon.**

The `ifdef / `else / `endif conditionally-compiled-code compiler directives have been a part of the Verilog language since before the Verilog-1995 Standard. Two additions have been added to help generate conditionally compile code: `ifndef and `elsif.

The `ifdef set of compiler directives have been synthesizable by most synthesis tools for a long time and they became synthesizable by Synopsys tools starting with Synopsys version 1998.02 (full usage within Synopsys tools requires that the switch hdlin_enable_vpp be set to true).

The `ifndef switch adds a small simplification to Verilog code where the intent is to compile a block of code only when a specific text macro has not been defined.

```
`ifdef SYNTHESIS
`else
  initial $display("Running RTL Model");
`endif
```

Example 8 - Verilog-1995 coding style to replicate the Verilog-2001 `ifndef capability

```
`ifndef SYNTHESIS
  initial $display("Running RTL Model");
`endif
```

Example 9 - Using the new Verilog-2001 `ifndef compiler directive

Since the `ifndef and `elseif statements are used to simply determine when code should be compiled, these compiler directives could easily be implemented in both simulation and synthesis without much effort.

### 5.5 Exponential Operator

**Expected to be synthesizable? Yes, if the operands are constants.**
**When? This could be implemented soon.**

The ** (exponential) operator is a straightforward way of determining such things as memory depth. If a model has 10 address bits, it should have 1024 memory locations.

If the two operands of the ** operator are constants at compile-time, there is no reason a synthesis tool could not calculate the final value to be used during synthesis.

### 5.6 Local Parameters

**Expected to be synthesizable? Yes.**
**When? This could be implemented soon.**

Parameters, local to a module, that cannot be changed by parameter redefinition during instantiation is another enhancement to the Verilog-2001 Standard. Local parameters are declared using the keyword localparam.

This enhancement is needed by IP developers who want to create a parameterized design where only certain non-local parameters can be manually changed while other local parameters are manipulated within a design based on the parameters that are passed to a particular design instance. Restricting access to some parameters helps to insure that a IP users cannot inadvertently set incompatible parameter values for a particular module. The memory models in Example 10 and Example 11 both use local parameters to calculate one of the memory parameters based on other memory parameters.

### 5.7 Comma separated sensitivity list
**Expected to be synthesizable? Yes.**
**When? Almost immediately.**

Verilog-1995 uses the keyword "or" as a separator in the sensitivity list. New users of the Verilog language often ask the question, "can I use ***and*** in the sensitivity list?" The answer is no.

The "or" keyword is merely a separator between signals in the sensitivity list and nothing more. The Verilog sensitivity list is one of the few places where Verilog is more verbose than VHDL. I personally found this to be offensive!

VHDL separates signals in the sensitivity list with a comma character, which most Verilog users would agree is a better separator token. For this reason, the comma character has been added as an alternate way of separating signals in a Verilog sensitivity list.

Because this enhancement is really just a parsing change, it should be very easy to implement. There is no reason this capability should not be available by all Verilog vendors as soon as the Verilog-2001 Standard is released by the IEEE.

The Verilog code for a parameterized ram model in Example 10 uses a comma-separated sensitivity list in the always block just two lines before the endmodule statement.

```
//-----------------------------------------
// ram1 model - Verilog-2001 @(a, b, c)
// requires ADDR_SIZE & DATA_SIZE parameters
// MEM_DEPTH is automatically sized
//-----------------------------------------
module ram1 (addr, data, en, rw_n);
  parameter ADDR_SIZE = 10;
  parameter DATA_SIZE = 8;
  parameter  MEM_DEPTH = 1<<ADDR_SIZE;
  output [DATA_SIZE-1:0] data;
  input  [ADDR_SIZE-1:0] addr;
  input                  en, rw_n;

  reg    [DATA_SIZE-1:0] mem [0:MEM_DEPTH-1];

  assign data = (rw_n && en) ? mem[addr] : {DATA_SIZE{1'bz}};

  always @(addr, data, rw_n, en)
    if (!rw_n && en) mem[addr] = data;
endmodule
```

Example 10 - Parameterized Verilog ram model with comma-separated sensitivity list

### 5.8 @* combinational sensitivity list
**Expected to be synthesizable? Yes.**
**When? Almost immediately.**

The Verilog-2001 Standard refers to the @* operator as the implicit event expression list; however, members of the VSG called the always @* keyword-pair, the combinational logic

sensitivity list and that was its primary intended purpose, to be used to model and synthesize combinational logic.

Experienced synthesis engineers are aware of the problems that can occur if combinational always blocks are coded with missing sensitivity list entries. Synthesis tools build combinational logic strictly from the equations inside of an always block but then synthesis tools check the sensitivity list to warn the user of a potential mismatch between pre-synthesis and post-synthesis simulations [4].

The always @* procedural block will eliminate the need to list every single always-block input in the sensitivity list. This enhancement will reduce typing, and reduce design errors. The intent was to reduce effort when coding combinational sensitivity lists and to reduce opportunities for coding errors that could lead to a pre-synthesis and post-synthesis simulation mismatch.

The @* was really intended to be used at the top of an always block, but the VSG chose not to restrict its use to just that location. The VSG could not think of a good reason not to use, nor did the VSG think it was wise to restrict, the @* operator only to the top of the always block. This enhancement was made orthogonal but it should be used with caution and has the potential to be abused.

The Verilog code for a parameterized ram model in Example 11 uses an @* sensitivity list in the always block just two lines before the endmodule statement.

```
//-----------------------------------------
// ram1 model - Verilog-2001
// requires ADDR_SIZE & DATA_SIZE parameters
// MEM_DEPTH is automatically sized
//-----------------------------------------
module ram1 #(parameter  ADDR_SIZE = 10,
              parameter  DATA_SIZE = 8)
             (output [DATA_SIZE-1:0] data,
              input  [ADDR_SIZE-1:0] addr,
              input                  en, rw_n);

  localparam MEM_DEPTH = 1<<ADDR_SIZE;
  reg    [DATA_SIZE-1:0] mem [0:MEM_DEPTH-1];

  //-------------------------------------------------
  // Memory read operation
  //-------------------------------------------------
  assign data = (rw_n && en) ? mem[addr] : {DATA_SIZE{1'bz}};

  //-------------------------------------------------
  // Memory write operation - modeled as a latch-array
  //-------------------------------------------------
  always @*
    if (!rw_n && en) mem[addr] <= data;
endmodule
```

Example 11 - Parameterized Verilog ram model with @* combinational sensitivity list

The Verilog-2001 Standard notes that nets and variables which appear on the RHS of assignments, in function and task calls, or case expressions and if expressions shall all be included

in the implicit sensitivity list. Missing from the Verilog-2001 Standard is the fact that variables on the LHS of an expression when used as an index range and variables used in case items should also be included in the implicit sensitivity list. In the 3-to-8 decoder with output enable shown in Example 12, the en input and the a-inputs should included in the @* implied sensitivity list.

```
module decoder (
  output reg [7:0] y,
  input      [2:0] a,
  input            en);

  always @* begin
    y = 8'hff;
    y[a] = !en;
  end
endmodule
```

Example 12 - 3-to-8 Decoder model using the @* implicit sensitivity list

### 5.9 Constant functions

**Expected to be synthesizable? Yes.**
**When? This might take some time to implement.**

Perhaps the most contentious enhancement to the Verilog-2001 Standard, the enhancement that raised the most debate and that was almost removed from the standard on multiple occasions in the past five years, was the constant function. EDA vendors opposed this enhancement because of the perceived difficulty in efficiently implementing this enhancement, and its potential impact on compile-time performance.

A quote from an EDA vendor who requested that constant functions not be added to Verilog summarizes some of the opposition:

"Constant functions are another example of how a VHDL concept does not map will into Verilog ... The Verilog language is simply too powerful and unrestricted to support such functionality."

The users on the VSG also recognize that constant functions might not only be difficult to implement, but also impact compile times. Despite this potential impact on compile-time performance, users deemed this functionality too important to omit from the Verilog-2001 Standard. Vendors might want to publicize that a design modeled without constant functions will compile faster than designs that include constant functions.

Constant functions are important to IP developers. The objective of the constant function is to permit an IP developer to add local parameters to a module that are calculated from other parameters that could be passed into the module when instantiated.

Constant functions will require vendors to calculate some parameters at compile time, which will require that some parameters not be immediately calculated when read, but that they will be calculated after a function is used to determine the actual value of a parameter.

Consider the example of a simple ROM model. To make a parameterized version of a ROM model, we need to know the number of address bits, number of memory locations and number of data bits. The data bus width should be passed to the model, but only the memory size or number of address bits should be passed to the model. If we are given the number of address bits, we should calculate the memory depth at compile time. If we are given the number of memory locations, we should be able to calculate how many address bits are required at compile time.

In order to make constant functions somewhat more agreeable to EDA vendors, they were defined with significant restrictions including some that do not apply to normal Verilog functions. Significant restrictions that apply to constant functions include:

- Constant functions shall not contain hierarchical references.
- Constant functions are defined and invoked in the same module.
- Constant functions shall ignore system tasks. This permits a regular Verilog function with system tasks such as $display commands to be changed into a constant function without requiring removal of the system tasks.
- Constant functions shall not permit system functions.
- Constant functions shall have no side effects (they shall not make assignments to variables that are defined outside of the constant function).
- If a constant function uses an external parameter within the internal calculations of the function, the external parameter must be declared before the constant function call.
- All variables used in a constant function that are not parameters or functions must be declared locally in the constant function.
- If the constant function uses a parameter that is directly or indirectly modified by a defparam statement, the behavior of the Verilog compiler is undefined. The compiler can return an unknown value or it can issue a syntax error.
- Constant functions cannot be defined inside of a generate statement.
- Constant functions shall not call other constant functions in any context that requires a constant expression.

The VSG anticipated that the typical use of a constant function would be to perform simple calculations to generate local parameters to insure compatibility with passed parameters. The above restrictions insure that constant functions do not cause undue compile-time problems.

### 5.10 Attributes

**Expected to be synthesizable? Partially.**
**When? As soon as the IEEE synthesis committee finishes its work and includes attributes into the synthesis spec.**

Verilog-2001 will add a new construct (new to Verilog) called an attribute. The attribute uses (* *) tokens (named "funny braces" by members of the VSG) as shown in Figure 1.

```
(* attribute_name = constant_expression *)
                                        -or-
(* attribute_name *)
```

Figure 1 - Legal attribute definition syntax using (* *)

Attributes were primarily added to the Verilog language enable other tools to use Verilog as an input language and still pass non-Verilog information to those tools. For many years now, vendors have been adding hooks into the Verilog language by way of synthetic comments. The most famous (infamous) example is the deadly[6] synthetic comment:

```
// synopsys full_case parallel_case
```

The biggest problem with the synthetic comment approach is that attaching tool-specific information to a Verilog comment forces those same tools to parse all Verilog comments to see if the comment contains a tool-specific directive.

To assist vendors who use Verilog as an input language, the VSG decided to add attributes to the Verilog language that for the most part will be ignored by Verilog compilers the same as any Verilog comment. The attributes permit third-party vendors to add tool-related information to the source code without impacting simulation and without having to parse every Verilog comment.

### 5.11 Required net declarations
**Expected to be synthesizable? N/A.**
**When? Soon.**

Verilog-1995 has an odd and non-orthogonal requirement that all 1-bit nets, driven by a continuous assignment, that are not declared to be a ports, must be declared. It is the only 1-bit net type that must be declared in Verilog. This non-orthogonal restriction is removed in Verilog-2001.

```
module andor1 (y, a, b, c);
  output y;
  input  a, b, c;
  wire   n1; // not required in Verilog-2001

  assign n1 = a & b;
  assign y  = n1 | c;
endmodule
```

Example 13 - Verilog-1995 required net declaration for the LHS of a continuous assignment to an internal net

### 5.12 `default_nettype none
**Expected to be synthesizable? N/A.**
**When? Soon.**

In the Verilog-1995 Standard, any undeclared identifier, except for the output of a continuous assignment that drives a non-port net, is by default a 1-bit wire. Verilog never required these 1-bit

net declarations and adding the declarations to a model yielded no additional checking to insure that all 1-bit nets were declared.

The Verilog-2001 Standard adds a new option to the `defult_nettype compiler directive called "none." If the "none" option is selected, all 1-bit nets must be declared.

Whether or not forcing all 1-bit nets to be declared is a good coding practice or not is open to debate. Some engineers believe that all nets should be declared before they are used. Other engineers find that declaring all 1-bit nets can be both time and space-consuming.

Editorial comment: I personally find the practice of declaring all 1-bit nets to be a waste of time, effort and lines of code. VHDL requires all 1-bit nets (signals) to be declared, and on a VHDL ASIC design that I worked on in 1996, while instantiating and connecting the major sub-blocks and I/O pads at the top-level model of an ASIC design, I spent as much time debugging flawed signal declarations as I did debugging real hardware problems. The only declarations that I personally found useful were multi-bit signals (buses), which are also required in Verilog-1995. My signal declarations extended over three pages of code and offered no additional useful information about the design. Nevertheless, one can now inflict similar pain and suffering into a Verilog design using the `default_nettype none compiler directive.

## 6.0 Array of Instance

**Expected to be synthesizable? Yes.**
**When? Soon.**

A noteworthy enhancement to the Verilog language is the Array of Instance that was added to the 1995 IEEE Verilog Standard. This enhancement was implemented by Cadence more than two years ago, but the other simulation vendors and all synthesis vendors were slow to follow.

An array of instance allows an simple one-dimensional linear array of instances to be declared in a single statement.

Most ASIC designers build a top-level module that only permits instantiation of other modules, no RTL code allowed. The RTL code is used in sub-modules but not in the top-level module.

In the top-level module, all of the major sub-blocks are instantiated along with all of the ASIC I/O pads. Consider the task of instantiating the I/O pads for a 32-bit address bus and a 16-bit data bus. Verilog engineers have always been required to make 32 address-pad and 16 data-pad instantiations in the top-level model as shown in Example 14.

```
module top_pads1 (pdata, paddr, pctl1, pctl2, pctl3, pclk);
  inout  [15:0] pdata;                    // pad data bus
  input  [31:0] paddr;                    // pad addr bus
  input         pctl1, pctl2, pctl3, pclk; // pad signals
  wire   [15:0] data;                     // data bus
  wire   [31:0] addr;                     // addr bus

  main_blk u1 (.data(data), .addr(addr),
          .sig1(ctl1), .sig2(ctl2), .sig3(ctl3), .clk(clk));
```

```
  IBUF c4 (.O(ctl3), .pI(pctl3));
  IBUF c3 (.O(ctl2), .pI(pctl2));
  IBUF c2 (.O(ctl1), .pI(pctl1));
  IBUF c1 (.O( clk), .pI( pclk));

  IBUF i15 (.O(data[15]), .pI(pdata[15]));
  IBUF i14 (.O(data[14]), .pI(pdata[14]));
  IBUF i13 (.O(data[13]), .pI(pdata[13]));
  IBUF i12 (.O(data[12]), .pI(pdata[12]));
  IBUF i11 (.O(data[11]), .pI(pdata[11]));
  IBUF i10 (.O(data[10]), .pI(pdata[10]));
  IBUF i9  (.O(data[ 9]), .pI(pdata[ 9]));
  IBUF i8  (.O(data[ 8]), .pI(pdata[ 8]));
  IBUF i7  (.O(data[ 7]), .pI(pdata[ 7]));
  IBUF i6  (.O(data[ 6]), .pI(pdata[ 6]));
  IBUF i5  (.O(data[ 5]), .pI(pdata[ 5]));
  IBUF i4  (.O(data[ 4]), .pI(pdata[ 4]));
  IBUF i3  (.O(data[ 3]), .pI(pdata[ 3]));
  IBUF i2  (.O(data[ 2]), .pI(pdata[ 2]));
  IBUF i1  (.O(data[ 1]), .pI(pdata[ 1]));
  IBUF i0  (.O(data[ 0]), .pI(pdata[ 0]));

  BIDIR b31 (.N2(addr[31]), .pN1(paddr[31]), .WR(wr));
  BIDIR b30 (.N2(addr[30]), .pN1(paddr[30]), .WR(wr));
  BIDIR b29 (.N2(addr[29]), .pN1(paddr[29]), .WR(wr));
  BIDIR b28 (.N2(addr[28]), .pN1(paddr[28]), .WR(wr));
  BIDIR b27 (.N2(addr[27]), .pN1(paddr[27]), .WR(wr));
  BIDIR b26 (.N2(addr[26]), .pN1(paddr[26]), .WR(wr));
  BIDIR b25 (.N2(addr[25]), .pN1(paddr[25]), .WR(wr));
  BIDIR b24 (.N2(addr[24]), .pN1(paddr[24]), .WR(wr));
  BIDIR b23 (.N2(addr[23]), .pN1(paddr[23]), .WR(wr));
  BIDIR b22 (.N2(addr[22]), .pN1(paddr[22]), .WR(wr));
  BIDIR b21 (.N2(addr[21]), .pN1(paddr[21]), .WR(wr));
  BIDIR b20 (.N2(addr[20]), .pN1(paddr[20]), .WR(wr));
  BIDIR b19 (.N2(addr[19]), .pN1(paddr[19]), .WR(wr));
  BIDIR b18 (.N2(addr[18]), .pN1(paddr[18]), .WR(wr));
  BIDIR b17 (.N2(addr[17]), .pN1(paddr[17]), .WR(wr));
  BIDIR b16 (.N2(addr[16]), .pN1(paddr[16]), .WR(wr));
  BIDIR b15 (.N2(addr[15]), .pN1(paddr[15]), .WR(wr));
  BIDIR b14 (.N2(addr[14]), .pN1(paddr[14]), .WR(wr));
  BIDIR b13 (.N2(addr[13]), .pN1(paddr[13]), .WR(wr));
  BIDIR b12 (.N2(addr[12]), .pN1(paddr[12]), .WR(wr));
  BIDIR b11 (.N2(addr[11]), .pN1(paddr[11]), .WR(wr));
  BIDIR b10 (.N2(addr[10]), .pN1(paddr[10]), .WR(wr));
  BIDIR b9  (.N2(addr[ 9]), .pN1(paddr[ 9]), .WR(wr));
  BIDIR b8  (.N2(addr[ 8]), .pN1(paddr[ 8]), .WR(wr));
  BIDIR b7  (.N2(addr[ 7]), .pN1(paddr[ 7]), .WR(wr));
  BIDIR b6  (.N2(addr[ 6]), .pN1(paddr[ 6]), .WR(wr));
  BIDIR b5  (.N2(addr[ 5]), .pN1(paddr[ 5]), .WR(wr));
  BIDIR b4  (.N2(addr[ 4]), .pN1(paddr[ 4]), .WR(wr));
  BIDIR b3  (.N2(addr[ 3]), .pN1(paddr[ 3]), .WR(wr));
  BIDIR b2  (.N2(addr[ 2]), .pN1(paddr[ 2]), .WR(wr));
  BIDIR b1  (.N2(addr[ 1]), .pN1(paddr[ 1]), .WR(wr));
  BIDIR b0  (.N2(addr[ 0]), .pN1(paddr[ 0]), .WR(wr));

endmodule
```

Example 14 - Verilog-1995 structural top-level ASIC model with multiple I/O pad instantiations

VHDL engineers have been able to use two generate for-loops to instantiate the same 32 address and 16 data pad models. With Verilog-2001, Verilog engineers can now use similarly simple generate for-loops to instantiate the 32 address and 16 data pads, as shown in Example 15.

```
module top_pads2 (pdata, paddr, pctl1, pctl2, pctl3, pclk);
  inout  [15:0] pdata;                    // pad data bus
  input  [31:0] paddr;                    // pad addr bus
  input         pctl1, pctl2, pctl3, pclk; // pad signals
  wire   [15:0] data;                     // data bus
  wire   [31:0] addr;                     // addr bus

  main_blk u1 (.data(data), .addr(addr),
               .sig1(ctl1), .sig2(ctl2), .sig3(ctl3), .clk(clk));

  genvar i;

  IBUF c4 (.O(ctl3), .pI(pctl3));
  IBUF c3 (.O(ctl2), .pI(pctl2));
  IBUF c2 (.O(ctl1), .pI(pctl1));
  IBUF c1 (.O( clk), .pI( pclk));

  generate for (i=0; i<16; i=i+1) begin: dat
    IBUF i1 (.O(data[i]), .pI(pdata[i]));

  generate for (i=0; i<32; i=i+1) begin: adr
    BIDIR b1 (.N2(addr[i]), .pN1(paddr[i]), .WR(wr));

endmodule
```

Generated instance names dat[0].i1 to dat[15].i1

Generated instance names adr[0].b1 to adr[31].b1

Example 15 - Top-level ASIC model with address and data I/O pads instantiated using a generate statement

For simple contiguous one-dimensional arrays, the array of instance construct is even easier to use and has a more intuitive syntax. Finally, simulation and synthesis vendors are now starting to support the Verilog-1995 Array of Instance construct that makes placement of 32 consecutively named instances possible with an easy instantiation by bus names as ports and applying a range to the instance name as shown in Example 16.

```
module top_pads3 (pdata, paddr, pctl1, pctl2, pctl3, pclk);
  inout  [15:0] pdata;                    // pad data bus
  input  [31:0] paddr;                    // pad addr bus
  input         pctl1, pctl2, pctl3, pclk; // pad signals
  wire   [15:0] data;                     // data bus
  wire   [31:0] addr;                     // addr bus

  main_blk u1 (.data(data), .addr(addr),
               .sig1(ctl1), .sig2(ctl2), .sig3(ctl3), .clk(clk));

  IBUF c4 (.O(ctl3), .pI(pctl3));
  IBUF c3 (.O(ctl2), .pI(pctl2));
  IBUF c2 (.O(ctl1), .pI(pctl1));
  IBUF c1 (.O( clk), .pI( pclk));

  IBUF i[15:0] (.O(data), .pI(pdata));

  BIDIR b[31:0] (.N2(addr), .pN1(paddr), .WR(wr));

endmodule
```

Arrayed instance names i[15] to i[0]

Arrayed instance names b[31] to b[0]

Example 16 - Top-level ASIC model with address and data I/O pads instantiated using arrays of instance

## 7.0 Conclusions

The Verilog-2001 enhancements are coming. These enhancements will increase the efficiency and productivity of Verilog designers.

## 8.0 Honorable Mention

Although the Behavioral Task Force benefited from the expertise and contributions of numerous synthesis experts, a particular honorable mention must go out to Kurt Baty of WSFDB.

Kurt has experience designing some 50 ASICs and has written a significant number of Design Ware models that are used in Synopsys synthesis tools. Kurt complains that he had to write all of the models using VHDL because Verilog lacked a few of the key features that are required to make parameterized models. Kurt's insight into the 1995 Verilog limitations lead to enhancements that will make future IP model creation not only doable, but also easier to do in Verilog than it was in VHDL.

## 9.0 References

[1] IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language, IEEE Std P1364/D5

[2] Douglas L. Perry, *VHDL*, McGraw-Hill, Inc., 1994, p. 1.

[3] IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language, IEEE Computer Society, IEEE Std 1364-1995

[4] www.chris.spear.net

[5] Don Mills and Clifford E. Cummings, "RTL Coding Styles That Yield Simulation and Synthesis Mismatches," *SNUG'99 (Synopsys Users Group San Jose, CA, 1999) Proceedings*, section-TA2 (1st paper), March 1999.

[6] Clifford E. Cummings, "'full_case parallel_case", the Evil Twins of Verilog Synthesis,' *SNUG'99 Boston (Synopsys Users Group Boston, MA, 1999) Proceedings*, section-FA1 (2nd paper), October 1999.

## Revision 1.2 - What Changed?

The ANSI style ports in previous versions of this paper incorrectly showed semi-colons between port declarations and between the parameter list and the port list. These errors were fixed in this version of the document.

## Revision 1.3 (April 2002) - What Changed?

Example 11 still incorrectly showed ANSI style parameters separated by a semicolon instead of a comma and also included an illegal comma at the end of the ANSI style parameter list. Note that the second **`parameter`** keyword is not required in the ANSI style parameter list and would typically be removed.

Example 11 also included a localparam declaration in the ANSI style parameter list, which is illegal. The localparam has been moved outside of the ANSI style header.

## Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 20 years of ASIC, FPGA and system design experience and ten years of Verilog, synthesis and methodology training experience.

Mr. Cummings, a member of the IEEE 1364 Verilog Standards Group (VSG) since 1994, chaired the VSG Behavioral Task Force, which was charged with proposing enhancements to the Verilog language. Mr. Cummings is also a member of the IEEE Verilog Synthesis Interoperability Working Group and the Accellera SystemVerilog Standardization Group

Mr. Cummings holds a BSEE from Brigham Young University and an MSEE from Oregon State University.

E-mail Address: cliffc@sunburst-design.com
This paper can be downloaded from the web site: www.sunburst-design.com/papers

(Data accurate as of December 17th, 2001)