New Verilog-2001 Techniques for Creating Parameterized Models (or Down With `define and Death of a defparam!)

Clifford E. Cummings
Sunburst Design, Inc.
cliffc@sunburst-design.com

Abstract

Creating reusable models typically requires that general-purpose models be written with re-definable parameters such as SIZE, WIDTH and DEPTH.

With respect to coding parameterized Verilog models, two Verilog constructs that are over-used and abused are the global macro definition (`define) and the infinitely abusable parameter redefinition statement (defparam).

This paper will detail techniques for coding proper parameterized models, detail the differences between parameters and macro definitions, present guidelines for using macros, parameters and parameter definitions, discourage the use of defparams, and detail Verilog-2001 enhancements to enhance coding and usage of parameterized models.

1. Introduction

Two Verilog constructs that are overused and abused are the Verilog macro definition statement (**`define**) and the infinitely abusable **defparam** statement. It is the author's opinion that macro definitions are largely over-used to avoid the potential abuse of the dangerous **defparam** statement by design teams.

Respected Verilog and verification texts over-promote the usage of the macro definition (**`define**) statement, and those recommendations are being followed without recognition of the dangers that these recommendations introduce.

Note: even though multiple questionable parameter and macro definition recommendations are cited from *Principles of Verifiable RTL Design* by Bening and Foster[13] and from *Writing Testbenches, Functional Verification of HDL Models* by Bergeron[8], I still recommend both texts for the other valuable material they both contain, especially the text by Bening and Foster.

2. Verilog Constants

In Verilog-1995[6], there are two ways to define constants: the **parameter**, a constant that is local to a module and macro definitions, created using the **`define** compiler directive.

A **parameter**, after it is declared, is referenced using the parameter name.

A **`define** macro definition, after it is defined, is referenced using the macro name with a preceding ` (back-tic) character.

It is easy to distinguish between **parameters** and macros in a design because macros have a **`***identifier_name* while a **parameter** is just the *identifier_name* without back-tic.

3. Parameters

Parameters must be defined within module boundaries using the keyword **parameter**.

A **parameter** is a constant that is local to a module that can optionally be redefined on an instance-by-instance basis. For parameterized modules, one or more parameter declarations typically precede the port declarations in a Verilog-1995 style model, such as the simple register model in Example 1.

```
module register (q, d, clk, rst_n);
  parameter SIZE=8;
  output [SIZE-1:0] q;
  input  [SIZE-1:0] d;
  input             clk, rst_n;
  reg    [SIZE-1:0] q;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 0;
    else        q <= d;
endmodule
```

Example 1 - Parameterized register model - Verilog-1995 style

The Verilog-2001[5] version of the same model can take advantage of both the ANSI-C style ports and module header parameter list, as shown in Example 2.

```
module register2001 #(parameter SIZE=8)
  (output reg [SIZE-1:0] q,
   input      [SIZE-1:0] d,
   input                 clk, rst_n);

  always @(posedge clk, negedge rst_n)
    if (!rst_n) q <= 0;
    else        q <= d;
endmodule
```

Example 2 - Parameterized register model - Verilog-2001 style

4. Parameters and Parameter Redefinition

When instantiating modules with parameters, in Verilog-1995 there are two ways to change the parameters for some or all of the instantiated modules; parameter redefinition in the instantiation itself, or separate **defparam** statements.

Verilog-2001 adds a third and superior method to change the parameters on instantiated modules by using named parameter passing in the instantiation itself (see section 7).

5. Parameter redefinition using

Parameter redefinition during instantiation of a module uses the **#** character to indicate that the parameters of the instantiated module are to be redefined.

In Example 3, two copies of the register from Example 1 are instantiated into the **two_regs1** module. The **SIZE** parameter for both instances is set to 16 by the **#(16)** parameter redefinition values on the same lines as the register instantiations themselves.

```
module two_regs1 (q, d, clk, rst_n);
  output [15:0] q;
  input  [15:0] d;
  input         clk, rst_n;
  wire   [15:0] dx;

  register #(16) r1 (.q(q), .d(dx),
          .clk(clk), .rst_n(rst_n));

  register #(16) r2(.q(dx), .d(d),
          .clk(clk), .rst_n(rst_n));
endmodule
```

Example 3 - Instantiation using parameter redefinition

This form of parameter redefinition has been supported by all synthesis tools for many years.

The biggest problem with this type of parameter redefinition is that the parameters must be passed to the instantiated module in the order that they appear in the module being instantiated.

Consider the myreg module of Example 4.

```
module myreg (q, d, clk, rst_n);
  parameter Trst = 1,
            Tckq = 1,
            SIZE = 4,
            VERSION = "1.1";
  output [SIZE-1:0] q;
  input  [SIZE-1:0] d;
  input             clk, rst_n;
  reg    [SIZE-1:0] q;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= #Trst 0;
    else        q <= #Tckq d;
endmodule
```

Example 4 - Module with four parameters

The **myreg** module of Example 4 has four parameters, and if the module, when instantiated, requires that just the third parameter, (for example the **SIZE** parameter) be changed, the module cannot be instantiated with a series of commas followed by the new value for the **SIZE** parameter as shown in Example 5. This would be a syntax error.

```
module bad_wrapper (q, d, clk, rst_n);
  output [7:0] q;
  input  [7:0] d;
  input        clk, rst_n;

  // illegal parameter passing example
  myreg #(,,8) r1 (.q(q), .d(d),
               .clk(clk), .rst_n(rst_n));
endmodule
```

Example 5 - Parameter redefinition with #(,,8) syntax error

In order to use the parameter redefinition syntax when instantiating a module, all parameter values up to and including all values that are changed, must be listed in the instantiation. For the **myreg** module of Example 4, the first two parameter values must be listed, even though they do not change, followed by the new value for the **SIZE** parameter, as shown in Example 6.

```
module good_wrapper (q, d, clk, rst_n);
  output [7:0] q;
  input  [7:0] d;
  input        clk, rst_n;

  // the first two parameters must be
  // explicitly passed even though the
  // values did not change
  myreg #(1,1,8) r1 (.q(q), .d(d),
                .clk(clk), .rst_n(rst_n));
endmodule
```

Example 6 - Parameter redefinition with correct #(1,1,8) syntax

Aware of this limitation, engineers have frequently rearranged the order of the parameters to make sure that the most frequently used parameters are placed first in a module, similar to the technique described by Thomas and Moorby[4].

Despite the limitations of Verilog-1995 parameter redefinition, it is still the best supported and cleanest method for modifying the parameters of an instantiated module.

Verilog-2001 actually enhances the above parameter redefinition capability by adding the ability to pass the parameters by name, similar to passing port connections by name. See section 7 for information on this new and preferred way of passing parameters to instantiated modules.

6. Death to defparams!

First impressions of **defparam** statements are very favorable. In fact, many authors, like Bergeron, prefer usage of the **defparam** statement because "it is self documenting and robust to changes in parameter declarations"[9].

The **defparam** statement explicitly identifies the instance *and* the individual parameter that is to be redefined by each **defparam** statement. The **defparam** statement can be placed before the instance, after the instance or anywhere else in the file.

Until the year 2000, Synopsys tools did not permit parameter redefinition using **defparam** statements. Synopsys was to be commended for this restriction. Unfortunately, Synopsys developers bowed to pressure from uninformed engineers and added the ability to use **defparam** statements in recent versions of Synopsys tools.

Unfortunately, the well-intentioned **defparam** statement is easily abused by:

(1) using **defparam** to hierarchically change the parameters of a module.

(2) placing the **defparam** statement in a separate file from the instance being modified.

(3) using multiple **defparam** statements in the same file to change the parameters of an instance.

(4) using multiple **defparam** statements in multiple different files to change the parameters of an instance.

6.1. Hierarchical defparams

It is legal to hierarchically change the values of parameters using a **defparam** statement. This means that any **parameter** in a design can be changed from any input file in the design. Potentially, the abuse could extend to changing the parameter value of the module that instantiated the module with the **defparam** statement and pass that **parameter** to the instantiated module that in turn re-modifies the **parameter** of the instantiating module again, etc.

In Example 7, the **testbench** module (**tb_defparam**) instantiates a model and passes the **SIZE** parameter to the register module (passed to the **WIDTH** parameter), which passes the **WIDTH** parameter to the **dff** module (passed to the **N** parameter). The **dff** module has an erroneous hierarchical **defparam** statement that changes the testbench **SIZE parameter** from **8** to **1** and that value is again passed down the hierarchy to change the register **WIDTH** and the **dff N** values again.

```
module tb_defparam;
  parameter SIZE=8;
  wire [SIZE-1:0] q;
  reg  [SIZE-1:0] d;
  reg             clk, rst_n;

  register2 #(SIZE) r1
             (.q(q), .d(d), .clk(clk),
              .rst_n(rst_n));
  // ...
endmodule

module register2 (q, d, clk, rst_n);
  parameter WIDTH=8;
  output [WIDTH-1:0] q;
  input  [WIDTH-1:0] d;
  input              clk, rst_n;

  dff #(WIDTH) d1
       (.q(q), .d(d), .clk(clk),
        .rst_n(rst_n));
endmodule
```

Example 7 - Dangerous use of hierarchical defparam
(example continues on next page)

```
module dff (q, d, clk, rst_n);
  parameter N=1;
  output [N-1:0] q;
  input  [N-1:0] d;
  input             clk, rst_n;
  reg    [N-1:0] q;

  // dangerous, hierarchical defparam
  defparam tb_defparam.SIZE = 1;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 0;
    else        q <= d;
endmodule
```

Example 7 - Dangerous use of hierarchical defparam

All of the ports and variables in the designs in Example 7 are now just one bit wide, while synthesis of the **register2** and **dff** modules will be eight bits wide. This type of **defparam** use can easily escape detection and cause design and debug problems.

```
module register3 (q, d, clk, rst_n);
  parameter WIDTH=8;
  output [WIDTH-1:0] q;
  input  [WIDTH-1:0] d;
  input              clk, rst_n;

  dff3 #(WIDTH) d1
        (.q(q), .d(d), .clk(clk),
         .rst_n(rst_n));
endmodule

module dff3 (q, d, clk, rst_n);
  parameter N=1;
  output [N-1:0] q;
  input  [N-1:0] d;
  input             clk, rst_n;
  reg    [N-1:0] q;

  // dangerous, hierarchical defparam
  defparam register3.WIDTH = 1;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 0;
    else        q <= d;
endmodule
```

Example 8 - Dangerous hierarchical defparams enclosed within the register3/dff3 models

Example 8 is similar to Example 7 except that the **defparam** redefines the bus widths of the **register3** model and appears to be self-contained. Unfortunately, even though this model will simulate like a 1-bit wide model, it still synthesizes to an 8-bit wide model.

6.2. defparams in separate files

It is not uncommon to find **defparams** being abused by placing them in a completely different file from the instances being modified[10].

Unfortunately, this practice was semi-encouraged by the following comment in section 12.2.1 of the Verilog-1995[6] and Verilog-2001[5] Standards documents:

> The defparam statement is particularly useful for grouping all of the parameter value override assignments together in one module.

The above text probably should have been deleted from the Verilog-2001 Standard, but it was not.

It should be noted that the Verilog Standards Group (VSG) introduced and encourages the use of the superior capability of passing parameters by name (see section 7), similar to passing ports by name, when instantiating modules. The VSG hopes that engineers will take advantage of this new capability and that **defparam** statements eventually die (see section 6.6).

6.3. Multiple defparams in the same file

defparams are abused by placing multiple **defparams** in the same file that modify the same **parameter**. The Verilog-2001 Standard defines the correct behavior to be:

> **In the case of multiple defparams for a single parameter, the parameter takes the value of the last defparam statement encountered in the source text.[5]**

In Example 9, two copies of the **register** from Example 1 are instantiated into the **two_regs2** module. The **SIZE** parameter for both instances is set to **16** by **defparam** statements placed before the corresponding **register** instantiations. A third **defparam** statement is placed after the second **register** instantiation, changing the size of the second register to **4** by mistake.

```
module two_regs2 (q, d, clk, rst_n);
  parameter SIZE = 16;
  output [SIZE-1:0] q;
  input  [SIZE-1:0] d;
  input             clk, rst_n;
  wire   [SIZE-1:0] dx;

  defparam r1.SIZE=16;
  register r1 (.q(q), .d(dx), .clk(clk),
               .rst_n(rst_n));

  defparam r2.SIZE=16;
  register r2 (.q(dx), .d(d), .clk(clk),
               .rst_n(rst_n));
  defparam r2.SIZE=4; // Design error!
endmodule
```

Example 9 - Instantiation using defparam statements

Because this is a small design and because compilers will issue "port-size mismatch" warnings, this design will not be difficult to debug.

Unfortunately, frequently when a second stray **defparam** statement is added by mistake, it is added into a large design with pages of RTL code because the designer did not notice that an earlier **defparam** statement had been used to redefine the same parameter value. This type of design is typically more confusing and more difficult to debug.

6.4. Multiple defparams in separate files

defparams are even abused by placing them in multiple different files.

The practice of placing multiple **defparam** statements in different files that make assignments to the same **parameter** is very problematic. Multiple **defparam** statements are treated differently by different vendors because the behavior for this scenario was never defined in the Verilog-1995 Standard.

The Verilog-2001 Standards Group did not want to encourage this behavior so we added the following disclaimer to the Verilog-2001 Standard.

> **When defparams are encountered in multiple source files, e.g., found by library searching, the defparam from which the parameter takes its value is undefined.[5]**

The Verilog-2001 Standards Group basically wanted to discourage this practice altogether so we left the behavior undefined and documented that fact, hoping to discourage anyone from requiring vendors to support this flawed strategy.

6.5. defparams and tools

Since **defparams** can be placed anywhere in a design and because they can hierarchically change the **parameter** values of any module in a design, **defparams** in their current incarnation make it very difficult to write either a vendor tool or an in-house tool that can accurately parse a design that is permitted to include **defparam** statements[14].

A Verilog compiler cannot determine the actual values of any parameters until all of the Verilog input files have been read, because the last file read might change every single parameter in the design!

I know of some companies that ban the use of **defparams** in their Verilog code in order to facilitate the creation of useful in-house Verilog tools. I agree with this practice and propose the following guideline:

Guideline: do not use **defparams** in any Verilog designs.

A superior alternative to **defparam** statements is discussed in section 7.

6.6. Deprecate defparam

The VSG is not the only organization that hopes that the defparam statement will die (see the end of section 6.2).

The IEEE Verilog Synthesis Interoperability Group voted not to support **defparam** statements in the IEEE Verilog Synthesis Standard[7].

And in April 2002, The SystemVerilog Standards Group voted unanimously (with one abstention) to deprecate the **defparam** statement (possibly remove support for the **defparam** statement from future versions of the Verilog language)[1].

After **defparams** have been deprecated, the author suggests that future Verilog tools report errors whenever a **defparam** statement is found in any Verilog source code and then provide a switch to enable **defparam** statement use for backward compatibility. An error message similar to the following is suggested:

> **"The Verilog compiler found a defparam statement in the source code at (file_name/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 and introduce potential problems but is bug-compatible with Verilog-1995 implementations.**
> **Defparam statements can be replaced with named parameter redefinition as define by the IEEE Verilog-2001 standard."**

The preceeding **defparam** warning is annoyingly long. Hopefully users will tire of these long annoying warnings and remove **defparams** from their code.

7. Verilog-2001 named parameter redefinition

An enhancement added to the Verilog-2001 Standard is the ability to instantiate modules with named parameters in the instantiation itself[3][5].

This enhancement is superior to and eliminates the need for **defparam** statements.

```
module demuxreg (q, d, ce, clk, rst_n);
  output [15:0] q;
  input  [ 7:0] d;
  input         ce, clk, rst_n;
  wire   [15:0] q;
  wire   [ 7:0] n1;

  not          u0 (ce_n, ce);
  regblk #(.SIZE( 8)) u1
          (.q(n1), .d (d), .ce(ce),
           .clk(clk), .rst_n(rst_n));
  regblk #(.SIZE(16)) u2
          (.q (q), .d({d,n1}), .ce(ce_n),
           .clk(clk), .rst_n(rst_n));
endmodule

module regblk (q, d, ce, clk, rst_n);
  parameter SIZE = 4;
  output [SIZE-1:0] q;
  input  [SIZE-1:0] d;
  input             ce, clk, rst_n;
  reg    [SIZE-1:0] q;

  always @(posedge clk or negedge rst_n)
    if      (!rst_n) q <= 0;
    else if (ce)     q <= d;
endmodule
```

Example 10 - Instantiation using named parameter passing

This new technique offers the advantage of specifically indicating which **parameter** is modified (like the **defparam** statement) and also places the **parameter** values conveniently into the instantiation syntax, like Verilog-1995 **#** parameter redefinition.

This is the cleanest way to instantiate models from any vendor and this is a technique that should be encouraged by designers and vendors of reusable models.

Because all of the parameter information is included in the instantiation of the model, this coding style will also be easiest to parse by vendor and in-house tools.

Guideline: require all passing of parameters to be done using the new Verilog-2001 named parameter redefinition technique.

8. `define Macro Substitution

The **`define** compiler directive is used to perform "global" macro substitution, similar to the C-language #define directive. Macro substitutions are global from the point of definition and remain active for all files read after the macro definition is made or until another macro definition changes the value of the defined macro or until the macro is undefined using the **`undef** compiler directive.

Macro definitions can exist either inside or outside of a module declaration, and both are treated the same. **parameter** declarations can only be made inside of module boundaries.

Since macros are defined for all files read after the macro definition, using macro definitions generally makes compiling a design file-order dependent.

A typical problem associated with using macro definitions is that another file might also make a macro definition to the same macro name. When this occurs, Verilog compilers issue warnings related to "macro redefinition" but an unnoticed warning can be costly to the design or to the debug effort.

Why is it bad to redefine macros? The Verilog language allows hierarchical referencing of identifiers. This proves to be very valuable for probing and debugging a design. If the same macro name has been given multiple definitions in a design, only the last definition will be available to the testbench for probing and debugging purposes.

If you find yourself making multiple macro definitions to the same macro name, consider that the macro should probably be a local **parameter** as opposed to a global macro.

9. `define Usage

Guideline: only use macro definitions for identifiers that clearly require global definition of an identifier that will not be modified elsewhere in the design.

Guideline: where possible, place all macro definitions into one "**definitions.vh**" file and read the file first when compiling the design.

Alternate Guideline: place all macro definitions in the top-level testbench module and read this module first when compiling the design.

Reading all macro definitions first when compiling a design insures that the macros exist when they are needed and that they are globally available to all files compiled in the design.

Pay attention to warnings about macro redefinition.

Guideline: do not use macro definitions to define constants that are local to a module.

10. `define Inclusion

One popular technique to insure that a macro definition exists before its usage is to use an **`ifdef**, or the new Verilog-2001 **`ifndef** compiler directives to query for the existence of a macro definition followed by either a **`define** macro assignment or a **`include** of a file name that contains the require macro definition.

```
`ifdef CYCLE
  // do nothing (better to use `ifndef)
`else
  `define CYCLE 100
`endif

`ifndef CYCLE
  `include "definitions.vh"
`endif
```

Example 11 - Testing and defining macro definitions

11. The `undef compiler directive

Verilog has the **`undef** compiler directive to remove a macro definition created with the **`define** compiler directive.

Bergeron recommends avoiding the use of macro definitions[11]. I agree with this recommendation. Bergeron further recommends that all macro definitions should be removed using **`undef** when no longer needed[11]. I disagree with this recommendation. This seems to be overkill to correct a problem that rarely exists. Using the **`define** compiler directive to create global macros where appropriate is very useful. Losing sleep over the existence of global macro definitions and tracking all of the **`undef**'s in a design is not a good use of time.

For the rare occasion where it might make sense to redefine a macro, use **`undef** in the same file and at the end of the file where the **`define** macro was defined.

Make sure that the last compiled macro definition is likely to be the macro that you might want to access from a testbench, because only one macro definition can exist during runtime debug.

Again, using a **`define**-**`undef** pair should be considered the last resort to a problem that could probably be better handled using a better method.

12. Clock cycle definition

Bergeron's somewhat justified paranoia over the use of the **`define** macro definition leads him to recommend that clock cycles be defined using **parameters** as opposed to using the **`define** compiler directive[12]. This recommendation is flawed.

Guideline: make clock cycle definitions using the **`define** compiler directive. Example:

```
`define CYCLE 10
```

Guideline: place the clock cycle definitions in the "**definitions.vh**" file or in the top-level testbench. Example:

```
`define CYCLE 10
module tb_cycle;
  // ...
  initial begin
    clk = 1'b0;
    forever #(`CYCLE/2) clk = ~clk;
  end
  // ...
endmodule
```

Example 12 - Global clock cycle macro definition and usage (recommended)

Reason: Clock cycles are a fundamental constant of a design and testbench. The cycle of a common clock signal should not change from one module to another; the cycle should be constant!

Verilog power-users do most stimulus generation and verification testing on clock edges in a testbench. In general, this type of testbench scales nicely with changes to the global clock cycle definition.

13. State Machines and `define do not mix

Bening and Foster[13] and Keating and Bricaud[15] both recommend using the **`define** compiler directive to define state names for a Verilog state machine design. After recommending the use of **`define**, Keating and Bricaud subsequently show an example using **parameter** definitions instead of using the **`define**[16]. The latter is actually preferred.

Finite State Machine (FSM) designs should use **parameters** to define state names because the state name is a constant that applies only to the FSM module. If multiple state machines are added to a large design, it is not uncommon to want to reuse certain state names in multiple FSM designs[1]. Example state names that are common to multiple designs include: RESET, IDLE, READY, READ, WRITE, ERROR and DONE.

Using **`define** to assign state names would either preclude reuse of a state name because the name has already been taken in the global name space, or one would have to **`undef** state names between modules and re-**`define** state names in the new FSM modules. The latter case makes it difficult to probe the internal values of FSM

state buses from a testbench and running comparisons to the state names.

There is no good reason why state names should be defined using **`define**. State names should not be considered part of the global name space. State names should be considered local names to the FSM module that encloses them.

Guideline: do not make state assignments using **`define** macro definitions for state names.

Guideline: Make state assignments using **parameters** with symbolic state names.

14. Verilog-2001 localparam

An enhancement added to the Verilog-2001 Standard is the **localparam**.

Unlike a **parameter**, a **localparam** cannot be modified by parameter redefinition (positional or named redefinition) nor can a **localparam** be redefined by a **defparam** statement.

The **localparam** can be defined in terms of **parameters** that can be redefined by positional parameter redefinition, named parameter redefinition (preferred) or **defparam** statements.

The idea behind the **localparam** is to permit generation of some local parameter values based on other **parameters** while protecting the **localparams** from accidental or incorrect redefinition by an end-user.

In Example 13, the size of the memory array **mem** should be generated from the size of the address bus. The memory depth-size **MEM_DEPTH** is "protected" from incorrect settings by placing the **MEM_DEPTH** in a **localparam** declaration. The **MEM_DEPTH** parameter will only change if the **ASIZE** parameter is modified.

```
module ram1 #(parameter ASIZE=10,
                        DSIZE=8)
   (inout  [DSIZE-1:0] data,
    input  [ASIZE-1:0] addr,
    input              en, rw_n);

   // Memory depth equals 2**(ASIZE)
   localparam MEM_DEPTH = 1<<ASIZE;
   reg [DSIZE-1:0] mem [0:MEM_DEPTH-1];

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

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

Example 13 - Verilog-2001 ANSI-parameter and port style model with localparam usage

We want to protect the local **MEM_DEPTH** parameter and calculate it from the size parameter value of the address bus.

Note: the Verilog-2001 Standard does not extend the capabilities of the localparam enhancement to the module header parameter list. Specifically, localparam currently cannot be added to an ANSI-style parameter list as shown in Example 14.

```
module multiplier2
    #(parameter  AWIDTH=8, BWIDTH=8,
      localparam YWIDTH=AWIDTH+BWIDTH)
     (output [YWIDTH-1:0] y,
      input  [AWIDTH-1:0] a,
      input  [BWIDTH-1:0] b);
  assign y = a * b;
endmodule
```

Example 14 - Illegal use of localparam in the ANSI-parameter header

15. `timescale Definitions

The **`timescale** directive gives meaning to delays that may appear in a Verilog model. The timescale is placed above the module header and takes the form:

```
`timescale time_unit / time_precision
```

The **`timescale** directive can have a huge impact on the performance of most Verilog simulators. It is a common new-user mistake to select a time_precision of **1ps** (1 pico-second) in order to account for every last pico-second in a design. adding a **1ps** precision to a model that is adequately modeled using either **1ns** or **100ps** time_precisions can increase simulation time by more than 100% and simulation memory usage by more than 150%. I know of one very popular and severely flawed synthesis book that shows Verilog coding samples using a `timescale of **1 ns / 1 fs**[17] (measuring simulation performance on this type of design typically requires a calendar watch!)

I have seen some engineers use a macro definition to facilitate changing all **`timescales** in a design. All modules coded by these engineers include the timescale macro before every module header that they ever write. Example 15 shows a macro definition for a global **`timescale** and usage of the global **`timescale** macro.

```
`define tscale `timescale 1ns/1ns

`tscale
module mymodule (...);
  ...
```

Example 15 - Global maroc definition of a timescale macro (not recommended)

These well-meaning engineers typically hope to control simulation efficiency by changing a global **`timescale** definition to potentially modify both the time_units and time_precisions of every model and enhance simulator performance.

Globally changing the time_units of every **`timescale** in a design can adversely impact the integrity of an entire design. Any design that includes **#**delays relies on the accuracy of the specified time_units in the **`timescale** directive. In Example 16, the model requires that the time_units of the **`timescale** be in units of **100ps**. Changing the time_units to **1ns** changes the delay from 160ps to 1.6ns, introducing an error into the model.

```
`timescale 100ps/10ps
module tribuf2001 #(parameter SIZE=8)
  (output [SIZE-1:0] y,
   input  [SIZE-1:0] a,
   input             en_n);

  assign #1.6 y = en_n ? {SIZE{1'bz}}:a;
endmodule
```

Example 16 - Module with 100ps time_units

Since the time_precision must always be equal to or smaller than the time_unit in a **`timescale** directive, additional guidelines should probably be followed if a global **`timescale** strategy is being employed:

Guideline: Make all time_units of user defined **`timescales** equal to **1ns** or larger.

Reason: if a smaller time_unit is used in any model, globally changing all time_precisions to **1ns** will break an existing design.

Note: If a vendor model is included in the simulation and if the vendor used a very small time_precision in the their model, the entire simulation will slow down and very little will have been accomplished by globally changing the time_precisions of the user models.

To enhance simulator performance, using a unit-delay simulation mode or using cycle based simulators are better options than macro-generating all of the **`timescales** in a design.

16. Conclusions

Macro definitions should be used to define system-global constants, such as a user-friendly set of names for PCI commands or global clock cycle definitions.

Each time a new macro definition is made, that macro name cannot be safely used elsewhere in the design (name-space pollution). As more and more modules are compiled into large system simulations, the likelihood of macro-name collision increases. The practice of making macro definitions for constants such as port or data sizes and state names is an ill-advised practice.

Macro definitions using the **`define** compiler directive should not be used to define constants that can be better localized to individual modules.

Verilog **parameters** are intended to represent constants that are local to a module. A **parameter** has the added benefit that each different instance of the module can have different values for the **parameters** in each module.

The following is a summary of important guidelines outlined in this paper:

Guideline: do not use **defparams** in any Verilog designs.

Guideline: require all passing of parameters to be done using the new Verilog-2001 named parameter redefinition technique.

Guideline: only use macro definitions for identifiers that clearly require global definition of an identifier that will not be modified elsewhere in the design.

Guideline: where possible, place all macro definitions into one "**definitions.vh**" file and read the file first when compiling the design.

Alternate Guideline: place all macro definitions in the top-level testbench module and read this module first when compiling the design.

Guideline: do not use macro definitions to define constants that are local to a module.

Guideline: make clock cycle definitions using the **`define** compiler directive.

Guideline: place the clock cycle definitions in the "**definitions.vh**" file or in the top-level testbench.

Guideline: do not make state assignments using **`define** macro definitions for state names.

Guideline: Make state assignments using **parameters** with symbolic state names.

Guideline: To improve simulation efficiency, make all time_units of user defined **`timescales** equal to **1ns** or larger.

In his book Writing Testbenches, Functional Verification of HDL Models, Bergeron claims that VHDL and Verilog both have the same area under the learning curve[8]. Due to the misinformation that has been spread through numerous Verilog books and training courses, I am afraid Bergeron may be right. When Verilog is taught correctly, I believe the area under the Verilog learning

curve is much smaller (and Verilog simulations run much faster).

"Long live named parameter redefinition!"

"Death to **`defparams`**!"

17. References

[1] Accellera - SystemVerilog 3.0 Accellera's Extensions to Verilog/Draft 6. www.accellera.org (not publicly available at this time).

[2] Clifford E. Cummings, "State Machine Coding Styles for Synthesis," *SNUG'98 (Synopsys Users Group San Jose, CA, 1998) Proceedings*, March 1998. Also available at www.sunburst-design.com/papers

[3] Clifford E. Cummings, "Verilog-2001 Behavioral and Synthesis Enhancements," *Delivered at HDLCON-2001 but missed publication in the Proceedings*, March 2001. Available at www.sunburst-design.com/papers

[4] Donald Thomas, and Philip Moorby, The Verilog Hardware Description Language, Fourth Edition, Kluwer Academic Publishers, 1998, pg. 142. (re-ordering the parameter redefinition list)

[5] IEEE Standard Verilog Hardware Description Language, IEEE Computer Society, IEEE, New York, NY, IEEE Std 1364-2001,

[6] IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language, IEEE Computer Society, IEEE, New York, NY, IEEE Std 1364-1995

[7] IEEE P1364.1/D2.1 Draft Standard for Verilog Register Transfer Level Synthesis, http://www.eda.org/vlog-synth/drafts.html

[8] Janick Bergeron, Writing Testbenches, Functional Verification of HDL Models, Kluwer Academic Publishers, 2000, pg. xxi. (Verilog learning curve)

[9] Janick Bergeron, Writing Testbenches, Functional Verification of HDL Models, Kluwer Academic Publishers, 2000, pg. 265. (why people use defparam)

[10] Janick Bergeron, Writing Testbenches, Functional Verification of HDL Models, Kluwer Academic Publishers, 2000, pg. 267. (defparam file)

[11] Janick Bergeron, Writing Testbenches, Functional Verification of HDL Models, Kluwer Academic Publishers, 2000, pg. 340. (avoid `define *and* `undef all `defines)

[12] Janick Bergeron, Writing Testbenches, Functional Verification of HDL Models, Kluwer Academic Publishers, 2000, pg. 341. (parameter CYCLE)

[13] Lionel Bening, and Harry Foster, Principles of Verifiable RTL Design, Second Edition, Kluwer Academic Publishers, 2001, pg. 146. (recommendation to use `define for FSM state names)

[14] Lionel Bening, and Harry Foster, Principles of Verifiable RTL Design, Second Edition, Kluwer Academic Publishers, 2001, pg. 147. (tool implementation of parameters are more difficult than `define)

[15] Michael Keating, and Pierre Bricaud, Reuse Methodology Manual, Second Edition, Kluwer Academic Publishers, 1999, pg. 110. (recommendation to use `define for state names)

[16] Michael Keating, and Pierre Bricaud, Reuse Methodology Manual, Second Edition, Kluwer Academic Publishers, 1999, pg. 112. (example state machine design uses parameters)

[17] Pran Kurup, and Taher Abbasi, Logic Synthesis Using Synopsys, Second Edition, Kluwer Academic Publishers, 1997 (book not recommended!)

Revision 1.2 (May 2002) - What Changed?

The text before Example 16 incorrectly stated that "Changing the time_units to **`1ns`** changes the delay from 1.6ns to 16ns, introducing an error into the model." The corrected text reads, "Changing the time_units to **`1ns`** changes the delay from 160ps to 1.6ns, introducing an error into the model."

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 April 19th, 2002)