The Verilog Preprocessor: Force for `Good and `Evil

Wilson Snyder

Cavium Networks
Marlboro, MA

http://www.veripool.org/papers

Last updated 2010-08-25

ABSTRACT

Join an exploration of some fun and horrid usages of the Verilog Preprocessor, and learn best practices from them. Including: Good and bad message and assertion macros, using `line in generated code, the mystery of where comments land, and the localparam-vs-function-vs-define trade-offs.

We then consider metaprogramming with defines, to build `if, `for, lookup tables and hashes inside the preprocessor, and use includes as a template language.

We present vppreproc, an open-source 1800-2009 preprocessor, and how to leverage it for custom scripts. We conclude with a wrap up of vendor preprocessor compatibility.

Table of Contents

1. Introduction

Having implemented my own version of the Verilog preprocessor[1], I've seen some interesting techniques, some useful, some downright scary. In this paper we'll look at some of what I've learned, and explore some future techniques.

We begin by looking at the evolution of the preprocessor in the Verilog standard. We compare some of its features with the nearest equivelent, the C preprocessor, and consider the differences.

We then consider some best practices for using the preprocessor, starting with the need for parenthesis. We also look at side effects, stripping semicolons, dealing with comments, and namespace collisions.

From there we discuss alternatives to using the preprocessor, metacomments, localparams, constant functions, and custom scripting.

We then study some common applications, including error an assertion wrappers, using variable arguments, `line in code generators, and building identical modules, stubs and lists.

Then we look at vppreproc and Verilog::Preproc, open source tools that can perform preprocessing, and can be leveraged to enhance other command languages and user scripts.

From the practical we move to the more theoretical. In the Metaprogamming section, where we use the preprocessor to create definitions-in-definitions, lookup tables, and loops.

Finally we compliance test some of our code against several simulators and provide recommendations on which constructs are stable enough for use.

2. The Standards

The Verilog preprocessor was first standardized in Verilog 1364-1995, and has been improved with each subsequent standard. Reviewing some of the changes:

Verilog 1995:

```
`define MACRO
`ifdef   `else   `endif
`include
```

Verilog 2001:

```
`define MACRO(arg...)
`ifndef   `elsif   `undef
`line
```

SystemVerilog 2005:

```
``   `"   `\`"
```

SystemVerilog 2009:

[1] Four times in three languages. Sigh. Now at least it's down to one code base.

```
`define  MACRO(arg=default...)
`undefineall
`__FILE__   `__LINE__
```

It's notable that the preprocessor has gained many, though not all of the features present in the C preprocessor. Some differences:

- Verilog lacks an `if, and as such the preprocessor can't directly perform math. Math would allow more metaprogramming features, but it's probably worth leaving out to avoid suffering bugs similar to those in C, where "#ifdef"s are often mistyped for "#if", and the wrong code is compiled.
- Verilog lacks `error, and has no standard preprocessor error mechanism. This would be a nice addition, however it is easy to work around:

  ```
  `ifndef X
    `error_define_X_must_be_set
  `endif
  ```

  The compiler will helpfully print "`error_define_X_must_be_set is undefined". While this isn't a perfect message, it will do.
- Verilog lacks defines with a variable number of arguments, which for C was standardized recently in C99[2]. This would be an excellent addition to allow wrapping $display functions.
- Verilog adds default parameter values. There are cases where this is useful, however it remains to be seen how widely used and supported this will become.
- Verilog requires the ` in front of all macro calls. While some have proposed this be eliminated in Verilog 2012(ish), the ` provides major advantages I would hate to lose: the visual obviousness of the call – macro calls are text substitution, not function calls. And, we'd lose the ability to report use of undefined macros as such.

3. Well Known Best Practices - Learning from "C"

In this section we look at some well known best practices. Most of these are already known by Verilog experts, but merit review. As was just noted, the Verilog preprocessor is very similar to the C preprocessor, and many Verilog techniques directly correspond to C best practices.

Precedence

The precedence problem occurs when a define is hastily created as follows:

```
`define OR(x,y) x|y
```

The problem occurs when the macro is used with unintended consequences:

```
wire b = `OR(a,b) & c;  // BAD!! == a|b&c == a | (b&c)
wire b = `OR(a&b, c);   // BAD!! == a&b|c == a & (b|c)
```

[2] For C, which has been reasonably stable since 1978, a ten year old feature is "new."

Because macros are text substitutions, the compiler doesn't see the author's assumptions about precedence. The solution is to always parenthesize both the formal parameters and the `define function results:

```
`define OR(x,y) ((x)|(y))

wire b = `OR(a,b) & c;  // Good! == ((a)|(b)) & c
wire b = `OR(a&b, c);   // Good! == ((a&b)|(c))
```

Side Effects

The side effect problem occurs in SystemVerilog when a parameter is used more than once:

```
`define D(x) (f(x)?(x):1)

always ...
   b = `D(a++);   // BAD!! == f(a++) ? a++ : 1
```

The increment of A will occur twice. It's worth double-defending against this; when writing a macro don't use an argument twice, and when calling any macro never put side effects in the macro call. (This is one benefit of Verilog's requiring the ` in front of the macro call; it warns us of this hazard, which wouldn't be present in a normal function call.)

```
`define D(x) (verilog_function(x,1))

always ...
   temp = a++;
   b = `D(temp);
```

Semicolon Eater

Consider a macro that performs some function:

```
`define WARN(msg) \
    begin $write("Warn:"); $display(msg); end
```

We've added the begin/end to attempt to make it work correctly in an if block. However it's not complete; we would like users to put a semicolon at the end where this is used, since it's a function. However the semicolon causes problems with if statements:

```
if (…) `WARN("yes"); else `WARN("no");
```

This results in a syntax error as there's an extra semicolon after the expanded macro. The solution again matches C, wrap it in a do...while loop:

```
`define WARN(msg) \
    do begin $write("Warn:"); $display(msg); end while(0)
```

The while will eat the semicolon.

Comments in Define Values

The Verilog LRM (reference [1]) requires that block comments (/**/) inside define values become part of the defines, while // comments at the end of a define do not. If the comment represents a meta-comment, this can be a source of bugs. Be sure to use /**/ block comments for all meta-comments.

Namespace Collisions

The preprocessor namespace is global. Unfortunately if the same define has two different values (often because two authors have used the same name for different purposes in different files) bugs will result, and many tools do not report the conflict.

To avoid this problem there are three solutions. First, add a unique prefix, usually similar to the name of the file the include is within:

```
// file.v
`define FILE_DEFNAME 1'b0
```

Alternatively, for local defines, add a `undef after the define has been used.

Finally, the new `undefineall may be used at the bottom of every module to clear out the namespace.

Name Header Files With .vh

A file that is to be `included is generally quite different from a file that contains a module. Include files should not be linted, as they may require additional includes to exist before it can compile successfully. To make this distinction clear, headers should have a unique suffix, ".vh" being the most common. This also simplifies Makefiles, as they can now use *.vh or *.v wildcards.

Include Guards

Included files should generally contain header guards to prevent multiple inclusion, and to also greatly reduce the quantity of code that needs compliation. For more details see reference [2].

```
// file.vh
`ifndef _FILE_VH
 `define _FILE_VH 1

  ...

`endif // Guard
```

Include Required Headers

Code that requires an includes be present should always `include the file itself, rather than assume it is loaded from the simulator command line or earlier file. This makes it easier to lint or compile each file standalone. For more details see reference [2][1].

An interesting way to enforce this would be to use a `undefineall at the end of every module source file, however I have not seen this used in practice yet. This would also defeat part of the purpose of include guards, as the includes would be reread every module, after each `undefineall.

4. Alternatives to Defines

Metacomment Conditionals

Synopsys made the unfortunate choice with their first design compiler to not implement a preprocessor, and instead created the dreaded pragma:

```
// synopsys translate_off
```

This has since been inherited by other vendors:

```
// ambit translate_off
// synthesis translate_off
```

Unfortunately this is a horrid technique, as it's easy to have misbalanced pragmas, and it doesn't allow other tools to select if they want the translated regions. The better practice is to use

```
`ifndef SYNTHESIS
```

Where the SYNTHESIS define is automatically set by Synopsys DC and other synthesizers. For more details and a program to clean these up, see reference [2][1].

Localparams vs. Defines

Verilog 2001 added the "localparam" keyword to define magic numbers local to a module. Localparam is often suggested as a good replacement to `define for constant values, but there are downsides.

Localparams have the benefit of being module local. This reduces the chance that there will be a namespace conflict. However, this is also a downside; different modules can have localparam of the same name with different values, which can cause bugs or confusion to those reading the code.

Localparams have a big advantage in that bits can be selected from the value, which is almost critical for constants representing register addresses. This benefit is reduced with SystemVerilog 2009 which finally added the {}[] syntax. So we have two choices:

```
`define GLOBAL_VAL 22
... {`GLOBAL_VAL}[1:0]

localparam LOCAL_VAL = 22;
... LOCAL_VAL[1:0]
```

Localparam values are typically visible in waveform capture tools, while `define values are not. This makes localparams easier to use, although having a large number of localparams can make it hard to see other parameters.

Taken together, the case is not as clear for localparams. My personal preference is to continue to use defines for true system global constants (such as register addresses), and use localparams for truely module specific information (such as the number of arbitration states).

Constant Functions vs. Defines

One IP vendor used the preprocessor as follows:

```
`define LOG2(x)  (x<=1?0:  (x<=2?1:  (x<=4?2: (x<=8?3: (...))))

wire [`LOG2(WIDTH_PARAM)-1:0] address;
```

This is interesting in that it required the Verilog 2001 feature of formal define arguments, and used them to work around tools that omitted another Verilog 2001 feature: constant functions. The catch is the `LOG2 was also used in another similar define, which expanded to nearly 32KB

every time it was called, making for very slow compiles. Today, unless it is a very simple function, it's better to use a real constant function.

Custom Scripts vs. Defines

If defines start getting really ugly, perhaps it's time to move to generated code. Classic examples of code that may need a custom generator include CSRs, RAMs and pad rings. The main downside is portability; both OVM and VMM have probably long passed the point where a code generator would generate cleaner and faster code, but a generator would greatly limit their adoption.[3]

5. Applications

After our digression into the world of metaprogramming, let's return to solve some real problems, using code that will work on most tools.

Messaging Macros Showing Calling Line

A common task is to write a message reporting macro, and have it report the calling line number. This wasn't possible in a standard way (short of the PLI) until SystemVerilog 2009 finally added the `__FILE__ and `__LINE__ directives. Let's start with those

```
`define MSG(msg) \
   $write("[%0t] %s:%d: ",$time, `__FILE__, `__LINE__); \
   $display(msg);
```

Users should expect to be able to call it from IF statements, so we should wrap it in a begin/end; however as described previously do-while is a better wrapper. We may also want to disable this code for faster simulation (we do NOT need to disable it for synthesis; synthesis will ignore the $calls.) So our completed attempt is:

```
`ifdef MESSAGING
  `define MSG(msg) \
     do begin \
        $write("[%0t] %s:%d: ",$time,`__FILE__,`__LINE__); \
        $display(msg); \
     end while(0)
`else
  `define MSG(msg) \
     do begin end while(0)
`endif
```

Message Macros with Variable Arguments

It would be great if messaging macros could look like $display and be callable as follows:

```
`MSG("Format a=%x, b=%x",a,b);
```

Unfortunately variable arguments aren't allowed anywhere but in the PLI, so what would work in C doesn't work in Verilog:

```
`define MSG(fmt,...) $display(fmt, `` __VA_ARGS)  // NOT LEGAL
```

Hopefully the next LRM will allow this.

[3] To avoid requiring everyone to have the generator, the code can use metacomments which are expanded in-line. An example of this technique is Verilog-Mode for Emacs' /*AUTOs*/.

The first workaround is to require extra parenthesis:

```
`define MSG(fmt) $display[4] fmt
`MSG((“Format a=%x, b=%x”,a,b));
```

Another workaround is to use SystemVerilog 2009 default arguments. Unfortunately this isn't well supported yet:

```
`define NONE
`define MSG(fmt,a1=`NONE,a2=`NONE,a3=`NONE) \
     $display(fmt,a1,a2,a3)
`MSG(“Format a=%x, b=%x”,a,b);
```

If only fmt is provided this will call "$display(fmt,,,)" which should be acceptable.

My solution of choice is to admit defeat using the preprocessor alone, and use either the PLI, or a custom preprocessor, such as vpassert (which again is part of Verilog-Perl):

```
$msg(“Format a=%x, b=%x”,a,b);
```

This works with even Verilog 1995 simulators, and all synthesis tools, since they ignore $ calls.

Proper `lines in Generated Code

Every few months I write a new Verilog code generator. The simplest possible generator is probably:

```
$ cat template.v >generated.v
```

Of course "cat" would become something fancier[5] to do substitution, etc, or a script that uses one of the many templating toolkits. The problem comes when the simulator reports an error message:

```
%Error: File “generated.v” Line 222: Syntax error
```

Since in this example line 222 was copied verbatim from template.v, a more proper message would tell the user what file they need to edit, namely:

```
%Error: File “template.v” Line 222: Syntax error
```

The `line directive is how to accomplish this:

```
$ echo ‘`line 1 “template.v” 0’ > generated.v
$ cat template.v >> generated.v
```

The error will now point to template.v.

The best practice is to emulate tools such as "Flex"[6]; their generated output alternates between `line to point to the generated file for generated text, and `line to point to the template for verbatim text. IE:

---

[4] Note the first level macro is the only one with the problem of variable arguments. The first level macro can use the new SV 2009 $sformatf call to reduce the format and format arguments to a single string that can be passed to lower level macros or the PLI.

[5] Even a simple copy can be useful; if a script "rsync"s all your code from NFS to a local disk before compiling, you're likely to see a compile time speedup. That's a perfect example of the problem described too; you don't want errors to refer to "/local/foo.v" but "/nfs/my/files/foo.v".

```
`line 2 “generated.v” 0  // Line 2 is the next output line number
// This file GENERATED by EXAMPLE PROGRAM
Code here is generated

`line 1 “template.v” 0
This code verbatim from line 1 of the template.
This code verbatim from line 2 of the template.

`line 10 “generated.v” 0  // Line 10 is the next output line number
Code here is generated
```

Wrapping Assertions

Assertions are great, but often require wrapping. For example, you need to ifdef them out for tools which do not support assertions, such as some synthesis tools, and on failure you want to call a common PLI error function. Here is what a user might have put into their code:

```
`ifndef SYNTHESIS
   label : assert property (when) \
               else $our_error(“Assertion ‘label’ failed”);
`endif
```

Duplicate code is bad, so we’d like a macro. We’ll start by using stringification to make a function to convert the label into the error message:

```
`define OUR_ASSERT_MSG_LABELED(label) `”Assertion ‘label’ failed`”

`define our_assert_prop(label,when) \
    label : assert property (when) else \
         $our_error(`OUR_ASSERT_MSG_LABELED(label))
```

The user is expected to add the trailing semicolon when it is used. This presents a problem though; the assert is used under a module statement, and if we disable this assertion we’ll be left with a stray semicolon in the module, which isn’t legal. One solution is to create a temporary localparam to consume the semicolon, which gives our solution:

```
`define OUR_ASSERT_MSG_LABELED(label) `”Assertion ‘label’ failed`”

// Assertion with message reporting, used inside a module.
// Add trailing ; where the macro is used
`ifdef USE_ASSERTIONS
  `define our_assert_prop(label,when) \
     label : assert property (when) else \
         $our_error(`OUR_ASSERT_MSG_LABELED(label))
`else
  `define our_assert_prop(label,when) \
     localparam _unused_ok_assert_``label = 1’b0
`endif
```

This is undoubtedly hackish, but synthesizes, and even if assertions are disabled we will get an error if a label is duplicated, which is a common mistake. Other ideas are welcome.

Immediate assertions pose a similar problem:

---

[6] Flex is a lexical generator which reads a specification that contains “C” code and emits “C” code.

```
   always @(posedge clk) begin
      //...
`ifndef SYNTHESIS
     label : assert (when) \
                 else $our_error("Assertion 'label' failed");
`endif
   end
```

These can be similarly wrapped, but a "do while" provides the semicolon eater, since we're now inside a statement block where do-while is legal.

```
// Immediate assertion with message reporting, used inside a block.
// Add trailing ; where the macro is used
`if USE_ASSERTIONS
  `define our_assert_blk(label,when) \
     do begin label : assert (when) else \
          $our_error(`CN_ASSERT_MSG_LABELED(label)) \
     end while (0)
`else
  `define cn_assert_blk(label,arg) \
     do begin end while (0)
`endif
```

Building nearly identical modules

You're creating a module for reuse, but some ports are only used in one version.[7] "Generate" statements are excellent for creating generic modules, but can't remove ports. There's no need to duplicate code, however. Create two modules and a common body, as follows:

```
// mod1.v
`define MODCOMMON_NAME mod1
`define MODCOMMON_SMALLER
`include "mod_common.vh"
`undef MODCOMMON_NAME mod1
`undef MODCOMMON_SMALLER
     // The `undef could be in mod_common, but it is
     // easier to read in the same file as the `define

// mod2.v
`define MODCOMMON_NAME mod2
`include "mod_common.vh"
`undef MODCOMMON_NAME

// mod_common.vh
// Note no include guards here, we want to include it twice!
module `MODCOMMON_NAME (
   input both
`ifndef MODCOMMON_SMALLER
   , input only_in_mod2
`endif
   );
...
endmodule
```

[7] Normally one module would still suffice; I'd recommend that unused ports still be brought out and tied at the level above, or tied in a wrapper module. Sometimes this isn't acceptable for blocks that represent schematics or are synthesized standalone.

The principle here is multiple inclusion; where the defines customize what the include does.

*** Including the Same File Twice Provides a Templating Language ***

Create and Substitute "Stub Modules"

"Stub" or "Null" modules are a module with the same pinout as another module, but minimal functionality inside. They are useful to stub-out sections of code for linting, or to accelerate block level simulations.

A stub module is easily created with Verilog-Mode [4]:

```
module ModnameStub (
   /*AUTOINOUTMODULE("Modname")*/
   );
   /*AUTOWIRE*/
   /*AUTOREG*/
   /*AUTOTIEOFF*/
   wire _unused_ok = &{1'b0,
               /*AUTOUNUSED*/
               1'b0};
endmodule
```

Then to use it, the preprocessor comes into play:

```
`ifndef Modname
 `define Modname Modname
`endif
`Modname mod (.*);
```

We've made the define the same name as the module so that any Verilog-Mode /*AUTOINST*/ pragmas will work, even without `defining Modname. Verilog-Mode contains a useful default assumption that if `Modname is referenced, and there's no `define Modname, then Modname.v will contain the source code.
To use the stub, we simulate with:

```
+define+Modname=ModnameStub
```

Note we haven't ifdefed out the entire module, so the connectivity remains the same - we don't have to `ifdef drivers to avoid undriven nets. This it the great advantage of stubs.

Building lists with multiple inclusion

Imagine you have enumerations and functions related to instructions:

```
typedef enum {
  IOP_INST1 = 1,
  IOP_INST2 = 2,
  _IOP_MAX
} IopEnum;

function bit iopAlu(input IopEnum en);
  case (en)
    IOP_INST1: return 1;
    IOP_INST2: return 0;
  endcase
endfunction
```

It's going to be hard to maintain this code when the number of instructions gets large, and changes. If it's a big project, it may be a good idea to use a custom script to auto-generate this code. But this paper is about the preprocessor, which is well up to this job:

```
// instr.vh
//     Name, iop, alu
`INSTR(INST1, 1,  1)
`INSTR(INST2, 2,  0)

// main.v

typedef enum {
  // Creates lines of the form:
  //    IOP_INST1 = 1,
`define INSTR(name,iop,alu)  IOP_``name = iop,
`include "instr.vh"
`undef INSTR
  _IOP_MAX_
} IopEnum;

function bit iopAlu(input IopEnum en);
  case (en)
    // Creates lines of the form:
    //     IOP_INST1: return 1;
`define INSTR(name,iop,alu)  IOP_``name: return alu;
`include "instr.vh"
`undef INSTR
  endcase
endfunction
```

Perhaps you don't like having a separate instr.vh file? You can do it all in one file by using `if-defs to select which purpose the file is being used for, and include the file recursively.

6. Preprocessing For Free[8] – vppreproc

How do you debug preprocessor issues? Often the compiler error messages are enough, but if you want to see what's going on, it's easiest to look at the post-pre-processed output.[9] One way to do this is to use the open-sourced "vppreproc," part of the Verilog-Perl tool suite[3], written by myself and 95 contributors.

Using vppreproc

Vppreproc takes a command line similar to most Verilog simulators, and prints the output to screen:

```
$ vppreproc  +define+SYNTHESIS=1  +incdir+/my/include  myfile.v
```

The output from vppreproc can then be visually debugged, or fed into any simulator or other tool.[10] Options are available to select stripping of comments, white-space, and insertion of `line

[8] Free as in "Free Lunch at SNUG" – No cost, but feedback is welcome and benefits everyone.
[9] Some vendors have debug dumps that can help. VCS has the ability to dump each stage of macro expansion through its rawtokens utility; contact your Synopsys Application Consultant for details.
[10] Feeding vppreproc output into another tool is one way to work around vendor preprocessor bugs or feature limitations.

directives. The –dump-defines option will show the value of all `defines; great for decoding `defines wrapped in undecipherable levels of ifdefs.

Verilog::Preproc

Verilog-Perl also provides a pre-processing Perl package, Verilog::Preproc. This allows a Perl script to gain a preprocessor, almost as simply as replacing IO::File with Verilog::Preproc:

```
use Verilog::Preproc;

my $opt = new Verilog::Getopt();
$opt->parameter("+define+SYNTHEIS");

my $vp = Verilog::Preproc->new(options=>$Opt);
$vp->open($file);

while (defined (my $line = $vp->getline())) {
    print $fh $line;
}
```

Ambitious folks can also override the internal preprocessor methods to perform complicated tasks, such as custom "translate on/off" rules.

Also note the above use of the Verilog::Getopt package, which understands how to parse the standard simulator flags first standardized by Verilog-XL (-f -v -y +incdir +libext). Verilog::Getopt makes it simple to add these standard flags to any hackware solution. It can also locate files using the Verilog search path rules, another common hackware problem.

Preprocessing Command Files

Once you have access to a standalone preprocessor, it's easy to apply it to other application languages; pipe the command language through vppreproc (or Verilog::Preproc if it's a Perl script) and then into the original tool. This enables `define and `ifdef, and allows direct reuse of the +defines sent to the simulator in the newly pre-processed tool.

Preprocessing in Emacs

When debugging defines, it would be handy to see the pre-processed code quickly. Based on this paper the Verilog-Mode package for Emacs [4] added a key sequence, C-c C-p, that will preprocess the current buffer, leaving the output nicely syntax highlighted.

7. Metaprogramming

Metaprogramming is the writing of a program that writes other programs. For our purposes, we are interested in the use of Verilog to generate other Verilog code. To some extent, this is what the preprocessor is all about – code that generates code. This technique is also used in OVM and VMM to reduce code authored by the user; less code, less bugs.

Can we use the preprocessor to make a more generic language, one with math, associative arrays, and loops? [11] Yes, to some extent.

[11] If you didn't guess yet, this is the `evil part of the paper.

First, a warning, we're breaking some new ground. Many of the techniques here aren't supported on all tools, we're doing this just to make discoveries; see Chapter 8 for a rundown of what works and what doesn't. The code to implement these functions is available on request from the author.

Logical Operations, and Defines Defining Defines

Perhaps the simplest way to store information in the preprocessor is by the presence or absence of a definition. We'll define the presence of a definition as a "one" and the non-definition is a "zero". Then, by nesting `ifdefs we can perform logical operations. We'll begin with a NOT operator:

```
// if (!defined(a)) define D
`ifndef A
  `define D
`endif
```

But the goal here is to abstract away the mechanics so we can eventually build more complicated structures. So, wrapping this in a define:

```
// If !defined(a) then define <d> else undef <d>
`define BSV_DEFIF_DEFNOT(d,a) \
    `undef d \
    `ifndef a \
      `BSV_DEFINEIT(`define d 1) \
    `endif

// For wrapping defines
// Note the backslash newline on the `define line
// and the newline following, required to form the end of a define
`define BSV_DEFINEIT(d) d \
```

This uncovered an important trick; to make another `define become defined when the macro is called, we used the BSV_DEFINEIT macro, which must contain a terminating return to end the macro's definition value. The second level macro is required, to have the newline terminate the generated define, not be burried as part of the first's substitution text. This discovery deserves emphesis:

*** Defines Can Create Other Definitions at Substitution Time ***

Likewise we can create AND and OR:

```
// If defined(a)&&defined(b) then define <d> else undef <d>
`define BSV_DEFIF_DEFAND(d,a,b) \
    `undef d \
    `ifdef a `ifdef b \
      `BSV_DEFINEIT(`define d 1) \
    `endif `endif

// If defined(a)||defined(b) then define <d> else undef <d>
`define BSV_DEFIF_DEFOR(d,a,b) \
    `undef d \
    `ifdef a `BSV_DEFINEIT(`define d 1) `endif \
    `ifdef b `BSV_DEFINEIT(`define d 1) `endif
```

We can then use them:

```
`BSV_DEFIF_DEFNOT(SIMULATION, SYNTHESIS)  // SIMULATION = !SYNTHESIS
`BSV_DEFIF_DEFOR(SIM_OR_LINT, SIMULATION, LINT)
`ifdef SIM_OR_LINT
    ...
`endif
```

It's not worth the obfuscation for single use, but when there's thousands of similar definitions, defines that create defines are worth considering. This also provides great power; the ability to pass data between one macro substitution and a later one.

Preprocessor Math, and Lookup Tables

The C preprocessor provides the #if directive. Can this be emulated in Verilog?

First, consider the C preprocessor "defined()" function that returns 1 if a define is defined, otherwise 0. This and its complement are easy:

```
// If  defined(<d>) then 1 else 0
`define BSV_DEFINED(d)  `ifdef d 1 `else 0 `endif

// If !defined(<d>) then 1 else 0
`define BSV_NDEFINED(d) `ifndef d 1 `else 0 `endif
```

How about the converse, converting a 0 to nothing, and a 1 to make a definition?

```
// If <n>==0 then define <d>, else undef <d>
`define BSV_DEFIF_Z(d,n) \
   `undef d \
   `ifdef _BSV_DEFIF_Z_``n \
      `BSV_DEFINEIT(`define d 1) \
   `endif

`define _BSV_DEFIF_Z_0(d) d
```

This uses a very important discovery – the preprocessor itself can read lookup tables. It's important, so again:

*** The Preprocessor can Read and Create Lookup Tables ***

It may be confusing to understand what's going on here, the key is the `` concatenation creates a new define name (either _BSV_DEFIF_Z_0 or _BSV_DEFIF_Z_1) that may be tested in the ifdef. Only _BSV_DEFIF_Z_0 is defined, so only a zero input will process the `ifdef body.

We continue with the complementary function:

```
// If <n>!=0 then define <d>, else undef <d>
`define BSV_DEFIF_NZ(d,n) \
   `undef d \
   `ifndef _BSV_DEFIF_Z_``n `BSV_DEFINEIT(`define d 1) `endif
```

The same idea can create all the logical operations:

```
// Logical NOT: If <n>!=0 then return 1 else 0
`define BSV_Z(n) `BSV_DEFINED(_BSV_Z_``n)
`define _BSV_Z_0 1

// Logical NOT: If <n>!=0 then return 1 else 0
`define BSV_NZ(n) `BSV_NDEFINED(_BSV_Z_``n)

// Logical AND: if <a>&&<b> then 1 else 0
`define BSV_AND(a,b) `_BSV_AND_``a``_``b
`define _BSV_AND_0_0 0
`define _BSV_AND_0_1 0
`define _BSV_AND_1_0 0
`define _BSV_AND_1_1 1
```

Addition, Subtraction and Comparison operations can be similarly created, and are available in the archive. The ugliness here is you need a define for each possible input value. It's thus reasonable to support integers (say) 0..63, but beyond that the code gets insanely long.

The above code read from a lookup table, but you can also create lookup tables, for example we can create associative array functions:

```
// Read hash: hash[key]
`define BSV_HASH_GET(hash,key) `hash``key

// Store hash: hash[key] = v
`define BSV_HASH_PUT(hash,key,v) \
    `BSV_DEFINEIT(`define hash``key v )

// Hash exists: defined(hash[key])
`define BSV_HASH_EXISTS(hash,key) `BSV_DEFINED(hash``key)
```

But... No Nesting

There's one huge catch with this arithmetic stuff, however. They won't nest. We'd like to use them like this:

```
// Less than: if <a> < <b> then 1 else 0

`BSV_DEFIF_NZ(FLAG, `BSV_OR(`BSV_DEFINED(SIMULATION),
                            `BSV_DEFINED(LINT)))
`ifdef FLAG ...
```

The problem is the order in which `` is expanded. Here's the simplified problem:

```
`define ZERO_0 0
`define NOTNOT(a) a
`define VALUE(a) `ZERO_```NOTNOT(a)

`VALUE(0)
   -> 0   // Would like this, but fails
```

The LRM is silent as to if the `NOTNOT gets expanded before or after the ``. We need it to be expanded before the `` pastes the result. Unfortunately the one simulator that works with most of the examples didn't make this choice; it throws a "`ZERO_ undefined" error because the `NOT-

NOT adds an additional space when substituted. This has been sent to the SystemVerilog committee, and hopefully the LRM will clarify this to allow it in the future.[12]

Repetition

Verilog coders often wish for a `for preprocessor statement. Though these cries have been greatly decreased since "generate" was introduced in Verilog 2001, `for can still be useful.

Using the lookup table technique just discussed, you can create a macro that repeats its arguments a specified number of times:

```
`BSV_REPEAT(4, "hello")
    ->  "hello" "hello" "hello" "hello"
```

Here's the magic to support n=0 to 5:

```
/// For <n> times (0..9), repeat <d>
`define BSV_REPEAT(n, d) `_BSV_REPEAT_``n(d)
`define _BSV_REPEAT_0(d)
`define _BSV_REPEAT_1(d) d
`define _BSV_REPEAT_2(d) `_BSV_REPEAT_1(d)d
`define _BSV_REPEAT_3(d) `_BSV_REPEAT_2(d)d
`define _BSV_REPEAT_4(d) `_BSV_REPEAT_3(d)d
`define _BSV_REPEAT_5(d) `_BSV_REPEAT_4(d)d
```

As usual, we have to hardcode for the maximum possible number of repetitions we will support.[13]

8. Compliance

The preprocessor is reasonably well supported across the different CAD vendors, however there are some notable differences and hazards, several of which I discovered when writing this paper. As usual, the more common a construct is, the better the support.

Feature	VCS 2009-12	Other1	Other2	Suggest
`define D(ARG) Defines with arguments is Verilog 2001.	ok	ok	ok	use
`define D(ARG=DEF) Define formal argument defaults is SystemVerilog 2009.	error	error	error	maybe
`line The other2 vendor added it after request.	ok	ok	error	use

[12] One interesting possibility would be to specify the new preprocessor operators `( and `) to allow the macro to specify the order of expansion.

[13] It's possible to make a "factory" macro that will automatically make the needed repeat macros, as they are required. Unfortunately, the factory needs subtraction. The subtraction can be done with another lookup table, but then we're still stuck with having to hardcode the maximum value for that subtraction table.

`undefineall Added to SystemVerilog 2009.	error	ok	error	maybe
`"d`" in arbitrary text Stringification is only specified to work inside macro text. For compatibility, wrap in a define, as shown below.	ok	ok	error	wrap in macro
`define STRINGIFY(d) `"d`" `STRINGIFY(hello) Stringification in standalone macro.	ok	ok	ok	use
`define DOINC `include "y" `DOINC Include delayed until define substituted.	ok	ok	ok	use
`define FILENAME "x.vh" `include `FILENAME The LRM specifies this case as legal.	ok	ok	ok	use
`define STRIFY_VH(d) `"d.vh`" `include `STRINGIFY(filebase) Stringification can compute an include filename.	ok	ok	ok	use
`define IF(d,v) `ifdef d v `endif	ok	ok	ok	use
`define DEF(d) d \  `define DEF2 DODEF(`define DEF 1) Note the newline is required in the DEF macro; see the text.	ok	error	ok	maybe
`define DEFINED `define INDIRECT(d) d `ifdef `INDIRECT(DEFINED) ... Perhaps obscure, but required for metaprogramming.	ok	ok	error	maybe
`define REPEAT(n,v) \ `REPEAT_``n(v) `define REPEAT_1(d) d `define REPEAT_2(d) `REPEAT_1(d)d `REPEAT(2,"v ") Might be legal; hopefully to be clarified in next LRM.	ok	error	error	maybe
`define REPEAT(n,v) \ `REPEAT_``n(v) `define REPEAT_1(d) d `define REPEAT_2(d) `REPEAT_1(d)d `define FUNC(n) n `REPEAT(`FUNC(2),"v ") Desirable; hopefully to be clarified in next LRM.	error	error	error	future

`define THRU(d) d `define MSG(m) `"m`" $display(`MSG(`THRU(hello))) Simulators give conflicting results, some give "hello" and some give "`thru(hello)". The former is probably more useful. The correct behaviour is undefined, the specification says only to "expand embedded macros."	`thru	hello	hello	future
`define MULTILINE line1 \ line2 `define MSG(m) `"m`" $display(`MSG(`MULTILINE)) Stringification of a multiline macro may result in an error, because the quoted result has a newline without a backslash escape. The standard should probably specify that newlines be converted to spaces.	error	ok	error	future

:

9. Conclusions

In this paper, we've shown several uses of the preprocessor, some good, some evil, and most in between. To summarize the best practices:

- Parenthesize define formals and outputs.
- Use each define formals only once to avoid side effects.
- Don't pass side effects into a macro.
- Wrap task macros in do-while.
- Beware comments inside defines.
- Prefix all defines to prevent name colisions.
- Undefine local defines at the end of files.
- Name header files with .vh.
- Guard your includes.
- Make all needed includes referenced by each module.
- Use `ifdef, not metacomments.
- Avoid defines where a constant function will do.
- Use `line in code generators.

And we introduced some power-user techniques:

- Defines can create other definitions at substitution time.
- The preprocessor can read and create lookup tables.
- Including the same file twice provides a templating language.
- Repeat and `if can be emulated, in some cases.

And along the path, we've uncovered several tool bugs, and LRM feature suggestions.

Finally, vppreproc provides an easy way to see the preprocessed output, and to integrate a preprocessor into existing scripts.
With these, the preprocessor can be a force to simplify code and reduce bugs, not horrify.

10. References

[1] IEEE 1800-2009 Standard for SystemVerilog. 2009 IEEE.

[2] Wilson Snyder. Ten IP Edits I To Most Code (and wish I didn't have to). SNUG Boston 2007. http://www.veripool.org/papers/TenIPEdits_SNUGBos07_paper.pdf

[3] Verilog-Perl. http://www.veripool.org/verilog-perl

[4] Verilog-Mode. http://www.veripool.org/verilog-mode

[5] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of C pre-processor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002.

[6] David Abrahams, and Aleksey Gurtovoy. C++ Template Metaprogramming, Appendix A: An Introduction to Preprocessor Metaprogramming. 2005 Pearson Education Inc.

11. Copyright