The Ten Edits I Make Against Most IP (And Wish I Didn't Have To)

SNUG Boston, September 2007.

Wilson Snyder

SiCortex, Inc.

wsnyder@wsnyder.org

ABSTRACT

This paper discussed ten specific edits I often make when integrating Verilog RTL IP. The edits tended to fall into two major categories; avoiding any requirements on an input.vc file, and being able to lint any module stand-alone. With only a small amount of time, and the tools mentioned herein, IP vendors could easily fix these issues and aid future customer integration.

Table of Contents

1.0 Introduction

In this paper I discuss specific edits I often need to make after receiving Verilog RTL from IP vendors. Though many of these edits correspond to commonly known Verilog or C++ coding techniques, most code I've received continues to need these repairs.

I've tried to limit this paper to the most common problems I've seen. If you're writing new code, I'd suggest looking at some of the papers at Verilog Hints and Tips (http://www.veripool.com/papers.html).

2.0 The Edits

2.1 I de-'do'.

What is wrong with this code?

```
module buffer (input do, output di);
```

It's a syntax error on all SystemVerilog compilers. That's because "do" is a new reserved word.

Now, as my compiler is recent, I could use the new Verilog 2005 feature `begin_keywords to get around it, but that's an incomplete solution, as it would not allow a SystemVerilog module to instantiate this module. Besides, if I'm going to edit to add `begin_keywords, it's only another minute to fix the real problem - rename the pin!

How do I find them?

I simply turn on SystemVerilog in my linter or simulator and look for the errors. Alternatively, a simple grep can turn them up. To be safe it is a good idea to grep against the complete list of new keywords that can be found in the SystemVerilog spec, although I've personally only seen "bit", "byte", "do", "int" or "ref" be a problem.

How do I automate repairing them?

The vrename script that is part of Verilog-Perl (http://www.veripool.com/verilog-perl.html) makes quick work of fixing these. I use the `--keyword` switch so vrename itself will let me edit the keywords that are really symbols, otherwise vrename won't let them be changed.

2.2 I split library files into one file per module.

Many vendors ship us a single huge Verilog file, which they expect to include be in our simulator as a library with the -v flag.

This requires the library name to be added to all of our input.vc files, even if the library won't even be needed for some parts of the design. So what? Well, reading extra libraries takes time, which matters for interactive programs like linters. Some tools also complain about unused

libraries, as they should, since there's no reason to have to read files that are never needed! Worse, you can't lint a single module in a library, it's the whole file or nothing. Finally, libraries are harder to read; it's painful to locate what file creates a given module; you need to search every library just to find the code.

Instead I split the library into one module per file, and make sure the module name matches the filename. Now I can just add to my input.vc:

```
+libext+.v  -v {dir} +incdir+{dir}
```

With that, if and only if an upper level module needs a cell will that cell's file and module be loaded. It's also now trivial to lint each module one at a time, we just lint the file containing it.

How do I find them?

Most lint tools will check the one module per file rule. If not, just grep for "module" and make sure there's only one module keyword in each file. Also verify the filename matches the module name.

How do I automate repairing them?

I use the vsplitmodule script that is part of Verilog-Perl (http://www.veripool.com/verilog-perl.html). This will read a library file and output multiple files, one for each module.

2.3 I add a common prefix to every module name.

Many vendors think they can use any module name they desire. This would be a fine practice if the names were randomly selected, but alas, engineers tend to invent the same names. One project I had three vendors that all decided that they deserved to own the module name "inv", but of course each version was slightly different with incompatible pin names! Please!

I rename every module, and therefore filename, to have a common prefix that is something unique to the vendor, or piece of IP (as chips may have multiple IP from the same vendor.) This prevents namespace conflicts, and makes it far easier given a error message that lists a module name to determine what vendor that module came from. If that seems a trivial problem, consider my last project had 2,500 modules. Without a naming convention, my brain's not capable of associating 2,500 unique names each to the corresponding IP vendor.

Of course, it’s also important that include files and any other files that filter into the design database are likewise prefixed.

How do I find them?

I simply look at the directory listing to make sure there's a common prefix on all of the .v files.

How do I automate repairing them?

I use the vrename script that is part of Verilog-Perl (http://www.veripool.com/verilog-perl.html). This only works for soft IP, but generally vendors creating hard IP seem to know better.

2.4 I convert translate_ pragmas to ifdefs.

How many files have you seen littered with synthesis on/off pragmas?

```
/* synthesis translate_off */
if (y===1'bx) $display("Something rotten is happening...");
/* synthesis translate_on */
```

Sigh. The translate off stuff is all pain caused by Synopsys being too lazy back in 1990 to add a hundred lines of code to implement the Verilog preprocessor. They fixed it a decade ago, so get over it!

The problem here is it's up to me to decide what code I want to simulate, or lint, or formally verify. It's not up to the IP vendor. What the vendor should have done was:

```
`ifdef MY_SYNTHESIS  // MY_ should be the IP vendor's prefix; see above
   if (y===1'bx) $display("Something rotten is happening...");
`endif
```

With the ifdef, by setting the MY_SYNTHESIS define I can determine what code is simulated or not. I can also lint both the synthesis and non-synthesis versions by linting twice, once with MY_SYNTHESIS defined, and once without.

Another advantage is the preprocessor will insure that the nesting is correct; while you can forget a translate_on, you can't forget a `endif without an error. One of my previous employers had a bug that made it into silicon that would have been easily avoided had they used this technique and caught a mismatched translate_off.

I should also note that in many cases where people insert translate offs they aren't even needed, and shouldn't be there. Synthesizers will **ignore PLI calls**, and will rip out logic that isn't needed. Thus there's no reason to ifdef out most simple assertions that use if statements, such as:

```
wire _internal_check = a|b|c;
always @* if (_internal_check !== slow_signal) begin
   $display("Internal mismatch....");
   $stop;
end
```

Avoiding turning off synthesis in these cases makes the code much easier to read, and can prevent simulator-synthesis mismatch bugs.

How do I find them?

I grep for the synthesis pragmas.

How do I automate repairing them?

Again, I use the vsplitmodule script that is part of Verilog-Perl. Vsplitmodule will globally replace the pragmas with ifdefs.

2.5 I add header guards.

Verilog files should be self-compliable, that is, I should be able to pick any module in the design, connect up the pins in a test bench, and compile. The compiler should pick up any submodules, and it should compile without warnings. Here's an example where this won't work:

```
// FILE: my_top.v:
`define MY_FOOGLE_WIDTH   32'h64
module my_top (
    my_modulea my_modulea (...);

// FILE: my_modulea.v:
module my_modulea (
   input [`MY_FOOGLE_WIDTH-1:0] ...

// FILE: my_moduleb.v:
module my_moduleb (
   input [`MY_FOOGLE_WIDTH-1:0] ...
```

The problem here is I can't lint my_modulea.v or my_moduleb.v standalone, because they won't know the value of MY_FOOGLE_WIDTH. I can solve this by moving the defines into a common file, and `including that file in every design, but I'd quickly find that I get errors about `defines being redefined.

Our C language friends long ago solved this problem. Standard practice in C is to have "header guards" around all headers, and to specify all needed headers in **each** source file. We need them in Verilog too!

Header guards allow **all** source files to include every headers they need, and let the compiler sort out which are redundant. This removes the requirement that the user to have a specific input.vc, something an IP vendor should never expect.

Using the example above, I change each file to include the common define files:

```
// FILE: my_moduleb.v:
`include "my_defines.v"
module my_moduleb (
   input [`MY_FOOGLE_WIDTH-1:0] ...
```

Then edit the defines file to add the header guards:

```
// FILE: my_defines.v:
`ifndef _MY_DEFINES_V
 `define _MY_DEFINES_V 1
 `define MY_FOOGLE_WIDTH   32'h64
 ...
`endif // Guard
```

In case this is unfamiliar I'll note the standard practice is to have name of the define be the filename upper-cased, plus a leading underscore. The trick here is the first time my_defines.v is included, _MY_DEFINES_V will not be defined, and the body of the file will be preprocessed. The second time my_defines.v is included, _MY_DEFINES_V will be defined, and the body will be skipped, preventing duplicate definitions.

With Verilog, there are two times you won't want the header guard: around a include file containing a `timescale, and in a include file that contains parameters. Both of these are cases where the include needs to be substituted into the includer's file multiple times, and thus any include should be used only for those unique purposes, never shared. That is, an include file should contains a single `timescale, or a list of parameters, or defines, never a mix of the three.

Also on the topic of includes, I never use directory names in the include filename, as that presupposes a directory structure. Instead I only specify the basename and use +incdir+ in my input.vc to select which directory the include exists in. This also matches standard C/C++ practices.

How do I find them?

I lint each file independently.

2.6 I add a common include to all modules.

There's often a define I need to set when compiling a vendor's modules. How to do this? You're probably guessing my answer isn't to add a series of defines to input.vc.

Instead, I make sure every file has at least two includes before the module statement.

```
`include "my_header.v"       // my_ is the IP vendor's prefix
`include "my_timescale.v"
module my_x (...);
```

This creates a common include header to provide a place to allow lint offs or other common defines that is needed for every module for this IP. Now I can simply add that code to my_header.v once, and be done with it.

Likewise just before each module should be an include of a file that contains only a single statement, the timescale. This makes it easy to change the timescale for the entire IP block in one easy edit. (This include does not get a header guard, as it needs to insert `timescale in every time it is included.)

How do I find them?

I grep for timescale and replace it with the include. Likewise, I grep for module and insert the `includes immediately above.

How do I automate repairing them?

I use the vsplitmodule script that is again part of Verilog-Perl. Vsplitmodule will globally replace a `timescale with a include statement.

2.7 I make every module lint clean(ish)

I don't require lint (or compiles) to only be run on the top of the design. I want to lint each module standalone!

Linting low level modules makes it easy to discover what problems are in the design; instead of thousands of warnings, you can lint from bottom-to-top, fixing warnings as you go. If the IP will be edited this is also invaluable, as I can simply edit a single module and lint from that point down. If you are using Verilog-Mode for Emacs (http://www.veripool.com/verilog-mode.html), linting requires only two keystrokes, `C-c C-s`.

How do I find them?

I create a bottom-up sorted list of all of the modules using vhier (again part of Verilog-Perl), and then edit that into a bash script that runs lint on each module. When I have all of the lint warnings for a module, I add the pragmas to the source to turn off the warnings, or get the vendor to clean up the issues themselves.

2.8 I convert casex to casez.

Cliff Cummings wrote an excellent paper including this topic, which I don't need to repeat here: Cliff's Full Parallel Case paper (http://www.sunburst-design.com/papers/CummingsSNUG1999Boston_FullParallelCase_rev1_1.pdf).

To summarize, the problem is Xs propagating through the design will cause simulation mismatches due to the casex statement. To avoid this, I replace "casex" with "casez", and "x" with "?" in the case values. It's that simple.

How do I find them?

I grep for casex.

How do I automate repairing them?

I ask the vendor to fix it, as it's the proper technique, and compatible with all versions of Verilog.

2.9 I fix width violations.

Width warnings from lint tools are there for our own good. While I could ignore them, the warnings often indicate a design problem, such as a control signal being a different width then a constant being comparing against.

First, consider constants. In Verilog 1995, an un-sized value is extended only to 32 bits:

```
wire [63:0] extend_x = 'bx;
```

The value of extend_x will be 64'h00000000_xxxxxxxx. Verilog 2001 fixes x/z extension of constants beyond 32 bits, but such constructs continue to rightfully result in lint warnings. Thus it's always good practice to specify the exact width of all constants. This can take the long form:

```
wire [63:0] extend_x = 64'bxxxxxxxx_xxxxxxxx;
```

It's easy to miss an X if you do that, so I think it's clearer instead to use a replication. Replication is also exactly what is needed when the width is variable, such as when it's determined by a define or parameter:

```
wire [`WIDTH-1:0] extend_x = {`WIDTH{1'bx}};
```

SystemVerilog also allows an unsized constant with '0. I think that's a fine thing to use when you really don't care about the width of the signal being assigned to; for example when resetting a bus. If you're resetting a FSM though, you'd be better off using a specifically sized IDLE define, parameter, or enumeration.

The second most common width violation is a missing width for a constant increment:

```
wire [15:0] original = ...;
wire [15:0] plusone = original + 1;  // Should be "+16'h1"
```

In addition to preventing the warning, specifying the width makes it explicit we're throwing away the carry.

Omitting widths can cause simulation problems; the classic example is from the Verilog specification:

```
wire [15:0] a, b;
wire [15:0] answer = (a+b) >> 1;
```

The answer will not include the carry, because a+b is a 16 bit value excluding the carry, and thus answer[15] will always be zero. The spec suggests adding 0, but that is stupid, as it simply hides the problem until the width of a or b exceeds 32 bits. I believe correct solution when you need a unsigned carry is to explicitly increase the width of the operands by one bit:

```
reg [15:0] a, b, answer;
always @* answer = ({1'b0,a} + {1'b0,b}) >> 1;
```

Concatenating a leading 1'b0 is the most common edit I need to make to fix width violations.

How do I find them?

Lint tools will report width violations. I'm partial to the Verilator (http://www.veripool.com/verilator.html) width warnings, as Verilator only warns about true mismatches - for example it takes 0 as an any-width zero; many tools will complain if you don't size zero, which is a bit silly in most cases. (Although zero must be sized when part of a concatenation; Verilator knows those rules and will warn appropriately.)

How do I automate repairing them?

I ask the IP vendor for fixes, add lint ignores for them, or make reasonable fixes manually myself.

2.10 I add unused_ok's

I've run lint, and get back a list of 50 signals with no sinks that presumably aren't needed. Are these signals ok, or did I misconnect or misconfigure something? It's a lot of work to find out.

The solution is to avoid the warnings when signals are not used. The technique I prefer is to create a signal with the words "unused" in it. Then, I know any unused warning on a signal name containing "unused" is acceptable. From there, I can either have a tool filter out the warnings, or add the appropriate lint pragmas to the source code to suppress the warnings.

Here's an example that specifies that the signal “nosink1” and bit 3 of “nosink2” will not be used:

```
wire _unused_ok = &{1'b0,
                    nosink1,
                    nosink2[3],
                    ...,
                    1'b0};
```

Several tricks are used here; first the reduction AND allows me to add any number of signals without needing to change the width of the _unused_ok signal. Second, the signals are listed one-per-line so it’s trivial to cut and paste them in, this also reduces version control system merge conflicts. Third, AND-ing with the constant 1'b0 forces _unused_ok to always output a zero, this will prevent the signal from toggling and increasing the size of wave files; furthermore, decent compilers will constant-propagate away the signal so it will not consume simulation time. Finally, as noted in an earlier item, we don't need to turn off synthesis around this code: since the _unused_ok is unused itself, the synthesis tool will rip out this logic, and all of the logic feeding it.

How do I find them?

Lint tools will report on unused nets.

How do I automate repairing them?

I take the unused nets from the lint report and add the signals mentioned to the _unused_ok equation, and put the appropriate lint ignore around it. I then feed the changes back to the IP vendor.

2.11 Extra Credit: I add Verilog-Mode AUTOs

Most of the time, I'm not going to be editing the IP I've received, beyond the automated minor fixed discussed here. However, occasional edits are unavoidable, especially using IP from OpenCores. When I'm going to make a slew of edits, I want the design to have minimal redundant information, and to be easy to connect hierarchically. To this end, I always use the AUTOs in Verilog-Mode (http://www.veripool.com/verilog-mode.html).

The nice thing about the AUTOs is that the code remains 100% compatible Verilog code; every Verilog tool still knows how to read it, but it's far easier to understand and maintain. The AUTOs could fill an entire SNUG paper by themselves, in fact they have! See the Veritedium paper (http://www.veripool.com/verilog-mode_veritedium.html).

Vendors that come with the AUTOs already inserted also get extra credit in our evaluation process, as it's an indicator that they're interested in maintainability and efficiency.

How do I automate repairing them?

Verilog-mode has an auto-inject feature, visit the module in Emacs and just type `C-c C-z`. I then make additional simplifications by hand. I also sometimes use Verilog-mode to automatically re-indent the file, if it wasn't indented consistently.

3.0 Conclusion

By applying the above edits, the code is now runnable in SystemVerilog. The files can be found by using an input.vc file search path, enabling easier substitution of IP versions, and compiling at any level. Finally, the code is lintable on a per-file basis, making it far easier to maintain.

4.0 Copyright