Simple and Correct Methodology for Verilog Include Files

22-Sep-2008

Too many times I have seen the same problem with the methodology related to the handling of Verilog include files.

The simplest and correct methodology is identical to the one used for software development. (There are some further details at: http://en.wikipedia.org/wiki/Header_file).

The simplest include file would look like:

```
// my_incl.vh

// If we have not included file before,
// this symbol _my_incl_vh_ is not defined.

`ifndef _my_incl_vh_
`define _my_incl_vh_


// Start of include contents

`define N 4

// Use parentheses to mitigate any undesired operator precedence issues
`define M (`N << 2)

`endif  //_my_incl_vh_
```

Then, in any module where we need these definitions:

```
// top.v

`include "my_incl.vh"

module top(input clk, input [`N-1:0] in1, output [`M-1:0] q);
  m1 u1(.clk(clk), .in1(in1), .q(q));
  //...
endmodule
```

Since the defines in `my_incl.vh` are put into a global namespace, it makes sense never to include or redefine those again.

```
// my_incl.vh

`ifndef _my_incl_vh_
`define _my_incl_vh_
...
`endif  //_my_incl_vh_
```

The `ifndef/`endif clause prevents redefinition (or inclusion) of the file's contents (if this same file was already included earlier).

For example, another file m1.v also requires the N and M definitions, so the source for m1.v is:

```
// m1.v

`include "my_incl.vh"

module m1(input clk, input [`N-1:0] in1, output reg [`M-1:0] q);
  always @(posedge clk) q <= {4{in1}};
endmodule
```

There is one case where you do need to re-include files; that would be for function and task definitions, since these are defined within module scope. The example below demonstrates this situation.

Some examples to play with

The SourceForce project v2kparse has an example utility analyze which can be used to do *very quick* analysis of Verilog files. This analysis is limited to:

- whether all referenced modules (I.e., instances) are defined in the source file set
- all include search-paths are correctly defined
- no duplicate `defines are done
- no syntax errors (for implementation) are present
    - supports Verilog IEEE Std 1364™-2005

We can use this utility with some (simplified) examples to demonstrate more of the include principles.

To download and install:

1. download the latest release.
2. create a directory and install software and examples:

```
> mkdir v2kparse
# x.y is release, as in 1.5, 1.6, ...
> cat v2kparse-x.y.tar.gz | (cd v2kparse; tar zxf -)
> cd v2kparse
```

3. Verify you have jruby and java/jre (at least version 1.6) installed:

```
> jruby -v
ruby 1.8.6 (2008-05-30 rev 6360) [x86-jruby1.1]
> java -version
java version "1.6.0_07"
Java(TM) SE Runtime Environment (build 1.6.0_07-b06)
Java HotSpot(TM) Client VM (build 10.0-b23, mixed mode, sharing)
```

Verify install:

```
> ./bin/analyze
Usage: analyze (--tcl out.tcl)? (--rb out.rb)? (--outf out.f)?
               --only_used? --exit_on_err? (--verbose n)? --abs_paths?
               (--redefn n)? (-E -C?)?
               topModule vlogOpts+
...
```

Once installed, we can proceed to work through the examples.

Example 1

In this example, we'll start with the use of an include file:

```
#From the install directory
> cd data/tc6
> ../../bin/analyze top -f e1.f

Info : ./e1/top.v:2: include file "./e1/my_incl.vh".  (INCL-1)
Info : ./e1/m1.v:2: include file "./e1/my_incl.vh".  (INCL-1)
Info : Link status: true.  (LINK-2)
```

The analyze utility returns a `Link status: true` since it was able to find definitions for all the modules instanced within the (*topModule)* `top` (1st argument to `analyze`).

You should examine the files under the `e1` directory.

Experiment with the options:

- The `-E` option will dump the preprocessed files:

```
> ../../bin/analyze top -f e1.f -E

Info : "./e1/top.v.E": creating pre-processed...  (VPP-1)
Info : ./e1/top.v:2: include file "./e1/my_incl.vh".  (INCL-1)
Info : "./e1/m1.v.E": creating pre-processed...  (VPP-1)
Info : ./e1/m1.v:2: include file "./e1/my_incl.vh".  (INCL-1)
Info : Link status: true.  (LINK-2)

#View using vi/view
> view -c 'set syntax=verilog' e1/*.E
...
```

- Experiment with the `--tcl`, `--rb` and `--outf` options. These are intended to expand a terse `.f` into a flat list of the actual files, include directories and defines used to analyze and link the *topModule*.
- Experiment with the `--verbose 2` option. This enables (most) verbose messages indicating:
  - the unresolved status/modules
  - the actual arguments passed to the underlying parser

during each (successive) link stage.

Example 2

In this example, the file `e2/m1.v` redefines `N`:

```
> ../../bin/analyze top -f e2.f
Info : ./e1/top.v:2: include file "./e1/my_incl.vh".  (INCL-1)
Info : ./e2/m1.v:2: include file "./e1/my_incl.vh".  (INCL-1)
Warn : ./e2/m1.v:4: macro 'N' redefined.
       First defined at ./e1/my_incl.vh:11.  (MACRO-3)
Info : Link status: true.  (LINK-2)
```

Looking at these 2 lines (which define `N`):

```
> egrep -n 'define[ ]+N ' ./e2/m1.v ./e1/my_incl.vh
./e2/m1.v:4:`define N 4
./e1/my_incl.vh:11:`define N 4
```

Hmmm, they both define the same value!

There is an option to `analyze` which sets the type of macro redefinition check:

```
> ../../bin/analyze

...
  --redefn n    : Level of macro redefinition check. "n" is:
                    1 -- checks only for different value.
                    2 -- (default) checks for different file+line.
```

Thus, if you expect multiple redefinition (file locations), and just want to check if any redefinition is a different value, then:

```
> ../../bin/analyze top -f e2.f --redefn 1
Info : ./e1/top.v:2: include file "./e1/my_incl.vh".  (INCL-1)
Info : ./e2/m1.v:2: include file "./e1/my_incl.vh".  (INCL-1)
Info : Link status: true.  (LINK-2)
```

In general, it is best to adopt a style which enforces the check that no macro is ever redefined. That is the default behavior of `analyze`.

- Sadly, I have seen my share of (RTL) IP which abuses this simple design guideline.
- Even more scary, I have had to use several IPs (from the same provider) which share a base set of macro names, like `RD_ADDR_N`, `WR_ADDR_N`. However, the use of both of these IP does in fact have to change these values. So, even if the designs **`analyze ... --redefn 1`**, there are many, many `(MACRO-3)` warnings.

- So, there is nothing to do (apart from editing the IP/RTL sources) except to make sure verification and implementation share the exact same order of files (otherwise the redefinitions could occur in a different order!)
- The next example illustrates this issue.

Example 3

In this example, we have 2 configurable IP blocks which both use macros `M` and `N` to configure them. (This example is not contrived, but merely a simplification of real IP I have seen/used.)

First, we'll verify each of the IP using their respective `.f` files:

```
1> ../../bin/analyze ip1 -f e3a.f
Info : ./e3/ip1.v:1: include file "./e3/ip1_cfg.vh".  (INCL-1)
Info : Link status: true.  (LINK-2)

2> ../../bin/analyze ip2 -f e3a.f
Info : ./e3/ip1.v:1: include file "./e3/ip1_cfg.vh".  (INCL-1)
Error: ip2: no definition for module.  (LINK-5)

3> ../../bin/analyze ip2 -f e3b.f
Info : ./e3/ip2.v:1: include file "./e3/ip2_cfg.vh".  (INCL-1)
Info : Link status: true.  (LINK-2)
```

At 2> above, the user was curious whether the `.f` were interchangeable: can module `ip2` be linked using the same `.f` as `ip1`? (The answer is... NO!)

Suppose `ip1` and `ip2` were simulated/verified, independently, using `e3a.f` and `e3b.f`, respectively.

In parallel, the integration person was told to instance the `ip1` and `ip2` and simply bring the IOs to the top. Abiding by the keep it simple principle, this hookup yields:

```
// top.v

`include "ip1_cfg.vh"
`include "ip2_cfg.vh"

module top(input clk,
           input [`N-1:0] in1, output [(`M*`N)-1:0] q1,
           input [`N-1:0] in2, output [(`M*`N)-1:0] q2);
 ip1 u_ip1(.clk(clk),.in1(in1),.q(q1));
 ip2 u_ip2(.clk(clk),.in1(in2),.q(q2));
endmodule
```

We can anticipate (in advance) the problems with this code already; and we'll analyze it with `--verbose 2` to demonstrate some of the inner workings of the multi-pass parsing (required by the particular style of the `e3c.f`):

```
> ../../bin/analyze top -f e3c.f --verbose 2

Info : "e3c.f": processing...  (FILE-3)
Info : e3/ip1.v:1: include file "e3/ip1_cfg.vh".  (INCL-1)
Info : e3/ip2.v:1: include file "e3/ip2_cfg.vh".  (INCL-1)
```

```
Warn : e3/ip2_cfg.vh:4: macro 'N' redefined.
       First defined at e3/ip1_cfg.vh:4.  (MACRO-3)
Warn : e3/ip2_cfg.vh:5: macro 'M' redefined.
       First defined at e3/ip1_cfg.vh:5.  (MACRO-3)
Info : e3/top.v:3: include file "e3/ip1_cfg.vh".  (INCL-1)
Info : e3/top.v:4: include file "e3/ip2_cfg.vh".  (INCL-1)
Info : Link status: true.  (LINK-2)
```

And, if this design were passed to simulation, elaboration would fail miserably.

- In all of the following examples, I am using (a free) ModelSim Verilog simulator available from the Xilinx website.

```
> vlog -f e3c.f
** Warning: e3/ip2_cfg.vh(4): [TMREN] - Redefinition of macro: N.
** Warning: e3/ip2_cfg.vh(5): [TMREN] - Redefinition of macro: M.
-- Compiling module top
-- Scanning library directory 'e3'
-- Compiling module ip1
-- Compiling module ip2

...
> vsim -c top

...
# ** Warning: (vsim-3015) e3/top.v(9): [PCDPC] - Port size (...) does not
match connection size (16) for port 'in1'.
#         Region: /top/u_ip1
```

So, while the root cause is pretty obvious/simple, the solutions are pretty ugly. In almost all situations, RTL from an IP provider is read only. So, while the obvious solution is to fix the source; it is verboten here, since this is IP. So, we **cannot**:

- edit any of: `ip[12].v ip[12]_cfg.vh`

In the next examples, we'll look at some possible solutions

Example 4

The first solution will solve using *just* preprocessor directives and modify `top.v` to (see `e4/top.v`):

```
// top.v

// We need to undef these so we are able to re-include the cfg files again.
// The cfg files were (correctly) written to prevent re-inclusion.
// But, we have an IP mess on our hands.

`undef _ip1_cfg_vh_
`undef _ip2_cfg_vh_
```

```
module top(input clk,
`include "ip_undef.vh"
`include "ip1_cfg.vh"
           input [`N-1:0] in1, output [(`M*`N)-1:0] q1,
`include "ip_undef.vh"
`include "ip2_cfg.vh"
           input [`N-1:0] in2, output [(`M*`N)-1:0] q2);
`include "ip_undef.vh"
 ip1 u_ip1(.clk(clk),.in1(in1),.q(q1));
 ip2 u_ip2(.clk(clk),.in1(in2),.q(q2));
endmodule
```

There are an awful lot of preprocessor directives required. But, it does work:

```
> ../../bin/analyze top -f e4a.f

Info : ./e3/ip1.v:1: include file "./e3/ip1_cfg.vh".  (INCL-1)
Info : ./e3/ip2.v:1: include file "./e3/ip2_cfg.vh".  (INCL-1)
Info : ./e4/top.v:11: include file "./e4/ip_undef.vh".  (INCL-1)
Info : ./e4/top.v:12: include file "./e3/ip1_cfg.vh".  (INCL-1)
Info : ./e4/top.v:14: include file "./e4/ip_undef.vh".  (INCL-1)
Info : ./e4/top.v:15: include file "./e3/ip2_cfg.vh".  (INCL-1)
Info : ./e4/top.v:17: include file "./e4/ip_undef.vh".  (INCL-1)
Info : Link status: true.  (LINK-2)
```

And, the verification folks will be pleased too:

```
> vlog -f e4a.f

...
-- Compiling module ip1
...
Top level modules:
        top

> vsim -c top

...
# Loading work.top
...
VSIM 1>
```

While this is *a solution*; it seems a bit clumsy, not to mention difficult to maintain (and perhaps even comprehend).

The real crux of the problem is the lack of a namespace, at least for macro values. We would really like to associate the macro values with a design or module: I.e., a set of `N` and `M` values for design `ip1` and a set of `N` and `M` values for design `ip2`.

Example 5

We can implement a namespace facility by defining a (module) scope for each of the (toplevel) IP modules called: `ip1_cfg` and `ip2_cfg`.

Within these scopes, the macro values will be *transferred* to parameter values; thus, the `ip[12]_cfg` module scopes will now be the cleaner, definitive source of the IPs configuration values (or more aptly named parameters).

(All example files under `tc6/e5`.)

```
// ip1_cfg.v

module ip1_cfg;
        parameter N = `N;
        parameter M = `M;
endmodule
```

```
// ip2_cfg.v

module ip2_cfg;
        parameter N = `N;
        parameter M = `M;
endmodule
```

The `.f` files are used to (cleanly) phase in the `` `define `` and `` `undefs ``. The `.f` also make it convenient to support pre-existing directory/install locations without burdening the user (of the IP) of the where.

A quick `analyze` shows the IP and `top` are clean, with these modifications:

```
> pwd
/.../data/tc6
> ../../bin/analyze ip1_cfg -f e5/ip1.f
Info : e3/ip1.v:1: include file "e3/ip1_cfg.vh".  (INCL-1)
Info : Link status: true.  (LINK-2)
> ../../bin/analyze ip1 -f e5/ip1.f
Info : e3/ip1.v:1: include file "e3/ip1_cfg.vh".  (INCL-1)
Info : Link status: true.  (LINK-2)
> ../../bin/analyze top -f e5.f
Info : e3/ip1.v:1: include file "e3/ip1_cfg.vh".  (INCL-1)
Info : e3/ip2.v:1: include file "e3/ip2_cfg.vh".  (INCL-1)
Info : Link status: true.  (LINK-2)
```

and, verification clean:

```
> vlog -f e5.f
-- Compiling module ip1  -- Compiling module ip1_cfg
-- Compiling module ip2  -- Compiling module ip2_cfg
-- Compiling module top

Top level modules:
        ip1_cfg ip2_cfg top
```

After elaboration, the parameter values are correctly established:

```
> vsim -c top ip1_cfg ip2_cfg
# Loading work.top
# Loading work.ip1
...
VSIM> examine -name /top/*
# {/top/N1 4} {/top/M1 8} {/top/N2 16} {/top/M2 2} ...
```

Example 6

An example showing the proper means to include functions and tasks.

Here, a decoder function is to be reused. The function definition assumes that N and M are predefined parameter values, such that they are inherited at the function scope (from the enclosing module definition).

```
// funcs.vh

// Assume M = 2^N are both defined (before inclusion)
function [M-1:0] decode(input [N-1:0] sel, input valid, input value)
  reg [M-1:0] tmp;
  begin
    tmp = ~{M{value}};  //initialize to inactive value
    if (valid) tmp[sel] = value;
    decode = tmp;
  end
endfunction
```

And, the toplevel, `e6/top.v`:

```
module m1
  #(parameter N=2, parameter M=1<<N)
  (input clk, input [N-1:0] x, output reg [M-1:0] q);

`include "funcs.vh"

  always @(posedge clk)
    q <= decode(x, 1'b1, 1'b0);
endmodule

module m2
  #(parameter N=2, parameter M=1<<N)
  (input clk, input [N-1:0] x, input [M-1:0] toMask, output reg [M-1:0] q);

`include "funcs.vh"

  always @(posedge clk)
    q <= toMask & decode(x, 1'b1, 1'b1);
endmodule

module top
  #(parameter N1=3, parameter N2=5,
    parameter M1=1<<N1, parameter M2=1<<N2)
```

```
  (input clk, input [N1-1:0] x1, output [M1-1:0] q1,
              input [N2-1:0] x2, input [M2-1:0] toMask, output [M2-1:0] q2);

  m1 #(.N(N1),.M(M1)) u1(clk, x1, q1);

  m2 #(.N(N2),.M(M2)) u2(clk, x2, toMask, q2);

endmodule
```

And checking:

```
> ../../bin/analyze top +incdir+e6 e6/top.v

Info : e6/top.v:5: include file "e6/funcs.vh".  (INCL-1)
Info : e6/top.v:15: include file "e6/funcs.vh".  (INCL-1)
Info : Link status: true.  (LINK-2)

> vlog +incdir+e6 e6/top.v
...
Top level modules:

        top

> vsim -c top
...

# Loading work.top
# Loading work.m1
# Loading work.m2

VSIM 1>
```