Using Verdi to Generate *vi* and *emacs* Tagging Databases

David Carson

Huawei Technologies Canada

Kanata, Ontario, Canada

http://www.huawei.com/ca-en/index.htm

ABSTRACT

A compiled and elaborated Synopsys Verdi Knowledge Database (KDB) contains valuable information that can be accessed via tcl scripts and utilized by the most widely used code editors to allow for advanced design browsing. Once a design has been compiled by Verdi the line numbers of variables, modules, packages, classes, etc. may be extracted from the database and written out to vi and emacs "tag files". These tag files may then be used by these editors to browse through a design's RTL with ease. For example, when examining a piece of code that contains an instance of a module or a structure definition, if the code editor has access to a "tag file" it can take the coder directly to the module definition or structure definition with a "key sequence" (just like a hyperlink) as opposed to having to find the file that contains the reference - perhaps in some distant directory tree - and open it in a separate buffer or edit session.

Table of Contents

Table of Figures

Table of Tables

1. An Introduction to Tagging

1.1 What is a Tag and What Does It Do?

A *tag* is a SystemVerilog *identifier* (or an identifier in any language, tagging is common to most programming languages) that can be *jumped* to (just like a hyperlink on the web). An *identifier* is any SystemVerilog variable, instance, package, input/output port, structure definition, label, task, function, etc and *jumping* means that when the cursor is placed over the identifier and key sequence or mouse click is entered the editor jumps to the definition of the identifier – either in the same edit buffer or a new one. A different sequence returns to the user to the tag location of the file that was being edited prior to the jump. The number of jumps a user make take is unlimited.

All of the *tags* associated with a design are stored in a *tag* database that is used by an editor to navigate from the tag to the tag definition.

Using tags to browse through a design is a very convienent way of navigating through RTL source code; especially if the file structure of the underlying code base is unfamiliar, large or spread over a file system that inclues multiple file trees.

For example, consider this trivial design where module ***top*** *(top.v)* contains two sub-modules ***mod_a*** *and* ***mod_b*** *(mod_a.v, mod_b.v)* and the three modules refer to the package ***PKG*** *(pkg.v)*.

```
1  `include "pkg.v"
2
3  module top
4    (
5      input  [PKG::BUS_W-1:0]  i_in,
6      output [PKG::BUS_W-1:0]  o_out
7    );
8
9    PKG::bus_s                 a2b;
10
11   mod_a U_MOD_A (
12     .i_in                    ( i_in  ),
13     .o_a2b                   ( a2b   )
14   );
15
16   mod_b U_MOD_B (
17     .i_a2b                   ( a2b   ),
18     .o_out                   ( o_out )
19   );
20
21 endmodule
22                    top.v
```

```
1  `ifndef _PKG_
2  `define _PKG_
3
4  package PKG;
5
6    typedef struct packed {
7      reg                          vld;
8      reg [7:0]                    data;
9    } bus_s;
10
11   parameter int   BUS_W = $bits(bus_s);
12
13 endpackage
14
15 `endif
16
17
18
19
20
21
22                    pkg.v
```

```
1  `include "pkg.v"
2
3  module mod_a
4    (
5      input  [PKG::BUS_W-1:0]       i_in,
6      output PKG::bus_s             o_a2b
7    );
8
9    assign o_a2b = i_in;
10
11 endmodule
12                  mod_a.v
```

```
1  `include "pkg.v"
2
3  module mod_b
4    (
5      input  PKG::bus_s             i_a2b,
6      output [PKG::BUS_W-1:0]       o_out
7    );
8
9    assign i_in = i_a2b;
10
11 endmodule
12                  mod_b.v
```

Figure 1. Tagging Example

If file top.v is being edited and a designer wants to examine the module ***mod_a*** rather than opening the file *mod_a.v* in another editing session, which requires that the designer knows the location of

the file, the designer could simply "jump" to the tag ***mod_a***. To do this in *vi* or *emacs* a designer simply puts the cursor on the identifier ***mod_a*** (on line 11 in top.v) and enters the keystroke to "jump to tag" and the editor will automatically open the file *mod_a.v.* The editor knows to open *mod_a.v*, where to find it, and to put the cursor on line 3 of this file, because this information is stored in a tag database that was previously generated for this design.

Similarily if a designer was editing file *mod_a.v* they could jump to the definition of the structure ***bus_s,*** and their editor would open the file *pkg.v* and place the cursor on line 6.

While tagging may seem unnecessary in a design of this simplicity, real designs typically include many dozens of files and could include thousands of unique identifiers (tags) spread across a complex file system with symbolic links and much source code that does not even belong to the design being browsed. Furthermore, the code may be conditionally compiled with variables hidden away in build systems or environment variables.

It is in this more typical scenario that tagging via Verdi SystemVerilog tags shines since it includes *design context* in the tagging process – tags are only generated from fully compiled designs and only tag of design interest are included. This is a significant improvement over traditional tagging practices.

1.2 A Brief History of Tags

Tagging has been utilized by programmers for decades. Tagging programs parse source files and generated tag databases for use by editors. Early versions of tagging programs include the original version written by Ken Arnold in the mid 1970s, the *ctags* program included in the *elvis* version of vi authored by Steve Kirkendall and others in the early 1990s. And finally the subsequent and more widely used *exuberant ctags* version written by Darren Hiebert in the mid 1990s that supports a wide varity of languages (including Verilog) and still widely used today.

Tagging is also built into many commercial and freeware Integrated Development Enviroments (IDEs) such as MicroSoft VisualStudio and Eclipse. Many IDEs rely on vendor supplied or community sourced language parsers and plug-ins. Programmers of widely used programming languages such as 'C', Java or Perl have access to high quality language parsers while users of less widely used languages such as SystemVerilog or VHDL do not.

Tagging programs are provided a list of source code files from which tags are extracted. The programs usually employ light weight language parsers to extract identifiers from the source code and then created a tag database of these identifiers for use by code editors. Because these parsers are neither pre-processors nor compilers, pre-processor macros and mischevious syntax errors would often cause tags to be missing or incorrect. Furthmore, duplicate or non-relevant tags may be generated if wild card searching of files is used to generate lists of files to be tagged or if conditional compiling is employed within files.

Some editors will produce an internal set of *tags* for every file that is loaded for rapid real time tagging. This approach can break down if the language parser does not deal well with syntax errors that may be present in the source code and cannot handle pre-processor macros or included files.

Some IDEs will use tagging information to display lists of tags in a separate editor window, or provide real time "pop up" preview information about variable types or function definitions. *vi* and *emacs* both have sophisticated plug-ins that support this kind of capability for use with *ctags.*

Verdi is also a very powerful design exploration tool which offers identifier tagging as described above and is often used by design and verification to understand and navigate large complex designs – however it is ***not*** an editor.

On the subject of new tagging tools, Gnu Global is a relatively new project and has recently made a move to take over from exuberant ctags as the defacto tag generation platform of the future.

1.3 Tags and SystemVerilog

With the introduction of SystemVerilog 2012 the language became extremely complex to parse and the traditional community supplied taggers did not cope well with the advanced language features. While previous versions of Verilog could be tagged with reasonable accuracy and produce accurate lists of registers and wires, new language features may cause some freeware language parsers to struggle to produce full and comprehensive sets of tags.

ctags and etags are still widely used and very effective tagging tools - with the appropriate regular expressions used during tag generation. What would be an added benifet for the current generation of SystemVerilog coders is a way to generate comprehensive and accurate tags for large SystemVerilog designs without need to maintain and tweak regular expressions or maintain tagging file lists. Furthermore it would be ideal of the tagging system was fully design aware – only tags that are compiled and used by the design are included in the tag database.

2. Tag Files and the Verdi Knowledge Database

For those with access to Synopsys's Verdi Debugging Platfrom there is a way to generate comprehensive and accurate tags for large SystemVerilog designs – use the *Synopsys VC Apps Native Programming Interface (NPI)*. Before discussing the NPI it is worth discussing what a tag actually is.

2.1 The Anatomy of a Tag file

The format for *vi* tag files and *emacs* tag files are both well defined, albeit different and easily found on the internet; for example, the Wikipedia entry for "ctags" provides complete details for each of them.

Consider the following simple Verilog example:

```
1 module sv_test_top
2   (
3     input logic                       i_in,
4     output logic                      o_out
5   );
6
7   assign o_out = i_in;
8
9 endmodule
```

Figure 2. Simple Verilog Example

2.1.1 vi

The format for a vi tags file is:

```
{tagname}<Tab>{tagfile}<Tab>{tagaddress}
```

Where fields given above are::

- `{tagname}` – Any identifier, not containing white space.

- <Tab> – Exactly one tab character.
- {tagfile} – The name of the file containing {tagname}.
- {tagaddress} – A line number or an ex mode command that will take the editor to the location of the tag.

For the above example the *vi* tags file would contain:

```
1 i_in  sv_test_top.v 3
2 i_in  sv_test_top.v 3
3 o_out sv_test_top.v 4
4 o_out sv_test_top.v 4
5 sv_test_top sv_test_top.v 1
```

Figure 3. *vi* Tags File Contents for Simple Verilog Example

The first line indicates that the identifier i_in is defined on line 2 of the file top.v, o_out is defined on line 4 of top.v and top is located on line 1 of top.v. The tag file does not offer an information about what the idenitifier is, only where to find its definition.

2.1.2 emacs

An emacs tag file separates the contents of input source files into sections separated by non-printable ASCII characters. These characters are represented as bracketed hexadecimal numbers below.

Each section contains a two line header. The first line contains a single <\x0c> character. The second line contains the file name and the total size of the identifier section in bytes including carriage returns.

```
{src_file},{size_of_tag_definition_data_in_bytes}
```

A list of tag identifiers follows the header with the following format

```
{tag_definition_text}<\x7f>{tagname}<\x01>{line_number},{byte_offset}
```

{tagname} and the <\x01> character may be omitted if the name of the tag can be deduced from the text at the tag definition.

For the same example as above, the *emacs* tags file would contain:

```
1 ^L
2 sv_test_top.v,58
3 ^?i_in^A,24
4 ^?i_in^A,24
5 ^?o_out^A,69
6 ^?o_out^A,69
7 ^?sv_test_top^A,0
```

Figure 4. emacs Tags File Contents for Simple Verilog Example

The first line is the section separator character <\x0c>. The second line indicates this section corresponds to the file top.v and that the top.v identifiers section is 115 bytes in length. The next three lines contain identifier information. The first line indicates that top is located on

line 1 of top.v at byte offset 0 of the file, the second line that identifier `i_in` is defined on line 3 of the file top.v at byte offset 19 and the third line that identifier `o_out` is defined on line 4 of top.v at byte offset 57. The non-printable characters and byte offsets are used by *emacs* internal searching mechanims.

2.2 The Verdi Knowledge Database

A design that has been compiled by Verdi or compiled by VCS for Verdi is stored in the Verdi Knowledge Database (KDB). A Verdi KDB has all of the information necessary to create tagging information for *vi* and *emacs.* It contains lists of all identifiers complete with scope and file location. Question: How is this information extracted and re-format it into the *vi* and *emacs* tagging formats described above? Answer: use the *VC Apps Native Programming Interface* which allows users access to KDB information via 'C' programs or tcl scripts from within a Verdi session.

Synopsys supplies ample documentation for its VC Apps in *VC Apps Toolbox User Guide* and the Native Programming Interface (NPI) in its *VC Apps Native Programming Interface (NPI)* documents.

2.2.1 Verdi Access Using tcl

The NPI provides an interface layer to let users access the KDB contained in the Verdi® Automated Debug Platform.

Users can work with the NPI models through two kinds of interfaces, one is the Tcl command interface and the other is the C interface. Compared to the C interface, Tcl commands provided by the NPI model can be played in both batch mode (through a command file) and interactive mode (through the Tcl console) when the Verdi executable is invoked. However, since Tcl is an interpreted language, the Tcl command interface will have worse performance due to the Tcl interpreter effort. It is recommended that you use the C interface if the application has stringent performance requirements.

Commonly used VC apps are pre-packaged in the installation package of the Verdi platform. Users can access these VC Apps from the VC Apps Toolbox area from the Verdi GUI. The VC Apps in the VC Apps toolbox are developed and tested by Synopsys. Users can also access the original source code of some Apps in the Toolbox and customize it to fit different requirements. VC Apps toolbox provides flexibility that allows users to add in house tools into VC Apps Toolbox group of .

The NPI Language Model treats hardware description language (SystemVerilog/VHDL) constructs as objects, and the NPI Language Model routines provide ways to locate any specific object or type of object within the design loaded from Verdi. An Application Programming Interface (API) within Verdi provides access to these HDL constructs.

To create *vi* and *emacs* tags databases a custom tcl script, *tags.tcl*, was written. *tags.tcl* traverses the KDB, extracts tag information for the relevant constructs and outputs the information to an intermediate raw data file which is converted to *vi* and *emacs tag data bases by additional scripts*. The *tags.*tcl script is run within Verdi on a previously compiled database that was opened by Verdi, or on an RTL compilation done by Verdi.

2.2.2 tags.tcl

The *tags.tcl* script is actually quite simple – all of the complexity of the task is hidden in the API. It traverses the design hierarchy where it identifies all *module types* and then traverses the hierarchy a second time and extracts all of the identifiers contained in these *module types*.

This first pass through the code is necessary because when a design hierarchy is traversed via

the Verdi API **all** modules are searched for identifiers. If a design contains multiple instances of the same module adding the same identifier to the tagging database for each instance would create unnecessary duplicate tag entries. What a tagger *should* do is create tags *per module type,* not *per module instance*. Consider the following figure.

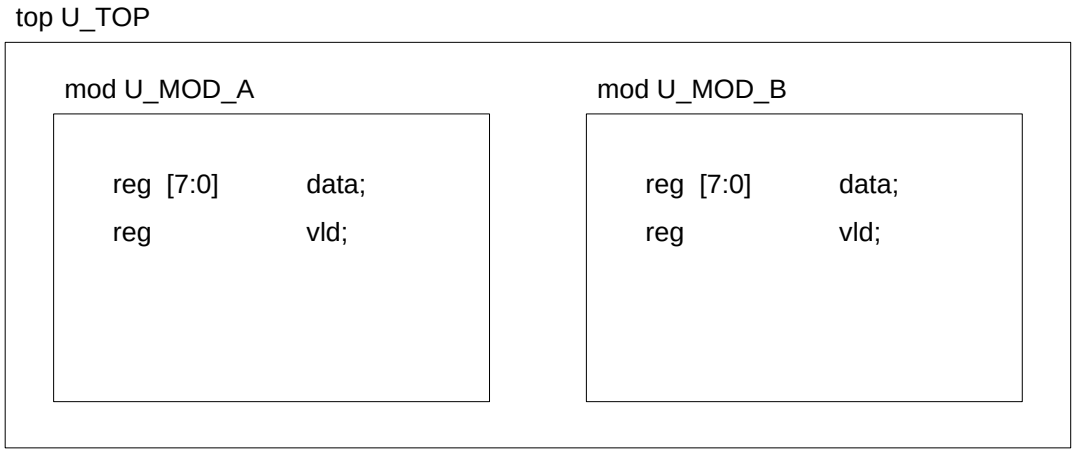


Figure 5. Extracting Tags From Only One Instance

In this design module *top* contains two instances of *mod.* When creating a tag database adding two entries for *data* and *vld,* one for each *instance of mod* would be redundant. Only a single entry for *data* and *vld,* as contained in the *definition of mod,* should be added to the database.

tags.tcl first pass through the design hierarchy creates a list of all module types and records a single instance name for each type. On the second pass through the hierarchy only identifiers from the instance that matches the recorded name are saved in the tag database. In the above figure *tags.tcl* would pass through the design hierarchy and identify two module types – *top* and *mod* – and record the instance names U_TOP and either U_MOD_A or U_MOD_B. On the second pass only identifiers for U_TOP and U_MOD_A or U_MOD_B would be recorded. Which U_MOD is recorded is arbitrary and depends on the Verdi's tree traversal.

The API function *npi_hier_tree_trv* is used to traverse the design object hierarchy tree (from the specified scope) and execute callback functions registered by the user for various object types when they are found in the tree. The design object hierarchy tree is the data structure that Verdi uses to hold design information. Call backs for objects are registered with the API function *npi_hier_tree_trv_register_cb*. Callbacks are user defined functions that are called when tree traversal locates object types of interest. *npi_hier_tree_trv_reset_cb* is used to reset callbacks. These functions are all documented in the *VC Apps Native Programming Interface (NPI)* document.

Consider this code fragment from tags*.tcl*:

```
123   # Create an associative array of 'items'.
124   array unset items
125   array set items ""
126
127   # Create an associative array of 'identifiers'.
128   array unset ident
129   array set ident ""
130
131   # Index for the associative array containing identifiers.
132   set index 0
133
134   # Create data structure and a list of data structures to be sent to the call
135   # back functions.  Append the data structures to the list.
136   #
137   set level [info level]
138   set cbList ""
139   lappend cbList $LOG
140   lappend cbList $level
141   lappend cbList "items"
142   lappend cbList "ident"
143   lappend cbList "index"
144
145
146   # Bind the call back function 'npiItemCb' to all items of interest.  Then
147   # traverse the Verdi Knowledge Database hierarchy tree to search for them.
148   # Each of thems items can appear in the database multiple times.  'cbList' is
149   # received by the call back function which will add the first instance of
150   # each item type to an array.
151   #
152   ::npi_L1::npi_hier_tree_trv_register_cb "npiModule"    "npiItemCb" "cbList"
153   ::npi_L1::npi_hier_tree_trv_register_cb "npiPackage"   "npiItemCb" "cbList"
154   ::npi_L1::npi_hier_tree_trv_register_cb "npiProgram"   "npiItemCb" "cbList"
155   ::npi_L1::npi_hier_tree_trv_register_cb "npiClassDefn" "npiItemCb" "cbList"
156   ::npi_L1::npi_hier_tree_trv ""
157   ::npi_L1::npi_hier_tree_trv_reset_cb
158
```

Figure 6. First Tree Traveral – Searching for Modules

Associated arrays *items* and *idents* are declared and included (as members of the call back list *cbList*) as arguments to the API *npi_hier_tree_trv_register_cb* function along with *npiItemsCb*, the name of the call back function to call when an object of type of interest is found during design hierarchy tree traversal. Object types of interest (items) include *modules, packages, programs and class definitions*. As previously mentioned on this first pass throught the hierarchy instances of modules are recorded and their identifiers are captured. On this same pass, identifiers in packages and programs are also captured since they are compilable objects with only one instance. Furthermore the definitions of all classes are recorded. Identiers in class definitions are not recorded so as not to create too many duplicate identifiers. A tree hierarchy traversal is requested with the API call to *npi_hier_tree_trv.*

The npiItemCb call back function is describe in following code segment.

```
11 proc npiItemCb { object ref } {
12   upvar $ref cbList
13
14   # Brake down call back list into its parts.
15   set LOG        [lindex $cbList 0]
16   set level      [lindex $cbList 1]
17   set itemsRef   [lindex $cbList 2]
18   set identRef   [lindex $cbList 3]
19   set indexRef   [lindex $cbList 4]
20
21   upvar #$level $itemsRef items
22   upvar #$level $identRef ident
23   upvar #$level $indexRef index
24
25   # Get the relevent information about the module.
26   if { $object } {
27     set npiType      [npi_get_str -property npiType      -object $object]
28     set npiDefName   [npi_get_str -property npiDefName   -object $object]
29     set npiDefFile   [npi_get_str -property npiDefFile   -object $object]
30     set npiDefLineNo [npi_get     -property npiDefLineNo -object $object]
31     set npiFullName  [npi_get_str -property npiFullName  -object $object]
32
33     # Generate Raw information for debug.
34
35     # If this type of module has not yet been seen, add it to the array of
36     # items and save the particular instance found as the array entry's data.
37     if {$npiDefName != ""} {
38       if { ![info exists items($npiDefName)] } {
39         set items($npiDefName) "$npiFullName"
40         set ident($index) "$npiDefName\t$npiDefFile\t$npiDefLineNo"
41         incr index
42       }
43     }
44   }
45 }
```

Figure 7. npiModule Callback

The *npiItemCb* callback function is called by the tree traversal API every time an item of interest is encountered during design tree traversal. *npiItemCb*, using other NPI API calls, identifies the name of the item definition, the file name and line number of the file that contains the item's definition and the full scope name of the instance of the item found. It then adds the item's definition to the *items* associated array (line 39) if this *item* is not already a member of the array. In addition all of the identifiers found in this item are added to the *ident* array on line 40.

In this first pass all items have been identified and all the identifiers of modules, programs and packages recorded.

The second pass through the design hierarchy tree collects all of the remaining identifiers that are to be tagged. The following abbreviated code segment shows just a few of the identifier types (ArrayNet, ArrayTypespec, etc) that are registered to the generic call back function *npiCb*.

```
160   # Bind all objects to be added to the tag list to the generic callback function
161   # 'npiCb'.  This call back will add all tags for these objects.
162   #
163   ::npi_L1::npi_hier_tree_trv_register_cb "npiArrayNet"       "npiCb" "cbList"
164   ::npi_L1::npi_hier_tree_trv_register_cb "npiArrayTypespec"  "npiCb" "cbList"
165   ::npi_L1::npi_hier_tree_trv_register_cb "npiArrayVar"       "npiCb" "cbList"
166                     .
167                     .
168                   [snip]
169                     .
170                     .
171   ::npi_L1::npi_hier_tree_trv ""
172   ::npi_L1::npi_hier_tree_trv_reset_cb
173
```

Figure 8. Second Tree Traveral – Searching for Identifiers

The generic call back function *npiCb* is called by the tree traversal API every time one of many objects registered for callback is encountered during the second design tree traversal. *npiCb*, using other NPI API calls, identifies the name of the object, the file name and line number of the file that contains the object and the full scope name of the object. It then adds the identifier to the array of identifiers **only if** the scope name of the object matches the scope name of one of the items in the list of items contained in the associated list *items* captured on the first pass. This is the check to ensure that the identifiers of only modules definitions are tagged, not the identifiers of all instances.

- NOTE: Verdi provides tags.tcl the absolute location of the files it used to compile the design and it is these paths that are used to generate the tags database. If the location of files changes between the time the design was compiled and when the tag database is to be used, the tagger will not be able to locate the file. If this is the case tags can simply be regenerated or additional scripting may be used to correct tag database file paths.

As can be seen in the following code fragment, information is gathered about the object in lines 70 to 79. A check is then made to see if the object is a *package, class or program* and if it is then the identifier information is added to the identifier array *ident*. All other objects are added to the *ident* array only if the scope they belong to is one of the object scopes gathered on the first pass (this check being done on line 97).

```
 54 proc npiCb { object ref } {
 55   upvar $ref cbList
 56
 57   # Brake down call back list into its parts.
 58   set LOG        [lindex $cbList 0]
 59   set level      [lindex $cbList 1]
 60   set itemsRef   [lindex $cbList 2]
 61   set identRef   [lindex $cbList 3]
 62   set indexRef   [lindex $cbList 4]
 63
 64   upvar #$level $itemsRef items
 65   upvar #$level $identRef ident
 66   upvar #$level $indexRef index
 67
 68   # Get the relevent information about the module.
 69   if { $object } {
 70     set npiType        [npi_get_str -property npiType     -object     $object]
 71     set npiName        [npi_get_str -property npiName     -object     $object]
 72     set npiFileName    [npi_get_str -property npiFile     -object     $object]
 73     set npiLineNo      [npi_get     -property npiLineNo   -object     $object]
 74
 75     # Get the scope name and type of the current object for comparison to list
 76     # of module instances.
 77     set npiScopeHandle [npi_handle  -type     npiScope    -refHandle  $object]
 78     set scopeName      [npi_get_str -property npiFullName -object     $npiScopeHandle]
 79     set scopeType      [npi_get_str -property npiType     -object     $npiScopeHandle]
 80
 81     # If the object is a package, class definition or program or contained
 82     # therein add the object to the tag list.
 83     if { $scopeType == "npiPackage"   ||
 84          $npiType   == "npiPackage"   ||
 85          $scopeType == "npiClassDefn" ||
 86          $npiType   == "npiClassDefn" ||
 87          $scopeType == "npiProgram"   ||
 88          $npiType   == "npiProgram" } {
 89       set ident($index) "$npiName\t$npiFileName\t$npiLineNo"
 90       incr index
 91
 92     # For all other objects only add a tag if 'the scope of the current object'
 93     # matches 'the recorded module instance' of 'one of items' in the array
 94     # of items that was generated when all module types were identified.
 95     } else {
 96       foreach { key data } [array get items] {
 97         if {$data == $scopeName} {
 98           set ident($index) "$npiName\t$npiFileName\t$npiLineNo"
 99           incr index
100         }
101       }
102     }
103   }
104 }
```

Figure 9. Generic Identifier Call Back

Now that all identifiers have been found the *vi* tags file can be generated as per the *vi* tags file format detailed earlier. As can be seen in the next code segment, the array containing all identifiers is flattened into a tcl list (line 251) and the list is then sorted by identifier (line 254) and output to the tags file. Because the format of emacs tags files are more complex, a list of all possible tagged source files is generated while the *vi* tags file is generated.

```
240  # Create an associative array that will hold the names of all source files.
241  array unset files
242
243  # Open the vi tag file.
244  set vtagFileName "tags"
245  set vtagFilePtr [open $vtagFileName "w"]
246
247  # Turn the associative array that contains all identifiers into a flattened
248  # list of strings.
249  #
250  # format of list:  <index> <identifier> <filename> <line number>
251  set flat_ident [array get ident]
252
253  # Sort the list by identifier.
254  set sorted_ident [lsort -stride 2 -index 1 $flat_ident]
255
256  # Write sorted list's data (no index) to tag file using vi formatting style.
257  # While doing so, create an array of source file names to be used for
258  # generating the emacs data base.
259  #
260  foreach {index tagInfo} $sorted_ident {
261    puts $vtagFilePtr "$tagInfo"
262
263    # Array of source file names
264    set srcFile [lindex [split $tagInfo] 1]
265    if { ![info exists files($srcFile)] } {
266      set files($srcFile) "$srcFile"
267    }
268  }
269
270  # Close vi tag file.
271  close $vtagFilePtr
```

Figure 10. Generating the *vi* tags

Once the vi tag file has been generated the emacs tag file is generated. All of the files in the list generated during *vi* tag file generation are opened (line 285) and the byte count to each line is calculated and stored in the *byteCounts* array. This information is then used in the loop at line 313 to generate the "per tag file" identifier information required by *emacs*. One this "per tag file" information is generated it is written out to the *emacs* tag file.

```
278   # To generate tagging information for all source files cycle through the
279   # array of source file names.
280   #
281   foreach { fileName data } [array get files] {
282
283     if { $fileName != ""} {
284       # Open a source file.
285       set srcFilePtr [open $fileName "r"]
286
287       # Array used to store the byte count offset for each line in source file.
288       array unset byteCounts
289       array set byteCounts ""
290
291       # emacs tag database format required byte offsets information for each
292       # tag.
293       set totalByteCount 0
294       set lineCount      1
295       set bytesThisLine [expr [gets $srcFilePtr str] + 1]
296
297       # Calculate the byte offset for each line in source file.
298       while { $bytesThisLine  >= 1 } {
299         set byteCounts($lineCount) $totalByteCount
300         set totalByteCount [expr $totalByteCount + $bytesThisLine]
301         set lineCount [expr $lineCount + 1]
302         set bytesThisLine [expr [gets $srcFilePtr str] + 1]
303       }
304       close $srcFilePtr
305
306       set entryList ""
307
308       # Cycle through the list of sorted identifiers.  If the filename for an
309       # identifier matches the current file, add the identifier and its line
310       # offset to a string of identifiers for this file following the emacs
311       # formatting style.
312       #
313       foreach {index tagInfo} $sorted_ident {
314         set srcFile [lindex [split $tagInfo] 1]
315         if {$fileName == $srcFile} {
316           set tag [lindex [split $tagInfo] 0]
317           set line [lindex [split $tagInfo] 2]
318           set offset $byteCounts($line)
319
320           set entry "\x7f$tag\x01,$offset"
321           append entryList "\n" $entry
322         }
323       }
324
325       # Calculate the total length of the string that holds all the entries for
326       # this file then write out the current file's tagging information to the
327       # tag file following the emacs formatting style.
328       #
329       set entryListLength [string length $entryList]
330       puts $etagFilePtr "\x0c\n$fileName,$entryListLength$entryList"
331     }
332   }
333
334   # Close emacs tag file.
335   close $etagFilePtr
```

Figure 11. Generating the *emacs* tags

2.2.3 Recognised SystemVerilog Identifiers

The following is a list of identifiers generated by *tags.tcl* as of Verdi 2017.03. These are the identifier types that are registered for callback during tree traversal. Dynamic objects such as Classes and verification construct such as Programs and Virtual Interface Variables are not

recognized prior to this release.

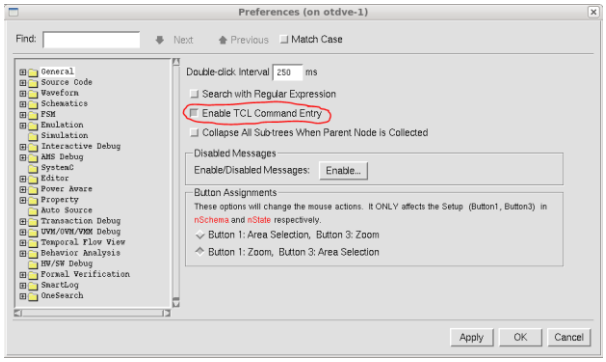
Table 1. Recognised SystemVerilog Identifiers

ArrayNet	ArrayTypespec	ArrayVar	BitTypespec
BitVar	ByteTypespec	ByteVar	ClassDefn
ClassObj	ClassTypespec	ClassVar	Constant
EnumNet	EnumTypespec	EnumVar	GenVar
GenScope	IODecl	IntTypespec	IntVar
IntegerNet	IntegerTypespec	IntegerVar	Interface
InterfaceArray	LogicTypespec	LongIntTypespec	LongIntVar
MethodFuncCall	Modport	Module	ModuleArray
Net	NetBit	Package	PackedArrayNet
PackedArrayTypespec	PackedArrayVar	Parameter	ParameterBit
Port	Program	RealTypespec	RealVar
RefObj	Reg	ShortIntTypespec	ShortIntVar
ShortRealTypespec	ShortRealVar	StringTypespec	StructNet
StructTypespec	StructVar	Task	TaskCall
TypeParameter	TypePattern	TypespecMember	UnionTypespec
UnionVar	VirtualInterfaceVar		

3. Running the tags.tcl Script

tcl scripts (VC apps) can be run from within the Verdi GUI or in batch mode. Running tcl scripts from within the Verdi GUI is done via the TCL Command Entry dialog.

To run a script first start Verdi and load a design. Then open the Command Entry dialog by selecting *Tools->Preferences* from the Verdi GUI main menu. From the Preferences dialog box click the *Enable TCL Command Entry* radio button. The Command Entry dialog will appear.



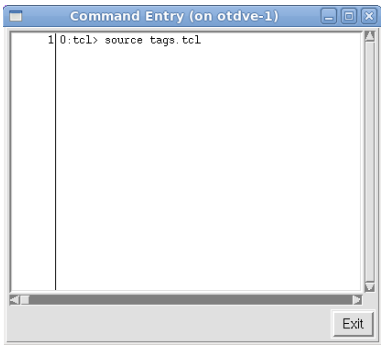


Figure 12. Opening Verdi's TCL Command Entry Dialog Box

Scripts can be sourced from within this dialog. Indeed this is a full tcl interpreter and any tcl command may be issued in this dialog. To run the tags.tcl script simply source it in this dialog.

To run a tcl script in Verdi in batch mode from the command line use the –play option:

```
verdi –nogui –nologo –q –f design.f –play tags.tcl > /dev/null
```

The –f option directs Verdi to a manifest file containing a list of design files and any other options necessary to compile the design. Optionally Verdi could be directed to open a precompiled database. The –nogui, –nologo –q options will keep the output "quiet" and re-direct all error output to devnull to squelch console error messages. These options are not necessary and may be removed if the user wants all feedback from the run. Furthermore, this command could be run from a makefile or a cron job to routinely generate tagging information or run when files are submitted into a shared storage location such as a version control system.

4. Using Tags

Once *vi* and *emacs* tag databases have been created for a design both editors must be configured to know where to find the database. Additionally custom key bindings can be set up that match users preferred interaction with the editor.

4.1.1 vi

To configure *vi* to use a tagging database the 'tags' option must be set. The 'tags' option is a comma separated list of file names that is searched for tags. Since the *tags* script generates a *vi* tag database with the file name *tags* the following command must be issued in normal mode to set the 'tags' option when editing files that have been included in a SystemVerilog tag database:

```
:map tags=tags
```

Alternativly the 'tags' option can be permanently set by putting this line in the *vi* configuration file .vimrc.

There are many ways to jump to a tag in *vi*. The simplest is the normal mode command

```
:tags {ident}
```

Where `{ident}` is the identifier to jump to and is manually typed in by the user. Using the `tags` command to jump to a tag does not require the cursor to be on the identifier since it is included in the command.

Not having to type the identifier to jump to is very convienent. The following mouse and keyboard control commands:

```
<C-LeftMouse>
CTRL-]
```

will both jump to the identifier ***under*** the cursor. When there are multiple matching tags for the identifier *vi* will jump to the first one or provide a numbered matchlist to allow the user to select which tag to jump to.

When *vi* jumps to a tag it records the current location in the tag stack. To return from a jumped location entering this command in normal mode:

```
:pop
```

will jump *back* to the last location on the stack. Similarily the following mouse and keyboard control commands will jump back:

```
<C-RightMouse>
CTRL-T
```

The tag stack can be examined with this command:

```
:tags
```

The output of ":tags" looks like this:

```
  # TO tag       FROM line  in file/text
  1   1 mod_a            11  top.v
> 2   2 PKG               5  pkg.v
```

So as to keep the current buffer open to know what you were editing prior to a jump, the jump can be opened in a separate window using:

```
CTRL-W CTRL-]
```

The jump can also open the tag in a 'preview' window, leaving the cursor in the current file:

```
CTRL-W CTRL-}
```

If there are multiple matching tags *vi* will offer a menu of matches for the user to select from. In this example *dat* is defined in two packages.

```
 1 package A;
 2 logic                                 dat;
 3 endpackage
 4
 5 package B;
 6 logic                                 dat;
 7 endpackage
 8
 9 module sv_test_top
10   (
11     input logic                       i_in,
12     output logic                      o_out
13   );
14
15   assign o_out = A::dat;
16
17 endmodule
```

Figure 13. Multiple Matching Tags Example

When an attempt to jump from *dat* on line 15 is made the user is queried as to which tag is to be jumped to.

```
  # pri kind tag               file
  1 F C      dat               /data/PTG/d904686/sv_test/ip_...rtl/sv_test/src/sv_test_top.v
               2
  2 F C      dat               /data/PTG/d904686/sv_test/ip_...rtl/sv_test/src/sv_test_top.v
               6
Type number and <Enter> (empty cancels):
```

Figure 14. Multiple Matching Tags Menu

4.1.1 emacs

To jump to a tag in *emacs* use:

```
Meta .
```

And to return from a jump

```
Meta *
```

All of the capability shown in the *vi* section above is available in *emacs*.

4.1.2 Shortcoming and Limitations

There often are multiple hits for some tags. Take a clock net such as *i_clk* for example. This net is usually defined port in top level module then re-defined in all lower level modules. If a jump request is issued for this net somewhere in the hierarchy editors will either jump to the first matching tag in a list of matching tags or present the user a match list and ask the user to select which tag to jump to. While sometimes irritating, this behavior is to be expected. An editor's tagging system has no notion of the scope of the design and therefore cannot be expected to

identify what the relivent definition of an identifier should be. In the case of example *i_clk* this is usually not an issue since all definitions of it would likely be the same. However, for an identifier name that is reused in multiple places in the design where each usage has a unique definition, it is important for the user to carefully examine the presented list of matches and ensure that the definition jumped to maches the identifier usage at the jump point.

Regarding Package Scope Resolution. *vi* and *emacs* typically interprete the colon character as a tag separator. This means that the *scope resolution operator* '::' is ignored when determining what tag to jump to. For example, in the case of a parameter definition that includes scope (e.g. "PackageName::Parameter"), both editors will select either the "PackageName" or the "Parameter" (depending on which word the cursor is positioned) as the tag and jump to the appropriate definition. If there are multiple "Packages" with the same "Parameter" and the cursor is placed on a "Parameter" identifier and a jump is requested, then either the first matching tag in the tag database is jumped to or the designer is presented a list of matching tags. At this point it us up the designer to manually determine which tag to jump to by identifying which tag hit would likely match the "PackageName" that preceeds the "Parameter". It is recommended in this type of scenario that the designer jump to the "PackageName" then perform a forward search for the "Parameter" of interest. In this way the designer will always find the *correct* "Parameter". Even if vi or emacs were set up to recognize the scope resolution operator, the tag database generated by the tags.tcl script does not include tags with scope.

Some editors and their plugins will attempt to tag all in buffer files. To do this the editors need access to a fast executable tagger such as ctags. Unfortunately compiling a design and generateing its tags in Verdi is too slow to support this.

Editors need to know the location of the tags database for the design they are working on; this, for example, is set by the 'tags' option in *vi* for and may either be relative or absolute. Managing the location of the tags database and the configuration editors and a full discussion of this issue is beyond the scope of this paper. One possible setup would be to store tag databases at the 'root' of a design tree where the top level module and manifest are located. Managing tag databases for multiple designs that share this same directory tree would be done with separate design check-outs from a version control system that takes advantage of environment variables to identify the active project.

5. Future Work

Using tcl in Verdi is slow given its interpreted nature. It would be straight forward to convert the tags.tcl script to 'C' to improve performance and use the Verdi 'C' interface to generate the tags. Documentation is available from Synopsys regarding VC development in 'C' in the *VC Apps Toolbox User Guide* and the Native Programming Interface (NPI) in its *VC Apps Native Programming Interface (NPI)* documents.

If a reduced set of tags was desired making the list of identifier types to tag programmable via a config file would be an easy addition.

Adding exuberant ctags' additional tagging information to the *vi* ctags database would be simple addition given that all of the relevant identifier type information is available in the KDB if a user desired to have this information to support some of the more popular *vim* tag exploration tools.

6. Conclusions

Navigating todays large System on Chip designs is more difficult than ever given the scope and magnitude of the RTL that generates them. While venerable editors such as *vi* and *emacs* are still up to the task of day to day editing tasks as are, to a certain extent ctags and etags, they can lack the capability of modern day IDEs to assist designers in locating all variable declarations, structure definitions, module definitions, tasks, functions, packages, parameters and far flung logic libraries and memories. Furthermore, not knowing how a design was conditionally compiled can make searching for these items even harder.

While *find* and *grep* may have been the go to approach most designers have taken to navigate unknown designs, designers now have a more powerful too. By taking advantage of Synopsys' *VC Apps Native Programming Interface (NPI)* the *tags.tcl* script unlocks vital information stored in the Verdi Knowledge Database and allows those designers still using legacy editing platforms to unleash the power of code tagging.