**SV Directed Test Bench Users Guide**

by

Ken Campbell

Version: 1.0

August 15, 2014

Copyright (c) 2014 Ken Campbell.

Licensed under the Apache License, Version 2.0 (the "License");

you may not use this file except in compliance with the License.

You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software

distributed under the License is distributed on an "AS IS" BASIS,

WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

See the License for the specific language governing permissions and

limitations under the License

**Revision History:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Revised By** | **Description** | **Date** |
|  |  |  |  |
| 1.0 | Ken Campbell | Initial release | August 15, 2014 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Table of Contents:**

1. Introduction 6

1.1 History 6

1.2 Over View 6

1.3 Scope 6

1.4 Interested Parties 6

1.5 Why? 6

1.6 Document Acronyms 7

2. Test Environment Usage Flow 8

2.1 The DUT 9

2.2 Choose The Methodology 9

2.3 Generating the Test Bench 9

2.4 Creating the Initial Instructions 9

2.5 Writing Test Cases (stm files) 10

2.6 The Regression Set 11

3. The SV Directed Test Bench 11

3.1 Recommended directory and file structure 11

3.2 Default Test Bench Structure 12

3.3 Implementation Variations 12

3.3.1 Internal Test Bench Variant 12

3.3.2 Multi Script Implementations 13

3.4 Script Parsing Conventions 13

3.4.1 Case 13

3.4.2 White space 13

3.4.3 Comments 13

3.4.4 Variables 13

3.4.5 Special Variables 14

3.4.6 Condition Variables 14

3.4.7 Number Notation: 15

3.4.8 Dynamic Text Strings 15

3.5 Search Order 16

4. Test environment Instructions 16

4.1 Default Instructions 16

4.2 User Defined Instructions 19

4.2.1 Tasks 20

4.2.2 Concurrency 20

5. Test Bench Working Details 20

5.1 SystemVerilog Classes 20

5.2 User Functions 20

6. Test Bench Generator tool 21

6.1 ttb\_gen Usage 21

7. Releases and Updates. 22

Appendix A: Examples 23

Appendix B: Updates 24

Table of Figures

Figure 1 - Default Test Bench Structure 12

Figure 2: Script Driven Processor Implementation 13

Index of Tables

# Introduction

## History

The VHDL test bench system was released in 2007. This version is the same implementation but is implemented using SystemVerilog (SV). Using some of the facilities of SV this implementation duplicates the scripting syntax of the VHDL version.

## Over View

The SV Directed test bench is a collection of SystemVerilog classes and functions which enable the user to create their own scripting instructions for directed test stimulus. The stimulus script or test case contains the instructions in a regular ASCII text file. The functionality of the instructions are coded in SystemVerilog as part of the test bench. The SV Directed package contains classes and functions to read, parse and execute the test script (stimulus file, test case, script). The script is evaluated in two passes. The first pass reads the instructions from the stimulus file, checks the validity of the instructions, adds valid instructions to the cmd\_lst object and creates the variable list. The first pass leaves everything needed in memory and happens at time zero of the simulation. The second pass is the execution pass. Instructions are referenced by their index numbers and return the instruction text, up to 6 parameters in integer form and possibly one text string. This is then fed down an “elsif” chain where the instruction text is used to choose the correct SV instruction sequence. At this point each instruction could be controlling the timing of the test case.

## Scope

This document provides the usage recommendations and detailed functionality of the test bench environment. It is expected that once this document has been read, the user will have the knowledge to use the environment.

## Interested Parties

All Verilog designers can benefit from the use of a simple but flexible foundation. The SV Directed test bench package provides a very good starting point for any effort requiring verification using the Verilog or SV language. This document should provide you all you need to use and implement the package.

## Why?

There are several reasons why you may want to use this test bench package

* Used the VHDL test bench package before and liked it.
* As a designer, want an environment that is quick and easy to use.
* You do not need a fully randomized test environment to do your verification.

## Document Acronyms

| Term | Definition |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# Test Environment Usage Flow

Obtain Verilog module

&

Decide the test bench

methodology

Run tb\_gen or

tb\_gen\_cmd to

generate base files

Edit generated files

to make them

complete

Connect clock driver

always output to

DUT

Get the environment

to compile without

errors

Create Reset, Read

and Verify

instructions in

tb\_mod file

Add required Models

and model access

instructions

Create an initial test

case that proves the

DUT will initialize

correctly

Create test case that

proves access to the

DUT

Create self checking

test case to prove

functional test plan

points

Code new instructions

or tasks into the

tb\_mod file

All Test Plan

test cases

complete?

New Instructions

or functionality

needed?

No

No

Yes

Yes

Stimulus file

regression set

ready

Illustration 1: Typical Test Bench Usage Flow

Consider the flow illustration above, Illustration 1: Typical Test Bench Usage Flow. This flow diagram presents a basic flow of how the test bench package can be used to create a verification environment, to test a DUT.

The following assumes that the DUT is a synchronous design that has a reset input.

## The DUT

To start using the environment it is required that the Verilog module of the DUT be available. With the DUT module file available, the test bench can be generated using the tb\_gen tool. The tb\_gen tool uses the pin definitions within the module definition to generate several parts of the test bench structure.

## Choose The Methodology

Depending on how you like to do your verification, the test bench will be coded differently. If you are going to create dynamic self-checking test cases or you are going to record vectors and compare to some good vector set, the test bench will be implemented in different ways. The commands that need to be created will be different.

As a preference, dynamic self-checking test cases should be the first choice. This will suggest that you will have to create commands to “read” various outputs, and “verify” the outputs are as expected.

Vector collection and compare should be the last choice of method. But this implementation may include creating instructions that enable the test environment to collect different outputs at different times.

## Generating the Test Bench

Using the tb\_gen tool, generate the test bench. Once the initial files have been generated, the test bench writer will then edit them for specifics. The tb\_mod.sv file is the users play ground. It is recommended that all edits, instructions and tasks be placed in this file, or a file that is included by this file. The other files that make up the test bench should not be edited because they can be regenerated if the module pinout changes significantly. The tb\_mod.sv file is optionally generated, as you would not want to over write the code that has been created for this test environment.

## Creating the Initial Instructions

The test bench package file cmd\_lst.sv contains several default instructions. Those are presented later in this document. It would be beneficial to review these instructions and be familiar with their implementation before creating your own instructions.

As a starting point, the first instruction that would be included in all test environments is the “reset” type instruction. This instruction is the one that applies the reset to the DUT and test bench elements. It is expected that all items need to be triggered to get into a default starting state. There may be several reset type instructions in one environment, one to reset everything, one to reset just the DUT and could be one to reset test bench models. These instructions would be created if the elements of the test environment need to be reset independent of each other.

Another very common instruction to consider is the “wait” type instructions. These instructions are used to time activities in a test case. For instance, an instruction to wait so many clock cycles is useful for just waiting for some know amount of time. A wait on interrupt is also useful to enable the test case to wait till some indicator triggers the test case to continue. As an addition to the wait instruction there should be a wait\_max type instruction that enables the test writer to set a time out for the wait. This is so that if the single event you are waiting for never happens, the environment can terminate the simulation.

To facilitate control of the DUT, write type instructions will be needed. This type of instruction will access the DUT possibly through an address and data buss or just the placing of values on input pins. Whatever the write type instruction(s) do, they will be the instructions used to setup and control the DUT.

To enable the collection of data, some kind of “read” instruction will be created. This instruction reads a target item, data bus, register, output pins, and puts the value into a common place. As in, all read instructions put the data in the same place, this could be a variable.

To confirm the data values are as expected, some kind of “verify” instructions will be created. The verify instruction(s) enable the test writer to check that the value of the last read item is as expected. Several types of verify instructions can be created, verify (whole word), verify\_slice, verify\_bit and they would all look to the same place for the data to compare.

The four basic instruction groups described above will enable the test bench developer to get started at the test bench / test case creation. As new instructions are needed, they are added.

Depending on the complexity of the test bench environment, there may be some instructions that are needed to initialize, access and or control test bench models.

## Writing Test Cases (stm files)

Once the test bench initial instructions have been created, test cases may start to be created. The test case is created by a person or persons through the use of a simple text editor. The test case writer will create the file containing instructions defined in the tb\_mod.sv file. As an example, please examine the following:

-- This is an example test case

-- using instructions defined in previous sections. << actual comment seen in test scripts

DEFINE\_VAR STAT\_ADDR x001000 -- status address variable (test bench default instruction)

DEFINE\_VAR CTL\_ADDR x001004 -- control register address variable

RESET\_SYS -- The reset instruction, DUT and test bench initialization.

WRITE $CTL\_ADDR x01 -- Write to the control register some value (assume DUT enable)

WAIT\_CYCS 1000 -- Waiting for 1000 clock cycles to go by.

READ $STAT\_ADDR -- Read some status to some internal variable after waiting

VERIFY x0055 -- Testing the read value is as expected.

FINISH -- Terminating statement for the test case (test bench default instruction)

The test case presented above is very simple. The syntax is presented later in this document. This test case could be created by a test writer with a simple text editor in a matter of moments. As the test case writer(s) progress into the test case writing phase of a verification effort, they will find that the initial instructions are not enough to do the testing they need to do. At any point new instructions may be added to a test environment. Once added to the environment the new instructions can be used in test cases. Test cases that are hand written are typically directed type tests. Unless the test bench is created with randomization instructions, most of the test cases written will be directed type test cases.

Another way to write test cases is to have a program generate them. Depending on the type of DUT, randomization may be a target test methodology. A test case generator can facilitate both ease of test writing and randomization. This is a method that should be considered when the DUT is very complicated.

## The Regression Set

It is assumed that the test case writer is working from a “Test Plan”. The Test Plan is a document which states what tests are going to be created and what functionality each test validates. The Test Plan is created from the document that contains the functional requirements of the DUT. The test case writer will create test cases (stimulus files) until all those stated in the test plan are complete and working as expected.

If the tools are available, it is now time to do code coverage. Using all the test cases created run them with code coverage enabled and merge the results. Any missing code should be evaluated and determined what was missed. Once the missed code is determined, existing tests can be upgraded or new tests created to cover the missed statements or branches. Once an acceptable level of coverage is achieved, it is considered that all of the test cases written constitute the full regression set.

# The SV Directed Test Bench

Usage of the environment can be done in many ways. Part of the objective of using a common verification environment is to use it in a common way. One thing about the test bench package and its components is that it is very flexible. If users implement in a common way then the ability to take up other's work becomes easier. Below are a few recommendations and then some details of the environment.

## Recommended directory and file structure

The following directory and file naming is used throughout the remainder of this document.

Design name directory -- the top level test directory name

sv -- The directory holding all test bench SV files

stm -- The directory holding all the test scripts (stimulus files)

sim -- The compile and run directory

The test bench comprises several files.

Package files:

gfuncts.sv general functions

tb\_types.sv definition of types

lst\_item.sv the list item class

tb\_cmd.sv the tb command class

cmd\_lst.sv the command list class

tb\_pkg.sv the package file, `includes the above files.

Test bench files:

dut\_if.sv the interface used to connect to the DUT (generated file)

tb\_mod.sv the module file, contains instructions (generated from a template)

tb\_top.sv the top module (generated file)

The package files need to be compiled with the environment. These files should not have to be modified by the user or test bench creator. The test bench files are generated or modified from a template by the tb\_gen or tb\_gen\_cmd scripts.

In a team setting it is recommended that team members follow a documented style and directory structure.

## Default Test Bench Structure

A pictorial representation of the default test bench environment is provided below in Figure 1 - Default Test Bench Structure. As can be seen the stimulus file is directly linked to the tb\_mod. This link is facilitated through the use of a parameter at the top level called STM\_FILE.

Usually, a test environment will contain many test scripts, and obviously they can not all have the same name. The user can control which test file will be loaded in many ways. One way is that the user copy the test file to stimulus\_file.stm. (this assumes the parameter points to this file by default, it may be modified to point to something else)

Other ways include using symbolic links, or passing in parameter changes when you start the simulator. (not detailed in this document, $value$plusargs)

DUT

Interface

tb\_mod.sv

tb\_top.sv

Stimulus file

Figure 1 - Default Test Bench Structure

## Implementation Variations

The default structure of the test bench is to have the test bench wrapped around the DUT. This is depicted in Figure 1 - Default Test Bench Structure. The file set generated by tb\_gen creates the structure that is presented in Figure 1 - Default Test Bench Structure. There are other ways that the test bench package can be used to facilitate other configurations.

### Internal Test Bench Variant

This implementation connects the script parser inside the DUT. For instance, if your DUT has an internal processor, the script parsing part of the test environment can be used to emulate the processor. This is depicted in Figure 2: Script Driven Processor Implementation, where hierarchy signals are used to drive and read the CPU module pins. The scripting commands are created such that they interface to the processor buses. They assign and react to signaling just as a processor would. Instructions can emulate assembly instructions exactly or instructions can be created to implement more abstract functionality. This is accomplished by instantiating the tb\_mod.sv in the top level test bench and using hierarchy connections down into an empty processor module.

Figure 2: Script Driven Processor Implementation, shows hierarchy connections can be used to drive and read top level I/O. This will be required to provide such things as clocks, and reset.

tb\_mod.sv

tb\_top.sv

Stimulus file

CPU

GPIO

ARB

MEM

DUT

Figure 2: Script Driven Processor Implementation

### Multi Script Implementations

Just say no. With the hierarchy connectivity of Verilog, it should not necessary to use two scripting parsers. That said, it is possible to create an environment that has two or more scripting parsers in it.

## Script Parsing Conventions

This section contains the details of the system with regards to parsing of the test scripts. The script parser is what the SV Directed Test Bench Package is. It is very limited, and all the particulars are stated in this section.

### Case

The case of all text in the stimulus file is significant.

### White space

There must be white space between fields with the exception being the comment. White space characters are, 'space' and tab.

### Comments

Comments can be added within the script file. The '--' is used as the comment delimiter, and when encountered parsing of the current line is halted. Anything on a given line, after and including the '--' character sequence is ignored.

Examples

ADD\_VAR TEMP 5 -- add 5 to the TEMP Variable

ADD\_VAR TEMP 5-- add 5 to the TEMP Variable

-- An all comment line

### Variables

Variables can be created with in the scripting environment. To create a variable the DEFINE\_VAR instruction is used. Variables can be defined anywhere in the script. The parser uses the DEFINE\_VAR instruction to create and add to the list of variables, a new variable. The first pass of the script parser puts the variables onto a link list in the order they were defined. Once a variable is defined, it can be referenced in two ways. One way is to specify to return the value of the variable: $var\_name returns the value of the variable. Second way, is to specify to return the index to the variable: var\_name returns the index to the variable.

Examples

DEFINE\_VAR VAR1 10 -- define variable 'VAR1' to have a value of 10

DEFINE\_VAR VAR2 20 -- define variable 'VAR2' to have a value of 20

.

.

READ\_DUT $VAR1 -- some kind of read instruction, passing a value of 10 to the instruction

ADD\_VAR VAR1 $VAR2 -- add to VAR1 the Value of VAR2

### Special Variables

To facilitate branching and jumping a special variable, we will call an “in-line variable”, can be created. This is a pointer type of variable which is not used like the variables created with the DEFINE\_VAR instruction. A text field terminated by the ':' character will create a variable with a value equal to the sequence number or 'line number' of the next valid instruction. This can then be used as a jump or call pointer.

Example

.

.

CALL $TEST\_FUNCTION

.

.

TEST\_FUNCTION:

ADD\_VAR VAR1 10

RETURN\_CALL

### Condition Variables

To facilitate the WHILE and IF instructions the parser will recognize condition fields. The cmd\_lst.get() function will search for condition code text. If the condition text is found, the relative value is returned into the elsif chain. The instruction must now act on the returned integer in the proper way. Usage examples can be seen in the WHILE and IF instructions.

The parser can detect the following fields and pass the indicated integer to the elsif chain.

| Text | Meaning | Return value |
| --- | --- | --- |
| == | Left side equals right side | 0 |
| != | Left side is not equal right side | 1 |
| > | Left side is greater than the right side | 2 |
| < | Left side is less than the right side | 3 |
| >= | Left side is greater than or equal to the right side | 4 |
| <= | Left side is less than or equal to the right side | 5 |

### Number Notation:

The stimulus file parser recognizes binary, decimal and hexadecimal numbers. For binary numbers the number is preceded by a 'b', lower case only. For a hexadecimal numbers the number is preceded by an 'x' or 'h', lower case only. If the first digit of the number is a decimal number character, the number is considered decimal notation. Any other character will make the parser consider the field a variable. All values are converted to integers within the test bench data records.

Examples

DEFINE\_VAR VAR1 10 -- define VAR1 to have value decimal 10

DEFINE\_VAR VAR2 xabc800 -- define VAR2 to have a hexadecimal value of 'abc800'

DEFINE\_VAR VAR3 habc800 -- define VAR3 to have a hexadecimal value of 'abc800'

EQU\_VAR VAR1 b0011001 -- equate the value of VAR1 to binary value '11001'

### Dynamic Text Strings

The stimulus file parser recognizes a text field within an instruction. The '”' (double quote) character defines the beginning of the text field. Anything after the '”' character is considered to be part of the text string, for that line of the stimulus file. There **must** only be one '”' in a line, indicating the beginning of the text field. The characters after the '”' are stored in a string and that string is returned into the elsif chain. The double quote is not included as part of the final string. There must be at least one white space character before the '”' character. Dynamic text strings must also be placed before any comment delimiter in that instruction line.

The text string feature is part of every instruction, and is not optional nor does it require any predefinition or pre-configuration. For every line of a stimulus file, if the '”' is encountered, the string is stored. The user may then do with this string as they wish. The tb\_pkg provides functions to print the text string to the console. Details of the two functions are provided in a section later in this document.

NOTE: Inline variables are not commands, any dynamic text on a line where an inline variable is defined, is ignored.

Note: any white space found at the end of a text string up to the comment delimiter will be included in any string output. The exception to this is when the text string is used for an include file, the extra white space is stripped off before using it for a file name.

Examples:

DEFINE\_VAR ACQ\_CTL1 xD0010034

DEFINE\_VAR VALUE 55

.

.

PPC\_WRITE $ACQ\_CTL1 x000DEF “Writing to acquisition control x0DEF

This will write to the console 'Writing to acquisition control x0DEF' each time this stimulus file line is run. This is provided that the user does a print\_str call. See section 8 for details of the print\_str procedure.

PPC\_WRITE $ACQ\_CTL1 $VALUE ” #&% Writing to $ACQ\_CTL1 $VALUE

This will write to the console ' #&% Writing to $ACQ\_CTL1 $VALUE' each time this stimulus file line is run. This is provided that the user does a print\_str call. See section 8 for details of the print\_str procedure.

Or, this will write to the console ' #&% Writing to 0xD0010034 0x37'. each time this stimulus file line is run. This is provided the user does a print\_str\_wvar call. See section 8 for details of the print\_str\_wvar procedure.

## Search Order

The test bench package implements a double linked list for the instructions. The tb\_trans class contains the full instruction list, return value and a next value. The next value is set by the user to state which instruction to retrieve. Depending on whether the desired instruction is after or before the current one, list pointers next and prev will be used to transvers instruction list.

# Test environment Instructions

## Default Instructions

As a starting point several default instructions are included in the test bench environment. These are considered among the most useful instructions or are nice to have as a common set among different test benches. Following is an explanation of each of the default instructions.

DEFINE\_VAR

is the only way to define / create a stimulus file variable. The implementation of this instruction is solely done within the test bench package. The DEFINE\_VAR instruction is not part of the instruction list.

ABORT

is the instruction that may be called in case of failure. If this instruction is encountered, the simulation is halted and a failure message is displayed.

FINISH

is the instruction which is called at the end of a simulation. When this instruction is encountered the simulation is halted and a message is displayed stating the simulation passed.

INCLUDE

is the instruction used to load in another stimulus file. The instructions found in the include file are inserted into the sequence of instructions as if they were part of the calling stimulus file. Includes cannot be nested. The file name may be specified in one of two ways. One way is to just state its path and name, no quoting required.

Example:

INCLUDE stm/include.stm

The second way is to use the text string method. This allows the text path name to be the length of type string.

Example:

INCLUDE “C:/work/dir1/dir2/dir3/dir4/dir5/stm/include.stm

EQU\_VAR

is the instruction used to change the value of an existing variable.

ADD\_VAR

is the instruction used to add a value to an existing variable.

SUB\_VAR

is the instruction used to subtract a value from an existing variable.

CALL

is the instruction use to jump execution to a subroutine. The use of this instruction should include a RETURN\_CALL instruction as there is a stack maintained in the back ground. The limit to the nested call depth is 8.

RETURN\_CALL

is the instruction which terminates a CALL sequence. This instruction will return sequence execution to the point from which is was called.

Example:

DEFINE\_VAR TEMP\_DAT x0

DEFINE\_VAR TEMP\_ADD x01000

.

.

.

EQU\_VAR TEMP\_ADD x02000

CALL $ACCESS\_DUT

.

.

.

FINISH

ACCESS\_DUT:

READ\_DUT $TEMP\_ADD

WRITE\_DUT $TEMP\_ADD $TEMP\_DAT

ADD\_VAR TEMP\_ADD 4

RETURN\_CALL

LOOP

is a simple loop instruction. Used to execute a set of instructions a number of times between it and the END\_LOOP instruction.

END\_LOOP

is the instruction used to terminate a loop instruction.

Example:

DEFINE\_VAR ADDR x80

DEFINE\_VAR DATA x20

.

.

LOOP 5

WRITE\_DUT $ADDR $DATA

ADD\_VAR ADDR 4

END\_LOOP

JUMP

This instruction is used to go to particular location in the script. NOTE: When a JUMP instruction is encountered, all WHILE and CALL stacks are zeroed. This is to prevent problems with jumping out of one of these constructs.

Example:

JUMP $NEXT1

.

.

NEXT1: -- <<< NOTE: Inline variable

ADD\_VAR $VAR1 20

IF, ELSEIF, ELSE, END\_IF

These instructions form the structure for an “if” type condition script sequence. The implementation is that of a regular if statement found in other languages. The current implementation cannot operate on nested if statements.

Example:

IF $var < 10

ADD\_VAR var 1

ELSEIF $var = 10

EQU\_VAR var 2

ELSE

EQU\_VAR var 2

END\_IF

WHILE, END\_WHILE

These instructions make up the structure for a “while” type condition script sequence. The implementation is that like any other language. If a JUMP instruction is used with in a WHILE loop, all while loop status is zeroed. This means that if you jump back into a while loop, it may not work as expected.

The WHIILE instruction can not be nested.

Example:

WHILE $var != 10

ADD\_VAR var 1

END\_WHILE

## User Defined Instructions

The user will be required to create instructions for use in test case writing. These instructions will be more specific for the DUT than those default instructions of the test bench environment. Once the instruction is defined using the define\_instruction function call, the user then codes the new instruction into the elsif chain. The default instructions are coded as part of the cmd\_lst class. Simply copy an elsif line and replace the existing instruction with the new instruction text, remembering case sensitivity. Then code the SV below the condition statement to implement the required functionality.

The best instruction is one that says something about what it does. Also, flexible instructions are very good to reduce test writing complexity. For instance, if you had a DUT with 20 ports, creating an instruction to read data from every port you would have 20 different instructions. By using the parameters, you can case on one of them and use one READ instruction to read all 20 different ports.

Example SV code in the tb\_mod.sv file

/////////////////////////////////////////

end else if(cmd\_string == “READ”) begin

case (r.rtn\_val.par1)

0: temp\_data = port1;

1: temp\_data = port2;

2: temp\_data = port3;

3: temp\_data = port4;

.

.

endcase

Script code example:

DEFINE\_VAR PORT1 0

DEFINE\_VAR PORT2 1

DEFINE\_VAR PORT3 2

DEFINE\_VAR PORT4 3

READ $PORT1

READ $PORT2

READ $PORT3

READ $PORT4

### Tasks

The use of tasks within the tb\_mod file is recommended. Interface procedures are good things to put into a task. Once created, any instruction created can take advantage of the task, saving coding and creating a single point of interface.

### Concurrency

At some point it will become apparent that something will have to happen at the same time as something else. Some kind of complicated instruction may be considered. At this time, a model to relieve the stimulus file from producing everything, should be created. The use of models will enable many things to be taking place at the same time, and the stimulus file controlling and checking as the test progresses. Such things to be considered for a model implementation are objects like memories, data generators, data checkers and standard protocol interfaces.

# Test Bench Working Details

The tb\_mod file contains all clock sources, models connected to the DUT, instruction definition and any other processes or models used to monitor or control the DUT. An initial block handles the stimulus file parsing and script execution process. Details are:

## SystemVerilog Classes

lst\_item:

this class is used to implement a basic list of items. In the test bench system it is used for the variable, files and valid instructions lists. The valid instruction list is the list that is created by the define\_instruction calls. The variables and files lists are created as the stimulus file is parsed.

tb\_cmd:

this class is used to hold the stimulus commands in the form of a double linked list. It holds all the variables needed to store a stimulus file command or line. This class contains a “self” parser method, which parses a string into itself.

cmd\_lst:

this class implements the command system. It has many functions that are needed to implement the test bench system. This includes user functions, define\_instruction, print\_str\_wvar and many others the user may never have to use.

tb\_trans:

this class is a container class used to pass in “next” and return the values from the command list. This class enables the user to have full access to the command list as required by their implementation.

## User Functions

cmd\_lst::define\_instruction

is the function that must be called to add a new instruction to the list of valid instructions. It is defined as define\_instruction("INSTRUCTION\_TEXT", Number of Parameters). The user must define the “INSTRUCTION\_TEXT” and number of parameters. The instruction text is what the user wants to make the instruction read in stimulus files, and case matters. The number of parameters is defined by the user and checked by the parser during the stimulus file reading. The dynamic text string does not require configuration and is not optional, it is part of every instruction.

NOTE: If the number of parameters is defined to be more than 6, the parser will not check for the correct number of parameters. This enables a variable number of parameters in an instruction.

cmd\_lst::get(tb\_trans inst)

is the function used to get a command from the command list. The inst.next is set to the index that is to be retrieved. The instruction is returned in inst.rtn\_val. Instructions modify the inst.next value as needed to access the instructions or by default inst.next is incremented.

# Test Bench Generator tool

To make it as easy as possible to get started quickly, a tool to generate some of the default code is provided. tcl/tk scripts in command line and GUI form are provided. The scripts will generate, from a module file input, the interface file (dut\_if.sv), the tb top (tb\_top.sv) and will write out the module file (tb\_mod.sv) from the template. The other files are part of the package and are static.

## ttb\_gen Usage

The test bench generation tool is called tb\_gen, a tcl/tk GUI. The command line version is tb\_gen\_cmd.tcl a standard tcl script. It is assumed that tcl\tk is installed on your Windows or Lenox computer. If not please follow this link: [Activestate](http://www.activestate.com/Products/ActiveTcl/) . Once tcl\tk ready, for Windows, you should be able to invoke the application by just double clicking on the program in the “My Computer” browser tool. In Lenox, with the tcl bin directory in your search path, type the script you wish to run on the command line of your shell.

With GUI version invoked the user can use the Browse ... button to select the file that contains the module definition of interest or you can type the path and file name into the Source field. If the Browse button is used the Destination field is filled by default to be the source directory.

The reason for the option to generate the tb\_mod.sv file or not, is so that you can avoid over writing an existing test bench module file. For instance, after some time the DUT may have a significant pinout change. This will cause the test bench files to have to be edited. The interface and top level files can be regenerated, avoiding hand editing errors.

tb\_gen will parse out the first module it finds in the source file, and halt the search for any other module definitions. Once the end of the module definition is found, file parsing halts, the file is closed and that stored in memory is used from then on.

So far only 3 pin directions are supported, input, output and inout. tb\_gen is NOT a full Verilog parser, and uses the pin input, output and inout to find pins within a module.

The tool copies the test bench module file from the template included in the package. Those constructs that are needed to implement the test bench are inserted at the appropriate places.

For those that do not like GUI’s, a command line version of tb\_gen.tcl is provided. tb\_gen\_cmd provides a command line interface for test bench generation.

# Releases and Updates.

This release is the first release.

Appendix A: Examples

Example standard test bench:

A simple example of a test bench implementation is provided with the SVN repository. This example contains a working behavioral model of a couple registers driving some output pins. The user can control the value of the output pins by writing to the stimulus access port address corresponding to the output desired. A read instruction is included to read the value of the example DUT outputs. A verify instruction is used to check the value that was read is as expected. This demonstrates self-checking test cases.

The following files are found in the SVN examples/standard/sv directory:

dut.sv

du\_if.sv

tb\_mod.sv

tb\_top.sv

The example stimulus file is found in the SVN examples/standard/stm

stimulus\_file.stm

Example internal test bench:

A simple example of how the test bench system can be connected to an internal block of the system. In this example the CPU module is used as the interface point. This enables the scripting system to emulate the processor and access the CPU facilities. The example includes a memory and a GPIO block that can be addressed from the CPU module. The stimulus file does several writes and reads to prove access to memory and input GPIO values.

The following files are found in the SVN examples/internal/sv directory:

dut\_top.v

arb.v

cpu\_mod.v

gpio\_mod.v

mem\_mod.v

tb\_top.sv

tb\_mod.sv

dut\_if.sv

The example stimulus file is found in the SVN examples/internal/stm

stimulus\_file.stm