VHDL Test Bench Users Guide

by

Ken Campbell

Version: 2.0 Beta

May, 2011

Revision History:

Version	Revised By	Description	Date
1.0	Ken Campbell	Initial release	March 20, 2007
1.1	Ken Campbell	Update for changes to the tb package	September 1, 2007
1.2	Ken Campbell	Update for faster script access	February 24, 2008
2.0	Ken Campbell	Update for variable parameters in a stimulus command Update for missing file name variable content	April, 2011

Table of Contents

Table of Figures

Index of Tables

1. Introduction

1.1 History

The VHDL test bench system contained in its release and described in this document, began in 1996 when computers were not as they are today. It was created to provide a lean but flexible test bench system. The environment was implemented based on a presentation at a 1996 VHDL conference, by a Semi-conductor manufacturer representative. The first implementation was created by an employee of a telco equipment manufacturer, after attending the conference. This enabled better testing of ASIC and FPGA designs. The computer initially running the environment was a Sun Systems Sparc 5. The environment was created to be sparse in order to save as much of the memory for the design as possible. It was about this time that I, "the author" of this document and GNU VHDL test bench, started using this environment and began to specialize in the HDL verification field. An application was created to make the initial test bench VHDL structure files, a push button operation.

As computing power increased the facilities of the VHDL test bench got upgraded and enhanced. Variable manipulation and the "include" function are a couple of the enhancements made. The user base got to an estimated 40+ users and was introduced to Europe offices of the telco company. One European user gave the environment a very nice review, sighting ease of use, size and documentation qualities. After six years of using and upgrading the environment, the author changed employers.

At the new location, the year being 2003, the lack of a test environment incited the author to write a new VHDL test bench package for use at the new employer. While doing so, many complaints from the past, about various lacking functionality were addressed. Many messages were added to provide better debugging information to the test bench user. The development of the new package was done on a Windows platform using Modelsim's PE VHDL simulator. This effort took approximately one week of full time coding. With the computing power increase, the test bench system was created to have as few limits as possible. The addition of more passed parameters and the output text string facility enabled the 14 person design team to implement some very complicated mixed HDL designs. The environment proved once again that the whole team can be writing test cases, at the same time, using a single common environment.

In 2007,with the environment having been used to develop some 50+ design blocks, 6 FPGA designs and one ASIC, it is believed that all the bugs have been discovered. All dependencies on tool specific packages have been removed. The VHDL test bench package is now considered stable and portable. At the request of the author, the employer agrees to allow the author to release the VHDL test bench package as a GNU free software offering.

1.2 Overview

The VHDL test bench is a collection of VHDL procedures and functions which allow the implementer to create their own scripting instructions for test stimulus. The stimulus script or test case contains the instructions in a regular ASCII text file. The function of the instructions is coded in VHDL as part of the test bench. The test bench VHDL package contains procedures 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 instruction sequence (inst_sequ) and creates the variable list (defined_vars). 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 line numbers and return the instruction text, up to 6 parameters in integer form and one text string pointer. This is then fed down an elsif chain where the instruction text is used to choose the correct VHDL instruction sequence. At this point each instruction could be controlling the timing of the test case.

1.3 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. For more information see the link in Other sources of Information.

1.4 Interested Parties

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

For those in school, needing a predefined verification environment or wanting to study a verification environment, this package should provide a good example of how VHDL can be used in different ways.

1.5 Why?

There are several reasons why you may want to use this VHDL test bench package. The fact that you may only have access to VHDL or VHDL is your main HDL language, is one good reason. The package implementation is pure VHDL with a little generation application that produces VHDL code.

Another reason to use this system could be that you and/or your team do not have a standard method of encapsulating your DUT in a test environment. The implementation of a common test environment, enables reuse, quicker refresh on old work, better interaction between team members and a general feeling of security in something you know.

Portability is another reason to use this VHDL test bench system. It is not that a test bench you create on a specific tool set that may be portable. The environment it self will be. If for instance you were forced to change tool sets in the middle of a project, if you did not use tool specific functionality in your implementation, you should be able to just recompile and run your tests.

A small list of reasons to use the VHDL test bench system:

- ease of use
- implementation limited mostly by user skill level
- encourages reuse and in turn reduces test bench creation time frames
- enables multiple test case writers to be writing on the same test bench at the same time
- changes to a test case do not require recompilation of anything
- Industry proven implementation that provides results
- You wish to only use one language, VHDL, for all design and testing
- this is a free package

1.6 Other sources of Information

A blog has been started by the author, many of the topics found in this document are elaborated on there.

http://vhdltb.blogspot.com/

1.7 Document Acronyms

Term	Definition
bhv	Short for behave
DUT	Device Under Test, could be the design RTL or a model
elsif	A VHDL construct, is the 'else if' part of an if statement
ent	Short for entity
FPGA	Field programmable Gate Array
implementer	the person editing and creating the test bench and test bench commands
Script file	Same as the Stimulus file
Stimulus file	The file which contains test case instructions, this is the default name of the file loaded by the test bench environment.
str	Short for structure
Test case	Same as a Stimulus file
tb	Test bench
ttb	Top level test bench
ttb_gen_gui	Top Level Test Bench Generator tcl\tk application
user	A user is the person editing and creating the test cases using test bench commands
VHDL	Programming language used to define FPGA / ASIC logic

2. Test Environment Usage Flow

This section presents the way the environment would typically be used by the author.

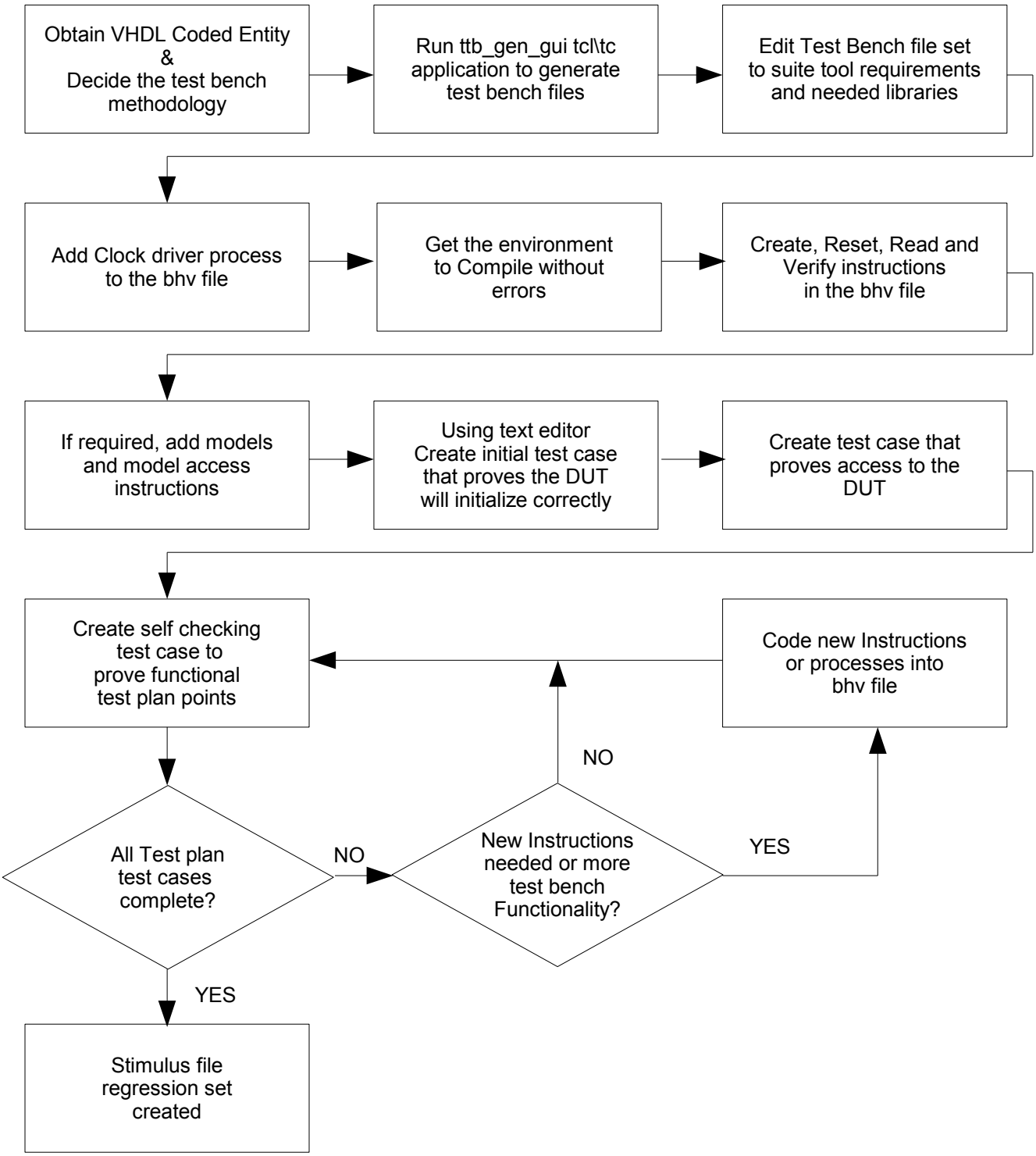

Illustration 1: Typical Test Bench Usage Flow

Consider the flow illustration above, Illustration 1: Typical Test Bench Usage Flow. This flow diagram presents a basic flow of how the VHDL 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.

2.1 The DUT

To start using the environment it is required that the VHDL coded entity of the DUT be available. It is not required that the architecture of the design be available, but the entity is the starting point. If the DUT architecture is not available and you would like to ensure your environment will compile, replace the DUT architecture with an empty definition. If later the DUT RTL is still not available, the verification person can code behavioral code into the empty architecture to enable initial test bench functionality to be created and confirmed.

With the DUT entity file available, the test bench can be generated using the ttb_gen_gui tool.

2.2 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.

As a last choice, vector collection and compare should be the chosen method. But this implementation may include creating instructions that enable the test environment to collect different outputs at different times.

2.3 Generating the Test Bench

Using the ttb_gen_gui tool, generate the test bench. Once the initial files have been generated, the test bench writer will then edit them for specifics. In the top level entity file, ttb_ent, the stimulus_file generic default value is replaced with the default path to the stimulus file. The stimulus_file generic is a pointer to the file that the test bench parses for instructions on what to do and when. The library includes and use statements need to be updated to include any DUT specific libraries. The ttb_str file needs to be updated at the instantiation statements depending on the structure of the design. The tb_ent file has to be updated only if you need more libraries for the behave file. The tb_bhv file will be added to and edited many times as the test bench grows and is developed. But as a starting point the clock generation is done in this file. The test bench creator will have to add the clocking process(s) to meet the needs of the DUT. Once all this is done, the test bench should be compiled. At this point it is an empty test bench environment and it should compile with out errors. Note that the elsif loop has no wait statements yet, and your simulator may complain. This is ok because you will be adding instructions with wait statements and this warning will go away.

2.4 Creating the Initial Instructions

The test bench bhv file template 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. What ever 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.

2.5 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 test bench bhv file. As and 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  -- define status address variable  (test bench default instruction)
DEFINE_VAR CTL_ADDR  x001004     -- define 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. Some of 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 you are testing, 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.

2.6 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. 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.

3. The VHDL 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 is a few recommendations and then some details of the environment.

3.1 Recommended directory and file structure

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

Design name directory -- the top level test directory name, point of simulator execution
vhdl -- The directory holding all test bench VHDL files
stm -- The directory holding all the test scripts (stimulus files)
work -- The compile directory for compiled test bench VHDL (not usually DUT)

The test bench comprises four VHDL files.
(name)_ttb_ent.vhd -- Top entity, empty except for stimulus_file generic
(name)_ttb_str.vhd -- Top level structure, connect DUT to test bench
(name)_tb_ent.vhd -- tb entity, is exact copy of DUT entity except for pin direction
(name)_tb_bhv.vhd -- tb behave, contains all that is not the DUT

It is highly recommended that a "team" document and follow a recommended test environment directory structure. When starting a new test bench creation, use ttb_gen_gui to generate the test bench for you and get the template from the central location. You should find a TCL/TK application called ttb_gen_gui included with the VHDL Test Bench package offering. The application is described section 6. Some modifications to ttb_gen_gui.tcl will have to be made to make it work in a centralized team or group environment.

3.2 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_bhv. This link is facilitated through the use of a generic at the top level ttb_ent called stimulus_file.

Usually, a test environment will contain many test scripts, and obviously they can not have all the same name. The user can control which test file will be loaded in one of two ways in windows environments. First the user can just copy the test file to stimulus_file.stm. (this assumes the generic points to this file by default, it may be modified to point to something else) Or the user could just edit the default file and rename it later as needed. The second way to control which stimulus file is loaded is to pass a new value to the top level generic. Modelsim allows this and the syntax is:

vsim work.pci_ttb_top -Gstimulus_file=sim/test3.stm

This says, start vsim and load compiled object pci_ttb_top from the work library, and all so assign to the stimulus_file generic a value of 'sim/test3.stm'. This method is used for regressions, and is the reason for adding it to the top level entity.

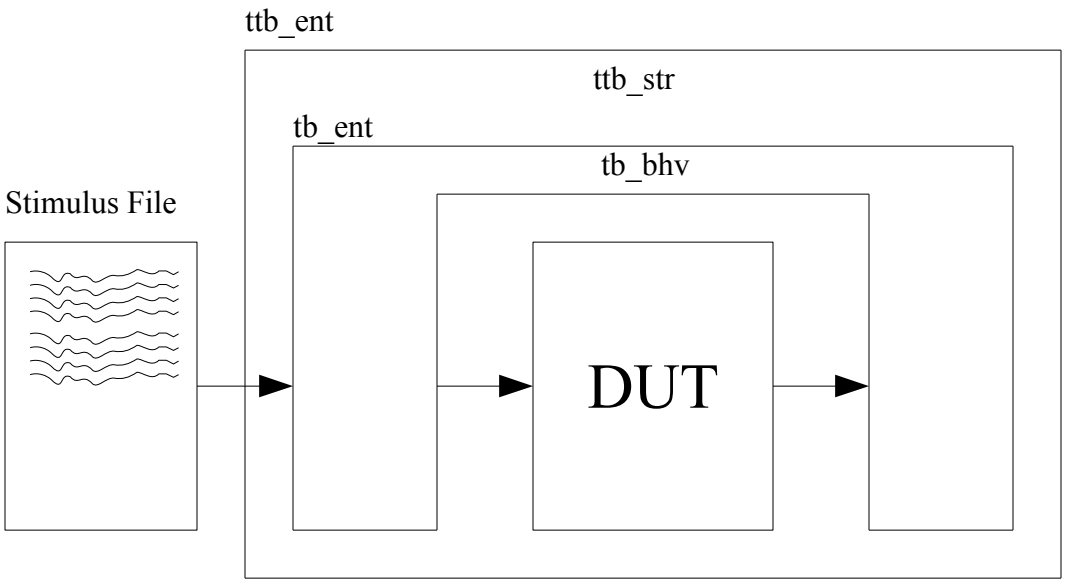

Figure 1 - Default Test Bench Structure

3.3 Implementation Variations

The default implementation of the VHDL 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 ttb_gen_gui 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.

3.3.1 Internal Test Bench Variant

This implementation puts 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 the the stimulus file is linked to the internal processor block. 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 making the bhv file the architecture of the processor entity. Then adding the instructions and VHDL code to implement them.

As can be seen in Figure 2: Script Driven Processor Implementation, there is supporting test bench logic out side the DUT. This will be required to provide such things as clocks, memory and reset. If control of the supporting test bench logic requires dynamic actions, there may have to be links from the processor to outside the DUT. Some tool sets provide this facility, but VHDL does not allow arbitrary access across entity boundaries. For instances where control of the over all test bench can not be facilitated from one control point, a duel or multi script system may be considered. See the following section.

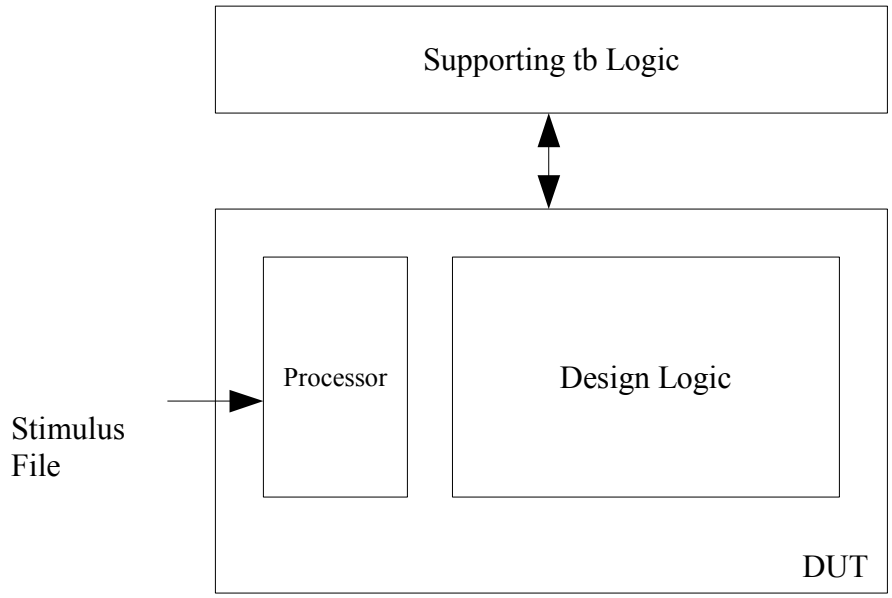

Figure 2: Script Driven Processor Implementation

3.3.2 Multi Script Implementations

The test bench implementation depicted in Figure 3: Multi Script Variant Implementation, is a variant of a previous figure.

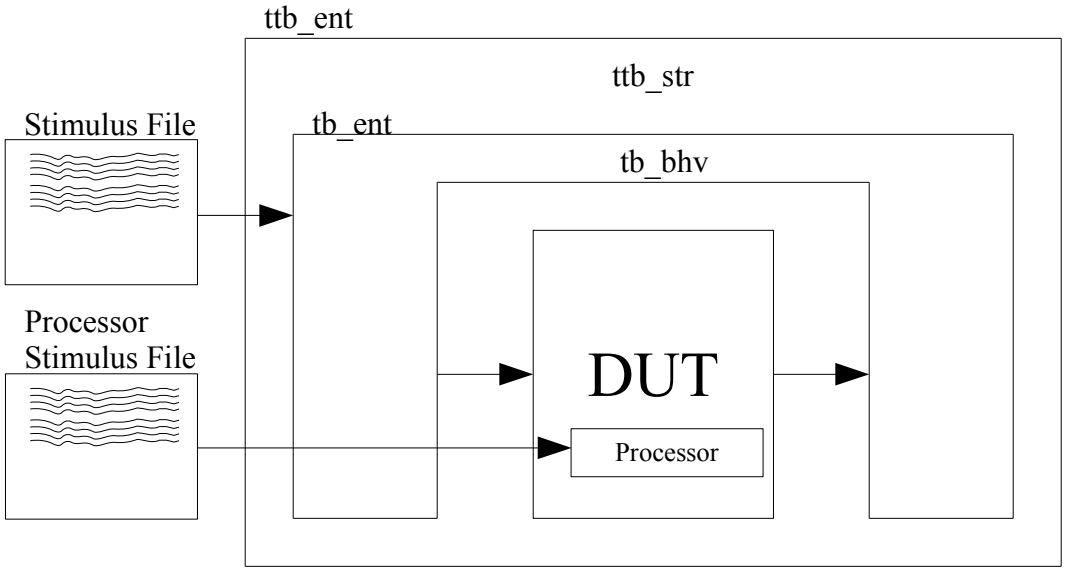

Figure 3: Multi Script Variant Implementation

As can be seen in Figure 3: Multi Script Variant Implementation, the test bench is reading two stimulus files. One script is being read by the top level bhv file and the other by an internal processor bhv file. The bhv file of the processor can be considered a bus functional model (BMF) of the processor. Each of the two bhv files, usually point to a different stimulus files by default. This is done by assigning the default stimulus file names to be different for each VHDL entity generic. The top level test bench stimulus file can be very simple if all that is required is to provide reset and then wait. It could also be very complicated by controlling external models, asserting external inputs, monitoring external outputs.

Once the user realizes that the VHDL bhv file is the heart of the system and that the rest is just structure, they will be able to implement complicated variations. It should be remembered that the system was created to be

simple, but flexible. Multi script systems will make it more complicated to create test cases and will result in a more complicated test bench implementation.

3.4 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 VHDL Test Bench Package is. It is very limited, and all the particulars are stated in this section.

3.4.1 Case

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

3.4.2 White space

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

3.4.3 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
```

3.4.4 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
```

3.4.5 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' it is currently on. This can then be used as a jump or call pointer.

Example
.
.
CALL $TEST_FUNCTION
.
.
TEST_FUNCTION:
  ADD_VAR VAR1 10
RETURN_CALL

3.4.6 Condition Operator Constants

To facilitate the WHILE and IF instructions the parser will recognize condition fields. The access_variable procedure will search the first character of each text_field for condition code text. This means that the special characters can not be used at the start of variable names. If the special 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 then 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

3.4.7 Number Notation:

The stimulus file parser recognizes binary, decimal and hexadecimal numbers. For binary numbers the number is proceeded by a 'b', lower case only. For a hexadecimal numbers the number is proceeded by a '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'
```

3.4.8 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 a pointer to that string is returned into the elsif chain. The only thing that will terminate the string parsing is the end of the string, the comment delimiter or max string length reached (200). The maximum string length is 200 characters, and any string longer than that will be truncated at character 200. The double quote is not included as part of the final string. There must be at lease 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 pre-definition or pre-configuration. For every line of a stimulus file, if the "" is encountered, a pointer to a string is created. The user may then do with this string as they wish. The tb_pkg provides procedures to print the text string to the console. Details of the two procedures are provided in a section later in this document.

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 txt_print call. See section 8 for details of the txt_print 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 txt_print call. See section 8 for details of the txt_print 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 txt_print_wvar call. See section 8 for details of the txt_print_wvar procedure.

Another way to use the dynamic text is for setting environment strings from the stimulus file. With the following definition, a string (fname) can be written to.

Define a signal of type stm_text, a string subtype defined in the test bench package.

```
Signal fname  :  stm_text;
```

.

.

Then define the instruction to assign the passed txt pointer to the signal.

```
elsif (instruction(1 to len) = "SET_FN") then
  i := 1;
  while(txt(i) /= nul) loop
    fname(i)  <=  txt(i);
    i := i + 1;
  end loop;
```

Once the file name has been set, the string could be used to open files for loading or dumping data.

3.5 Search Order

The test bench package, release Feb 2008, changed the searching method for searching the list of instructions, during the execution phase. Two variables were added to the access_inst_sequ call that enable the searching to start from where the last instruction was accessed most of the time. This differs from previous package versions in that they searched from the top of the list each time. This change saves the CPU having to search the list of instructions each time, except if the line is before the one that was last accessed. In the case where the instruction line being searched for, is before the last instruction accessed, the system starts searching from the beginning of the list of instructions. The order of the instructions on the list of instructions, is the same as the were encountered when the package parsed the script. As an example, LOOP type instructions will cause the system to search for the top of the loop, from the start of the instruction list, each iteration through the loop. Large scripts will also benefit greatly from the new search implementation.

4. Test Environment Instructions

4.1 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 added to the instruction list, as of version 1.2 of the test bench package.

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 can not be nested. The file name may be specified in one of two ways. One way is just stated, its path and name, no quoting required. This limits the path name length to the size of type text_field (48 characters).

  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 stm_text (200 characters).

  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 can not 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 be nested to a level of 8.

Example:
```
WHILE $var != 10
  ADD_VAR var 1
END_WHILE
```

MESSAGE_ON

is the instruction which sets the messages variable to false. This then makes all assert statements fail and messages are put out.

MESSAGE_OFF

is the instruction which sets the messages variable to true. This then makes all assert statements pass and messages are suppressed.

4.2 User Defined Instructions

The test bench implementer is 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. An instruction is defined using the define_instruction procedure call, the user then codes the new instruction into the elsif chain. instructions are created below the default instructions of the test bench template. copy an elsif line and replace the existing instruction with the new instruction text, remembering case sensitivity. code the VHDL 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 VHDL code in the bhv file

```
--------------------------------------------------------------------------------
    elsif (instruction(1 to len) = "READ") then
      Case par1 is
        when 0 =>
          temp_data  <=  port1;
        when 1 =>
          temp_data  <=  port2;
        when 2 =>
          temp_data  <=  port3;
        when 3 =>
          temp_data  <=  port4;
         ....
```

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
```

4.2.1 Commands with Variable Number of Parameters

The 2011 release of the test bench package added one feature, commands with variable parameters. This feature is configurable on a command by command basis through the “args” field. If the “args” field is defined to be greater than six, the check for the correct number of parameters is by passed. The onus is on the implementer to test the command for correct number of parameters, or ignore anything more than needed. By default, the parsing of the script involves testing for incorrect syntax before the simulation starts. This feature is in place, for all other commands, to save time by avoiding the discovery of scripting syntax errors at the time of execution. Which as stated, could be hours into a simulation.

The number of parameters that are valid is contained in the stimulus line record. This is one of the few additions to the test bench package. The implementer has access to this value through the last_sequ_ptr variable, the valid_fields field. This enables the implementation to check for the number of valid parameters and act accordingly. Still, the discovery of scripting errors in a test case will not be discovered until that line executes.

4.2.2 Procedures

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

4.2.3 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 VHDL 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 interfaces.

5. Test Bench Working Details

Of the four files that make up the test bench, as stated above, the (name)_tb_bhv.vhd file is the most important. This file should house all models and all VHDL needed to deal with the DUT. While generating the test bench, using the ttb_gen_gui application, the bhv file is an optional generation. If the DUT changes its pin out, ttb_gen_gui can be used to regenerate the other three files, but not the bhv file. This file is initially copied from the template, and once you are well into a test environment you will not want to replace that file. The template is coded to meet the VHDL 93 coding standard. The required VHDL standard packages are:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use std.textio.all;
```

The bhv file is intended to contain all clock sources, models connected to the DUT, instruction definition and any other processes or models used to monitor or control the DUT. The process that handles the stimulus file parsing and execution is the Read_file process. Details are:

5.1 VHDL Variables

inst_list
: is the linked list of defined instructions. This is the list that is created by the define_instruction procedure as per user requirements. Also several default instructions are defined to provide some basic functionality. The order of the list is the same as the order of the "define_instruction" calls. All searches to this list are done from the top (first) down. Warning: this variable must not be modified by the user.

defined_vars
: is the linked list of defined variables. This list is created as DEFINE_VAR instructions are encountered or in-line variables are found in the stimulus file during parsing. The order of the list is the same as the order of the "DEFINE_VAR" stimulus file calls. All searches to this list are done from the top (first) down. Warning: this variable must not be modified by the user.

inst_sequ
: is a linked list of instructions. This is the test as it was read from the stimulus file, with only the instructions. Warning: this variable must not be modified by the user.

file_list
: is a link list of file names with index. The index is used instead of the full file name in the inst_sequ. Warning: this variable must not be modified by the user.

last_sequ_num
: is the sequence number of the last instruction recovered from the list of instructions. This variable is used to determine which way to search through list of instructions. Warning: this variable must not be modified by the user.

last_sequ_ptr
: is the address of the last instruction recovered from the list of instructions. This variable is used to access the list of instructions. Warning: this variable must not be modified by the user.

instruction
: is the text field of the instruction, the instruction text itself. This is a return parameter of the access_inst_sequ procedure call, and is of type text_field.

par1
: is the first field of an instruction. This is a return parameter of the access_inst_sequ procedure call and is of type integer.

par2

is the second field of an instruction. This is a return parameter of the access_inst_sequ procedure call and is of type integer.

par3
: is the third field of an instruction. This is a return parameter of the access_inst_sequ procedure call and is of type integer.

par4
: is the fourth field of an instruction. This is a return parameter of the access_inst_sequ procedure call and is of type integer.

par5
: is the fifth field of an instruction. This is a return parameter of the access_inst_sequ procedure call and is of type integer.

par6
: is the sixth field of an instruction. This is a return parameter of the access_inst_sequ procedure call and is of type integer.

txt
: is the pointer to the text string, if any, of this stimulus line. Is of type stm_text_ptr.

len
: is the length of the instruction in number of characters. This is a return value so that the exact length of an instruction is known for compares. len is of type integer.

file_name
: is the file passed into the system by the stimulus_file generic. file_name is of type text_line.

file_line
: is the line number in the text file that this sequence is from. This is a return value from the access_inst_sequ procedure call and is type integer. This is provided so that user commands can message out the line number from the stimulus file as needed.

line
: is the sequence number of the current instruction. This value is the actual line or sequence number in the test sequence. It is of type integer and is used to index into the instruction sequence. User modification of this value will effect the next line recovered from the sequence. (not normally modified by the user instructions)

5.2 VHDL Procedures

define_instruction
: is the procedure that must be called to add a new instruction to the list of valid instructions. It is defined as define_instruction(inst_list, "INSTRUCTION_TEXT", Number of Parameters). The user definable parts are, 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 file read procedure. If the number of parameters is set to seven or more, the checking by the parser is by passed. This enables the implementation of commands with a variable number of parameters. The dynamic text string does not require configuration and is not optional, it is part of every instruction.

read_instruction_file

```
------------------------------------------------------------------------
-- Read, test, and load the stimulus file
read_instruction_file(stimulus, inst_list, defined_vars, inst_sequ, file_list);
```

: is the procedure that reads the stimulus file into memory as an instruction sequence. Various checks are performed and if errors are found, the simulation is halted and the user is given an error message. This procedure is considered the first pass of the stimulus file parsing. It is called once and is part of the system. There is no user intervention required, nor should the user modify this procedure call.

access_inst_sequ

access_inst_sequ(inst_sequ, defined_vars, file_list, v_line, instruction,
par1, par2, par3, par4, par5, par6, txt, len, file_name, file_line,
last_sequ_num, last_sequ_ptr);

is the procedure which accesses the instruction sequence in memory, indexed by its sequence number. (line) The procedure returns the instruction text, returns converted variables to integer or index, returns a pointer to any text, and returns the line number in the stimulus file this sequence is from. Also, the last sequence number and pointer to the last sequence number are passed in and out.

6. Test Bench Generator Tool

The test bench files that make up the structure of the connectivity are rather tedious to create. To make this process a push button operation, a new tcl\tk code generator is provided.

6.1 TTB Gen Plus 2 Usage

7. The Test Bench And BFMs

The test bench system does, in no way, force any particular implementation methodology for BFM creation and usage. From the authors past experience, it has been observed that a common BFM strategy has many benefits. A standard method for BFM building and interface has been created for use with the VHDL test bench. This section presents and describes this standard method. An example provided in the release, uses the stimulus access port on a model to show and example of test bench usage.

7.1 The Stimulus Access Port

Version 2 of the test bench package defines record types for use as stimulus access ports.

```
-- define the stimulus slave control record types
type stm_sctl is record
  rst_n       : std_logic;
  addr        : std_logic_vector(31 downto 0);
  wdat        : std_logic_vector(31 downto 0);
  rwn         : std_logic;
  req_n       : std_logic;
end record;
type stm_sack is record
  rdat        : std_logic_vector(31 downto 0);
  ack_n       : std_logic;
  rdy_n       : std_logic;
  irq_n       : std_logic;
-- define the stimulus master control record types
type stm_mctl is record
  addr        : std_logic_vector(31 downto 0);
  wdat        : std_logic_vector(31 downto 0);
  rwn         : std_logic;
  req_n       : std_logic;
  breq        : std_logic;
end record;
type stm_mack is record
  rdat        : std_logic_vector(31 downto 0);
  slv_rdy     : std_logic_vector(15 downto 0);
  slv_irq     : std_logic_vector(15 downto 0);
  ack_n       : std_logic;
  bgrant      : std_logic;
end record;
```

Type stm_sctl is the stimulus slave control bus and is originated from a master. Type stm_sack is the stimulus slave acknowledgment bus and is driven by the selected slave. The master types are for advanced test benches where BFMs take control of the stimulus bus. Otherwise, the stimulus file is the default master in the stimulus bus system. The data bus of the stimulus bus system has been made separate for read and write operations. This prepares the bus system for implementation on an FPGA with synthesized test bench BFMs.

The stm_sctl type contains all the connections needed to control the BFM from the stimulus file or master BFM. On a BFM this record type is defined as an input. The record elements are intended as detailed below:

rst_n: a reset signal input to reset or initialize the stimulus interface. Active low.

addr: a 32 bit std_logic input that represents the address being accessed.

wdat: a 32 bit std_logic input that represents the data being written to the accessed location.

rwn: a read / write_not signal for selecting read and write operations.

req_n: a request or select signal input, to select the BFM being addressed. Active low.

The stm_sack type contains all the connections that the slave would output in acknowledgment of a request. This includes read data, rdat and ack_n outputs. The record elements are intended as detailed below:

rdat: a 32 bit std_logic output that is the read data from the addressed location during a read cycle.

ack_n: an output signal that acknowledges a request. Active low.

rdy_n: an output signal that indicates readiness. Used by the BFM to indicate it is busy with the last request. The BFM is ready when this output is low.

irq_n: an output signal to indicate that this BFM needs servicing. Active low.

The idea of the stimulus bus implementation changes is to detach stimulus IO on a BFM from DUT interfaces. Make the stimulus bus have a small profile in the over all system, yet be able to provide full functionality. The record types make the number of pins for stimulus access much smaller, yet introduce huge flexibility. The changes to the stimulus bus are presented below.

The addition of the reset input, rst_n, enables the BFM to be initialized separately from the DUT reset. It is part of the record and is easy to connect to any source of reset signals.

The change to a separate read and write bus enables the stimulus bus to migrate to a hardware implementation more easily. It is not recommended that by-directional buses be implemented in an FPGA.

The addition of the ready, rdy_n, output is to facilitate the need for some BFMs to indicate they are busy. This is usually the result of a previous command, the packet has not been fully transmitted or applied yet, I am not ready. The implementation of this output is BFM specific and user created.

The addition of the interrupt, irq_n, output is to facilitate service requesting from a slave. This output could also be used as an indicator of errors from a monitor BFM.

The record types provide flexibility to change the bus structure as needed. If the BFM is to be synthesized, a clock input will have to be added to the stm_sctl type and a new architecture created for clocked access.

The master record types will not be detailed here, as some exploration is needed. The blog VHDL Test Bench Blog http://vhdltb.blogspot.com/ will have postings of examples of possible implementations of master BFMs.

The stimulus access port is handled by the STM_access process found in the model code. This process handles the physical signaling and timing of the STM_ACK_N signal. The few ps of delay are added in to make the accesses viewable in simulation. The STM_access process triggers the REG_access process to perform the read or write to the models register set. When creating a new model the STM_access process can be copied from

another model. Though the REG_access process can be copied from another model, it most likely will have to be modified to suit a new register set implementation.

7.2 Connecting BFMs into the Test Bench

With the assumption that models are not accessed simultaneously, the following connection strategy can be implemented. The address, data and rwn pins can be connected to all models in a bus like fashion. A single acknowledge signal can be formed by "anding" all the models acknowledge signals together. Each request pin will require a separate signal and controlling statement in a stimulus instruction.

```
-- WRITE_STM
--    par1   model selection
--    par2   address
--    par3   data
   elsif (instruction(1 to len) = "WRITE_STM") then
     stm_addr  <=  to_stdlogicvector(par2,32);
     stm_data  <=  to_stdlogicvector(par3,32);
     stm_rwn   <=  '0';
     case par1 is
        when 0 =>
           stm_req_bfm1 <=  '0';
        when 1 =>
           stm_req_bfm2 <=  '0';
        when others =>
            null;
     end case;
     wait until stm_ack'event and stm_ack = '0';
     stm_req_bfm1 <=  '1';
     stm_req_bfm2 <=  '1';
     wait until stm_ack'event and stm_ack = '1';
     stm_addr  <=  (others => 'Z');
     stm_data  <=  (others => 'Z');
     stm_rwn   <=  '1';
     wait for 0 ps;
```

The code above is an example of a multi BFM access type of write instruction. Depending on the value of parameter #1 a specific BFM will be selected for the access operation. This code would be part of the elseif change of instruction definitions. The selection statement is found in the case statement. The stm_ack signal is the product of the anding of the ack signal from bfm1 and bfm2. Waiting for 0 ps causes a break from the process and signals to be updated / applied.

8. Functions Provided By The Test Bench Package

Several functions and procedures are available from the test bench package. Provided here is a description of the provisions of the package and their usage.

8.1 c2std_vec

This function converts a character to a standard logic vector of 4 bits.

```
function c2std_vec(c: in character) return std_logic_vector;
```

The input character is converted to a 4 bit std_logic_vector. If a non-hexadecimal character is encountered an error message is displayed and the simulation is terminated. This is usually used to covert text read in from a file to vectors in the test bench.

8.2 define_instruction

This procedure is called to define a new instruction for use in stimulus files. The instruction set "inst_set" variable is passed in. The new instruction "inst" is added to the list with indication of how many arguments will be passed in "args". (The 2011 addition enables the user to specify an unconstrained command length. If the args value is greater than six, it will cause the parser to by pass the check for the correct number of parameters.) The appended list is returned in "inst_set".

```
procedure define_instruction(variable inst_set: inout inst_def_ptr;
constant inst:    in    string;
constant args:    in    integer);
```

The instruction text must be less than 48 characters long and must not be a duplicate. If not, an error will be indicated and the simulation terminated.

8.3 index_variable

This procedure is called to get the value of a variable by indexing it. Passing in the variable list, "var_list" and the index, the value "value" and a valid indication "valid" are returned.

```
procedure index_variable(variable var_list : in  var_field_ptr;
                         variable index    : in  integer;
                         variable value    : out integer;
                         variable valid    : out integer);
```

The only indication from this procedure call is the valid field value. If the index was valid, it returns as 1 else as a 0. The user will have to test this status if they consider this needed. Valid variables are tested for and an invalid variable should have been detected before this call could happen. This procedure is intended to give user access to the variables. An example of usage can be found in the default variable manipulation instructions.

8.4 update_variable

This procedure is called to update the value of a defined variable. Passing in the variable list, “var_list”, the index and the value “value”, a valid indication “valid” is returned.

```
procedure update_variable(variable var_list : in  var_field_ptr;
                          variable index    : in  integer;
                          variable value    : in  integer;
                          variable valid    : out integer);
```

The only indication from the procedure is the valid output. It will be returned as a 1 if the transaction was successful. The user will have to test this status if they consider this needed. Valid variables are tested for and an invalid variable should have been detected before this call could happen. This procedure is intended to give user access to the variables. An example of usage can be found in the default variable manipulation instructions.

8.5 tokenize_line

This procedure is called to break the fields from a text_line. The test bench package defines a three types, a text_line, size 256, a text_field, size 48 and a stm_text, size 200. This procedure can be used anytime text is being read from a file and needs to be parsed into fields. It will parse up to 7 items from the line, one text string and return the count of valid tokens in the valid integer field.

```
procedure tokenize_line(variable text_line:   in  text_line;
                        variable token1:      out text_field;
                        variable token2:      out text_field;
                        variable token3:      out text_field;
                        variable token4:      out text_field;
                        variable token5:      out text_field;
                        variable token6:      out text_field;
                        variable token7:      out text_field;
                        variable txt_ptr:     out stm_text_ptr;
                        variable valid:       out integer);
```

If a “--” sequence is encountered parsing halts and any tokens found are returned. If a "" (double quote) is encountered, the characters following it are copied to a string and txt_ptr is returned. All white space is considered delimiter for fields. There is no error messages or error detection done by this procedure. If while parsing out a txt_pointer string, the “--” sequence is encountered, any white space between the end of text and the comment delimiter is included in the text string. This extra white space is removed if the txt_pointer string is used for an INCLUDE instruction file name.

8.6 print

This procedure can be called any time the user wants to just print something out. This procedure will not print out anything but the text provided. No time stamp like the assert type output.

```
procedure print(s: in string);
```

8.7 txt_print

This procedure is used to print to the console the txt string. This string is actually a pointer to the string that is returned with each instruction accessed. It is contained in the field named 'txt' that is part of the access_inst_sequ call. This procedure will print the string as it is defined with no alterations to the text.

```
procedure txt_print(variable ptr: in stm_text_ptr);
```

8.8 txt_print_wvar

This procedure is used to print to the console the txt string and substitute variables found. This string is actually a pointer to the string that is returned with each instruction accessed. It is contained in the field named 'txt' that is part of the access_inst_sequ call. This procedure will print the string and substitute any variable definitions with current variable values.

```
procedure txt_print_wvar(variable var_list : in var_field_ptr;
                         variable ptr      : in stm_text_ptr;
                         constant b        : in base) ;
```

Parameters passed are the variable list, var_list, the string pointer, txt and the base to display in, b. The variable list is the one created by the test bench environment as the stimulus file is parsed. The ptr field is the text pointer, txt, just have to pass it along. The display base must be included and is one of four choices, bin, oct, hex, dec. The choices correspond to Binary, Octal, Hexadecimal, and Decimal. All variables found in a string are converted to the same base representation.

The procedure parses the string and when a '$' is encountered it collects the field until the next white space. It then tries to get the variable value, and if successful places it in the output string in the number base format specified by b. This will most likely change the size of the over all string and in doing so, may cause truncation if the string size limit is exceeded. The concept of a variable index is not know by this procedure, it only knows of accessing, so the '$' must be included for the procedure to recognize there is a variable.

Examples:

```
elsif (instruction(1 to len) = "TEST_INST") then
  print("A Test Message from TEST_INST");
  txt_print(txt);
```

The above instruction example, contains one call to print and one call to txt_print. The call to print will put out that same message every time the instruction is executed. The call txt_print will print out what ever text is part of this particular “TEST_INST” instruction in the stimulus file.

```
elsif (instruction(1 to len) = "TEST_1") then
  print("More Test Message");
  txt_print_wvar(defined_vars, txt, bin);
```

The example above prints out a static message each time the instruction is encountered, this is done by the print function. The TEST_1 instruction also prints out any text found for any TEST_1 instruction encountered and substitutes any variables found with binary representation.

```
elsif (instruction(1 to len) = "TEST_3") then
  txt_print(txt);
  txt_print_wvar(defined_vars, txt, hex);
```

The example above will print out the txt part of the TEST_3 instruction twice. The first time, the txt field will be printed as is. The second time, it will be printed with variable substitution with hexadecimal representation.

9. Version 1.2 Test Bench Package

The test bench package, version 1.2, provides some enhancements and bug fixes.

The one bug that would affect Linux users pertained to the INCLUDE instruction and the use of the txt variable method. If a comment was placed after the file name, any extra white space between the end of the file name and the comment delimiter, was included in the file name. This cause the file not to be found because spaces are legal in Linux file names. The INCLUDE instruction implementation was changed to strip off any white space found in the file name.

Another problem was with the discovery of an invalid variable some time into a simulation. This could cause hours to be wasted when the test case halts before completion. A procedure was added to scan the instruction sequence for invalid variables before returning from the read_instruction_file call. This could save hours for you if you make a typo.

While evaluating structure of the test bench it was seen that the DEFINE_VAR instruction was being put onto the inst_sequ list. Hence was being searched through each time an instruction was retrieved from the instruction list. This is wasted computing and so the DEFINE_VAR instruction no longer is put on the inst_sequ list. Also as a result of this effort was the removal of the dependency of where variables are defined. Now variables can be defined any where in a script using the DEFINE_VAR instruction.

It was also found that the full file name of the file, a line in the a stimulus came from, was being included in every stim_line record. A full 200 characters was being reserved for each line in the test case. This character string was replaced with a pointer to a record in the file_list, linked list of file names. The file names found in the file_list are the main file name and any file names from INCLUDE instructions. This change was made to reduce the amount of memory used by the test bench system.

9.1 Upgrading tb_bhv from 1.1 to 1.2

If you want to upgrade a version 1.1 test bench to use the version 1.2 test bench package, you will have to make some small edits to the tb_bhv file.

Add:

```
variable file_list    : file_def_ptr;  -- pointer to the list of file names
```

The definition of the variable for the list of files, file_list of type file_def_ptr. This is shown below in full from the read_file process definition statement.

```
Read_file: process
  variable current_line : text_line;  -- The current input line
  variable inst_list    : inst_def_ptr;  -- the instruction list
  variable defined_vars : var_field_ptr; -- defined variables
  variable inst_sequ    : stim_line_ptr; -- the instruction sequence
  variable file_list    : file_def_ptr;  -- pointer to the list of file names
```

Change:

Change the read_instruction call to look like the statement below. Note the addition of the file_list variable.

```
-- Read, test, and load the stimulus file
read_instruction_file(stimulus_file, inst_list, defined_vars, inst_sequ, file_list);
```

Change the access_inst_sequ call to look like the statement below. Note the addition of the file_list variable.

```
access_inst_sequ(inst_sequ, defined_vars, file_list, v_line, instruction,
    par1, par2, par3, par4, par5, par6, txt, len, file_name, file_line);
```

10. Version 1.3 Test Bench Package

A user found and reported a typo in the Template WHILE instruction. This was corrected and checked in under version 1.3 of the test bench template behave file.

Updating a pre-version 1.3 test bench to version 1.3 will require the user to change the test bench bhv file. A pre-version 1.3 test bench retrieves instructions from the list of instructions by searching from the top each time. In order to improve efficiency, two new variables have been added to the test bench template (bhv file) and the access_inst_sequ call. The new variables store the sequence index and pointer to the last instruction accessed. If the current instruction is after the last one, in the list, the test bench will search forward in the list from the last instruction accessed. Otherwise it will search for the current instruction from the start of the list, like it use to. The changes to the package files are small but the affect on instruction searching can be drastic.

11. Beta 2.0 Test Bench Package

This release will not require the user to do any changes to existing test benches based on version 1.3.

There was one bug fix, the returning of the file name the stimulus line is from. Some how this value was not being returned in my downloaded version of the package. The value is now returned as expected.

The main addition/change to the package is the ability to create commands with and undefined number of parameters. This is achieved by setting the “args” value in the define_instruction call to have a value greater than six.

The package also contains the definition of record types to be used as stimulus bus ports on BFM's. This includes functions defined to set the buses into a neutral state.

Appendix A: Example Tb

Example test bench:

A simple example of a test bench implementation was posted to the CVS repository. This example contains a working behavioral model of a register driving some out put 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 pins 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 CVS examples/example1/vhdl directory:

example_dut_ent.vhd

example_dut_bhv.vhd

example_dut_tb_ent.vhd

example_dut_ttb_ent.vhd

example_dut_tb_bhv.vhd

example_dut_ttb_str.vhd

The example stimulus file is found in the CVS examples/example1/stm

stimulus_file.stm

Appendix B: Updates

Updates:

Version 1.2 tb_pkg_header.vhd

Fix package header name to match body and intended naming.

Version 1.3 tb_pkg_header.vhd

Update c_stm_text_len constant to be 200. Enabling txt strings to be longer for file path naming in INCLUDE instructions.

Add file_def and file_def_ptr types

Change stim_line record file_name, type stim_text, to file_idx, type integer

Version 1.2 tb_pkg_body.vhd

Add check for valid variables before starting simulation.

Change inst_sequ so that DEFINE_VAR instructions are not added to it.

Add removal of white space at the end of file names of INCLUDE instructions when the txt_ptr version is used.

Add file_list linked list of file names functionality

Remove some unused code

Version 1.2 template_tb_bhv.vhd

Comment out the INCLUDE instruction from the elsif chain

Add txt_print_wvar as the default text output function to put out text after instruction completion.

Add missing ELSEIF define_instruction call

Add file_list variable to read_file process

Update end of elsif chain comment to be more informative

Update function calls to include file_list

Version 1.2 example_dut_tb_bhv.vhd

This file was updated for version 1.2 of the test bench package

Add variable: file_list : file_def_ptr; -- pointer to the list of file names

Change the read_instruction call:

read_instruction_file(stimulus_file, inst_list, defined_vars, inst_sequ, file_list);

Change the access_inst_sequ call:

access_inst_sequ(inst_sequ, defined_vars, file_list, v_line, instruction, par1, par2, par3, par4, par5, par6, txt, len, file_name, file_line);

Version 1.3 tb_package_body.vhd

Add parameters to the access_inst_sequ and associated code to implement new search method

Version 1.4 tb_package_header.vhd

Add to access_inst_sequ last_num and last_ptr parameters

Version 1.4 template_tb_bhv.vhd

Update parameters for access_inst_sequ calls, add and initialize variables for new search method

GNU Free Documentation License

GNU Free Documentation License
Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by

proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the

Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities
responsible for authorship of the modifications in the Modified
Version, together with at least five of the principal authors of the
Document (all of its principal authors, if it has fewer than five),
unless they release you from this requirement.
C. State on the Title page the name of the publisher of the
Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications
adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice
giving the public permission to use the Modified Version under the
terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections
and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add
to it an item stating at least the title, year, new authors, and
publisher of the Modified Version as given on the Title Page. If
there is no section Entitled "History" in the Document, create one
stating the title, year, authors, and publisher of the Document as
given on its Title Page, then add an item describing the Modified
Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for
public access to a Transparent copy of the Document, and likewise
the network locations given in the Document for previous versions
it was based on. These may be placed in the "History" section.
You may omit a network location for a work that was published at
least four years before the Document itself, or if the original
publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications",
Preserve the Title of the section, and preserve in the section all
the substance and tone of each of the contributor acknowledgements
and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document,
unaltered in their text and in their titles. Section numbers
or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section
may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements"
or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or
appendices that qualify as Secondary Sections and contain no material
copied from the Document, you may at your option designate some or all
of these sections as invariant. To do this, add their titles to the
list of Invariant Sections in the Modified Version's license notice.

These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document
> under the terms of the GNU Free Documentation License, Version 1.2
> or any later version published by the Free Software Foundation;
> with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
> A copy of the license is included in the section entitled "GNU

Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.