Teal User's Manual

Version: 1.40b
Part number/version: TUG_1_40b
Release date: Spetember 12, 2006
www.trusster.com

Contents

Using this guide

This guide provides information about using Teal, a c/c++ Test Environment Abstraction Layer.

Who this guide is for

This guide is for verification engineers who use, or want to use C++ for verification. Specifically, engineers who want to use the Teal library to accomplish verification tasks.

What you need to know

To understand and use the information in this guide, you must:

- Be familiar with Hardware Description Languages (HDL) such as Verilog or VHDL.
- Have a general knowledge of the problem of hardware verification.
- Be familiar with software programming in c++.
- Know the use of the how to use the Linux operating system.

Conventions used in this guide

Throughout this guide, text written in this font is intended to be written in either c/c++ or HDL.

Customer support

To report an error in Teal, go to www.trusster.com and click on the teal section.

For support, search the forums on trusster.com or send e-mail to support@trusster.com

Chapter 1: Introduction to Teal

This chapter provides a general description of Teal.

Overview

Engineers perform verification to:

▪Minimize the probability of hardware functional errors.

▪Ensure that the hardware meets performance requirements.

▪Ensure that the hardware is usable by software.

Teal helps you perform verification by providing a set of capabilities that access HDL signals and enable actions based on changes in the values of these signals. In addition, Teal encourages independent generators, transactors, and checkers by providing for management of independent user-created threads.

Because Teal is a c/c++ library, you can develop algorithms that both validate the hardware design and can be re-used in the production software. Verification is C\C++ is appropriate because the language is well-defined and well-documented.

System requirements

To install and run Teal, your system must meet these requirements.

Hardware

The base Teal library is independent of any specific hardware platform, yet, as of this printing, it has only been compiled on Liunx based operating systems. However, a windows port is in progress. Check the trusster.com web site for the availibility.

You may have to modify the base types used in Teal as specified in Teal.h to compile and run Teal on other hardware platforms.

Software

As teal is written in C++, it requires a c++ compiler.

You also need an HDL simulator.

Teal in General

Teal is a collection of C++ classes, functions and data placed within the teal namespace. This set of code, along with build and run scripts that you write, provides an environment for verification tests.

Realize that this collection of code is only a very small part of the work of verification. Given Teal, you must write code to stimulate the Design Under Test (DUT), code to check the output from the DUT, and files to control the stimulus generation,. Also, you must write some some top level code to start and stop the various stimulators, generators and, of course, guide these to perform a test.

Chapter 2: Components of a Teal Verification System

This chapter describes the various parts that make up a Teal Verification System.

Overview

There are many different definitions of a verification system. Within this manual, the term is taken to mean those files needed to run a test and those files that are produced by ruinning the test. These components are elaborated in the next section.

Basic Teal verification components

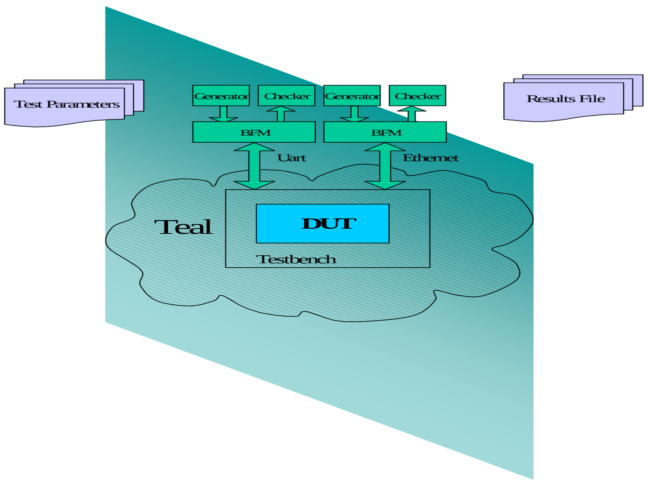
The basic Teal verification components are:

- A Teal library
- A set of Verilog design files, a design top
- A set of c/c++ source files, organized as traffic generators, checkers, and Bus Functional Models (BFMs).

There is also probably a run shell script and some makefiles.

Components of Teal based verification

The diagram below shows these basic components.



Chapter 3: Installing Teal

This chapter describes how to download and install the Teal software.

Overview

Before you download and install the Teal software, you need to verify that your system meets the requirements specified in the "Using this Guide" chapter.

As a check on the download and install process, you should run the tests listed in the test subdirectory of teal.

Downloading and installing Teal

To install Teal:

1. Using a browser, go to trusster.com.
2. Click the download tab.
3. Extract Teal to a directory, making sure that the directory you use is on your INCLUDE path (to enable C/C++ compilations).

Running the Teal tests

4. Set the environment variable SIMULATOR_HOME to the path for your simulator.
5. Set the environment variable SIM to your simulator (currently only ivl, mti, ncsim, vcs, or aldec)
6. Set the environment variable ARCH to your operating system (currently linux, solarus, and windows)
7. If you want to build teal, change to the teal directory and do one of these steps:

   -To remove all the binaries and object files, enter this command:

   ```
   make clean
   ```

   To compile the sources, enter this command:

   ```
   make
   ```

8. Change to the test sub-directory and do one of these steps:

-To run all the examples, enter this command:

```
./run –c –clean $(SIM) –l test_list
```

To run a single test, enter this command:

```
./run –c -$SIM –t <some_test>
```

where you replace <*some_test*> with either: reg_test, vreg_test, trandom_test, synch_test, mutex_test, or dictionary_test.

Note that the test is implemented by a pair of `.cpp` and Verilog files with the same root name.

Note also that the run script will display whether the test (or test list) passed or failed.

Chapter 4: Verification Top

This chapter describes how Teal interacts with the simulator; that is, what you need to add to your HDL testbench to use Teal.

Overview

Teal uses the Programming Language Interface (PLI 1.0 or 2.0) to allow c/c++ code to co-exist with the HDL. To this end, you must put a call somewhere in the HDL to start Teal. You usually do this in an initial block in the top level, but can be in any module at any statement. Of course, it it is not in an initial block, Teal will not start at simulation time zero.

This PLI call that is used to start teal is $teal_top. Other than backdoor memory access, this is the only required call to hookup Teal.

Theory of operation

When the simulation starts, the call to `$teal_top` causes Teal to start the threading system and call your `verification_top()` function. Your `verification_top()` function typically initializes global objects like the random number generator, dictionary, and your own top level code, and then waits for an `init_done` or `out_of_reset` signal from the HDL testbench.

After the Device Under Test (DUT) is out of reset, you typically start a series of transactors and checkers. During execution, these generators and checkers print log messages of checks that pass and errors that occur. Of course, it’s up to you to decide what your verification_top should do. A block level test or directed test may not need any threads and instead do everything in the `verification_top()`.

The `verification_top()` usually then waits for some signal from the lower level units that they are done. The signal can happen when, for example, a certain amount time has passed, or traffic generators are done, or maybe an error threshold has been reached.

Finally, the `verification_top()` typically prints some statistics and signals to the HDL testbench to quit. The HDL testbench calls $finish to shut down the simulation Note that the HDL $finish() call is not required in terms of Teal, but most systems have clocks that keep generating events. In this case, some mechanism is needed to stop the simulation. Alternatively, you can call teal_finish().

Here's what you have to do:

1. Add a initial `$teal_top()` in your top level HDL testbench. Typically, add an "`reg test_end; initial begin test_end = 0; wait (test_end); $finish; end` " as well.
2. Create a directory in which to run your sims.
3. Copy the example/example_1.cpp file and modify the `verification_top()` function for your test.
4. Copy the Makefile from the examples directory into your directory.
5. Copy the run script for the examples directory into your directory.
6. Make sure the environment variables ARCH, SIM, SIMULATOR_HOME, and TEAL_HOME are set.
7. Type run -c -clean -$SIM –t <your_file_name>

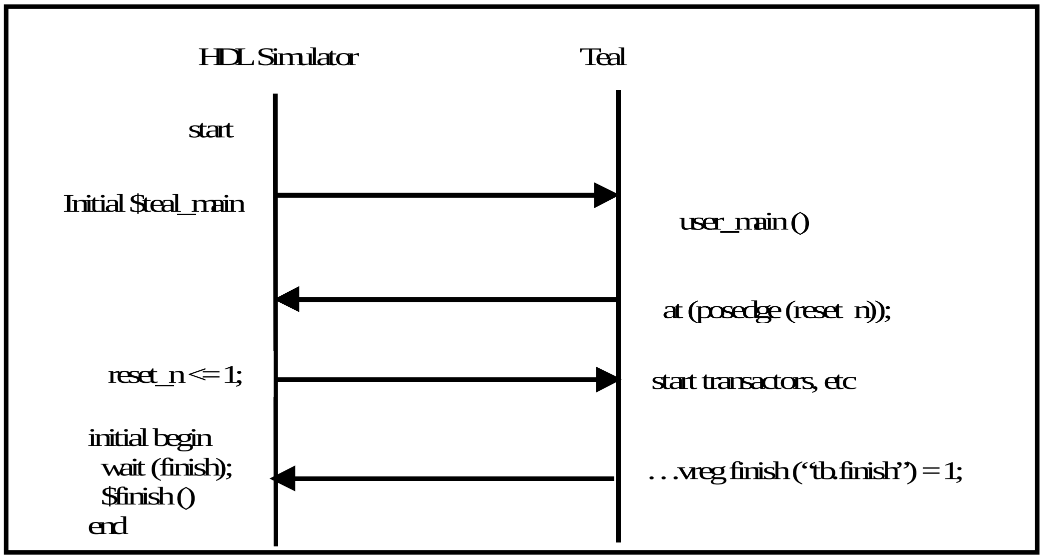


Figure 1: Example Process flow

C/C++ Interface

There is only one entry point (other than the memory system) for the Teal system. It is the verification_top. The prototype is shown below.

```
void verification_top ();
```

Teal calls this function during initialization:.

Shown below is a typical verilog testbench.

```
timescale 1ns/1ns
module testbench
<your wires, clocks, etc>
reg clk;
always #0.5 clk = ~clk;
<your DUT instance here>
initial $teal_top;  //hookup to Teal
reg test_end;
initial begin
  test_end = 0;
  wait (test-end);
  $display ([%t] [%m] [Verilog] Received Exit from Teal;
  $finish ();

end
```

Figure 2: A sample Verilog testbench.v

Examples

Given the testbench example in the previous section, this simple program just runs for 20 clock pulses.

```
#include “teal.h”

using namespace teal;

int verification_top ()

{

  vreg clock (“testbench.clk”);

vout log (“Chapter 4- Example 1”);

dictionary::start (“simple_clock_test.txt”);

uint number_of_periods (dictionary::find (“number_of_clocks”,
20));

for (int i(0); i < number_of_periods; ++i) {

  log << note << “i is “ << i << clock is << clock << endm;

}

dictionary::stop ();

vlog::get (expected) << “test completed” << endl;

}
```

In this example, the test runs for a number of positive edges in a clock register. The `simple_clock_test.txt` test file picks up the duration (The dictionary is discussed in a later chapter). This file only has one line - “`number_of_periods 23`” - which defines the length of the run.

Chapter 5: The Reg Class

This chapter describes one of the most basic classes in the Teal library, the reg class. It's main purpose is to provide arbitrary length 4-state operations.

Overview

When you go about designing a class library, an important decision is the creation of the "common currency" of the system. This chapter and the next few describe the common currency of the Teal system; that is, the basic generic building blocks of a Teal based verification system. The reg class, which is the most basic, is described first.

Theory of operation

Hardware simulators compute in four possible values for a bit[1]:

- 1
- 0
- X
- Z

In addition, hardware languages support arrays of bits, called reg. HDLs also support a rich set of operators on regs.

The Teal class `reg` implements this four-state logic and makes sure Xs propagate through calculations.

The `reg` class supports the usual HDL wire/register operations such as addition, subtraction, shifting, Boolean operations, and four-state comparison. Note that because multiplication and division are not part of the standard HDL operations, these opersations are not provided. As in HDL languages, bitfields or subranges of `reg` are supported. The subranges can be on either the left or right side of an expression.

1. Some simulators use 8 logic vlaues, but Teal uses only four because the concept of drive strenghts are not typically relevant to functional verification. Teal uses the HDL simulator for signal value resolution.

C\C++ interface

Creating, copying and, destroying a reg

Reg has only a few constructors and a copy constructor. The default constructor creates a one-bit register and is marked explicit to prevent you from accidentally creating such a register. The most common constructor build 64-bit register. Theer is a second, optional parameter that specifices the bit width. Some examples are shown here.

`reg a (45);` //create a 64-bit register initialized to 45

`reg a (0x22, 73);` //create a 73-bit register initialized to 0x22

The `reg_slice` class is another helper class that is automatically created (and destroyed) when a register subrange is used. The `reg_slice` constructor is automatically used when a register is needed in an expression. For `example reg a(45); reg b (a(10,0))`; creates a reg_slice of a. Because `reg_slice` is a temporary object automatically created during left-hand assignment, it is not documented further.

The last `reg` constructor takes in an integer and creates a 64-bit `reg` set to that value. This constructor allows statements such as:

```
 reg a(length(32)); a = a + 5; and a(4:0) = 1;
```

The `operator=()` method sets the current value of the instance to the `rhs` but does not change the length of the instance. Specifically, the operator=() extends (with zeros) or truncates, as appropriate, depending on the length of the `rhs`.

The virtual destructor allows for subclasses to clean up any allocated storage.

- reg () [explicit]
- reg (const reg_slice &)
- reg (uint64, uint32 length = 64)
- reg (const reg &)
- reg& reg::operator= (const reg &)

▪~reg () [virtual]

Reg access functions

This set of functions convert registers to intergral types, and allow access to parts of a register.

The `to_int()` method converts the register to a 64 bit integer. Any X or Z bits are returned as their encoded aval, so be careful when using this method. Also, the result will be truncated if the register is more than 64 bits long.

There are three register slicing methods. These methods allow access to a subset of the total bits in a register. They are overloaded methods of `operator()`. The single argument `operator()` is the single bit read-only access function for a register. You use this method to get a single bit. To get more than a single bit, you use the two-argument method. There are two forms of the two-argument `operator()` method.

The constant two-argument method returns a copy of the bits of the `reg` as specified by the arguments. This method is called automatically by the compiler when the slicing operation is on the right-hand side of an expression.

The two-argument non-constant method returns a reference to the bits of the `reg` as specified by the arguments. This method is called automatically by the compiler when the slicing operation is on the left-hand side of an expression.

`uint64 to_int () const`

`char operator() (uint32` *b*`) const`

`reg operator() (uint32` *u*`, uint32` *l*`) const`

`reg_slice operator() (uint32` *u*`, uint32` *l*`)`

Math functions

The math functions come in two flavors:

- **Global functions.** These functions exist because they are symmetrical. They are used when it is not appropriate to define the function as a member function, usually because either:

  –Reg objects are passed in.

  –The operation is communative.

The functions listed next are for the mathematical binary addition and subtraction operations. Note that the registers are considered unsigned.

```
    reg operator+ (const reg & lhs, const reg & rhs)

    reg operator- (const reg & lhs, const reg & rhs)

    reg& operator+= (const reg & rhs)

    reg& operator-= (const reg & rhs)
```

- **Member functions**. These functions sends the mathematical result back to itself (the <x>= operators).

The functions listed next perform left and right shift operations. There are equivalent symmetrical operators as well. These functions, while seemingly complex, provide capabilities that make `reg` act like a built-in type.

Note that zeros are shifted in during a right shift.

```
reg& roperator<< (uint32 rhs)

reg& operator>> (uint32 rhs)

vlog& operator<< (vlog & c, const reg & rhs)

reg operator>> (const reg & lhs, const uint32 rhs)
```

Logic functions

These functions implement the Boolean operations that are associated with a register. Some functions also implement the logic as a four-state enumeration, as in the HDL. As with the math functions, some are global and symmetrical, while others are methods.

By default, the relational (less than or greater than) operators act like the normal two-state mathematical less than. However, when a single bit is X, the result is X.

The `reduce_xor` method models the unary “^” operator of Verilog.

The functions listed below perform the usual logical operations.

```
enum four_state {zero=0, one, X, Z}

bool operator== (const reg & lhs, const reg & rhs)

reg operator~ (const reg & lhs)

reg operator| (const reg & lhs, const reg & rhs)

reg& reg::operator|= (const reg & rhs)

reg operator & (const reg & lhs, const reg & rhs)

reg& reg::operator &= (const reg & rhs)

bool operator!= (const reg & lhs, const reg & rhs)

four_state operator< (const reg & lhs, const reg & rhs)

four_state operator< (const reg & lhs, const reg & rhs)

four_state triple_equal (const reg & lhs, const reg & rhs)

four_state reduce_xor (const reg &)
```

Printing functions

The printing functions allow a `reg` and any derived classes to be printed as part of a message line (see Chapter 7). The `reg::operator<<()` is the virtual method that allows you to print reg and all subclasses as built-in types. By default `operator<<()` senses the output format (hex, dec, or binary) and calls the appropriate string formatter, described below.

The `format_hex_string()` method returns a hexadecimal representation of the `reg`, for example, 4’hf. The `format_decimal_string()` method prints the register as in integer and the `format_binary_string()` method prints the register as a sequence of ones and zeroes, like 5’b01001.

```
vlog & reg::operator<< (vlog & c) const [virtual]

vlog & operator<< (vlog & c, const reg&)

std::string format_hex_string () const
```

```
std::string format_binary_string () const

std::string format_decimal_string () const
```

Functions for HDL coherency

The methods shown in the next examples are used by derived classes like `vreg` to allow them to make sure the HDL signal and the c/c++ world agree.

Within `reg`, all these methods do nothing. The `read_check`() method is called before every access to the `reg` internal storage (`aval/bval` array). The `write_check`() method is the corollary to `read_check`(), in that it is called whenever any part of the `reg` is updated.

```
virtual void read_check () const [virtual]

virtual void write_through () const [virtual]
```

Examples:

This section shows some basic examples of `reg`. These examples can be used as an introduction of the capabilities and use of the reg class. Note that some of these examples use the derived class vreg, which is discussed in the next chapter.

Declarations

```
reg a(23);  //64 bit register/initialize it with the value 23

reg b(length (323)); //323 bit register with the initial value of all X's

b = 0x11; //assign b to 11 (clearing the upper bits)

b(315,300) = a; //set bits 315 to 300 of b to 23.

a = b(63,0); //uses the lower 64 bits of b

reg c(b); //323 it register with the initial value of b's current value

c(440, 413) = 1;  //illegal – run time error, bit index out of range

vreg d("tb.chip.reset_n); //create a vreg that is tied to reset_n
```

```
c = d; //clear all but the lowest bit of c, c(0) = current value
of reset_n

d = 0x0; //push reset_n to 0;
```

Math on reg/vreg

```
std::string path (“testbench.top.main_bus”); //the root of the
bus module

vreg addr(path + “.address”); //address, bit length copied from
HDL

reg b(length (32));

b(31:0) = 0x12; //init b

addr += 2; //increment address, push to HDL

addr = b << 3; //same as addr(addr.length() –1, 3) = b;
addr(2,0) = 0;
```

Bit fields

```
reg b (101);

std::string root (“tb”);

std::string module (“bus”);

vreg addr (root + “.” + module + “.rd_addr”);

addr(1,0) = b(28,27);  //copy the two bits, push to HDL

int c (addr (31,28).format_int()); //get current upper nibble
of address

addr(27,20) &= 0x55; //do some bit bashing
```

Chapter 6: The vreg class

This chapter describes the class that is used to connect your c++ code to the HDL. This class provides mechanisms to use signals in the DUT as though they are built-in c++ variables. The vreg class is derived from the reg class.

Overview

The `reg`, by itself, has very little to do with simulation. The `vreg` class, which is derived from `reg`, implements the hookup to the Hardware description Language (HDL).

In order to do verification, pre-silicon co-verification, or software algorithm development, interacting with the hardware is a basic necessity. The vreg class provides the ability to stimulate inputs and interrogate/respond to outputs.

Theory of operation

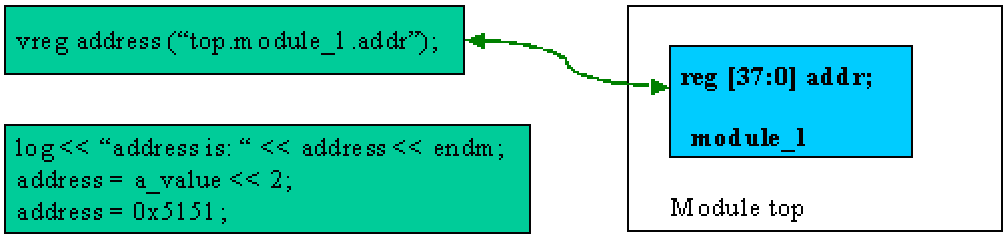
The vreg object uses the Verilog Procedural Language (VPI) or Programming Language Interface (PLI), depending on the compile options, to find the wire or register in the HDL. The path to the register or wire is the handle to the HDL register or wire. The path is formed by concatnating the module names and then the wire/register name. For example, assuming that your top level module is called testbench, and you have a top level wire called usb_dp and a module called uart with an internal register called clk, this is how the path would be passed in to the vreg constructor.

▪vreg (“`testbench.uart.clk`”)

ßvreg (“`testbenchb.usb_dp`”)

A `vreg` differs from a `reg` in that its value is initialized and conceptually exists in the HDL. Therefore, any assignment (or sub-range assignment) is pushed to the HDL as if it were a non-blocking assignment.

The example below shows a top level module called module_1 and a register within the module_1 called addr. The C/C++ code on the left shows how to declare and use the vreg variable address.



```
vreg address ("top.module_1.addr");
```

```
log << "address is: " << address << endm;
address = a_value << 2;
address = 0x5151;
```

Figure 1: Example of how the vreg interacts with the DUT

Because `reg` was designed with `vreg` in mind, there are classes to virtual methods for determining when a `reg` should be written to the HDL or when the most recent HDL value is needed.

C/C++ interface

This section describes the programming interface. Note that there may be undocumented private interface data and members.

Creating, copying, and destroying a vreg

To create a `vreg`, pass in a string path down to the wire or register. The path is copied so that later printing functions can print the signal by name. If the path does not map to an HDL signal, you get an error, and all subsequent usage of this `vreg` will fail.

```
 vreg (const std::string & path_and_name)

 ~vreg ()

vreg & vreg::operator= (const reg &)
```

Sometime, when designing reusable Intellectual Property (IP), you may not know if a particular register is going to be implkemented in every instance of the DUT. Yet, to minimize the chance of having different versions of IP, you want to write the IP such that it always exists. This is supported in the vreg class by allowing the path to be zero (NULL). In this case, the vreg is not “hooked up” and all operations proceed as though this was a reg obejct. You can test this condition by the enabled() method.

```
 bool enabled ()
```

Similar to the above discussion, there are times when you do not know the name when the object must be constructed. This case should be rare, as two stage construction is more prone to errors. To supported delayed hookup to the HDL, the name() method is used. This resets the path to the register.

```
 void name (const std::string & path_and_name)
```

Finally, the verification effort may be ahead of a particular implementation or there may be many varients of a chip with subsets of the full feature set. In this case, the C++ modules may want to test for the presense of a module or wire to determine if the functionality is present. In this case, the static method present () can be used.

```
bool present(const std::string & path_and_name)[static]
```

Functions for HDL coherency

The methods below are overridden by `vreg` to allow it to make sure the HDL signal and the c/c++ representation agree.

Specifically:

ßThe `read_check`() method checks the current integer global state, and if its value is not the same, calls the HDL to get the current value of the signal. Then `read_check`() sets its internal state variable to the global one.

ßThe `write_check`() implementation pushes a value to the HDL, as though it were a non-blocking assignment.

ß`Invalidate_all_vregs`() is a method used only by the threads management system. This method is called when the c/c++ code is called from the HDL side, causing the global state value to change and causing all `vregs` to get a new value when or if they are referenced.

```
virtual void read_check () const

void write_through () const
void invalidate_all_vregs () [static]
```

Examples

This section shows some basic examples of `reg` and `vreg`. You can use these examples as an introduction to the capabilities and use of these classes.

Declarations:

```
std::string path (“testbench.top.main_bus”); //root of the main bus

vreg addr(path + “.address”); //access the address, bit length from
HDL

vreg clk((“testbench.top.clk”);

vreg a(23);  //illegal. Need a path

reg b(length (72)); //create a 72 bit register, initial value of
all X’s

b(71,32) = addr; //set bits 71 to 32 of b to current value of
address.

clk = 0x0; //force clk to 0;
```

Bit fields

```
reg b (101);

std::string root (“tb”);

std::string module (“bus”);

vreg addr (root + “.” + module + “.rd_addr”);

addr(1,0) = b(28,27);  //copy the two bits, push to HDL

int c (addr (31,28).format_int()); //get current upper nibble of
address

addr(27,20) &= 0x55; //do some bit bashing
```

Chapter 7: Logging Simulation Output

This chapter describes how to write to a results file. This chapter covers why a uniform output format is desirable and the facilities included in Teal to make this happen.

Overview

Often, you use a log file as a trace of what happened during a simulation. It is important to have a consistent message format to enable post processing, error counting and possibly filtering. The Teal classes `vout` and `vlog` encourages such uniformity.

To increase the probability that a piece of verifcation code can be reused, Teal provides a very flexible formatting system. Why ar ethey related? Well, different teams have different specification for output format. Teal is flexible enough to match almost any desired output format. However, this flexibility comes at a cost of some complexity. It is hoped that suitable initial settings make this complexity only apparent when sophisticated flexibility is needed.

Theory of operation

Logging seems so simple when you first start writing to cout or using printf()s. As your code or team size grows, however, it becomes quite complex. This is because different people think about logging in different ways. Some like generous amounts of data (and may post process the results file). Others prefer to think of a “level” of debugging, where a bigger number represents more verbosity and zero represents an unmaskable message. Others prefer a mask-based approach.

Orthagonal to the amount of data to be displayed, there is the presentaton order of the data. Should the simulation time be first, or the type of message (note, error, expected, etc) ? Also, in more advanced applications, you may sometimes want to redirect or duplicate the messages to a display or another unit in the system. Add to all that, the possibility of getting verification IP from an outside vendor or another part of the company and logging becomes downright painful.

Teal attemps to provide a maximum of flexibility while keeping the simple outputting simple. Because of the sloping complexity of logging, this section is divided into basic and several advanced parts.

Creating a results file within Teal has two basic components. One is the vout class, which is intended to be instianiated at each component or functional area of a design (e.g a each transactor, checker, and maybe the verification_top). The other object is called a singleton, which means there is only one of them in the system. This is the vlog class. The figure below shows the relationship between your code, the vout class and the vlog class.

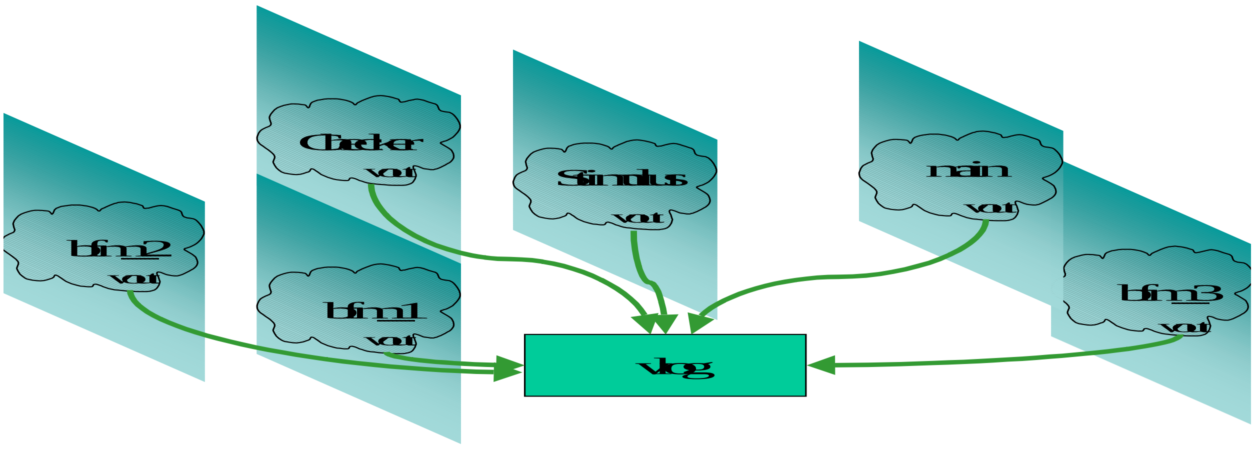


Figure 1: View of many vout objects and a single vlog

Basic Logging

As shown above, all vout instances call the single vlog to print a message. This is done so there is a single point of control to reorder, demote, change or delete parts of any message. This ability is provided by the vout::get() function chain and it is described below. Since all messages start at the vout, this will be described first.

The vout class is modeled after the `c++ cout` object. It directly supports output of the standard types. By following a few simple guidelines, you can print complex objects as conveniently as the standard types. See example three in this chapter. To end a message, call the endm function (see the basic example). To describe a multi line message, use endl (just like cout) where needed and use a final endm at the end.

When you create a vout, you give it a string that represents the functional area it is providing messages for. You can then build any mumber of message statements. For example:

```
vout log (“a test”);

log << teal_info << “A number” << hex << 562393 << endm;
```

This example prints (assuming a file os simple.cpp, at line 101, thread of “uart_o_tx”, and a simulation time of 77 ns):

[77 ns] [simple.cpp] [line 101] [uart_0_tx] [a test] A number 64’h894D9

Note that when you finish a message statement (using endm), the vout instance adds the simulation time, the file, the line number, the current thread name and the functional area to the message and sends it to the vlog singleton. It does not send it as a text string, which would not allow efficient modification of the message. Rather, it sends the message as a set of pairs of IDs and strings. This allows you to instruct the vlog instance to modify messages based on their components.

The `vlog` class also supports decimal, hexadecimal, or binary output. You select the output basis by placing either a `hex` or `dec` or bin in the message statement.

Before moving on to the vlog class and it’s message manipluation capabilities, you can also turn off display of parts of a message directly at the vout instance. The message_display() function takes in an ID and a boolean. If the boolean is false, the message part represented by the ID is not displayed. The standard message IDs are “time, thread_name, functional_area, error, note, warning, expected, and message_data”. Note that since the message_display() function takes in an integer, you can add your own message IDs and control their display.

As shown above, completed messages are sent to the vlog object. Generally, you include a message ID pair that describes the type of the message. The message type is generally either :

- `teal_info`.Used for standard messages.
- `teal_fatal`. Used when a test must exit the simulator immediatly.
- `teal_error`. The error type is used when the expected behavior is different from the expected.

The number of info and error messages is counted and can be used for end of test checking or summary reports.

The vlog has two main functions. One is to assemble the message items into a string. The other is to print them out. The standard vlog (see Advanced vlog below for other possibilities) performs these two tasks using an output_message() and a local_print() method.

The local_print() method is basic and prints the string to standard out (using printf). Also, there is a subclass of the standard vlog called foile_vlog, which can send the text to the file (via fprintf).

Level based Logging

Sometimes its more convient to implement logging as based on a debug level. What you are getting in the level feature more expressive power, at the expense of additional complexity. The vout class supports debug printing with ethe teal_debug manipulator and the level (<#>) io manipulator.

The `vout` class can be constructed with a functional area string (like its base, vout) and a default uint32 level. The dictionary parameter "<functional area> + *show*_level" is searched for a command line or file override to the level for this functional area. This is so that debug levels can be overridden without changing or rebuilding the verification environment (or object file, if a Verification IP provider did not provide source code). When a message is written, the manipulator function level(uint32) is used, like so:

```
vout log ("a_test", 3);

log << teal_debug << level (4) << "level 4 message" << endm;
```

In this case, without a dictionary override, the message will not be printed. This is because the level_vout was created at level 3 and the message is at level 4.

The show_debug_level() method allows the verbosity to be set programatically.

Note that info, error, and fatal messages are printed out at level 0, so they cannot be suppressed using the show_debug_level(). There are other ways, and they are shown in the Advanced vlog object section.

Debug messages are set to level 1 until a level(<#x>) is encountered, so you may want to not use level(), and thus treat all debug messages as either all on or all off.

Advanced vlog objects

The vlog class supports a static method get(), which returns the currently vlog object. This is what is used by the vout to output a message.

However, vlog is only logically a singleton. It is actually implemented as a linked list of filters, each with the ability to manipluate the list of mesage items or the formed string.

By constructing a new vlog, the vlog constructor automatically assigns the new object to the logical singleton "get()". The vlog class also keeps track of the previous value of get(). In this way a chain is formed. So, when a message is printed, your object is the first to be able to suppress or change the message.

Your derived class of vlog that overrides the local_print_() method, you have the ability to get the std::string that is the completed message and perform any post processing that is appropriate.

Also, if you build a derived class of vlog that overrides the local_print_() method, you have the ability to get the std::string that is the completed message and perform any post processing that is appropriate.

C/C++ interface

Creating, copying, and destroying a vlog

`vlog` is unlike a "normal" class in that it is expected to always be there - so there is no explicit construction step. You create a `vlog` by calling the `get()` method[1]. Also, because `vlog` is implementing a logical singleton, it cannot be copied or assigned. Finally, because `vlog` will exist for the duration of the simulation, it is not destroyed[2].

The message type describes the type of message that is being printed:

```
enum message_type = {info, expected , error, fatal}
```

However, you generally use the teal IO manipulators instead of directly putting the message type in the message. These functions are placed in-line with the output message.

```
teal_info () - for normal messages

teal_debug () - for developer messages

teal_error () - for incorrect DUT behaviour messages

teal_fatal () - print the message and exit the sim
```

This function returns the object at the top of the vlog chain.

```
vlog & get () [static]
```

Statistics

As each type of message is printed, `vlog` increments a counter. You retrieve the current count of any message id by using this call.

```
int how_many (int) //where int is usually error !
```

Output of the standard types

These functions handle outputting of the standard types in c/c++:

1.Inside of this call, there is a test to see if no vlog has been created. If this is the case an object called local_vlog() is created. As this is an implementation detail, it is not documented further. The class is implemented in the Teal file teal_vlog.cpp.

2.Of course, you can create local scoped vlogs and they will be added abd removed from the chain as appropriate.

```
vlog& operator<< (vlog &(* f)(vlog &))

vlog & operator<< (double)

vlog & operator<< (const std::string &)

vlog & operator<< (long long unsigned int)

vlog & operator<< (long)

vlog & operator<< (unsigned int)

vlog & operator<< (int)

vlog & operator<< (char)
```

Output Format

The next functions set the output type, add a line feed, and end a message. Note that, although these are functions, the iomanip template allows then to be used in-line as part of the "<<" expression, as in the cout model.

Be aware that you must call endm() to end a line. However, you may want to have multiple lines in the same message. In this case, call endl(). This inserts a note to the vout object to begin a new line.

```
vlog& dec (vlog & a_vlog)

vlog& hex (vlog & a_vlog)

vlog& bin (vlog & a_vlog)

vlog& endl (vlog & a_vlog)

vlog& endm (vlog & a_vlog)
```

Examples

1: Simple

```
vout a_log ("Basic");

a_log << note << "Hello World " << 42 << endm;
```

Assuming the previuos lines are called at simulation time 5 nanoseconds, the output is:

```
[5 ns] [simple.cpp] [line 3] [verification_top] [Basic] Hello World
42
```

2. Logging a reg

```
reg a(23);

reg b(length (48)); b[7:4] = 3;

vout a_log (“Basic”);

a_log << note << “basic a is” << dec << a << endm;

a_log << expected << “a is” << hex << a << “ and also “ << bin << a
<< endm;

a_log << error << “b is “ << hex << b () << endm;
```

Assuming the previous lines are called at simulation time 565 nanoseconds, the output is:

```
[565 ns] [basic.cpp] [line 4] [verification_top] [Basic] Note:
basic a is 64’d23

[565 ns] [basic.cpp] [line 6] [verification_top] [Basic] EXPECTED:
a is 64’h23 and also
64’b00000000000000000000000000000000000000000000000000000000000101
11

[565 ns] [basic.cpp] [line 6] [verification_top] [Basic] ERROR: b
is 512’hXXXXXXXXXX3X
```

3. Output of an object:

This example shows code that enables complex objects to be printed as native types. First, declare the base operator<<() method as virtual, to allow derived classes to have their own output:

```
class base {

  virtual vout& operator<< (vout& v)  const {v << “Base class”;
return v;}

};
```

Now, declare a single global function to call the virtual one. The compiler automatically finds this function when you do the following:

```
my_log << note << base_instance << endm;

inline vout& operator<< (vout& v, const base& b) {

                       return b.operator<< (v);}

class derived : public base {

 virtual vout& operator<< (vout& v) const { v << “Derived “;
return v;}

};

base a_base;

derived a_derived;

vout a_log (“Basic”);

a_log << note << a_base << “ and “ << a_derived << endm;
```

Assuming the previous lines are called at simulation time 565 nanoseconds, the output is:

```
[565 ns] [object.cpp] [line 7] [verification_top] [Basic] Base
class and Derived
```

Chapter 8: The Random Number Module

This chapter describes what is means to have independent streams of random numbers and why that is important for verification. It then describes Teal's facilities for random number generation and shows some random number generation examples.

Overview

Using random numbers for test values is a staple of modern verification. The numbers must be well-distributed and stable across runs. The first is important because you need to find all the possible valid values and the second is important because after you find a bug, you need to rerun that simulation at least twice:

- Once to create a `vcd` file.
- Once to confirm that the bug is fixed.

You might want to put that run in a regression suite as well. The reruns of that simulation must produce exactly the same sequence of random numbers.

This chapter describes Teal's random number capability.

Theory of operation

Teal's trandom class provides a stable random number generator. To provide independent streams of random numbers, the random class uses a string and an integer to create the *start seed*.The start seed is the initial value for the random number generator. In addition to the start seed for each instance, the random class itself is initialized with a master seed, which provides a tie-in to all the streams. In this way, a single number, given to the static `random::init`() function, possibly from the command line or test file, is used to guide all random number streams.

The basic random number class generates a [0..1.0) random double on every draw. The `random_range` class maps this double to a range of integers.

To support stability, you must carefully choose the seed string or number passed into the constructor. The example section shows some common techniques that can be used to provide the appropriate level of flexibility and stability.

The random class is a basic one. It provides all the basic operations needed, it's often the use and interaction of the random numbers that are the complex part of verification. Feel free to derive other classes to implement different distributions.

C\C++ Interface

Required Initialization

Before using any random numbers, you must initialize the generator. There are two ways to do this. The first way is to pass in a path to the master seed file. This file, if it exists, is searched for a dictionary entry of master_seed. If a line beginning with "master_seed" is found, the remainder of the line is assumed to be a set of hex digits used for the master seed. If the "master_seed" string is not found, a master seed is chosen and the "master_seed" and hex digits are written to the file.

```
void trandom::init_with_file (const std::string &
master_seed_path) [static]
```

The other way to initialize the random number master seed is to explicitly pass it to the init() method. This is useful when you pickup the master seed from the dictionary. This is the method used by the example run script and test files.

```
void trandom::init_with_seed (uint64) [static]
```

Note that these functions can be called multiple times if a combination of master "key" seeds are desired.

Common macros

Before describing the `trandom` class, you need to know about some common macros that you may use often. The RAND_8() and RAND_32() macros generate random numbers of the appropriate bit length. The RAND_RANGE() macro generates a bounded random number.

```
 RANDOM_RANGE (output_value, min_value, max_value);

 RAND_8 (output_value);

 RAND_32 (output_value);
```

Creating, copying and destroying a trandom

After the random number class is initialized, you can build `trandom` objects. The constructor takes in a string and an integer, both of which are optional. The macros described above pass in the ansii (americal national standard for information interchange) standard __FILE__ and __LINE__ macros. Note that omitting both creates identical random number generators. The default copy constructor and operator=() can be used to copy a random number stream or set one stream to match another.

```
trandom (const std::string & file, uint32 line)
```

```
~trandom ()
```

Getting random numbers

The draw() method is used to draw a random number. The number will be between 0 and 1, specifically [0..1).

```
double draw ()
```

Creating, copying and destroying a trandom_range

A simple derived class of `trandom` is `trandom_range`. This class shifts the 0..1 double into a `uint32`-based range.

```
trandom_range (const std::string &, uint32)
```

Examples

1. Stable across source file editing

As a general technique, you may find it useful to enclose each call to a random number generator (`RAND_RANGE`, `RAND8`, and so on) within a static function at the top of a file. This is because the default macros use `__FILE__ and __LINE__` as the seed, and you don't want the line number to change as you add or remove code from a source file. For example:

```
static uint32 get_next_channel (uint32 a_min, uint32 a_max) {

uint32 r; RAND_RANGE (r, a_min, a_max); return r; }

};
```

2. Direct use of the trandom class

You also can bypass the macro and provide your own string (or number). This would also be stable across code changes. For example:

```
trandom my_random (“Hello World”, 0);

uint32 my_random_value = my_random.draw ();
```

3. Cycling through random numbers

Sometimes you need to track which random number has been handed out. This is usually either when you want to walk an entire range before returning to a previous value or make sure you never hand out a duplicate. In this case, the number represents a resource, like an endpoint number, that can be “allocated” and “released” randomly during a simulation.

In this case, you can use a bool array of the appropriate size. You can also use a std::map to track the random number usage. For example, assume that you are handing out uint8s. Using the technique in example one to isolate random number streams in a static function at the top of the file, you could have:

```
static uint8 next_channel () {

  uint8 x;

  uint32 deadlock = 100000; //Good enough for most cases.

  static bool used[256] = {0};

  do {  RAND_8 (x);}while (--deadlock &&used[x]);

    if (!deadlock)

      vout::vout (“next_channel”) << error << :Deadlock! no more
channels at” << __FILE__ << __LINE__ << endm;

  }

};
```

Note that the deadlock ensures that the system never is hung up, which is important whenever you constrain a random value. As the interactions of the random numbers increases, so does the possibility of a deadlock.

4. Selecting a boolean via a 0..100% probability

Sometimes it’s useful to skew a random distribution so that it’s not gaussian. There are several ways to do this. This and the following examples explore some of the ways. For a single bit (or a Boolean), one simple way is to provide a threshold, which represents probability of success. Then by generating a random number between 0 and 99, and comparing it to the threshold, you can skew a distribution.

```
static bool get_use_tone_detection () {

  uint8 threshold = dictionary::get (“tone_detection”, 50); //
default is balanced random%

  uint8 x; RAND_RANGE (x, 0, 99);

  return (x < threshold);

};
```

5. Selecting a enum

A way to randomize the selection of an enum is to all the possible values, and then pick a random index. For example:

```
#include <vector>

using namespace std;

typedef enum valid_address_type {config, io, cached, uncached};

static valid_address_type get_next_address () {

   std::vector<address_probability> addresses; {

   addresses.push_back (config);

   addresses.push_back (io);

   addresses.push_back (cached);

   addresses.push_back (uncached);

   uint32 x; RAND_RANGE (0, addresses.size () -1);

   return addresses[x];

};
```

6. Subclassing trandom

Another way to provide a non-gaussian distribution is to subclass the `trandom` or `trandom_range` object and apply a post-processing step. For example, taking the log of the result of the `draw`() method produces a logorithmic distibution.

7. Constrained random example

Often it is useful to constrain the generation of random numbers using an external to the test mechanism. This allows one test to have several different runs. One way to do this is to use the master seed picked differently for each run. Another way is to use a test file that provides variables to be used as bounds in the thresholds or distributions shown in the previous examples.

For some examples of how to get external variables into your test, see the dictionary namespace in Chapter 11.

Chapter 9: Accessing Memory

This chapter describes how you use Teal to read and write memory in zero simulation time. It also explains how the Teal memory system can be used for more generic abstraction of reading and writing registers (in zero time or through a DUT interface).

Overview

Almost every chip that is verified has internal memory or interacts with external memory. This chapter describes Teal's capability for testing those interfaces. It describes the types of memories that are supported and how these are mapped to integer address ranges. This chapter also describes how you can use memory building blocks to support error injection, grouped memory, and bank/front door memory access.

Theory of operation

For simulation, you can implement the memory in either c/c++ or the HDL. Implementing the memory in c/c++ is appropriate in these cases:

- For extremely large memories
- When all accesses must be checked
- When statistics must be gathered

Because a c/c++ implementation could be built on top of Teal (using the `vreg` class, the `run_loop` class (see), and the `memory_bank` class (see next), this chapter does not discuss it. This chapter is primarily concerned with accessing memory implemented in the HDL. The information in this chapter is based on the assumption that you are implementing the memory as a HDL register array.

You can get access to HDL implemented putting a hook task into the module that contains the register bank. See `teal_memory_note()`[1].

A `memory_bank` object is created for each `teal_memory_note()` call. This object contains code to access the internal memory of the simulator that is implementing the HDL register array. The `memory_bank` that is created by the hook function contains `to_memory()` and `from_memory()` methods, which carry out the access *without advancing simulation time*.

1.One could augment Teal to provide a memory acessor via a string as well, but it is not provided at this time.

The `memory_bank` also has low and high address 64 bit integers, which are used to allow access to memory using an integer address.

The `memory` namespace keeps track of all memory banks. Note that memory banks are the workhorses of the memory namespace. They do the actual work, while the namespace just figures out which one to hand the access request to. For the interface method to find the right memory_bank when using an integer address, you must map the memory banks into an integer address range. The `memory::map`() function maps a `memory_bank` to an address range. You can put this mapping in the initialization code of your `verification_top`(), and all other code can just read/write by integer address.

Sometimes several memory banks are grouped together to provide one logical memory bank. In this case, you need to write a special memory bank that first retrieves all the sub-memory banks (using `memory::bank_lookup` ()) and then adds itself to the memory namespace's list of memory banks, using `memory::add_bank`().

Sometimes you want to inject errors in the memory system. You can do so in a manner similar to the grouped case, where you look up the bank in question, and add a new bank that handles the memory accesses.

Often it ´s helpful to randomly use either front door access (via a memory bank that uses a bus transactor, like PCI or AHB) or back door access, using a memory bank that is connected to an HDL model. The front door access is slower and takes simulation time, but front door access tests that HDL code path and may also test contention.

You can also build a memory bank that aggrgrates several vreg objects, so that back door register access occurs as a result of memory reads/writes.

C\C++ Interface

Memory functions

The memory manager has a small interface. You can map memory to some integer range. You also can add banks of memory for special purpose memory, such as ECC, parity protected memory, or a Context Addressable Memory (CAM). Later, other parts of your simulation can retrieve this memory by address or by path, and you can read and write this memory.

`void add_map (const std::string & path, uint64 first_address, uint64 last_address)`

`void add_memory_bank (memory_bank *)`

`memory_bank * lookup (const std::string & parial_path)`

`memory_bank * lookup (uint64 address_in_range)`

`void read (uint64 global_address, reg*)`

`void write (uint64 global_address, const reg & value)`

One item to note is that the read does not just simply return a reg. It may seem, at first, appropriate to do this, but then multiple data path width memory could not be supported. It is fairly common for todays memory subsystem to support byte, word, and long word access.

Creating, copying, and destroying a memory_bank

An internal class derived from the `memory_bank` class connects to the DUT. This class is within the memory namespace and is created whenever a `$note_memory_bank()` is placed in the HDL. However, you may want to aggregrate or add special functions for a memory range. In such a case, you want to create your own memory bank. The only parameter is the name of the memory bank:

`memory_bank::memory_bank (const std::string &)`

`memory_bank::~memory_bank () [virtual]`

Determining the right memory bank

For the read and write memory functions to work (when accessed by integer address), they must determine which `memory_bank` object will handle the access. Since the memory manager namespace contains a list of memory_banks, the lookup functions use the `contains()` methods to find a match. The first one to return true is the bank that is used. Note that there are two contains functions:

- One for lookup by address
- One for lookup by name

`bool memory_bank::contains (uint64 address) const`

```
bool memory::memory_bank::contains (const std::string & path)
const
```

Actually reading and writing memory

After a bank has been selected, the memory function calls one of these two methods:

- virtual void from_memory (uint64 *address*, *reg**) [pure virtual]
- virtual void to_memory (uint64 *address*, const reg & *value*) [pure virtual]

Examples

1. Simple memory use

This example shows how to hookup the HDL memory to Teal. It shows the most common way to read and write memory entries.

In your top level testbench, define some regsiter banks and hook them to Teal, like so:

```
module some_memory

reg[1:0] bank_0[1024:0];

initial $teal_memory_note(bank_0) //in module tb.memory_1

endm

module tb

some_memory memory_1();

some_memory memory_2 ();

endm
```

Now, in the verification_top() of a C/C++ source file, you tell Teal how to view this memory as an address range. Like so:

```
memory::add_map (“memory_1”, 0x100, 0x200);

memory::add_map (“memory_2”, 0x201, 0x400);
```

At this point, your program can read and write this memory by address. Here are some examples:

```
memory::write (0x10a, 22); //write local offset 0xa in memory_1
to 22.

reg val; memory::read (0x10a, &val);

if (val != 22) {

   vout log (“memory_example_1”);

   log << << teal_error << “At memory_1[“ 0xa << “] “ got “<<
val << “ expected “ << 22 << “.”

}

memory::write (0x20b, 33); //write local offset 0xb in memory_2
to 33.

reg val; memory::read (0x20b, &val);

if (val != 22) {

vout log (“memory_example_1”);

log << teal_error << “At memory_2[“ 0xb << “] “ got “ << val <<
“ expected “ << 33 “.”

}
```

2. Directly interacting with a memory_bank

This example shows how to read and write a memory bank directly. This can be useful when a large number of reads/write must be preformed to a single bank (as in initing a memory or in a pre-aging test) or when the memory is not designed for an “external” address range, in other words, internal RAM.

In your top level testbench, define some regsiter banks and hook them to Teal, like in the previous example. Then, you can get a pointer to the memory bank using Teal, like so:

```
memory_bank* bank = memory::lookup (“memory_2”); //partial path

bank->to_memory (0x10, 44);  //directly write a 44 to local
offset 0x10

reg val; bank->from_memory (0x10, &val);

if (val != 44) {
  vout (“memory example_2”) << teal_error << “At memory_2[“ 0x10 <<
“] “ got “
                    << val << “ expected “ << 44 “.”
}
```

Chapter 10: Concurrency in Teal

This chapter discusses the mechanisms Teal provides for dividing the verification task into separate, independent threads of execution. There are three main components to any such system: (1) a way to start, stop and join threads, (2) a mechanism for threads to signal one another, and (3) a way for threads to serialize data access. This chapter discusses how Teal provides these components.

Overview

Verification is a complicated task. If a complicated task can be broken down to a set of simpler, independent tasks the problem becomes less complicated. These tasks often are concurrent. This chapter describes creating, managing, and interacting with these concurrent tasks.

Theory of operation

As discussed in Chapter 4, the `verification_top`() function is your top-level controller thread. Generally, this thread waits for the DUT to be ready and then initializes the dictionary and random number subsystems. Next, a series of generators/checkers/transactors is started, and some end condition is tested. When the end condition occurs, all threads are stopped and the main thread exits.

You create threads using the `run_thread`() function. `This function` takes in the function you want to run and a pointer to a data area. A thread is created and the function is called. Normally, the thread never returns; your “main” thread cancels it with `stop_thread`()[1]. If, however, the thread is a temporary one, and so it returns, the last line must be `note_thread_completed`(). Teal needs this call for internal reasons.

When a thread is started, including `verification_top`(), it runs until it reaches a waiting point, which tells Teal that control can be returned to the HDL simulator. After a wait condition has been satisfied, the blocked thread runs.

There are three types of waiting points:

- at()
- `mutex::enter`()
- `semaphore::wait`()

1. The exception to this rule is generally the verification_top() thread, which spawns other threads and then returns.

The at() function

The `at()` function, the most common wait point, is intended to model the `@(sensitivity list)` statement in Verilog. This function takes in a sensitivity list of `vreg` signals. The signals are matched on the posedge, negedge or any change. A statement such as:

```
at (posedge (clk) || change (reset_n));
```

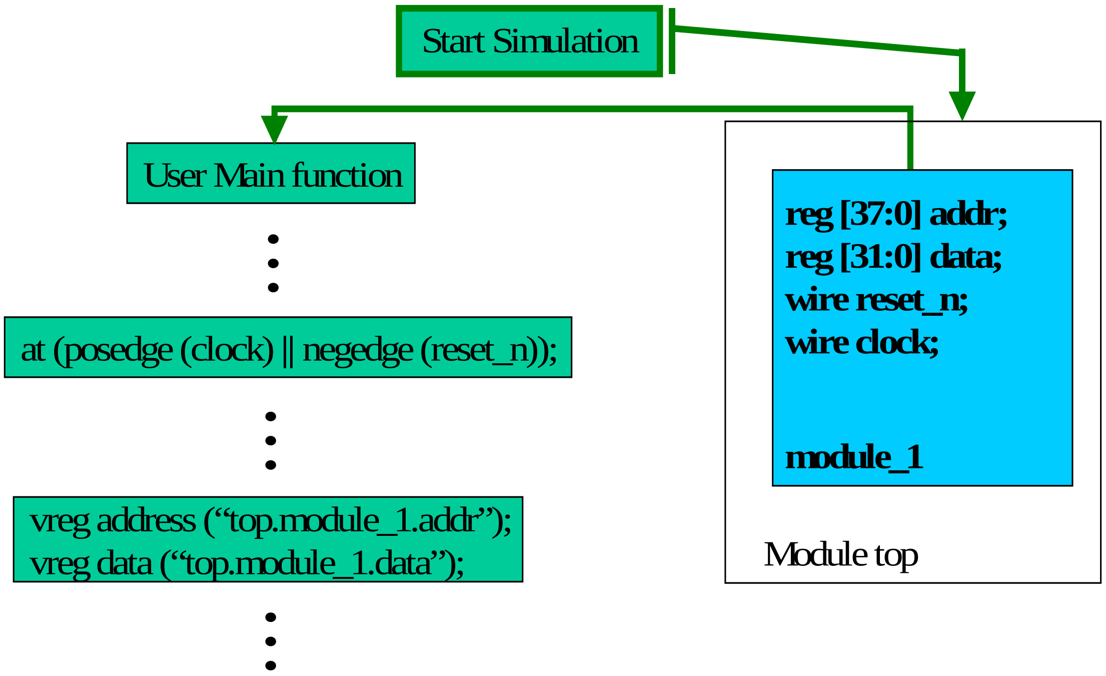
would mean to pause the thread until either :

▪The `clk` signal went from an X, Z, or 0 to a 1.

▪Any change occurred in the `reset_n` signal.

Execution would then continue after that statement.

The example above shows that, after simulation is started (in module Top) the verification_top() function is executed. It will run (in zero simulation time) until the at (posedge... clause. At this point, the thread pauses until either clock goes positive or reset_n



Start Simulation
User Main function
at (posedge (clock) || negedge (reset_n));
vreg address ("top.module_1.addr");
vreg data ("top.module_1.data");
reg [37:0] addr;
reg [31:0] data;
wire reset_n;
wire clock;
module_1
Module top

goes negative. Execution then continues after the at() clause, at the now more advanced simulation time.

Figure 1: Concurrency - Pausing the thread with the at() clause

The example above shows that, after simulation is started (in module top), the verification_top() function is executed. It will run in zero simulation time until the “at (posedge (...)” clause. At this point, the thread pauses until either the clock goes positive or reset_n goes negative. Execution then continues after the at() clause, with the now more advanced simulation time.

Mutex::enter ()

The `mutex::enter`() wait point is used when two or more threads need access to some common hardware resource. A good example is the PCI or AHB bus in the system. In the test system and in the software, many threads can be competing to access the bus (as the same master). The `mutex::enter`() call waits until no other thread is using that `mutex` object and then locks the object. After your thread finishes using the hardware, you must call `mutex::unlock` () to tell Teal that you have finished. At that point, any other waiting thread is allowed to access the hardware. For simple `mutex` management, see the `mutex_sentry()`utility object, in Teal.h.

semaphore::wait ()

The semaphore class is intended to loosely model the event object in Verilog. In this context, one thread is waiting for another thread to signal to it. The semaphore class provides this functinality with signal() and wait() methods[1]. The semaphore class is most often used for inter-thread communication. For example, suppose you have a monitor thread, and your test is waiting for an ACK packet to be sent. You would declare a semaphore object for `ack` in the monitor object (which also is a separate thread) and have a `method wait_for_ack`() that calls `semaphore::wait`() on the `ack` semaphore. When the main loop of the monitor object sees an `ack` on the wire, it would call `semaphore::signal` () on the `ack` object. This call would cause the `wait_for_ack`() method to return and the calling thread to continue running.

Simulation Time

The current simulaton time is returned by the vtime()function, which returns a uint64.

1. Note: This is exactly the same as the pthread condition variable, except that the Teal one is tied in to the simulation. Using the pthread condition variable will most likely crash the simulation.

Getting the Thread name from a thread_id

The function called `thread_name(pthread_t)` returns the name associated with the thread ID, which is returned by an earlier call to `run_thread`().

C/C++ interface

Creating and destroying threads

You create a new task with `run_thread`(). This function takes in a function to run, a data parameter, and a task name. `run_thread`() returns an ID to be used to stop the thread (or any other `pthread` function). The `stop_thread`() function takes in a thread ID and stops that thread

In general, a thread runs until it is stopped by another thread. If, however, a thread runs to completion and returns from the `user_thread` function, it must call `note_task_completed`() to let Teal clean up internal data structures.

```
void (user_thread) (void* user_data)
```

```
pthread_t run_thread (user_thread, void * user_data, const
std::string & name)
```

```
void * stop_thread (pthread_t)
```

```
void teal_finish () - iafter stopping all threads, end the
simulation
```

```
void note_task_completed ()
```

The `thread_name`() function converts between the thread_id and the task name.

```
std::string thread_name (pthread_t)
```

The vtime() function returns the current simulation time:

```
uint64 vtime ()
```

The at() function is the main way a task pauses. It is given a sensitivity list, which is an object that is automatically created (and destroyed) by calling posedge(), negedge() or change() with a vreg as its argument. The list is also extended when you have multiple vreg changes (negedge, posedge, or change) separated by the || operator.

Be aware that, by default, posedge() and negedge()test only the lowest bit for the appropriate edge. If you need to test for a different bit, pass a second parameter indicating the specific bit to test.

```
void at (const sensitivity &)
```

```
 posedge (vreg & v, uint32 bit_pos = 0)

 negedge (vreg & v, uint32 bit_pos = 0)

 change (vreg & v, uint32 bit_pos = 0)
```

Creating and destroying a mutex

Often several independent threads need to use a common hardware resource, such as a bus. If each thread has its own master identifier and the bus has hardware arbitrartion, each task can arbitrate for the bus wing the DUT hardware. In this case, a mutex is not needed.

However, if the independent tasks are the same master (as if they are emulating multiple software threads on a CPU), they need a simulation mechanism to arbitrate. This is the purpose of the mutex class. Once constructed, its pointer is passed to all affected modules (or hidden in a common transactor, like pci_bus_master, see the mutex test in the test directory of Teal). Each task calls lock() to either gain access to the hardware or block until the current task is finished with the hardware. Once the lock() method returns, you access the hardware, and then call release() to inform Teal that you are done with the hardware. At that point, any waiting threads re-arbitrate for the mutex.

The constructor only takes in a name for the mutex:

```
 mutex (const std::string & name)

~mutex ()
```

Working with a mutex

There are only two operations on a Mutex. One is to acquire the mutex, by calling lock() and the other is to release the mutex by calling unlock():

```
void lock ()

void unlock ()
```

Creating and destroying a condition

Whenever you have multiple tasks, there is a good chance that they will need to communicate. While tasks can put data into work queues and take data out, there must be some mechanism to signal that one task as just put some data in (or taken something out). Also, a monitor task may need to announce a particular condition (like ack, nak, or byte_sent). There may or may not be a receiver to note the announced event. The condition class provides such inter-task communication.

A condition is given a name when constructed:

```
condition (const std::string & name)
```

```
 ~condition ()
```

Working with a condition

Once a condition is created, its pointer is passed to the affected tasks (or buried in a method of a class, like monitor::wait_for_ack ()). At the appropriate time, one task calls wait() and it blocks until another thread calls signal.

```
void signal ()
```

```
void wait ()
```

Examples

Simple at() expressions

This example demonstrates how a thread can be paused on a number of signals. It is assumed that the testbench top module contains a register for an address and a clk.

```
vreg address (“testbench.address”);
```

```
vreg clk (“tb.clk”);
```

```
at (posedge (clk) || change (address)); //wait until next rising
clk or any address change
```

```
at (negedge (clk)); //falling edge test
```

Mutex

This example demonstrates how two threads cooperate to place their address on the shared hardware register called address. In addition to the testbench to having address, clk and data registers, it is assumed that there is a module that placed data on the bus in response to the address changing.

First, two similar functions are defined. Each one tries to get the mutex (passed in the context parameter) and then puts its address on the address register. One clock later, they retrive the value put in the data register (by the DUT) and then release their mutex.

```
void master_one (void* context) {
  mutex* m = static_cast<mutex*> context;
  m.lock ();
  vreg address (“tb.address”); address = 0x10;
  vreg clk (“tb.clk”);
  at (posedge (clk)); //wait one clk pulse
  vreg data (“tb.data”);
   vout log (“master_one”);
   log << teal_note << “data is “ << data << endl;
}
void master_two (void* context) {
  mutex* m = static_cast<mutex*> context;
  m.lock ();
  vreg address (“tb.address”); address = 0x16;
  vreg clk (“tb.clk”);
  at (posedge (clk)); //wait one clk pulse
  vreg data (“tb.data”);
   vout log (“master_two”);
   log << teal_note << “data is “ << data << endl;
}
```

The verification_top function creates the common mutext and starts the two threads running. It then waits for the two threads to complete.

```
void verification_top () {

mutex main_bus_mutex (“main bus”);

pthread_id one = run_thread (master_one, &bus_mutex, “task one”);

pthreda_id two =run_thread (master_two, &bus_mutex, “task two”);

join_thread (one); join_thread (two);

}
```

Semaphore

This semaphore example shows a producer/consumer pair of threads. One thread creates data and then signals the other to get the data. The consumer waits for data and then consumes the data.

A context structure is created that holds the semaphore and the data to be communicated. Note that this is a simple mailbox scheme.

```
struct context {

   void context (): mailbox (“main mailbox”) {};

   semaphore mailbox;

   std::vector <uint32> work_queue;

}
```

The producer gets the context and pushes two uint32s into the queue. Note that, in another design, the mailbox could have signalled just once. In this case the producer would have to drain the entire work queue.

```
void producer (void* c) {

  context* the_context = static_cast<context*> ( c );

  the_context.work_queue.push_back (10);

  the_context.mailbox.signal ();
  the_context.work_queue.push_back (20);

  the_context.mailbox.signal ();

}
```

The consumer gets the context and reads the two uint32s.

```
void consumer (void* c) {

  context* the_context = static_cast<context*> ( c );

  for (uint8 i(0); i < 2; ++i) {

    the_context.mailbox.wait ();

    vout log(”consumer”);

   log << received << the_context.work_queue.front ();

    the_context.work_queue.pop_front ();

  }

}
```

The verification_top() function builds a context, runs the two threads and waits for them to complete.

```
void verification_top () {

context my_context;

pthread_id one = run_thread (producer, &my_context,
“producer”);

pthread_id two (run_thread (consumer, &my_context,
“consumer”));

join_thread (one); join_thread (two);

}
```

Chapter 11: The Dictionary Module

This chapter describes Teal's way of getting parameters into a test. The dictionary is useful if the test can cover a range of features and capabilities, as it can be used to constrain the features on a per test basis.

Overview

This chapter introduces the dictionary, a namespace with several functions to solve a common problem in simulation: getting test parameters. Test parameters are separate from tests (although the test should have defaults) in that they bind a test to some specific range (or value) of parameters. The dictionary namespace integrates a file heirachy and command line parameters into a uniform interface.

Theory of operation

Sometimes you want to control a test using an external file. This allows a single test to have many different directed runs. These runs would still use random numbers, but that their range might be constrained by the test file. For example, the probability of a feature or error may be an external variable. Alternatively, your test file might be used to turn on and off features.

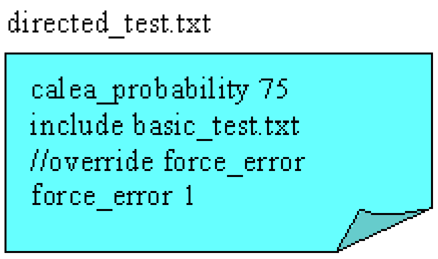
Teal provides a dictionary namespace for this purpose. After the dictionary namespace is initialized with a filename, the file is opened and the first word in every line is cached. Then, your test can query the dictionary (with the find() function) and recover the value after the keyword. For example, if the file had “number_of_streams 33” on a line, a call to dictionary::find (“number_of_streams”); would return 33.

There is one special first word “include”. If this word is found, the next word is taken as a filename and the dictionary module processes this file as well.

If the same word shows up in several lines (in any of the files), the last one processed will be the value that is stored.

For example, assume a file named “directed_test.txt” had the following lines:

Figure 1: directed_test.txt

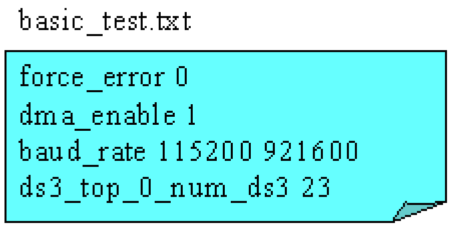


directed_test.txt

```
calea_probability 75
include basic_test.txt
//override force_error
force_error 1
```

Also assume that the file “basic_text.txt” has the following lines:

Figure 2: basic_text.txt



basic_test.txt

```
force_error 0
dma_enable 1
baud_rate 115200 921600
ds3_top_0_num_ds3 23
```

Since the parameter force_error is in both files, and the force_error setting of 1 is after the include with the force_error set to 0, the setting of 1 will override the setting of 0.

Since it’s often the case that command line parameters also act as test parameters, they are folded into the dictionary module. One the command line, a parameter such as “+my_parameter+my_value” will be entered as the word “my_parameter” with the value “my_value”. The command line parameters override any words of the same name in any file processed.

Since a test may want to get the dictionay file name from the command line, the find_on_command_line() function can be used before the dictionary::start() function is called.

C/C++ Interface

Dictionary main functions

To use the dictionary, you must first call `start`(). This function opens the file, processes all include directives,and copies all the entries as strings. When you are done with the file, call `stop`():

```
void start (const std::string & path)
```

```
void stop ()
```

To see if a parameter is on the command line, you can use the function below. This function is useful if you want to get the name of the dictionary file to open, or do not want to use a dictionary file. If the parameter is not on the command line, te default is returned. Note: You can pass in “” for the default if you want to just see if the parameter is there.

```
std::string find_on_command_line (const std::string &
parameter, const std::string & default_name)
```

Working with the dictionary

There is only one function to get the value of a parameter. The find() function returns the string associated with the parameter, or “” if it cannot find the word.

Since most parameters are not strings, there is another templated function to convert the string into other forms. This is based on the std::istringstream class defined in C++. The templated find() function relies on operator>>(const std::istream&) being implemented for the data type you need. This is a bit complicated, but fortunately, C++ provides this function for all the built-in types (int, char, long, double, etc). Note that it is the default parameter that keys C++ to the right template instance. So you can just call find(“my_parameter), my_integer_default) to get an integer value from the dictionary. See the examples, below.

```
std::string find (const std::string & name)
```

```
template<class data_type> inline data_type  find (const
std::string & name, data_type default_val)
```

Examples

Getting basic parameters

This example shows how to find integer parameters. Assume that a file, “teal_test.txt” has only three lines “foo 10”, “bar 12.34”., and “hex_value 0xffdd0”. The following code will read and print those parameters.

```
void verification_top () {

dictionary::start (“teal_test.txt”); //assume it has a line
“foo 10”

vout my_log(“basic parameters”);
my_log << teal_info << “Foo is “ << dictionary::find (“foo”, 20) <<
endm;

std::string not_here (dictionary::find (“not_here”));

if (not_here == “”) {

  my_log << teal_info << “expected \“not_here\” not found.” <<
endm;

} else {

my_log << teal_error << “\”not_here\” found.” << endm;

}

my_log << “bar is “ << dictionary::find (“bar”, 20.0) << endm;

//a fancy hex value

std::istringstream ss (dictionary::find (“hex_value”));

uint32 hex_value (9);

 ss >> std::hex >> hex_value;

my_log << teal_info << “Hex value is “ << hex_value << endm;

  }

}
```

A min max integer

This example shows how to find integer parameters. Assume that a file, “teal_test.txt” has only one line “ds0_channel_range 0 23””. The following code will read and print that parameter.

```
void verification_top () {

dictionary::start (“teal_test.txt”); //assume it has a line
“ds0_channel_range 0 23”

vout my_log(“two integer parameters”);

std::string channel_text (dictionary::find (“ds0_channel_range));

if (channel_text == “”) {

  my_log << teal_error << “channel range not found” << endm;
}
else {
 std::istringstream bar (channel_text);
 uint32 min_val (0);
  uint32 max_val (0);
 bar >> min_val >> max_val;
  my_log << teal_info << “ expected Min is “ << min_val << “ and
max is “ << max_val << endm;
}
```