Part II: An Open-Source Environment with C++

The previous part of the handbook was a high-level look at C++ and how to architect a verification system by using layers. Now we focus on a specific implementation of such a system.

This part of the handbook introduces two open-source libraries, called Teal and Truss, that together implement a verification environment that uses C++. The authors and others have used these libraries at several companies to verify real projects.

The libraries are free and open source because the authors feel strongly that this is the only way to unite and move the industry forward. Locking up people's "infrastructure" is not the way to encourage innovation and standardization—both of which are needed if the verification industry is to improve.

Consequently, you'll find no simulator-company bias in these libraries. These libraries work on all major simulators.

In this part we discuss the following:

- Teal, a C++-to-HDL interface that enables C++ for verification
- Truss, a layered verification framework that defines roles and responsibilities

- How to use Teal and Truss to build a verification system
- A first example, showing how all the parts we talk about fit together

Teal Basics

CHAPTER 5

Coming together is a beginning. Keeping together is progress. Working together is success.

Henry Ford

Building a verification system is a daunting task, but build we must. That is why we use the technique of layering, to break the problem down. By starting with the lowest layer—that is, the one that directly drives and senses the wires—we can start to get some real work done. Still, because C++ is not what most hardware engineers use for their HDL, we'll need an interface layer to connect the HDL with C++. Teal is just such an interface. Teal tries to be as unobtrusive as possible, using terms borrowed from the HDL domain, such as *posedge* and *reg*.

This chapter introduces Teal and shows how to use it. We'll talk a bit about the main parts of Teal—for example, how you can (fairly seamlessly) get and set values in the HDL, and how you can pause execution until HDL signals change.

Overview

Teal is a C++ class library for functional verification. Teal is tiny, consisting of only a handful of source files, yet it provides the necessary minimum features for verification. (A version of Teal is on the companion CD.)

Teal, like Vera, SystemVerilog, and "e," provides the illusion that the verification system is in control of the chip. In Teal, you write a `verification_top()` function, and create tests, generators, checkers, drivers, and monitors. Each of these objects can appear to be running independently of the chip, with each in its own thread of execution. Of course, in reality these threads only execute in response to a chip wire or register change. However, by driving wires and registers, the threads do, in some sense, control the chip.

Teal is unobtrusive; it does not get in the way of your C or C++ structure. You don't put Teal calls everywhere you want to sample or drive a signal, so Teal is also unobtrusive in the HDL code.

The authors realize that many companies have developed their own version of an HDL-to-C/C++ interconnect. We encourage those companies to contact us and share their experiences, so Teal can be made better. This is one of the reasons why Teal is open source.

What Teal provides

Teal enables functional verification by providing connections to HDL signals and allowing actions based on changes in the HDL simulation. It encourages the development of independent generators, checkers, drivers, and monitors by providing management for user-created threads that execute concurrently with the HDL simulation. Teal provides repeatability and constrained random-number generation, as well as a simple interface to pass in runtime arguments, either through the code or command line, or through "scenario" text files. Teal also provides flexible message printing.

This functionality provides the basis for functional verification, but it serves as only a small part of a verification project. You must still write code that stimulates the design, checks the output, and controls the

randomness. That is the real work of a verification project. (The next chapter talks about an open-source verification environment.)

Teal's similarity to HDLs

Although Teal does make use of classes and inheritance in C++, your algorithms for driving and sensing the wires can look close to what a hardware engineer is used to.

As an example, suppose you had a signal, located at `top.chip.address`, in your simulation and you wanted to get the value of it at the positive edge of a clock. The Teal code would look this:

```
teal::vreg clock("top.clock");
teal::vreg address("top.chip.address");
teal::vout log("logger id");
at(posedge(clock));
log << "The current address is "<< address" << endm;
```

Don't worry if this example is not clear. We'll walk through each of these Teal classes later. The point is that the `at(posedge(clock))` should be recognizable to Verilog coders. In addition, the address variable can be used as a regular C++ integral variable.

A tiny but complete example

This chapter delves into the details about all of Teal's classes, but let's look at a basic example of what a complete C++ example using Teal looks like. It should not be hard to understand the code presented here, assuming the reader has some familiarity with C++ (or C) and a general knowledge of Verilog (or VHDL).

In this example our chip implements a black-box function. Given a reference clock, it samples a stimulus on the positive edge of the clock and generates a response on the negative edge. To make things interesting, let's assume there is a three-clock latency from the stimulus to the response.

Here's how Teal might be used to drive the stimulus and get the response:

```
#include "teal.h"
using namespace teal;
#include <deque>


void verification_top(){
  const int latency = 3;
  const int number_of_iterations = 10;

  teal::vreg system_clock("testbench.reference_clock");
  std::deque<integer> stimulus_sent;
  vreg stimulus("testbench.stimulus");
  vreg response("testbench.response");

for (int i = 1; i <= number_of_iterations; i++) {
    //drive the stimulus to the chip and remember it
    at (posedge(system_clock));
    integer stimulus_int; RAND_UINT32(stimulus_int);
    stimulus = stimulus_int; //drive value to the chip
    //save value sent
    stimulus_sent.push_back(stimulus_int);

    //Read from HDL register "response" and print result
    at (negedge(system_clock));
    if (i >= latency) {
      //Note! 'response' in line below reads from HDL
      cout << "For stimulus " << stimulus_sent.front()
           << " the chip produced " << response << endl;
      stimulus_sent.pop_front();
    }
  }
  //need to collect last responses
  for (int i(0); i < latency; ++i) {
    cout << "For stimulus " << stimulus_sent.front()
         << " the chip produced " << response << endl;
    stimulus_sent.pop_front();
  }
}
```

It should be noted that the above example puts all code in the `verification_top()` function. However, this is not recommended for

real projects, where a lot more structure is needed (as will be shown throughout this handbook). The point here is that if you use Teal, you won't end up with code that is hard to understand. Teal is straightforward.

In this example we randomized a stimulus input and applied it to the chip, then just printed the response. In a more realistic test, you would have a model of the chip and compare the results to that model.

Teal's Main Components

It is important to decide on a "common currency" when designing a class library. The rest of this chapter describes the common currency of the Teal system—that is, the fundamental building blocks of Teal-based verification.[1]

The following is a summary of the most important classes and namespaces of Teal; more detail is given in the following sections.

- *The* `reg` *class*—This is one of the most basic classes in Teal. Its main purpose is to provide arbitrary-length, four-state (1, 0, X, Z) "registers" with corresponding operations. The `reg` class is useful for performing algorithms in the precision of the hardware. It also provides register-slicing operations.
- *The* `vreg` *class*—This is probably the most commonly used class in Teal, as it connects C++ code to the HDL. The `vreg` class provides mechanisms to connect wires and registers in the HDL simulation so they can be used in C++ code as though they are built-in C++ variables. The `vreg` class is inherited from the `reg` class.
- *The* `vout` *class*—This Teal class is used for logging, to help trace what happens during a simulation. Modeled after the standard C++ `cout` object, the `vout` class provides the ability to report, for example, debug, error, and other informative messages in a consistent format that is coordinated with HDL outputs.
- *The* `vlog` *class*—This class is a global resource that coordinates all the logging from your C++ code. It receives all `vout` messages

[1] This is not a complete reference manual, but rather an overview of the capabilities of Teal.

from the simulation and implements a filter chain, so you can add useful features such as replicating output to a file and removing messages or parts of messages.

- *The* `memory` *namespace*—This namespace provides an abstract interface for reading and writing memory. Internally, a group of memory banks are used to handle memory read and write requests, providing great flexibility.
- *The* `vrandom` *class*—Because using random numbers for test values is a staple of modern verification, this class is Teal's stable random-number generator. Though small, it provides thread-aware, independent streams of stable random numbers that can be guided by a single master seed. Of course, the numbers all have their own seed as well.
- *The* `dictionary` *namespace*—This namespace is a global service that abstracts how to set parameters in your test. It provides functionality to get and retrieve parameters from code, the command line, or external "scenario" files.
- *The* `run_thread()` *function*—This function forks off a thread. You'll use this whenever you have a function, such as a generator or monitor, that needs to operate independently of the test. This function provides a base capability for building transactors, drivers, checkers, and so on.
- *The* `at()` *function*—This function allows a thread to pause until any of the HDL signals has changed. You provide a sensitivity list of `vreg` objects, with modifiers such as `posedge`, `negedge`, or `change`. Used with the `run_thread()` function, it allows several independent tasks to run simultaneously.

All of these classes are described in the following sections, along with the small requirements that Teal puts on the HDL testbench (Teal needs to be initialized from the HDL testbench), and a discussion of how to create the "user-code entry point" function called `verification_top()`.

Using Teal

It's time to dive into some details regarding how Teal can be used for functional verification. This walk-through of Teal makes it easier to understand the "real world" examples presented in subsequent chapters, while illustrating how Teal can be used in your environment.

Initialization

Let's start at the beginning. For Teal to be used, it must be initialized from the HDL. This is done through an HDL function call that launches Teal. When Teal starts up, it initializes itself, then calls a user-provided function called `verification_top()`.

Verilog is the HDL of choice in this handbook. Because Teal was developed to work with Verilog, many of Teal's syntax and naming conventions mimic those of Verilog.[2]

Teal uses the Programming Language Interface (PLI 1.0 or 2.0) to connect C++ code to HDL simulators. To this end, you must put a PLI call somewhere in the HDL code to start Teal. This call is normally put in an initial block at the top-level testbench, but it can be put anywhere and called at any simulation time. The call is called `$teal_top;` and other than a call for "back-door" memory access, it's the only required HDL call for Teal.

Your Verilog testbench should include the following:

```
module testbench;
  ...
  initial
    $teal_top;
  ...
endmodule;
```

That's all there is to it. Teal will now start and run your test.

[2] Unfortunately, while a Teal for VHDL is in the works, it wasn't finished in time for publication. Contact the authors at www.trusster.com if you are interested.

Your C++ test

When the simulation begins, the call to `$teal_top` causes Teal to start the threading system and thereafter call a user-defined function called `verification_top()`. The `verification_top()` could be as simple as the following:

```
#include "teal.h"
void verification_top()
{
  teal::vout log("first code");
  log_ << "Hello Verification World" << teal::endm;
}
```

The `verification_top()` must be defined by the user, or Teal won't link. It normally instantiates other classes and calls their methods. (Subsequent chapters will show example of this.)

Registers

Teal's `reg` class implements a four-state logic, as well as all commonly supported HDL operations while making sure that X's and Z's propagate correctly. The class supports addition, subtraction, shifting, boolean operations, and comparisons. As in any HDL, bit fields (or subranges) are supported, and they can be on either side of the equal sign. (This will be described in more detail in the next section.)

The `vreg` class builds on the `reg` class and adds the connection to an HDL simulation. All events that happen to a wire/`reg` in either the C++ or HDL simulation get reflected on both sides. The `vreg` is one of the most used classes in Teal, as it serves as the connection point between HDL and C++.

Creating registers

Creating either a `reg` or `vreg` is easy. However, here is one of few places where the two classes differ. When you create a `vreg`, you supply the string HDL path to the corresponding HDL register, port, or wire you want the variable to reflect. Teal then automatically links together the C++ variable and the HDL signal, and also figures out the correct bit length for the C++ class.

When you create a `reg` object you don't supply a Verilog path, as there is no connection to an HDL. There are several ways to create a `reg`. The default is just like in Verilog, a one-bit variable. You can also give the `reg` an initial value, in which case the register is 64 bits wide. If you need a specific bit width, you can specify that after the initial value.

Here is an example of how you would construct a `vreg` and a `reg`:

```
vreg chip_register("testbench.chip.data");
reg cpp_register(0x7FFFFFFFFFF,47);
```

The first line connects the variable `chip_register` to the HDL signal located inside the chip instance of the testbench. The second line creates a 47-bit register array and assigns it an initial value.

Working with a reg or vreg

Teal registers are written to act like built-in types as much as possible. This makes working with them easy, and they support assignment to and from most other built-in types; for example, assigning the value of an `int` to a `vreg` and vice versa would look like this:

```
int drive_value(0x52571);
vreg v_signal("testbench.chip.signal");
// Assign a value to HDL signal
v_signal = drive_value;
//...
// Sample an HDL signal, assign it to a_sampled_value
int a_sampled_value = v_signal.to_int();
```

The assignment above would work for `reg`'s as well.

In functional verification it is common to access individual fields in a register, whether the contents are individual bits or strings of bits. Teal provides the capability to access both, as the following example shows:

```
uint32 x; RAND32(x); //Assign a random value to x
reg a_reg(x,32);
reg a_field = a_reg(32,25); //bit 32..25 to a_field
cout << a_field;
```

Registers have a number of logical and mathematical functions as well. These functions help define the correct four-state behavior for operations, and make the registers similar to C++'s built-in types.

The following is an example of some of the supported register operations:

```
vreg addr(path + ".address");
addr += 2;
addr = addr >> 2;
addr = addr << 4;
if (addr > 0x64)...
if (addr != 0x1)...
```

Teal does something a little differently when comparing two registers. Because `reg` is a four-state variable, Teal implements the `operator==()` as the Verilog triple equal in HDLs. That is, Teal looks at both the 0/1 value and the X/Z value when comparing two registers.

Logging Output

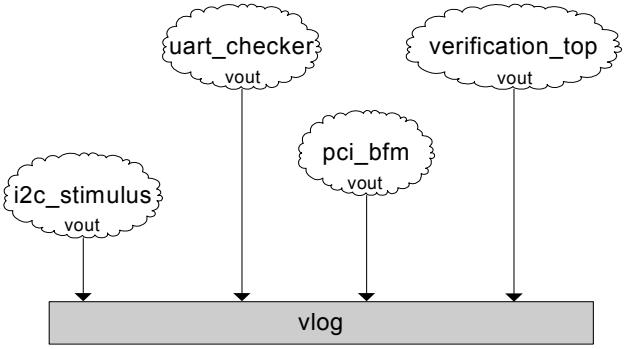
Because a lot of debugging is done by reading simulation log files, in order to see a progression it is important to organize simulations well. In other words, to enable postprocessing, error counting, messages, and possibly filtering, it is important to have a consistent message format. Fortunately, the logging facility in the Teal classes encourages such uniformity. Teal comes with a standardized, customizable logging mechanism, called `vout`, which mimics C++'s standard streaming mechanism.

Teal uses a two-level logging scheme, as shown in the following figure. In any code that needs to print information, a `vout` object is created. As many `vout` objects as needed can be created—which is good, because each `vout` object can have a relevant instance name.

Each `vout` object "under the covers" calls a global service `vlog` object. This is done so that there is a single point of control where reordering, demotion, changing, or deletion of parts of any message can be done.[3]

[3] Although describing this capability completely is beyond the scope of this handbook, subsequent chapters show several examples.

Vout and Vlog Objects in Teal



Because `vout` is modeled after the standard C++ library stream `std::cout` object, `vout` directly supports the output of the standard types. However, by following a few simple guidelines, you can print complex objects as conveniently as the standard types.

To end a message, just call the Teal `endm` function. To describe a multiline message, just use `endl` (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 in. You can then build any number of message statements. For example, note the following:

```
#include "teal.h"
using namespace teal;
void verification_top() {
  vout log("a test");
  log << teal_info << "val" << hex << 207218 << endm;
}
```

This example prints the following (assuming a thread of `tx`, a file of `uart.cpp`, a line number of 313, and a simulation time of 77 ns):

```
[77 ns] [tx] [a test] [uart.cpp] [313] val 64'h32972
```

Teal displays the file and line number in the source code that originated the message. This is useful when the same message comes from several different places in the code. (Of course, this information can be suppressed.)

Note that when you finish a message statement (by using `endm`), the vout instance adds the simulation time, the current thread name, and the functional area to the message, then sends the message to the `vlog` global service. It does not send the message as a text string, which would not allow the 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 with respect to their components—for example, to demote errors to a warning, or stop all output from a file or a functional area.

The `vout` class also supports decimal, hexadecimal, and binary output. You select the type of output by placing either a `hex` or `dec` or `bin` in the message statement. The `reg` class also looks at the setting when the `reg` is converted to a string.

However, you often do not need to use the global filtering mechanisms of `vlog`. Instead, you can turn off the display of parts of a message directly, at the `vout` instance. This is described in the Teal reference manual (available on the accompanying CD).

Most verification systems have several levels, or types, of messages. Teal, being no exception, uses the following general categories:

- `teal_info`— Used for standard messages.
- `teal_debug(<level>`—Used when a test wants to display a little more diagnostic information. This is a level-sensitive output; the `vout` class has level-setting methods and accepts a level for debug messages. The message is displayed only if the level of the message is less than or equal to the level that is set.
- `teal_error`—The error type is used when the chip's expected behavior is different from the expected.
- `teal_fatal`—This more-severe error type ends the simulation after displaying the message.

Examples of the above are provided in later examples.

Using Test Parameters

It is often important in functional verification to provide test parameters. These are frequently used, among other things, as constraints for random tests. For, example, a single test case may have several different sets of constraints, each of which covers a selected range of parameters or directs the test into interesting corner cases.

Because such parameters are commonly used, Teal provides a standard, flexible way of working with them. Test parameters can be defined by means of text files, code, or command line entries. Teal handles simple integer and string parameters as well as complex parameters.

The functionality of Teal's parameters is defined in the `dictionary` namespace. Teal maintains a list of parameter names and values, so that a test, for example, can query the dictionary and recover the value.

When you call the `dictionary::read(std::string)` function, Teal reads a text file, takes the first word on each line as the parameter name, and saves the rest of the line as data for that parameter. A special keyword, `#include`, is used to open other files from within files. If a parameter is repeated, the last definition is saved.

In addition to using files, you can also use code to add parameters. When you do this, you have the option of replacing an entry or not.

Parameters can also be entered on the command line. In this case, they override any parameter set by a file or the code. In this way, a parameter can have a default value but still be overridden by a script.

As an example, let's suppose we are testing a UART interface. We have a default parameter file that sets up default constraints, and then each specific test overrides a few values as well as defines its own parameters. The default parameter file could look like this:

```
//in default_parameters.txt:
force_parity_error 0
dma_enable 1
baud_rate 115200 921600
```

A specific parity-error test case could use the default parameter file and override the `force_parity_error` setting like this:

```
//in parity_error_test_parameters.txt:
stop_error_probability_range 32.81962 75.330
#include default_test_parameters.txt
force_parity_error 1
```

The `#include default_test_parameters.txt` line above tells the dictionary to open the `default_test_parameters.txt` file. The `force_parity_error 1` repeats the `force_parity_error` parameter and overrides the default value.

It is not always appropriate to use files to pass parameters. Using files can be good if you need to have many different test parameters and a few basic tests. However, it can be clumsy to make sure the files stay with the respective test code. Therefore, the examples later in this handbook use the code mechanism. Nevertheless, including such files, or even passing parameters on the command line, can be done after most of the test is written, without having to modify the test itself.

So how do we pick up the parameters? The following is a complete basic example of how these parameters could be retrieved:

```
#include "teal.h"
using namspace teal;
void verification_top() {
  // reads file shown above
  dictionary::read ("default_parameters.txt");
  vout log ("first_parameter_example");
  log << teal_info << "force_parity_error is " <<
  dictionary::find ("force_parity_error") << endm;
  }
```

Because most parameters are not strings, Teal provides a templated function, `find()`, to convert parameters to the correct variable format. The `find` function always returns a string—either an empty string ("") if the parameter is not found, or the actual string associated with the parameters. This function relies on the `operator>>` to be defined for the variable class used. The `operator>>` is defined for all built-in types (such as `int`, `char`, `long`, `double`, and so on). For your classes, you can define your own `operator>>` and then use Teal's `find()`.

If defining an `operator>>()` is not appropriate for your class, or if you don't have a class but instead have a collection of built-in types, you can use `std::istringstream`. This allows the code to create a stream from

a string, from which you can then extract the `chars`, `ints`, `doubles`, and so on, as needed.

For example, to read the `stop_error_probability_range` (from the example above), you would use the following:

```
#include "teal.h"
using namespace teal;

void verification_top(){
  dictionary::open("parity_error_test_parameters.txt");
  //reads "32.81962 75.330" from stop_error parameter
      into ss
  std::istringstream ss(
    dictionary::find("stop_error"));
  double stop_error_min (0);
  double stop_error_max (0);
  ss >> stop_error_min >> stop_error_max;
  vout log("showing double double reads");
  log << "Stop error range is "<< stop_error_min <<;
        " to " << stop_error_max << endm;
  }
```

The example above works for all integral built-in types.

Accessing Memory

For most verification projects it is important to be able to access memory. Sometimes you want to do this in zero simulation time. Allowing "back-door" accesses of memory improves simulation performance, allows the monitoring of memory for automatic checking, and makes it possible to insert errors into memory for test purposes. Teal provides such a "back-door" mechanism but also, of course, supports "front-door" access, which can map some memory address ranges to a transactor-based model.

Teal defines each accessible memory (transactor model or memory array) as a `memory_bank` object. A `memory_bank` object can be accessed directly through member functions called `to_memory()` and `from_memory()`, but each memory can also be associated with an address range, through

the `add_map()` function. In this way, memory can be accessed through addressing by means of `read()` and `write()` functions.

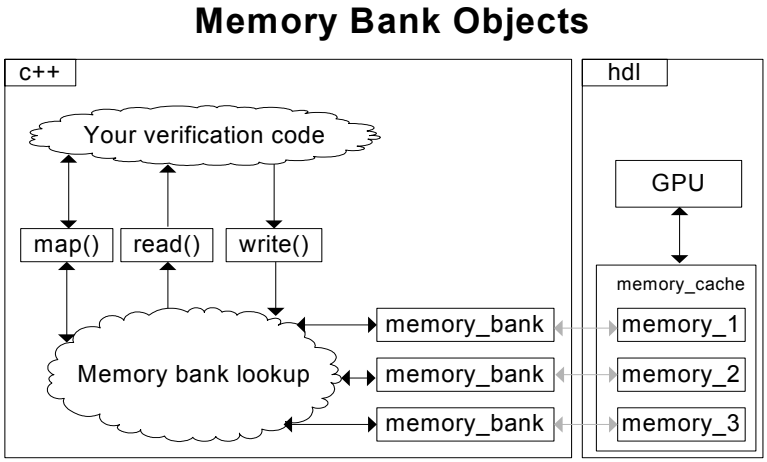
Working with address ranges has many advantages, because it creates code that is easier to understand and is closer to production software. The `read()` and `write()` functions can even be redefined in production software to become simple integer pointers, as is often appropriate.

When writing a memory transactor, you must define your own `memory_bank` object, but when working with HDL memories, put a `$teal_memory_note()` in the HDL. Teal uses this call to make a `memory_bank` object for you.

The following example shows how HDL memory arrays can be associated with an address range and accessed. An example of how to write memory transactors is in the UART example chapter.

A memory note example

The following diagram shows a small part of a larger testbench structure. This environment verifies a graphics chip that saves graphical texture information in its memory cache. In order to speed up simulation, back-door loading of the texture into the chips memory is used.



To support direct memory access you need three things. First you need the Teal PLI function `$teal_memory_note()`, which is called for each

memory register array to be accessed. This registers the array with Teal and creates a memory object for that memory. Then you need to deffine an address range for each memory instance to be used, allowing Teal to translate from an address to a specific memory. Finally, once the address range is established, you access that memory through `read()` and `write()` functions.

To do this in the environment pictured above, an initial line block is added to the memory model (that is, the model is, instantiated as `memory1`, `memory2`, and `memory3`, above). So part of the memory model looks like this:

```
reg[31:0] memory_bank_1[1024:0]; //Actual memory array
//to register memory_bank_1 with Teal.
initial $teal_memory_note(memory_bank_1);
reg[31:0] memory_bank_2[1024:0];
initial $teal_memory_note(memory_bank_2);
reg[31:0] memory_bank_3[1024:0];
initial $teal_memory_note(memory_bank_3);
```

As can be seen in the illustration, the memory model gets instantiated three times as `memory1`, `memory2`, and `memory3`. In the `verification_top()` function, the three memories get address ranges declared like this:

```
memory::add_map ("testbench.dut.memory_unit.memory_1",
                 0x100, 0x200);
// The following assums the subpath memory_2 is unique
memory::add_map ("memory_2", 0x201, 0x400);
memory::add_map ("memory_3", 0x401, 0x600);
```

Now any test can access these memory spaces through simple read and write function calls. Furthermore, neither reading nor writing memory consumes any simulation time. A simple memory access would look like this:

```
memory::write(0x10a, 22); //i.e 0xa in memory_1 = 22
if (memory::read(0x10a) != 22) {
  vlog("memory_example_1") << teal_error
              << "At memory_1[0xa] got "
              << memory::read(0x10a) << " expected 22."
}
```

Note that while the memory is written and read at `0x10A`, the actual memory is accessed at `0x0A`. This is because of the `add_map()` that we performed. This allows the rest of the verification system to read and write memory as specified in the chip's memory map. Teal takes care of finding the correct memory bank to access, and then removing the mapping offset.

Constrained Random Numbers

It is important to have a stable, repeatable, seeded random-number generator. Teal's `vrandom` class provides independent streams of random numbers that can be initialized from a string or file. There are also convenient macros for the most common random calls, such as getting a random integer value or getting a value from within a range.

The rest of this section describes the required initialization of the random generator and some simple examples.

Required initialization

Before using any random numbers you must initialize the random-number generator. This is done by calling the `init_with_seed()` function and passing it a 64 bit-seed value. It is recommended that higher-level code keep track of this seed value and pass it to the random-number generator. To initialize the random seed generator you would call the following:

```
uint64 master_seed;
...
// master seed gets initilized by higher layer
vrandom::init_with_seed(master_seed);
```

After the random-number generator is initialized, it is ready to be used. The examples in this handbook use the dictionary to get the master seed.

Using random numbers

Because integers are so commonly used here, Teal provides a couple of macros to deal with integers. After you have initialized the random-

number generator you can call these macros directly. The most often used macros are RAND_32 and RANDOM_RANGE, which generate a 32-bit random value and a 32-bit value within a range, respectively.

Here are some examples:

```
#include "teal.h"
using namespace teal;
void verification_top() {
  uint32 a_rand32; RAND_32(a_rand32)
  uint32 a_random_range;
         RAND_RANGE(a_random_range, 0, 0x030837);
  vout log (" random number test");
  log << "a_rand32 is " << a_rand32 << endm;
  log << "a_random_range is " << a_random_range << endm;
}
```

When you want to create more-elaborate random numbers, you need to work with the vrandom class directly. The vrandom class is a simple class that you can draw numbers from after it is created. This gives you more direct control over the generation of random numbers. The base vrandom class provides a uniform distribution, but you can create your own classes to have segmented, logarithmic, or other distributions.

You would create an object for your inherited class and draw a number like this:

```
vrandom a_random("some string", some_integer);
uint32 a_random_value = a_random.draw();
```

The macros use the ANSI standard __FILE__ and __LINE__ for the string and the integer. These parameters are hashed with the master seed and are used to initialize this particular random-number generator. You may want to pass in your own values. (For more details, see the reference manual available on the CD.)

Working with Simulation Events and Concurrency

Chips are massively parallel, which means they have many interfaces working at once. So when we test them, we need parallelism in testing as well. Teal provides the ability to start a thread and, if you want, wait for it to complete.

Why does Teal provide this capability when there are already several packages, both operating-system specific and in public domain? Most of the current simulators will "core dump" if any thread runs after control returns to the HDL simulator. Teal's threads ensure that this does not happen. It is this capability that provides the illusion that the C/C++ code is in control of the simulation.

However, having many threads of execution is no good if we cannot pause for some change in an HDL signal or to wait for another verification thread. Fortunately, Teal provides this capability. As soon as you have threads, you'll need a mechanism for exchanging events between threads. Teal calls this mechanism a *semaphore*.

Let's back up a bit and talk about running a thread. The Teal function `run_thread()` allows you to call a c-function in a new thread of execution. This is exactly how Teal starts your `verification_top()` function. The next chapter shows how the `run_thread()` function can be made more "object oriented," but the base mechanism is a c-function. This allows you to decide how object-oriented you want your threads.

It is possible that you may have several `verification_top()` types of functions and want to use that style for starting threads. To wait for a thread to finish, the `thread_join()` function is used. How does Teal know which function to wait on? The `run_thread()` function returns a `thread_id`, which is passed to the `thread_join()` function.

Once you have started a thread, you'll probably set some wires in the chip and then wait for some response. To wait for a wire change, use the `at()` function, which is intended to model the `@(sensitivity list)` statement in Verilog. This function operates on a sensitivity list of `vreg` signals, and the signals are matched on the `posedge`, `negedge`, or any `change`.

Take, for example, a statement such as the following:

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

This statement would pause the current thread until either the `clk` signal changed to a one or any change occurred in the `reset_n` signal. Execution would then continue.

As you build layers above this low-level wire layer, the threads themselves need to communicate. Teal's uses its signal and wait class, `semaphore`, to accomplish this. As with threads, the reason Teal provides these capabilities is to prevent a "core dump" in the simulator.

To communicate among threads, two threads need to share a `semaphore` instance. Then one thread (or any number of threads) pauses by means of a `semaphore::wait()` call. Another thread eventually gets some data or reaches some condition and issues a `semaphore::signal()`. That call unblocks the waiting thread. Because you cannot know the order of the thread's execution, a `wait()` may occur after the signal has occurred. The decision regarding whether a thread should honor this previous signal is up to you. If you want to wait for signals that occur only at the current simulation time or later, use the `wait_now()` method.

There is one last point to make about threads. Sometimes you want to make sure that only one thread is using a piece of code at a time. This is common in a BFM that is accessed directly (as opposed to when a queueing mechanism is used). In this case, the BFMs send, read, or write methods must use a `mutex` class. A `mutex` is a mechanism that ensures only one thread uses some shared resource at a time (as will be described in the OOP part of this handbook).

Summary

This chapter introduced an open-source C++-to-HDL interface, called Teal. We talked a bit about how Teal starts up and is connected to your testbench.

Teal's `register` class was covered, along with its inherited class `vreg`. These two common-currency classes are the backbone of the interface to the HDL.

Logging is a very important capability of a verification system. Teal's `vout` class and the global service class `vlog` provide a uniform, yet very flexible, logging capability.

Almost all tests need to have control parameters set by code or files. Teal's dictionary provides a global service for managing parameters.

The `memory` namespace of Teal can be used for both register access and internal chip memory accesses. If reads and writes are extracted from the actual underlying mechanism, different transactors can be used.

Random numbers are essential in verification systems. Teal provides a stable, independent random-number generator.

We ended the chapter with a look at concurrency and Teal's `at()` function. We looked at the `semaphore` class and the `mutex` class for coordinating different threads.

For Further Reading

- The *Teal User's Manual*, available at www.trusster.com, describes Teal in far more depth than this chapter does.
- For connecting the C++ code to the chip, a great handbook is *Principles of Verilog PLI,* by Swapnajit Mitra.
- A standard reference manual on PLI/VPI is *The Verilog PLI Handbook: A Tutorial and Reference Manual on the Verilog Programming Language Interface,* by Stuart Sutherland.

- The pthreads package, officially IEEE POSIX 1003.1c-1995, describes most of the capabilities that a multithreaded system needs.

Truss: A Standard Verification Framework

CHAPTER 6

Truss, and verify.
Anon.

Have you ever watched a building being constructed? Early in the project, when the frame of the building is just a skeleton, it's not clear what the finished building will look like. However, as construction continues, from the windows down to the cubicles that are our workplaces, the intent of the framework becomes clear. In fact, a large part of the building's presence depends on the fundamental structure.

This same basic process occurs when we build a verification system. Early in the project, the application framework is built. The result of years of best practices from both the verification and software fields, Truss is an application framework for verification. It is an *implementation*, and therefore makes some decisions about how things should be structured. With verification as with construction, the framework sets the tone for the system.

Truss is a layered architecture, so you can choose how to implement the layers. Although it makes very minimal assumptions, Truss does provide some base classes and conventions as a guide.

Overview

This chapter presents three main topics:

- The roles and responsibilities of the various major Truss components
- How these components work together
- How you adapt this framework for your verification system

This chapter builds on the two previous chapters of the handbook. It implements an open-source verification infrastructure based on the discussion in the Layered Approach chapter. It also uses the Teal library described in the last chapter as a connection between C++ and the simulation.

Teal provides the fundamental elements of a verification system and supports a wide array of methodologies. Truss, on the other hand, provides the infrastructure layers above Teal, adding a set of classes, templates, idioms, and conventions to facilitate the construction of an adaptable verification system.

One of the tricks in building a reasonable system is to find the *key algorithm*. The rest of the algorithms can usually fit around that key algorithm. For example, in a video editing program the key algorithm is all about getting the pace of the edits right. When you watch a movie, that happy, sad, or scared feeling you get comes from how well-timed and precise the changes in scene are.[1] The authors, having developed software for video editing systems, know that in this domain the key algorithm is implemented by adjusting the edit points of a few seconds of video while the video is constantly looping around those edits. This is not a trivial thing to do, because multiple streams of video and audio,

[1] Okay, emotions also come from the music, but everything works together.

possibly with software algorithms to implement effects, are changing as the user is adjusting the edit points.

In the verification domain, the key algorithm is the sequencing of the various components of the system. The authors refer to this as "the dance," as there are usually a few interacting components involved. As we talked about in the Layered Approach chapter, the top-level dance takes place between the test, the testbench, and the watchdog timer. Truss implements this dance in the `verification_top()` function—but Truss does not stop there. The authors believe that this dance is the key algorithm in several layers of the system, so we created a `verification_component` abstract base class. Also, we created `test_component` and `irritator` base classes to be the "`top`" at the interface and feature layers of the system. Recognizing and reusing the dance is a significant part of Truss.

This chapter explains the major components of Truss, providing code examples where appropriate. Subsequent chapters provide more-detailed examples.

General Considerations

The authors have worked on several different implementations of verification systems before Truss was available. While at a high level verification systems can be described uniformly, the language used to build them has a lot to do with how a specific framework is constructed.

Using a language other than C++

It is possible to build an OOP-based verification framework in languages other than C++, but no other verification language on the market has the OOP capabilities of C++. For example, when a language that does not support operator overloading is used, the generic `operator==()` or copy constructor cannot be used. To provide this basic required functionality, a common generic base must then be used. Unfortunately, this warps the framework and produces a fragile architecture—mostly because of the unsafe type casting. As another example, with a language that has a

compilation library (such as current HDLs), there is usually a failure to make a distinction between interface and implementation. This leads to a more-complicated framework, as test writers must separate the interface from the implementation manually and repeatedly. C++ avoids these problems.

Keeping it simple

A stated goal with both Teal and Truss is to avoid unnecessarily complicated code. C++ has many powerful features, but many times they are not appropriate. It is easy to get distracted with C++ techniques and forget that the real goal is keeping the whole team productive.

For example, implementing a generic interface for a verification component, such as a transactor, as a template can be tricky. Sometimes using the template can be more complicated than simply replicating code.

Sometimes only a convention should be used. An example of this is the *generator* concept. One could define an abstract base class, yet the common methods come down to just `start()`, `stop()`, `report()`, and a few others. It turns out that this concept of `start()`, `stop()`, and so on is common to a large set of verification tasks, and is represented in Truss as the abstract base class `verification_component`. However, the concrete subclasses are inherited from `verification_component` only if they use the bulk of the methods. Any smaller subset uses the same named methods as a convention instead.

> *In this way, the framework is not warped to fit a generic class. Even more important, your design is not warped to fit the generic class.*

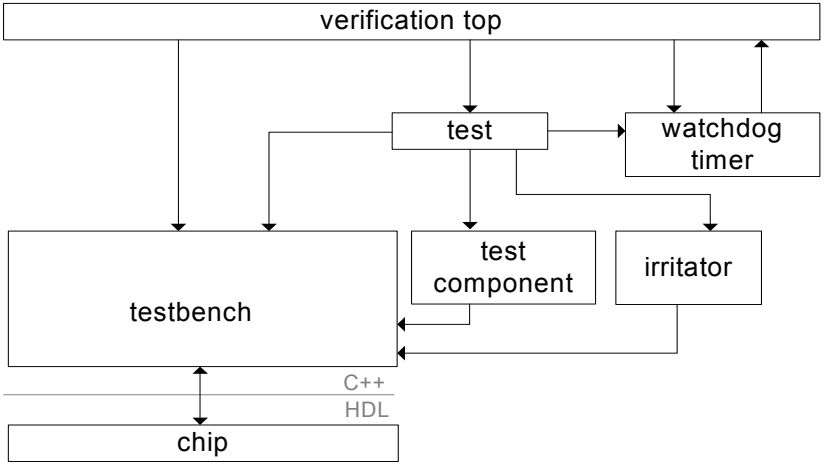
Truss implements a specific methodology for functional verification. As in any endeavor to generalize, the terrain is fraught with peril. Nevertheless, as writing code entails making judgments about what is the "right" decision, Truss attempts to generalize a style of verification. Deciding on the right balance between generic and specific is a judgment call for the team. The idea behind Truss is to foster a a small, usable, and adaptable methodology for beginners through experts. As such, Truss

provides an example of the techniques presented in Part III of this handbook.

Major Classes and Their Roles

Truss is an implementation of the layers talked about in the Layered Approach chapter. Consequently, there are only a few top-level components—the verification top, the testbench, the test, and the watchdog timer. Each component has a specific role. These components and their roles have been architected to allow a large amount of flexibility with a relatively simple interface. These top-level components (and those the next level down) are shown below:

Verification Component Hierachy



The top-most C++ component is the `verification_top()` function, whose role is to create and sequence the other components through a standard test algorithm. (The algorithm is explained in detail in the next section.) In addition, `verification_top()` initializes all global services, such as logging, randomization, and the dictionary.

The watchdog timer is a component created by `verification_top()`. This component's role is to shut down a simulation after a certain amount of time has elapsed, to make sure the simulation does not run forever.

The testbench top-level component is the bridge between the C++ verification world and the HDL chip world. As such, the testbench's role is to isolate the tests (and test writers!) from having to know how C++ transactors, traffic generators, monitors, and so on interact with the chip. Whether a bus functional model (BFM) writes to registers or forces wires should not be of concern to the test writer.

In addition, the testbench holds the configuration objects of the chip. This is needed by the BFMs, transactors, and similar agents to be able to configure the chip correctly. There is probably a configuration object for each interface of the chip. For chips that contain internal functions, such as dynamic memory allocation (DMA), there may be a configuration object for each function.

The last, but certainly not the least, top-level component is the test itself, whose role is to execute a specific functionality of the chip. It does this by using the testbench-created BFMs, monitors, and generators. The test is responsible for choosing among the testbench's many configurations and capabilities and exercising some subset of the chip's functionality. In general, the test contains very little code. This is because any code it contains may need to be used in other tests as well. To support code that is more adaptable, a test normally consists of several test components, as will be discussed later. The exception is for directed tests, in which case registers may be overwritten, specific traffic patters sent, or specific corner cases exercised directly in the test component.

Key test algorithm: The "dance"

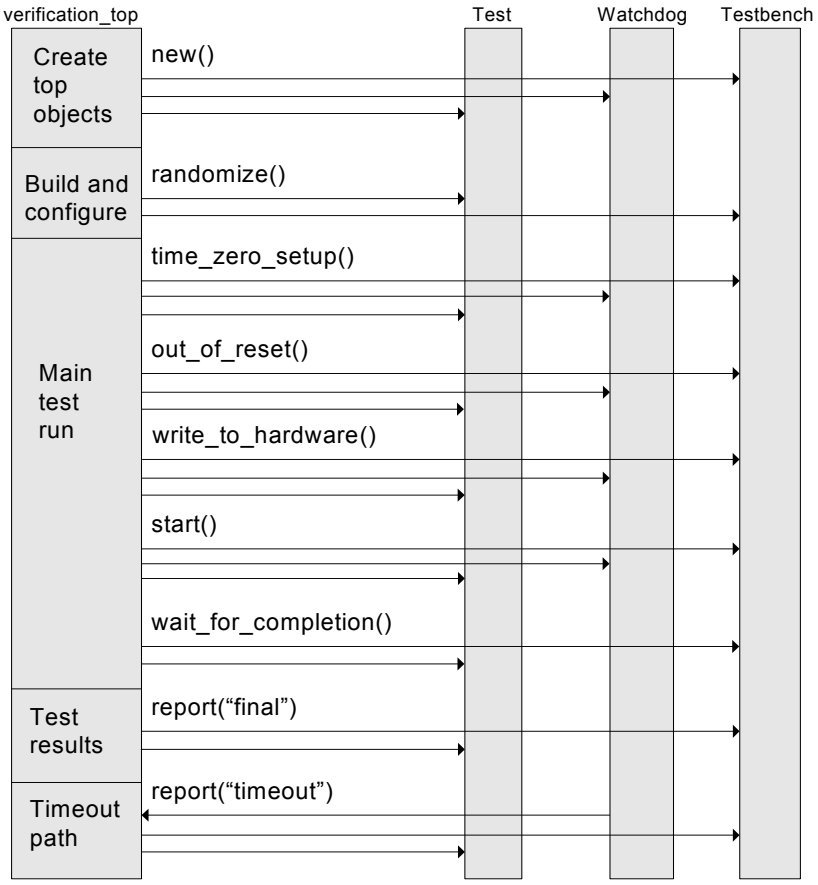
The top-level components of the previous section have a complex, yet necessary, set of interactions. This ensures the maximum flexibility for a test, while providing a known set of interactions. This is one of the tricky parts of a verification system. This section discusses this standard algorithm, which we call the "dance."

In general, the top-level components are created, randomized, and then started. Then `verification_top()` waits for the test and testbench to be completed. This is called the "polite" path. If the watchdog timer

decides that a timeout has occurred, the "impolite" path is taken and the simulation ends.

The order of these calls can be better visualized on an event diagram, as shown below. The four columns show the main components. Execution starts at the top left line, and the arrows represent function calls to the other components.

The Dance



The first thing that `verification_top()` does is build the global logging objects. These provide logging to a file and shut down the simulation after a threshold number of errors have been logged. (See `truss_vout.h`.)

Then, after the global logging objects have been created, `verification_top()` allocates the top-level objects. The test is given a pointer to the testbench, so that it can interact with the testbench. It is also given a pointer to the watchdog timer, in case a part of the test wants to force a shutdown or override the `verification_top()` default timeouts. The watchdog is given a pointer to an object so that it can call the final report method with a "watchdog timeout" string prefix.

At this point, all the top-level objects are constructed. As part of their construction they are expected to have established default constraints.

Then `verification_top()` reads the dictionary file (if it exists). This is to allow the test constraints file to override any default settings put there during the construction of the test, the testbench, and their subordinate components.

After initializing the random-number generator, `verification_top()` calls `test->randomize()`. Once the test is randomized, then `testbench->randomize()` is called.

At this point, it is expected that the test and testbench have built their respective subcomponents and are ready to run the test. The first step is the `time_zero_setup()` method, which is used to force wires and initialize interfaces prior to bringing the chip out of reset.

As expected, the next step is `out_of_reset()`, which is used to bring the chip out of its reset state and set it for initialization through the backdoor or register writes.

The next step, `write_to_hardware()`, is where the BFMs are called to initialize the chip. This can be done by either the test, the testbench, or a combination of the two. What is appropriate depends on your situation, as discussed in subsequent sections.

At this point the system is ready for traffic flow. The `start()` method directs the testbench and test to start running. The testbench is started first, to allow monitors and BFMs to start, followed by the watchdog timer. Finally, the test is told to `start()`, which generates the actual traffic.

Next, `verification_top()` calls `wait_for_completion()` on the *testbench*. If your design makes the testbench aware of what checkers are in use, this call waits for the testbench checkers to complete. If not, this method simply returns.

Then `verification_top()` calls the *test*'s `wait_for_completion()`. If your design makes the test aware of what checkers are in use, this call waits for them to complete. (This is the style used in the examples.)

At this point, the test is almost finished. The testbench and test are called to report their final status.[2]

Then `verification_top()` checks to see if any errors were reported. If none were reported, the test is considered to have passed. It may seem weak to accept that the absence of errors is sufficient to consider a test passing. In practice, however, there is no other choice. At the top level, one must trust that the lower-level objects do their jobs. Note that this usually means that in-flight data must be weeded out as the checker proceeds.

Now if the watchdog timer triggers, a different path is taken. The watchdog immediately calls the report method on `verification_top()`. Note that the watchdog itself uses an HDL-based timeout, so that if the report method hangs, the simulation still ends.

The verification_component Abstract Base Class

While the test and the testbench are completely different classes as far as their roles and responsibilities are concerned, their interface to `verification_top()` is the same. For this reason a common class was created. This common class, used as a base for both the test and testbench, is called the `verification_component`.

The `verification_component` is an abstract base class. As such, it provides pure virtual methods for the dance described in the previous section. In addition, `verification_component` provides a constant

2. The authors have tried using the destructor as the final report mechanism. In practice, however, this becomes a difficult part of the design. This is because some destructors try to access deallocated memory or other objects that have already been destroyed. It then becomes tricky to "shut down" the simulation in the correct order, so as not to cause a crash or hang and still get errors printed out. This is one area where verification is different from software, which generally does use destructors as part of the system design.

name and a logger. The interface for `verification_component` is shown below:

```
namespace truss {
  class verification_component {
    public:
      verification_component(const std::string& n);
      virtual ~verification_component();
      virtual void randomize() = 0;
      virtual void time_zero_setup() = 0;
      typedef enum {cold, warm} reset;
      virtual void out_of_reset(reset) = 0;
      virtual void write_to_hardware() = 0;
      virtual void start() = 0;
      virtual void stop() = 0;
      virtual void wait_for_completion() = 0;
      virtual void report(const std::string prefix) = 0;
      const std::string& name;
    protected:
      mutable teal::vout log_;
  };
};
```

Although `verification_component` is a base for the test and the testbench, it is also useful as a base for other objects.

Detailed Responsibilities of the Major Components

The previous sections discussed the roles of the major components and how they were sequenced to run a test scenario. This section dives down a level, discussing in more detail the specifications of the major components. (Because `verification_top()` was discussed in detail in the previous section, it is not discussed further here.)

The testbench class

The `testbench` class has two main responsibilities. One is to isolate the test writers from the actual wire interfaces. The other is to provide "one-stop shopping" for all the generators, checkers, monitors, configuration objects, and BFMs/drivers in the system. The reason to put all of your components into a single object is to facilitate the adaptation of components into multiple tests. In this way, a test writer can see all of the possible "building blocks" that are available.

The `testbench` class can be a passive collection point for all these components, or it can play an active role in bringing the chip out of reset, generating traffic, and knowing when the test is done. In theory, only the global functionality should be handled by the testbench. For example, the testbench probably should bring the entire chip out of reset, while the test can bring separate functionality out of reset. In practice, the test and the testbench share the work.

In general, it is better to let the test or test components control the simulation. This is because a test or test component can then be adapted for several different types of tests.

A more active testbench may, as a counterpoint, simplify a large number of tests in a way that a test base class cannot, because the testbench has direct access to all the chip's wires.

Understand that the more test knowledge a testbench has, the more all tests must act the same or have control over that testbench's functions. This can be good or bad. The specific responsibilities for control and functionality—test or testbench—are, of course, up to the verification team.

As an implementation detail, Truss provides only a `testbench_base` class. What `verification_top()` builds, however, is a `testbench` object. You must provide a `testbench.h`, which declares a `testbench` class. You will probably also have a `testbench.cpp`, which is inherited from `truss::testbench_base`.

Watchdog timer

The watchdog timer component is responsible for providing an "impolite" shutdown if the test has executed for too long. The timer has two timeout mechanisms: one triggers when the watchdog HDL timer triggers, and the other triggers after the first trigger has occurred.[3]

The watchdog timer uses the dictionary to get its timeout values, which are sent to the HDL on `time_zero_setup()`. The `start()` method starts the timers. The HDL watchdog uses an internal timer. If it were to use a passed-in clock, that clock may inadvertently be shut off.

Once either timer triggers, the watchdog HDL timer is notified and a second timer is started. If this timer expires, `$finish` is called. This might happen, for example, if there is some code in the report that is still reading registers, but the chip is unable to respond.[4]

After the watchdog is notified of an HDL timeout, the `report()` method in `verification_top()` is called. This allows the test to report which checkers have completed and which have not, helping to provide a clue as to why the simulation ran too long.

Test class

The test class is responsible for selecting, configuring, and running all the appropriate generators, BFMs, monitors, and checkers. It is also responsible for selecting the configuration of the chip to be used.

While you could directly implement the above responsibilities in the test class, Truss encourages another style. In Truss the test is intended to consist of a number of independent, smaller components called *test components*. These components are the ones that actually do the work; the test's role is to create, constrain, configure, and sequence the com-

3. The watchdog timer is simple in theory, but often hard to execute correctly. To be sure, it must have a clock and a countdown time, but even this basic level can be problematic. Should you use wall clock time, simulation time, or both? Should the HDL timer be internal or external? What resolution should it have? Should the test be able to extend or communicate the expected time of the run?

4. The authors worked on a project where the final report code read the status registers to make sure that functional area of the chip did not have any errors. However, when we added a power-down test irritator, the read hung the system. It took us a while to find the offending code.

ponents, as appropriate for the test at hand. The reasoning behind having multiple independent components is that this is close to the real operation of the chip, where each feature is expected to operate simultaneously. In reality, the chip has common resources that must sequence or arbitrate the use of features. It is in these common resources where the more tricky bugs lurk.

Using this method, the test's direct responsibility is to map the features of the chip (as presented by the testbench's data members) to a set of classes inherited from the `test_component` base class. The test would then add constraints to adapt the test component to the test at hand, as in the following example:

```
class ethernet_basic_packet : public test_base {
  public:
  ethernet_basic_packet(testbench* tb, watchdog* wd) :
    ethernet_data_1(tb->e_generator_1, tb->e_bfm_1,
                    tb->e_checker_1),
    ethernet_data_2(tb->e_generator_2, tb->e_bfm_2,
                    tb->e_checker_2),
    pci_express_1(tb->pci_generator_1, tb->pci_bfm_1,
                    tb->pci_checker_1) {}
  void time_zero_setup(){
    ethernet_data_1.time_zero_setup();
    ethernet_data_2.time_zero_setup();
    pci_express_1.time_zero_setup();
  }
  void out_of_reset(reset r) {
    ethernet_data_1.out_of_reset(r);
    ethernet_data_2.out_of_reset(r);
    pci_express_1.out_of_reset(r);
  }
  void write_to_hardware() {
    ethernet_data_1.write_to_hardware();
    ethernet_data_2.write_to_hardware();
    pci_express_1.write_to_hardware();
  }
   void start(){
    ethernet_data_1.start();
    ethernet_data_2.start();
    pci_express_1.start();
  }
```

```
    void wait_for_completion() {
     ethernet_data_1.wait_for_completion();
     ethernet_data_2.wait_for_completion();
     pci_express_1.wait_for_completion();
    }
    void report(const std::string& prefix) {
     ethernet_data_1.report(prefix);
     ethernet_data_2.report(prefix);
     pci_express_1.report(prefix);
    }
  private:
    ethernet_test_component ethernet_data_1;
    ethernet_test_component ethernet_data_2;
    pci_irritator pci_express_1;
  }
```

In the above example, the `ethernet_basic_packet` test uses three test components, two of which are identical. It connects up the appropriate testbench objects and forwards to every test component the following test calls:

`time_zero_setup()`, `out_of_reset()`, `start()`, `wait_for_completion()`, and `report()`

So why do testing in this more complicated manner? In addition to the previously mentioned idea of simulating close to real-world conditions, an important reason is to maximize the adaptability of the test components. In the example above, we used the same test component for both Ethernet ports. Also, when the test components take in only the parts of the testbench that they need, they (1) make explicit what they are using, and (2) minimize the assumptions on the rest of the chip. This, as will be highlighted in the single UART example in Part IV, allows a test component to be reused for other chips that have only a subset of the original chip's functionality.

Test components are critical to the adaptability of a verification system. In general, the test components themselves do not know whether they are running in parallel with other test components or are part of a series. Thus, the most adaptable components are these test components, as will be discussed further in the following sections.

As an implementation trick, `verification_top()` builds a test by using a `define` called `TEST`. This trickery, set up by the makefile, allows the

`truss` run script to compile in a different test, while leaving the rest of the build image the same for all tests. This allows each test to be its own class (inherited from `test_base`). This cleverness helps one avoid a bad experience in the future. Assume that your team had written on the order of 50 tests, and then a new test was created that required a new subphase to be added to the dance. Although the other tests did not need this new method, you cannot add the default method. This is because all the tests are implemented as a test class. There is only one header `test.h`, and 50 different `test.cpp` files. By defining a base class, and then having the actual test be an inherited class (with a different header file), one can add methods to the base without affecting the existing tests.

There is one more part to a test that needs to be discussed. Often a test is made better by the addition of random background traffic. This traffic, be it register reads and writes, memory accesses, or just the use of other interfaces, can uncover corner cases, such as bus contention, that would not be found otherwise.

These background-traffic test components are called *irritators* and inherit from the `test_component` class. They differ from the standard test component in that they continue their traffic generation until told to stop by the test. Test components, by contrast, decide themselves when they are done, as determined by specified metrics, such as a stop time or the number of packets to send. (Irritators will be describe in more detail later in this chapter.)

With background traffic irritators, the test is written essentially as before. The exception is that the `wait_for_completion()` of the test calls the primary test components' `wait_for_completion()`. When the primary component returns, the test calls `stop_generation()` on all the irritators and waits for them by means of their `wait_for_completion()`. Then the test returns control to `user_main`. (This is explained further in subsequent sections and in the examples in the chapters that follow.)

Test Component and Irritator Classes

As discussed in the previous section, test component-based design is central to a Truss-based test system. The authors have found that separating the test scenarios into test components has maximized the adaptability of the system. By using test components and irritators, test writers have been able to minimize their assumptions and distractions and concentrate on exercising the chip. Furthermore, other test writers can adapt what was done in other functional areas and inherit irritators (if they are not already present) for use as background traffic.

This section describes the responsibilities and interfaces of the `test_component` and `irritator` abstract base classes.

The test component abstract base class

The `test_component` is an abstract base class whose role is to exercise some interface of the chip. As discussed above, this functionality has traditionally been included in the test. The `test_component` describes the interface that all concrete implementations must follow.

In fact, you may have several types of `test_component` for a single interface, for example, a register read/write one, a basic data path one, and an error case one. The fact that these different exercises implement the same interface simplifies reasoning about them.

In practice, most test components use a generator and a wire-level object. Sometimes they may also be given a checker, depending on the designer's intent.

The `test_component` class is not directly a `verification_component`, but it has all the same phases.[5] The `test_component` breaks down some

[5] The primary reason for this is because `verification_component` represents a pattern, while `test_component` is an example of this pattern. The `test_component` has specific implementations of four of the `verification_component` methods. Also, `test_component` introduces some of these same methods as nonvirtual. Finally, the sequencing of the methods is different from the test and testbench, the two top-level components that are verification components. These differences are critical for the integrity of the class.

of the verification_component methods into finer detail, as one would expect of a lower-level object.

Below is the interface for the test_component base class.

```
namespace truss {
  class test_component :
      protected virtual verification_component,
      protected thread {
    public:
      test_component(const std::string& n);
      virtual void time_zero_setup() = 0;
      virtual void out_of_reset(reset) = 0;
      virtual void randomize() = 0;
      virtual void write_to_hardware() = 0;
      void start();
      void stop();
      void wait_for_completion();
      void report(const std::string& prefix);
    protected:
      virtual void start_();
      virtual void run_component_traffic_();
      virtual void start_components_() = 0;
      virtual void generate() = 0;
      virtual void wait_for_completion_() = 0;
      bool completed_;
  };
}
```

The methods time_zero_setup(), out_of_reset(), and write_to_hardware() are provided to allow the test component to interact with a BFM or driver. Note that a different, but equally valid, architecture would keep the wire-layer components private in the testbench and sequence them by means of the top-level dance. This assumes that the testbench knows what subset of the BFMs, drivers, and monitors, to start up.

The start() method is used to start the test_component's generator, BFM, and so on. This method is implemented by a Truss utility class called thread. A thread class runs another virtual method, start_(), in a separate thread or execution. This allows a test class to do the obvious thing and just call start() on all the test components the test uses.

Let's look at the `start_()` method, as it is the main starting point for an interface of the chip. The `start_()` method runs two methods: a `start_components()` pure virtual method, and a virtual `run_component_traffic_()` with a default implementation. The idea behind the `start_components_()` method is that you call `start()` on your generators, BFMs, and so on, as appropriate. (The examples part of this handbook contains examples of `test_component`.)

The default `run_component_traffic_()` method calls `randomize()` (to randomize the test component and its components), and then calls `generate()`. In your `randomize()` method, randomize the data members that will be used by `generate()` to cause some traffic to be generated. In your `generate()`, take these data members and make the appropriate calls to the generators in the testbench.

An AHB example

An example might make the roles a little clearer. (Remember that there are several fully implemented examples in Part IV.) Suppose you are creating a test component to test an AHB[6] arbiter. The test component acts as a master, generating read and write requests to a number of slaves.

The generator in the testbench can generate a burst of reads or writes to a given slave, using a specific burst length. Assume that the generator has a channel interface that can take in an AHB transaction object. The randomize function of your `ahb_test_component` might look like this:

```
void ahb_test_component::randomize() {
  burst_length_ = generate_burst_length(min,max);
  is_read_ = generate_type(min_type, max_type);
  slave_ = generate_slave(min_slave, max_slave);
}
```

The corresponding `generate()` might look like this:

```
void ahm_test_component::generate() {
  //addresses are picked by the generator
  generator_->queue_burst
    (new AHB_transaction (burst_length, is_read_,
```

6. AMBA (Advanced Microcontroller Bus Architecture) high-performance bus.

```
        slave_));
    done_.signal();  //Signals that test_component is done
  }
```

Notice that by nature these calls are executed in a one-shot manner. That is, together they perform a single transaction. This is useful to allow an `irritator` to inherit from this test component later, to sequence this pattern any number of times and possibly change the randomization constraints as well.

So why have two separate methods?

By separating the randomization from the generation phases, one can inherit different classes that either (1) have different randomization characteristics (for example, logarithmic distributions of the burst length, or a pattern); or (2) send the data through a filter first, then to the generator.

So now that the transaction has been generated, what should the `wait_for_completion()` method do? Because the generation is occurring in another thread, there should be a condition variable to communicate when it is done.

So the code might look like this:

```
void AHB_test_component::wait_for_completion_() {
  done_.wait();
}
```

Test-component housekeeping functionality

The `test_component` class also provides a basic housekeeping boolean that tracks when you return from the `wait_for_completion_()` method. This allows the `report()` method to determine whether you have considered the work of the component to have been completed or not. This can be very useful in a timeout situation, to see which components have not completed.

What you decide to do in the `wait_for_completion_()` depends on how you view your `test_component`. One view is that it is a traffic generator only, which can complete when the generation of traffic has been queued. It is then up to the testbench or test to determine when the chip has processed all the data. This will most likely involve a checker or monitor.

Another view is that your `test_component` represents a generate and check path through the chip. In this case, the completion of `test_component` signifies the completion of the entire exercise. (The examples in this handbook use this view.)

As always, the team must decide which view is better for their project.

The irritator abstract base class

As discussed above, the `test_component` is set up as a one-shot traffic generator. This works for tests that are directed, and for tests where the completion event is predetermined—that is, tests that know before the `start()` call what the end conditions are.

However, sometimes it is not good design to have the `test_component` determine when completion is achieved. This is the case when, for example, you want to achieve a certain metric, and the measurement is not appropriate information for the `test_component`.

For example, you may want to send 100 bursts of some AHB traffic. While this could be included in the `ahb_test_component`, you might not want to measure completion by 100 bursts all the time. Instead, you might want to write a test that looks at the number of hits each slave device gets, and stop the test when all slave devices have been targeted. As another alternative, you might want a test to run until some coverage occurs, which could be any of the previous scenarios, or could involve some internal state in the arbiter.

The `irritator`, inherited from `test_component`, is used for situations such as these. The interface is shown below.

```
namespace truss {
  class irritator : public virtual test_component {
  public:
     irritator(const std::string& n);
     virtual ~irritator() {}
     void stop_generation() {generate_ = false;}
  protected:
    virtual void start_();
    virtual void run_traffic_();
```

```
    virtual bool continue_generation();
    virtual void inter_generate_gap() = 0;
    bool generate_;
  };
};
```

The irritator overrides the `run_traffic_()` method of the `test_component` base class. It sets up a loop, calling the one-shot randomization and generation in the `test_component`'s `run_traffic_()` methods. The implementation is shown below.

```
virtual void truss::irritator::run_traffic_() {
  while (continue_generation()) {
    test_component::run_component_traffic_();
    intergenerate_gap();
  }
}
```

The method `continue_generation()` just looks at a boolean, which is toggled to `false` by a call to the `stop_generation()` method. This allows an external class to stop the continual loop of randomization and generation.

Note that there is a new virtual method in the `irritator` class, called `intergenerate_gap()`. Because the irritator is continually generating traffic, you might need a delay mechanism to prevent the generator from flooding the chip.

There are many ways to get this delay. For example, in one solution the generator and attached BFM/driver could execute the generate request as soon as it is called and thus take simulation time. In another solution, the way to get a delay would be to have a fixed-depth generator and BFM/driver channel.[7] This would put back-pressure on this generate loop. In still another solution, the generator could have a delay in clock cycles before returning.

Any of the above solutions is acceptable, but there is yet another choice. That option is to have the irritator itself provide the delay mechanism.

7. This method is supported in Truss's `channel` class.

The `intergenerate_gap()` is a virtual method allowing you to implement an irritator-based delay. This allows the irritator to decide on the throttle mechanism. Different subclasses could implement different policies. For example, an irritator could wait for a variable number of clock cycles. Another example would be to measure some parameter on the checker (such as packets in flight).

As always, the team must decide what is appropriate.

Using the irritator

The irritator continues this generate/wait loop until a `stop_generation()` is called. But how do you decide when to stop the irritator? The answer, of course, is "When the test reaches its goal." One goal could be that the "main reason" for the test has been achieved. For example, you can have the main goal be a test component, perhaps one that generates a fixed, but randomized, number of packets through a particular chip interface. The global goal in this case would be for the test component to achieve completion. Here is how the test code might look:

```
void noisy_packet_test::wait_for_completion() {
  //assume the data members include
     base_packet_exerciser,
  //the test component of interest and some std
     container
  //class with a list of irritators.
  basic_packet_exerciser_->wait_for_completion();
  std::for_each(irritators_.begin(), irritators_.end(),
                stop_generation());
  std::for_each(irritators_.begin(), irritators_.end(),
                wait_for_competion());
}
```

Ignoring the nontrivial constraining, selecting, and creating of the test component and irritators, what is accomplished in a few lines of code is a shutdown sequence that is powerful, while being a fairly simple idiom.

Note that a verification team could decide to use only irritators in their implementation. In that way, when to stop the test can then be determined by looking either at a checker or possibly at elapsed simulation time.

The complex part of the test would then become the randomization and selection of irritators. The authors have worked on a variant of this methodology, and the resulting verified chip was a first silicon success.

Summary

This chapter introduced Truss, an open-source application framework.

We revisited the benefits of an OOP language such as C++, but stressed the need to keep things simple despite the power of this language, to avoid writing code that is unnecessarily complicated.

We talked about the key algorithm of verification, which the authors called the "dance." We showed how the dance is used by the `verification_top()` program to run a test. We discussed the roles and responsibilities of the test, testbench, and watchdog timer, the main parts of the top-level dance.

We discussed the `verification_component` abstract base class, which provides pure virtual methods for the dance.

We then discussed the `test_component` and `irritator` classes, including their responsibilities and interfaces.

Truss Flow

CHAPTER 7

Expensive solutions to all kinds of problems are often signs of mediocrity.

Ingvar Kamprad, founder of IKEA

Have you ever bought and assembled a piece of furniture from IKEA? In the store most of their furniture looks very simple, but when you get it home and try to assemble it, you realize that it's built from several smaller and often confusing pieces. Even with IKEA's famous assembly instructions, showing the "intent" for each piece graphically, assembly can still be confusing. Imagine how hard it would be without instructions.

The authors have had to learn many verification environments through the years, and this has often been a very confusing experience. What seems like a great concept with a well-defined structure at a high level of abstraction is often obscured by troublesome details when you first try to implement it. Many times the confusion is increased because of a lack of description regarding how the high-level ideas are actually implemented. To help reduce the confusion around Truss, this chapter describes the *"dance"* in more detail.

Overview

This chapter looks at how the "dance" described in the preceding chapter is actually implemented. It shows the order in which each method is called, and describes the files to find the method, or its base. The chapter then looks at the structure for the major components of Truss.

First to be described is `verification_top()`, the first function called in Truss and the base of the "dance." Following this is a description of the methods, and their class, through which files are called for each step.

Then the test component is described. This component follows a dance similar to that of `verification_top()`, but for a different set of classes and files.

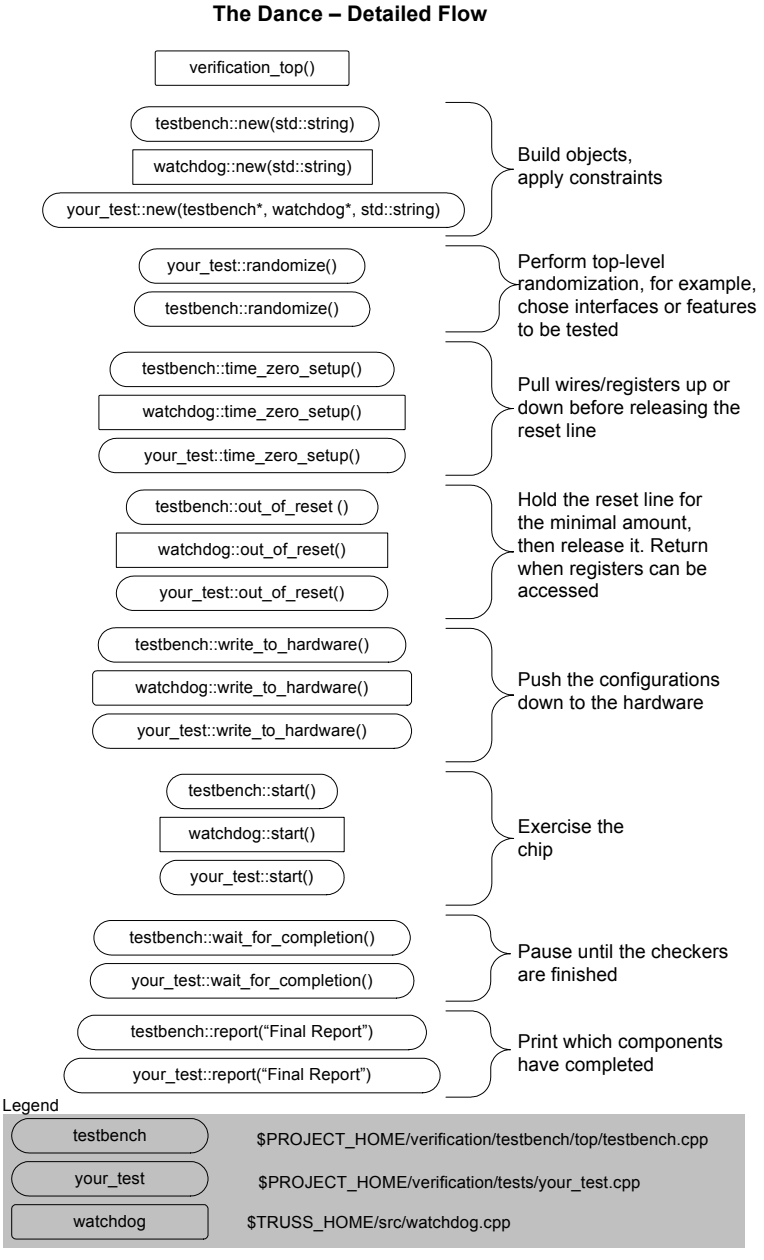
The `irritator` class is described next. While similar to a `test_component`, irritators have some unique method calls worth pointing out.

The last part of the chapter talks about steps that need to be taken to build a new Truss project, by taking the more-abstract description of classes and applying them to the first few tests in a new project.

About verification_top.cpp

When the simulator executes the `$teal_top` call in the HDL, control is passed to the `verification_top()` function in `verification_top.cpp` under the *truss* directory. In this handbook we refer to this function as the "dance," or top function. It is this function that interacts with your top-level components: the test, the testbench, and the watchdog timer.

Let's look at the dance with respect to the methods you have to write. This is illustrated in the figure on the following page. A square box indicates that the method has a default implementation, and a rounded box indicates it needs to be defined for your project.


The Dance – Detailed Flow
verification_top()
testbench::new(std::string)
watchdog::new(std::string)
your_test::new(testbench*, watchdog*, std::string)
Build objects, apply constraints
your_test::randomize()
testbench::randomize()
Perform top-level randomization, for example, chose interfaces or features to be tested
testbench::time_zero_setup()
watchdog::time_zero_setup()
your_test::time_zero_setup()
Pull wires/registers up or down before releasing the reset line
testbench::out_of_reset ()
watchdog::out_of_reset()
your_test::out_of_reset()
Hold the reset line for the minimal amount, then release it. Return when registers can be accessed
testbench::write_to_hardware()
watchdog::write_to_hardware()
your_test::write_to_hardware()
Push the configurations down to the hardware
testbench::start()
watchdog::start()
your_test::start()
Exercise the chip
testbench::wait_for_completion()
your_test::wait_for_completion()
Pause until the checkers are finished
testbench::report("Final Report")
your_test::report("Final Report")
Print which components have completed
Legend
testbench
$PROJECT_HOME/verification/testbench/top/testbench.cpp
your_test
$PROJECT_HOME/verification/tests/your_test.cpp
watchdog
$TRUSS_HOME/src/watchdog.cpp

The `watchdog` class is already written and should be sufficient for most purposes. (We will not discuss the watchdog timer's methods, because they are relatively straightforward.) You'll have to write the test and testbench classes.

In the `testbench` constructor, instantiate your generators, checkers, BFMs, and so on. (This assumes that your team has decided to put these interface objects in the testbench rather than in the test components.) Then add your constraints by using the *dictionary*. These constraints will be picked up by your generators and configuration objects to guide the randomization. Initially, you will probably have no constraints.

The test's constructor will create all the test components and irritators that it needs.

In the `testbench::randomize()` method, randomize your local variables and then call `randomize()` on lower-level components, as appropriate. Your testbench may have configuration objects for each interface or feature that is used to configure the chip.

The `test::randomize()` method is similar, in that the test randomizes each `test_component` it owns. In addition, the test may select some subset of all the components and irritators it owns.

The `testbench::time_zero_setup()` method is where you drive wires prior to letting the chip out of reset. You may need to wait for the PLL to lock, or set up "sensor" pins on the chip in this method.

The `test::time_zero_setup()` usually just calls all the active test component's `time_zero_setup()`. This is to allow test components that have a "plug-in" behavior, such as USB and PCI Express, to perform their initial training. (To use this method is a judgment call, as you may want to bring up an interface later in the simulation.)

The `testbench::out_of_reset()` will bring the chip to a stable state that can accept register access. If the team so decides, you could use `test::out_of_reset()` to reset the chip.

The `write_to_hardware()` methods in both the test and the testbench are where you perform register writes to move your selected configurations to the chip. The test's `write_to_hardware()` method usually just calls the same named method on all its test components. This is because the actual register writes will occur in the BFM or driver. One exception

is when you are writing a direct test, and it's easier just to write the registers at the test level.

The `testbench::start()` method, if it knows which interfaces and features are in use, starts up all the BFMs, monitors, and drivers. Depending on your architecture, it may also start the generators and checkers.

The `test::start()` method usually just calls the `start()` method on all its owned test components.

The `wait_for_completion()` methods in the test and testbench are used to pause the verification system until the test is finished. Although there are many ways to do this, the examples in this handbook just allow the checkers to say when the test is completed.

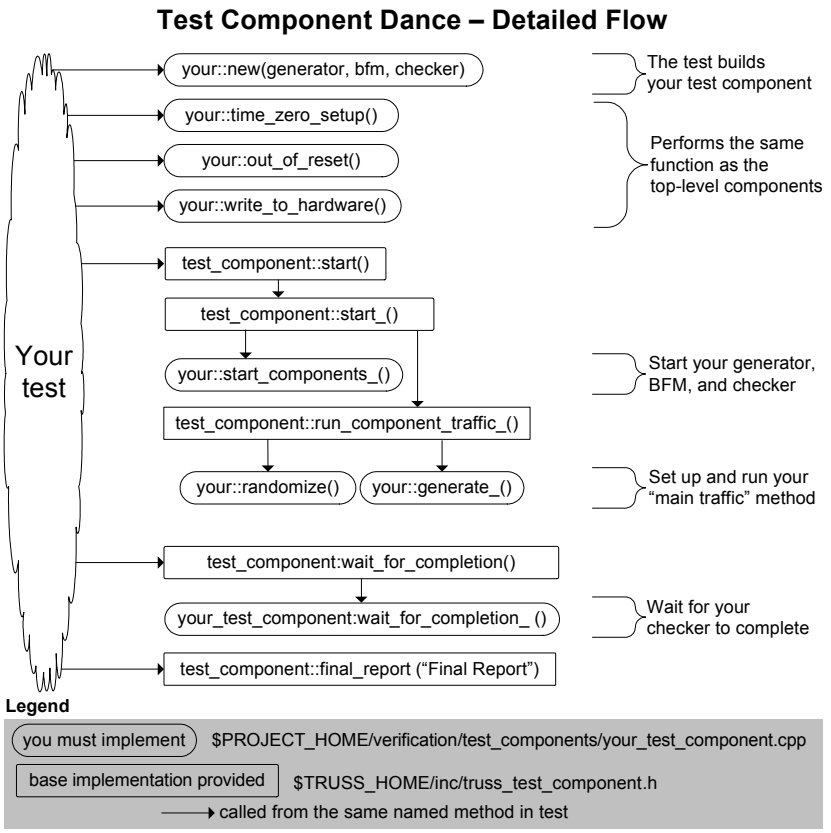
The `report()` method in both top-level objects reports their status. For the testbench, it is usually appropriate to report the configurations selected. For the test, it usually just calls the test components.

That's it. This may seem like a lot of methods to write, but you probably do not need to perform tasks in all the methods. Later in this chapter, we will talk about the order in which you might want to implement these methods.

The Test Component Dance

Did you notice that most of the time the test just called the same named methods on the test component? That's because verification has a fractal structure, with repeated patterns. The top-level dance is repeated, with a few changes, in the test. This time, instead of `verification_top()` calling the steps, the test does. The `test_component` also plays a role, subdividing the `start()` method into several lower-level methods, as shown in the following figure.

The `run_component_traffic_()` method has a standard implementation, which calls `randomize()` and then `generate_()`. The `randomize()` method has the same purpose it had for the top-level components: to randomize your random variables. The next method called,



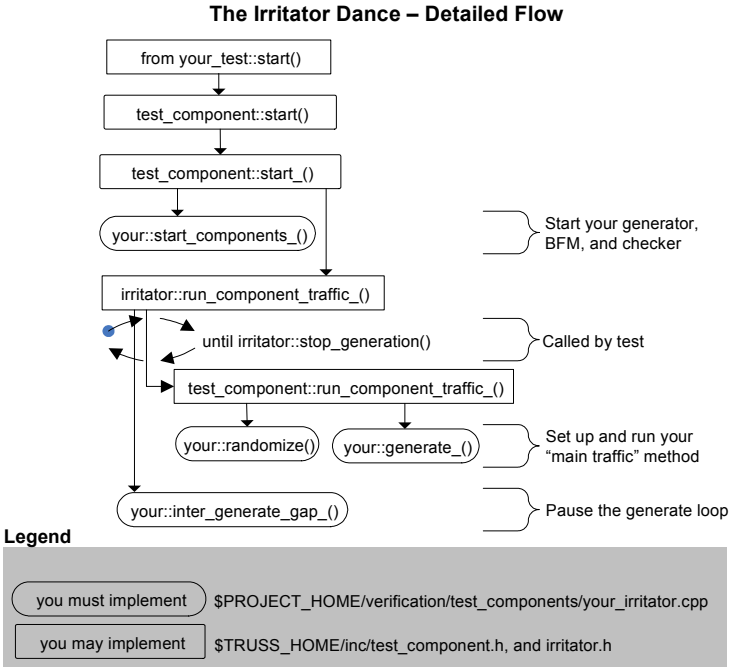
`generate_()` picks up the results of the randomization and interacts with the generator to exercise a feature or an interface of the chip.

Now, it may seem strange that these methods are implemented like this. However, the idea is to separate the various concerns of the test component: starting, randomization, and generation. This, as will be discussed in Part III of the book, creates more adaptable and less brittle code. The organization also sets up the *irritator*, making the transition from a fixed test to an irritator relatively painless.

The Irritator Dance

The `irritator` is an inherited class of `test_component`. Its purpose is to generate background "noise" while the test concentrates on some specific area of the chip. In some sense, using irritators is a way to emulate the real world, where many of a chip's features and interfaces are used simultaneously.

So what does an `irritator` add to or change from the `test_component`? Only one method is changed, and two methods are added. All these changes involve the new `run_traffic()` method, shown in the figure below.



The `irritator` overrides the `run_component_traffic_()` method from the `test_component` base, and calls the base class `run_component_traffic_()` method in a loop. This is the nature of an `irritator`: it just keeps on going until told to stop. The method that stops the loop is `stop_generation()`, which is usually called by your

test once the main feature or interface has finished being tested. This will be shown in detail in the Part IV of this handbook.

One method you will have to implement is `inter_generate_gap_()`. This method may be empty, for a couple of reasons.

- Your channel has a limited depth, and this limit is used to apply back-pressure to your system.
- Your generator has a built-in delay of some form.

In this handbook we use the checker to throttle the system—because we want to keep a certain amount of data in flight, and the checker is the only agent that knows what has been generated and what has been received. (The chip can handle an unlimited number of back-to-back transactions.)

That's all there is to building an irritator. Note that you will probably start with a test component, and then evolve it into an irritator. It will probably be many weeks into your project before the first irritator is built, but for coverage and finding congestion bugs, irritators are a good choice.

In fact, your first test will probably be even more rudimentary. This first test is the focus of the next section.

Compiling and Running Tests

The sections above described the main building blocks of Truss. The following chapters, as well as later examples, will show how these still somewhat abstract concepts get implemented for real projects. However, before we start looking at more concrete examples, there is one more problem to consider: that of compiling and running a verification environment.

All verification environments need some type of run script to compile and build both the RTL and verification code. In a large project this is not a simple task, because one must track a lot of code, as well as many tools and options.

A goal for Truss is to provide a production-grade run script and makefiles as open-source components. At the moment, a reasonable run script and

rudimentary makefiles are provided. They are a good starting point for a run script and provide enough functionality to handle the examples in this handbook. It is the authors' hope that through community effort, these scripts can be fleshed out into something better.

Truss run script

The Truss run script controls which files are compiled and run. It is written in Perl and has a number of switches that controls its actions. The script will first compile all the C++ files, then compile all the HDL files, then link all files into a single executable, and finally launch the simulation. After the simulation finishes, it checks the status of the test run. (This script is used to build and run all the examples on the companion CD.) The script is written in Perl, and can be located at $TRUSS_HOME/bin/truss).

Truss uses some environment variables to "understand" its environment. By using environment variables (instead of `.tool_rc` files, for example), the system's assumptions are both obvious and flexible.Truss uses only a small number of environment variables, as listed below.

Variable	Function
SIM	Simulator name (such as ncsim, mti, aldec, or vcs)
SIMULATOR_HOME	Path to the simulator install area
TEAL_HOME	Path to Teal's source files
TRUSS_HOME	Path to Truss install area
PROJECT_HOME	Path to top of the current verification project

The file named `setup` in each of the `bin` subdirectories of each example on the CD has default values for the TEAL_HOME, TRUSS_HOME, and PROJECT_HOME environment variables. You'll need to set SIM and SIMULATOR_HOME as appropriate for your environment.

Switches

The Truss run script has a number of switches to control its execution. Below is a table that expands on descriptions of the most important switches.

Switch	Function
--help	Prints longer help message
--test <test_name>	Runs the $PROJECT_HOME/testcases/<test_name> test.
--clean [options]	Cleans appropriate selection of the system. Default selection is USER. The following options are available: LOGS - Deletes simulation log files CPP - Deletes user-compiled C++ code HDL - Deletes user-compiled HDL code USER - Deletes all user-generated code (LOGS, CPP, HDL) TRUSS - Deletes compiled Truss files TEAL - Deletes compiled Teal files ALL - Deletes all of the above This switch can be repeated (--clean CPP --clean HDL)
--simulator <SIM>	Selects appropriate simulator from supported list. If switch is not used, then run script reads $SIM. If neither $SIM or --simulator is used script will fail.
--seed <seed value>	Sets random seed to integer <seed value>
--run <number>	Runs the selected test a number of times
--sim <sim>	Builds and runs using <sim> as the simulator.

For a full description of all switches from a command line, run the following:

```
$TRUSS_HOME/bin/truss --help
```

The Truss makefile

As is customary in the coding world, a makefile is used to build the objects and archives.[1] The Truss makefile may be a good starting point for your makefiles. Almost all of the directories in the examples include a standard makefile, located in the `/inc/Makefile` subdirectory of Truss. As with the `truss` script, this makefile is both too simple and too complex.

The makefile for three sources is shown below.

```
STATIC_LIB = $(LIB)/directory_name.$(SIM).a
INC = -I../a_referenced_directory
SRCS = \
    $(SRC)/file_one.cpp \
    $(SRC)/file_two.cpp \
    $(SRC)/file_three.cpp

include $(TRUSS_HOME)/inc/Makefile
```

The first line identifies the output static library name. The next three lines identify the source files. The last line includes the standard makefile. Most makefiles follow this form.

The Truss makefile has all the compiler switches to build the sources for a variety of simulators.

The First Test: A Directed Test

Because starting something new is not always easy, this section helps make the process easier by addressing how a first test can be written in Truss. This section concentrates on the steps you need to do, and how a test can be built up from scratch. The next chapter shows a complete first example and focuses more on the flow.

Your first test will probably be a simple directed test, with a `test_component` that does not have a generator and possibly not even a checker. It will probably interact directly with the BFM or driver.

[1] The final shared object is built by the `truss` script.

Focus your initial efforts on the driver and BFM. Write a "first cut" at the driver class, making it have the methods that seem right to you. You may or may not need a monitor, depending on the protocol or feature to be tested.

Next, create a testbench that includes that driver/BFM and think about how to get clocks to the chip and get it out of reset.

Now make a test class and get the whole thing compiling. Before moving on to connecting the test to the driver with a test component, make sure the chip is cleanly out of reset, as this can be done by the testbench's `out_of_reset()` method.

The next step is to make a simple `test_component`. This component will probably just be a directed exercise, with perhaps a few reads and writes or just a few calls to the driver. Note that you may use the `test_component` pre-implemented methods if you are comfortable with them, but for a first test it might be better just to override the `start()` method directly. This is because that's easier than remembering where to put your randomization and traffic-generation code.

If there is any configuration, use the chip's default configuration. Don't try to randomize anything yet.

Doing the checking can be tricky, so let's worry about that last. We'll probably be looking at waveforms for the first few days anyway.

Now build a test that has your `test_component` as a data member. Initially, have the test call the same named methods on your test component.

Note that the `wait_for_completion()` method probably just returns, if you implemented the `start()` method. However, if you used the `generate_()` method of the standard `test_component`, you'll want to trigger a `condition` variable at the end of your `generate()`. Then, the `wait_for_completion()` would just wait for the signal to be triggered, as shown below:

```
class your_test_component {
  //...your other code here...
  private:
  teal::condition done_;
}
```

Then, in the last line of the `your_test_component::generate_()` method, do this:

```
void your_test_component::generate_()
{
  //...your directed exercise code here...
  done_.signal();
}
```

Then your `wait_for_completion_()` would look like this:

```
void your_test_component::wait_for_completion_()
{
  done_.wait ();
}
```

That's it! You have created your first Truss-based test.

The Second Test: Adding Channels and Random Parameters

Software engineers count "one," "two,"—and then "many." This is because only the first three times they use a technique are significant. After that, everything looks like "many." By writing the first test, we've counted "one." Now we will count "two." The next section will cover the "many."

In this, the second test, we'll get more sophisticated. We'll add the agent layers and also add the generator and checker. These are the steps you need in order to create more advanced, randomized tests. You will probably create several directed tests before you need these additional features, but because this is a book we need to keep moving along.

Remember that the generator and monitor generally have pure virtual methods to communicate the results of their work. We'll add our agents to these methods. There will be an agent for the generator, the driver/BFM, the monitor, and the checker. Why all this complexity? Because there are many interconnection techniques, each one involving some

architectural trade-offs. These trade-offs are talked about at length in the OOP Connections chapter in Part III of this handbook.

To make the connection between the agents, we'll use a Truss channel. So let's digress a bit and look at a channel.

The channel classes

Verification systems have a lot of producer/consumer relationships. For example, a generator can be considered a producer and a BFM considered a consumer. However, it is a good idea to minimize the knowledge and assumptions of the interface between these two loosely cooperating objects. One way to decrease the coupling between these components is to use an *intermediary object*. An intermediary object would allow the two communicating objects to be anonymous or separated in time. The concept behind this object is called a pipe, mailbox, or channel. Truss uses the term *channel*.

Truss separates the roles of producer and consumer by having two abstract base classes, `channel_get` and `channel_put`. This clarifies the roles of the two communicating objects. For example, the constructor of a generator would take in a `channel_put` class, because it puts data into a channel. A monitor's constructor would also take in a `channel_put` class. A BFM or checker's constructor, on the other hand, would take in a `channel_get` class.

In Truss, the `channel_get` and `channel_put` classes are templated. This is one of the very few places where Truss uses templating. Don't worry if you don't understand all the details of the code in a first read-through; these templates are not hard to use, as will be shown in the chapter discussing a single UART example.

The reason we use a template is to encourage a strongly typed channel. Another reason is to allow you to have the choice of using pointers or actual objects and data in the channel. In general, the authors use pointers to objects only when virtual methods are needed. Otherwise, we put the objects themselves in the channel. This simplifies any memory-management issues.

The `channel` class joins the two concepts `channel_put` and `channel_get`. This class adds the storage for the actual data, as well as

the signaling and mutual-exclusion mechanisms. In addition, `channel` also supports a `depth` concept, for designs that want to implement back-pressure in that way. The interface for a `channel` class, as well as the base classes, are in `/truss/inc/truss_channel.h` on the CD.

The `channel` class also provides for other `channel_put` objects to be attached to a channel. This allows the data of one `put()` to be replicated across many channels. The common use for this is when a generator creates a data item and both the checker and BFM should get the data. It is also useful if there are multiple listeners to a channel, such as in an Ethernet broadcast, or multiple monitors for a data interface.

Building the second test

Now that we have channels, let's use them for the agents. This section is a bit high level, because every situation is different. We'll give general direction, but after you read this chapter, take a look at the next chapter for a first complete example.

Let's say that you are working on a chip interface called `my_interface`. You might have a generator that looks like this:

```
namespace my_interface {
  class my_data;
  class generator {
    public:
      void generate(); //make one
      virtual void post_generate_(const my_data&) = 0;
  }
}
```

We are concerned with the `post_generate_()` method. This is a pure virtual method, so we must implement it in our inherited class. Let's assume we want to add a channel interface, like so:

```
#include "generator.h"
namespace my_interface {
  typedef channel_put <my_data> generated_channel;
  class generator_agent: public generator {
    public:
      generator_agent(generated_channel* out)
                                          : out(out){}
```

```
      virtual void post_generate_(const my_data& d) {
        out_->put(d);
      }
  private:
    generated_channel* out_
  };
};
```

By building a `generator_agent`, we have abstracted how the generator gets the created data to the driver/BFM.

A similar situation exists in the monitor:

```
namespace my_interface {
  class results;
  class monitor: public truss::thread {
    public:
      void start();
      //the connection method
      virtual void data_received_(const results&) = 0;
  };
};
```

And likewise for an agent for the monitor:

```
#include "monitor.h"
namespace my_interface {
  typedef channel_put <results> generated_channel;
  class monitor_agent : public monitor {
    public:
      monitor_agent(out_channel* out) : out_(out) {}
      virtual data_received_(const results&) {
        out_->put(r);
      }
    private:
      out_channel* out_;
  };
};
```

But what about the other side of the channels? These objects are the `driver_agent` and `checker_agent`, respectively. Their job is to take the data out of a channel and act on the data.

Remember, we are discussing channels here because that's how we wanted to implement the agent layer. This could have easily been a more generic producer/consumer model, or an event-driven method, but implement what feels correct for you. (All the examples in this handbook use channels.)

Here are the classes for the driver and checker and the inherited classes for their agents:

```
namespace my_interface {
  class driver {
    public:
      void send_data(const my_data&);
  }

  typedef channel_get <my_data> input;

  class driver_agent : public driver,
                       public truss::thread {
    public:
      driver_agent(input* drain) : drain_(drain) {}
      //must have a start to drain the channel
      void start_() {
        for (;;) {
          send_data(drain_->get());
        }
      }
    private:
      input* drain_;
  };

  class checker {
    public:
      check_data(const my_data&, const results&);
  };

  typedef channel_get <results> checker_in;
  class checker_agent : public checker,
                        public truss::thread {
    public:
      checker_agent(generated_channel* generated,
             checker_in* actual) :
             generated_ (generated), actual_(actual) {}
```

```
      void start_() {
        //Check the data!
        for (;;)
          check_data(generated_->get(),
                     checker_in->get());
      }
    private:
      generated_channel* generated_;
      checker_in* checker_in_;
  };
};
```

The authors realize that there is a lot of code to look at, but just skim it over to get the general idea. The general technique is to inherit a class, add a channel, and append `_agent` to the name.

After the agents have been built, they should be added to the testbench. The testbench holds the generators, drivers, monitors, and so on. The test, on the other hand, holds the test components.

Building the second test's test_component

The `test_component` is relatively straightforward. A `test_component` constructor takes in the parts of the testbench you need. Remember, the entire testbench is not taken as a parameter, because then we would have to make assumptions about the name of the parts we needed. Also, by taking in only the parts we need, several of our test components can be used in the same chip.

The most likely candidates for the constructor's parameters are the generator, the driver, and the checker.

The rest of the test component usually just forwards its calls to the appropriate objects. An example test component is shown below.

```
namespace an_interface {
  typedef class bfm;
  typedef class generator;
  typedef class checker;
};

#include "truss.h"
```

```
namespace an_interface {
class a_test_component : public truss::test_component {
  public:
    a_test_component(const std::string n,generator* g,
                     bfm* b, checker* c);

    virtual void time_zero_setup() {
     bfm_>time_zero_setup();};
    };
    virtual void out_of_reset(reset r) {
      bfm_—>out_of_reset(r)};
    };
    virtual void randomize() {/* next section */;}
    virtual void write_to_hardware() {
      bfm_—>write_to_hardware();
    };
  protected:
    virtual void generate() {generator_—>generator();};
    virtual void wait_for_completion_() {
      checker_—>wait_for_completion();
    };
    virtual void start_components_() {
      bfm_—>start(); checker_—>start();
    };
  private:
    generator* generator_;
    bfm*       bfm_;
    checker*   checker_;
  };
};
```

Although your actual test component will be a bit different from the code above, the general form will probably be the same.

Adjusting the second test's parameters

As soon as you introduce randomization into a test, you'll probably want some knobs to control the randomization. Sweeping most parameters through an entire integer range would chew up a whole lot of simulation time. Besides, it's probably either (1) not interesting, or (2) unacceptable to the register associated with the integer.

A knob is a technique that uses other variables to control the range of a random variable, either directly or indirectly. In this example we'll concentrate on controlling the random variables directly. (The examples in the handbook use the Teal dictionary feature to pass parameters from a number of sources to the method that will use the knob variables.)

For example, consider a test for a CPU. Assume that a `cpu_generator` class has a `send_one_operation()` method that is called by a `test_component` to tell the `cpu_generator` to create one random operation. The generator is guided by dictionary variables. It is best to put the variables to randomize in a separate function at the top of the source file, because the seeding depends on line number. That way, the sequence of values selected does not change if the code below is reorganized. Of course, new random values chosen will be different for each master seed.

Here is an example function for generating the `operand_a` variable of a CPU operation:

```
namespace {
  uint32 get_operand_a(uint32 min_v, uint32 max_v) {
    uint32 returned; RAND_RANGE(returned, min_v,
       max_v); return returned;
  }
};
```

In the `cpu_generator`, the following lines could be used:

```
static uint32 min_operand_a =
  dictionary::find(name_ +     "_min_operand_a", 0);
static uint32 max_operand_a =
  dictionary::find(name_     + "_max_operand_a", ~0);
operand_a = get_operand_a(min_operand_a,max_operand_a);
```

This same style is used for the other operand and the operator variables.

So who sets the knobs? There are four ways: (1) use the default specified in the `dictionary::find()` call as the second parameter; (2) put the knob value on the command line; (3) use a knob configuration file; or (4) (finally) write code to use the `dictionary::put()` call, which is the mechanism used in our example. Note that because the Teal dictionary is used, both the command line and the knob file can be added later without the need to modify any of the example code.

The test constrains the test component with respect to the number of times the generator is called. Of course, this specifies the number of operations sent to the arithmetic logic unit (ALU). The code is shown below.

```
teal::dictionary::put(test_component_->name +
    "_min_operations", "4",
       teal::dictionary::default_only);
    teal::dictionary::put(test_component_->name +
    "_max_operations", "10",
    teal::dictionary::default_only);
```

Note that the name of the `test_component` is used. This allows the test to pick any name for the `test_component` and still have the code work.

However, be careful with the spelling of the knob variables. They must be spelled the same in both the `find` and the `put` routines in order to make a connection.

Now that the randomization and knobs are connected, we have completed writing the second test. In some ways, this test is rather sophisticated. It uses the Truss framework, and adds agents by using channels to connect the wire-layer classes to the transaction-layer classes.

The testbench created and wired up the generator, driver, monitor, and checker. The testbench can bring the chip out of reset and start the monitor.

The test itself is rather reasonable. It creates and connects the test component to the generator, driver, and checker in the testbench.

The Remaining Tests: Mix-and-Match Test Components

So now what do you do after creating this second, more-sophisticated test? You do what we verification engineers always do—create more tests! As these tests are being written, new test components will also be created, some of which could be used in several tests. Deciding which test components to adapt to different tests is the major activity (besides writing more tests) after you have written the first two tests. This is the "many" count that we talked about earlier.

Of course, you'll be doing other test-related activities, such as adding randomness to the existing tests and looking over your verification test plan to make sure you know when you're done.

And how do you go about adapting a test component from one test into another? You could just put the new test component in the test and wait until both of them are completed. However, as explained in the Truss Basics chapter, there is another way: use the Truss concept of irritators, and warm over, or "recrystallize," the existing test component to an irritator.

Converting the test components to irritators usually just involves deriving the existing test component with the `truss::irritator` component. Then, the appropriate methods will be overridden and the only method you have to write is `inter_generate_gap_()`. There are many ways to implement a gap, from the simplest (pausing a number of clock cycles), to the more complex (using back-pressure and bursty traffic). If the checker were inherited from Truss's checker, you can also just wait for generated data to be checked.

This process of writing a new test continues for all the rest of the features and interfaces of the chip. Remember, the more irritators a test has, the more likely it is to model what actually happens when the chip design is realized in silicon.

Summary

This chapter tried to clear the fog of how to go about using Truss. We started with a review of the top-level dance, and then showed that the dance also existed in other layers of the system.

We looked at the tools provided by Truss, which are the `truss` execution script and a standard makefile.

We covered writing the first test, concluding that it will probably be a directed test. Then, we took the test up a notch, adding connection agents to the generator, driver, monitor, and checker. We introduced the Truss channel as the interconnect technique, but noted that there are many other techniques.

We looked a bit at control knobs, a technique for passing parameters to constrain randomization. (There are many techniques for constraining random-variable generation.) This chapter showed how to harness Teal's dictionary to hook up bounds for randomization.

We finally discussed what to do after the second test. The idea is to write more tests for that interface or feature, and also test the rest of the chip. The key part of writing more tests is to keep an eye out for what you can "steal" (rather, "adapt") for other tests. By creating irritators, you can use the functionality of other tests as background activities. In this way, the chip is stressed more—and more faults are found prior to production.

Truss Example

CHAPTER 8

I know that you believe you understand what you think I said, but I'm not sure you realize that what you heard is not what I meant.

Robert McCloskey

Coding is tricky, because we take the great ideas, techniques, and trade-offs and actually make decisions. We put fingers to the keyboard, and decisions are made and trade-offs are fixed in code. Furthermore, learning a new technique only makes the coding task more difficult. An example, or several examples, can help put the technique into perspective.

This chapter is the first example of how to use Teal and Truss in a verification system. It's useful to build and run some example code when learning something new. So, install the code on the CD and noodle around with it a bit. You can add printf's and change the code a bit.

If you want, use this chapter as a guide to some of the more interesting parts. This chapter is not quite a map to the "homes of the movie stars." Instead, it is more like a mariner's map. It helps you to navigate in tricky waters.

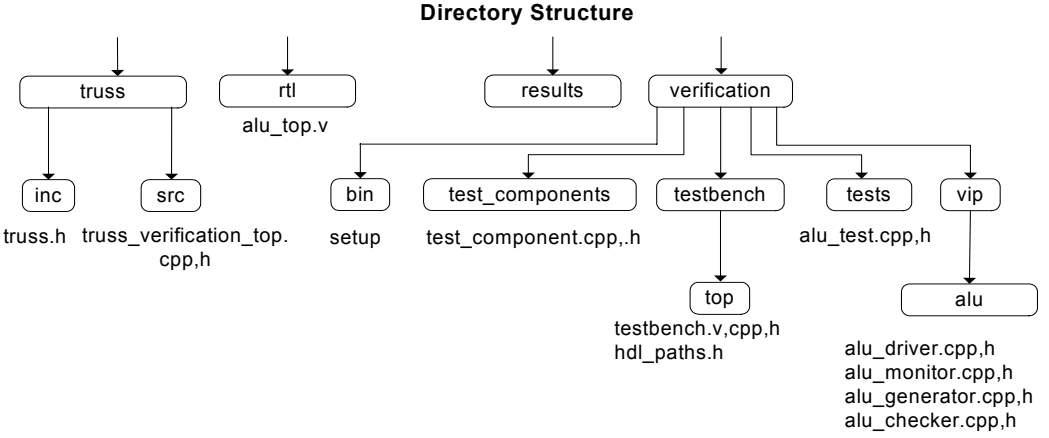
Overview

This chapter provides a first complete example of using Truss, where you can actually compile and run the code. The code is not as complex as what you would encounter in a fully featured chip. However, all the main parts are here to consider. The source files may seem silly or overly complex for the chip we are trying to test, but we are trying to demonstrate how to structure a verification system for a real project. Your chips will have plenty of complexity to manage.

This chapter does not walk through every code file. We are all capable of reading code. What it does instead is look at some of the more important aspects of the verification system.

Directory Structure

In order to help you navigate the source files, it's good to show the main directories that comprise a Truss-based system (shown below). We've also included only the main files we will be working with.



The source code for the chip is in the /rtl directory. How does the truss run script know this? The file /verification/testbench/top/hdl_paths.vc is used to specify the paths to the RTL and the RTL include directories. This is so that the RTL files can be rooted in a place different from the verification directory.

The /results directory is where you run the tests from. It also can be wherever you want. The authors generally put this directory in some non-backed-up networked storage area that is independent of the source-code control system. In the handbook example, the /results directory is placed in /examples/alu, at the same level as the /verification directory.

The /verification directory contains all the source code for the verification system. The /bin directory is there for the project's local scripts. The authors usually put a setup script there and alias setup to it.

The other four subdirectories—
/tests, /testbench, /vip, and /test_components
—are where the actual source files are. The /tests directory is where your test_name.cpp and test_name.h exist. These files are used when you give the --test <test_name> option to the truss script. (Use the --config option to the truss script to select a directory.)

The /testbench/top directory contains the C++ and HDL sources for the top-level testbench. If you have more than one chip in your simulation, it may be useful to have /testbench/<chip_name> directories.

The /vip directory is where chip interface classes go. There should be a subdirectory for each interface and major feature you need to test. The idea is that the code in these directories is fairly portable, and may contain purchased VIP as well as project and company-created VIP. In our example, there is only the /alu directory.

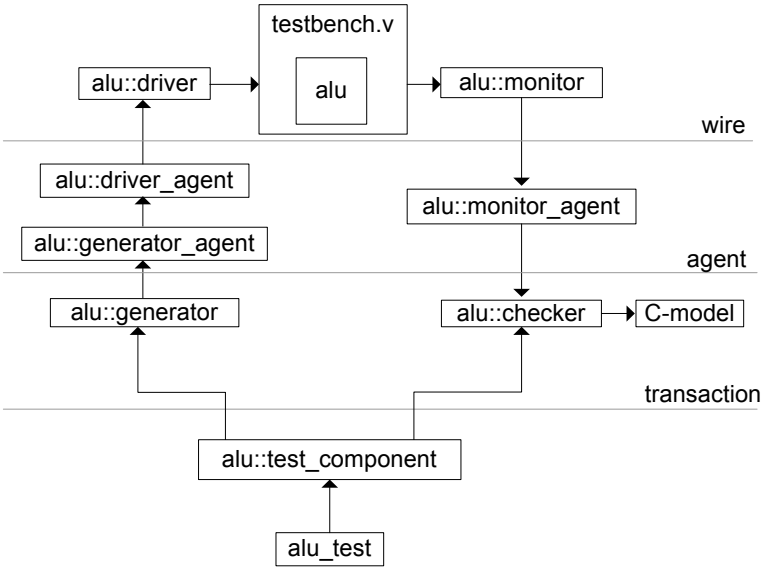
The /test_components directory contains the scenarios that you want to run. For this example, we'll only run one scenario, called test_component.

Theory of Operation

We'll be testing a really basic ALU chip. It takes in two 32-bit operands and performs a simple logic or arithmetic function. We'll use a legacy c-model for comparison with what the chip produces. The output consists of a result and an "operation complete" status interrupt. The `test-bench.v` will instantiate this ALU module and provide system clocks for the chip and verification system.

The main objects are shown below.

ALU Example: Objects and Connections



Because there is only one interface in the chip, we'll just refer to the components by their functionality. In other words, we'll say "driver," although in a chip with many drivers we would need to say which interface we are talking about. (Note that we do scope the code in an ALU namespace.)

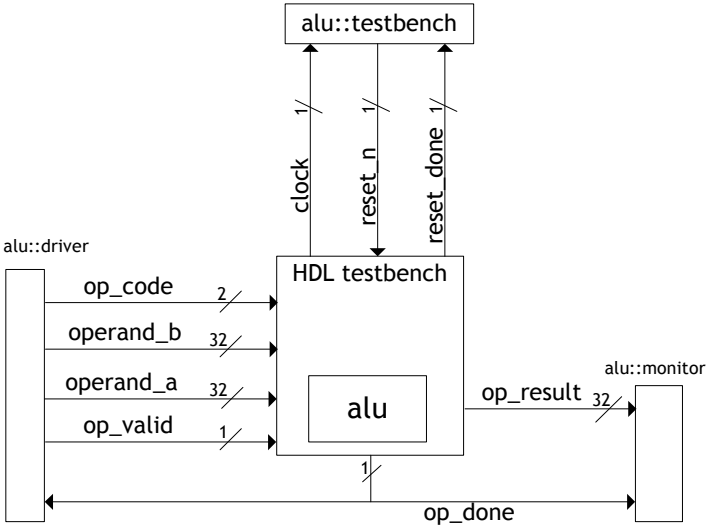
In the testbench C++ class, we have all the components of the ALU interface layer. There is a wire-layer driver and monitor, with their accompanying agents. There is a generator and a checker. The checker

is interesting, because we have a legacy c-model of the chip, which will be used by the checker.

There is also a `test_component` class, which runs a random number of operations through the chip. And, of, course, there is a `alu_test` class, which builds a `test_component`, giving it the generator, checker, and driver from the testbench.

The following illustrates the wires used by the verification system:

ALU Example: Wires and Objects



The driver and the monitor take care of the protocol into and out of the chip. The testbench takes care of bringing the chip out of reset.

The remaining sections highlight some specific "points of interest" in the code. The code itself, being the first example, is not that big. If you want to follow the code through its execution, start with the Truss `verification_top.cpp`, then move on to `testbench.cpp` and `test.cpp`.

Running the Simple ALU Example

You might want to see the log messages on the screen, so let's talk about how to run the example. In the `/examples/alu_tutorial/bin` directory, there is a setup script. If you look at the `setup` file, it sets up a few environment variables that are needed by the run and make tools.

First, source the `setup` file, then execute the following:

```
$TRUSS_HOME/bin/truss --test tutorial_test
```

The `truss` command has many more options; type `truss --help` for a synopsis.

You should see the C++ source files being compiled, and then the test should run. When the test runs, a series of printouts will announce the flow through the test. Remember that, by default, Teal prints the file and line number of the message.

Points of Interest

The next few sections address specific places in the code. These sections follow the general way you go about hooking up a chip to a Truss-based verification system.

For example, the first thing to be concerned with is bringing the chip out of reset. After that, you'll probably want to pick an interface and write the driver and monitor classes. Then, you decide upon some specific operations you want to perform and write the test component to exercise the interface or feature.

In general, the test builds the test components and ends when the last operation completes—that is, when the test component's `wait_for_completion()` returns.

Power-on Reset

Most chips have a power-on reset sequence. This sequence can be basic, or rather complicated. In this example we address a basic sequence.

The chip has a *reset* line, which is pulled low to initiate a reset. After the line is asserted, the chip performs its reset sequence. This chip only needs a fixed-duration pulse.

The `testbench` class is responsible for bringing the chip out of reset. The testbench methods `time_zero_setup()` and `out_of_reset()` are called by the top program to bring the chip on line. In our ALU example, we'll use a reference clock to count a number of cycles to keep the `reset_n` low.

Below are the snippets of code that perform the chip reset. The methods are located in `testbench.cpp`. Note that the `verification_top()` provided a path to the top of the testbench; the path is stored in the variable `top_`.

This method is called first by `verification_top()`:

```
void testbench::time_zero_setup() {
  teal::vreg reset(top_ + ".reset");
  reset = 0;
}
```

Then, this method is called:

```
const teal::unit32 reset_count = 10;
void testbench::out_of_reset(reset r) {
  teal::vreg reset(top_ + ".reset");
  teal::vreg clock (top_ + ".clock");
  reset = 1;
  for (int i(0); i < reset_count; ++i) {
    teal::at(teal::posedge(clock));
  }
  reset = 0;
}
```

That's all there is to it. Now the chip is ready for operation.

Driver and Monitor Protocol

Now that the chip is out of reset, we can start to drive it. This chip has a simple protocol for sending operations to perform. Assuming `op_done` is asserted, the driver puts `op_code`, `operand_a`, and `operand_b` on the wire. Then it asserts `do_op` and waits for `op_done` to be asserted. The code to do this is in `alu_driver.cpp` and is shown below:

```
void alu::driver::send_operation(const operation& op){
  op_code_ = op.code;
  operand_a_ = op.operand_a;
  operand_b_ = op.operand_b;
  op_valid_ = 1;
  at (posedge(op_done_)); //wait until accepted
  op_valid_ = 0;
  at (negedge(op_done_));
}
```

The variables above with the trailing "_" are data members and are Teal `vreg` objects that are connected to the chip.

The monitor code is fairly simple as well. The monitor uses a Truss utility thread class called `run_loop`. It consists of two methods, `loop_condition()` and `loop_body()`, which are run in a separate thread. The idea is that a large number of monitors are infinite loops of "wait for trigger" and then "gather data." This class represents that concept.

The `loop_condition()` method of the monitor waits for `op_done` to go high. The `loop_body()` method then copies the result into a local variable. It then calls the pure virtual method `operation_completed()` to connect to the monitor agent. Here is the code, in `cpu_monitor.cpp`:

```
void alu::monitor::loop_condition_()
{
  at (posedge(operation_done_));
}

bool alu::monitor::loop_body_()
{
  receive_completed_(result_.to_int());
  return true; //continue loop
}
```

Other than the reset logic (and the watchdog timer), the monitor and driver are the only code to interact with the chip wires.

Next we'll look at how we come up with the operations to be sent to the driver.

The alu_test_component

We now run a random sequence of operations through the ALU, testing the basic operations with random operands. The `test_component::start_components_()` method is used to run this exercise.

The code is shown below.

```
void alu::test_component::start_components_()
{
  driver_->start();
  checker_->start();
}
```

Like most test components, this one just starts the lower-level components.

Checking the Chip

Because we do verification for a living, the automated checking of the chip's results is important. In our case, we have a legacy c-model of the ALU and will use it to check that the answer is what we expected. The checker waits for the monitor agent to deliver a completed operation. Then it uses the inputs sent by the generator to have the c-model come up with the expected result.

The c-model prototype is shown below.

```
#if defined(__cplusplus)
  extern "C" {
  #endif

  unsigned int alu_model(unsigned int a,unsigned int b,
                         unsigned char op);

#if defined(__cplusplus)
  }
#endif
```

Note that the `ifdefs` allow the code to be compiled by both C and C++ code.

This key algorithm is in `checker.cpp` and is shown below.

```
void alu::checker::start_()
{
  for (;;) {
    operation gen = generated_->get();
    teal::uint32 actual = actual_->get();

    if (alu_model (gen.operand_a, gen.operand_b,
                   gen.op_code) == actual) {
      log_ << teal_info << " EXPECTED: sent " << gen
           << " == " << actual << endm;
    }
    else {
      log_ << teal_error << " sent " << gen
           << " !=  " << actual << endm;
    }
    if (!generated_->size()) {
      completed_flag_.signal();
      return;
    }
  }
}
```

The checker works fine as long as the `operation_done` is in synch with the result. However, the checker can be wrong if the monitor misses a result or somehow inserts an extra one. We could have registered the chip inputs at the same time as we got the results. However, by doing this we

make the assumption that there are no queuing or pipe stages in the ALU. This assumption works fine for our example, but it is probably not valid for most ALUs.

Completing the Test

When does the test stop? When `verification_top()` calls the test's `wait_for_completion()`, which in turn calls the test component's `wait_for_completion()`.

In turn, the test component's `wait_for_completion()` calls the checker's `wait_for_completion()`. The authors agree that this sounds silly, but in the later examples we actually do a bit more than just forward the call.

In the end of the forwarding chain, it's the checker that actually decides when the test is done. This makes sense, because the checker is the best able to "judge" what the chip did and when all the inputs have been checked.

But how does the checker know? There are many possible ways, but in this example the checker assumes that when the generated data channel runs dry, the test is over. This is a valid assumption—as long as you make sure that the generator can always be one step ahead of the checker. (If your chip has any latency, this is not a hard assumption to sustain.[1])

The checker code is shown below—

```
void tutorial::checker::wait_for_completion()
{
  completed_flag_.wait();
  //note that the checking thread completed normally
  completed_ = true;
}
```

—and at the bottom of the main check loop:

[1] Note that an intergenerate delay should not affect when the expected data are sent to the checker. The point is that even when delays are inserted, this model should be valid.

```
if (!generated_->size()) {
  completed_flag_.signal();
  return;
}
```

Remember that after the `wait_for_completion()` returns, the top calls the `report()` method in the test. The test calls the test_component's `report()` method, which in turn calls the checker's `report()` method.

The `report()` method prints the state of the `completed_` boolean. In this way, when you have multiple test components and the watchdog timer shuts the simulation down, you can tell which checkers have not completed.

Summary

This chapter is a tutorial on the Truss framework. We exercised a simple ALU, but implemented all the parts of a Truss-based verification system. The main objects and their connections were shown. The directory structure was introduced so we can find our way around the code. Then, the chip and the HDL connections were shown.

After laying out the verification system and showing how to run the example, we looked at how the chip was to be brought out of reset. We did a quick side tour to talk about how to run the example. Running the example produces many log messages, but this is probably a good thing when one is learning.

We showed how to bring the chip out of reset and how the driver and monitor interfaced with the chip. One point to note is that while this interface consisted of only a few wires, many interfaces in real protocols are this small. Of course, your code will be more detailed.

We looked at an important part of the verification system, the checker. In this example, the checker used a c-model to check that the chip was working correctly.

The last thing we looked at was how the test stopped. We looked at the normal path, ignoring the watchdog timer. We showed how the checker was in charge, pausing the end of the test until all the generated data had

been checked. The interesting point to note is that the checker may have had errors, but it will continue until all generated data have been checked. The Truss utility class `error_threshold` can be used to terminate the simulation in the case of excessive errors. The Truss `verification_top()` does this.

Whew! We made it through the first example. Time for a coffee break and some foosball!