VISA: a state-based, hierarchical, architecture-independent random test generation environment for high-performance multiprocessors

Neil McKenzie
Chris Weller
Michael Sedmak
Adam Snay

AMD, Inc.
Sunnyvale, Calif., and Fort Collins, Colo., USA

www.amd.com

ABSTRACT

Verification tool for an Instruction Set Architecture (VISA) is an architecture-independent random test generation (RTG) environment based on the Ruby scripting language and C++. VISA contains an embedded instruction set simulator (ISS) to keep track of the current state of each thread. VISA input templates are hierarchical: they can recursively include other VISA templates, and template code can make decisions regarding instruction generation on the fly based on the simulator state. After test generation is complete, VISA can create assembly language files for the standard directed test flow or directly output binary files for execution in co-simulation using Synopsys® VCS®. This paper discusses the current status of VISA, templates under development, and future directions. It contains introductory content and is intended for all verification engineers.

Table of Contents

Table of Figures

1. Introduction

Random test generators (RTGs) are software tools necessary for verification engineers to verify the designs of general-purpose microprocessors. Modern microprocessors are too complicated to verify in pre-silicon co-simulation by means of an exhaustive approach. In practice, it is feasible to run only a relatively small number of co-simulation test cases that cover just a tiny fraction of the space of all possible inputs. In effect, an RTG samples the space of all possible input test cases. The input specification steers the RTG to target different parts of this input space. An RTG is like a fractal generator: both types of generators create complex images from simple input specifications.

Figure 1 shows a diagram of the standard tool flow for running randomly generated tests in RTL co-simulation.

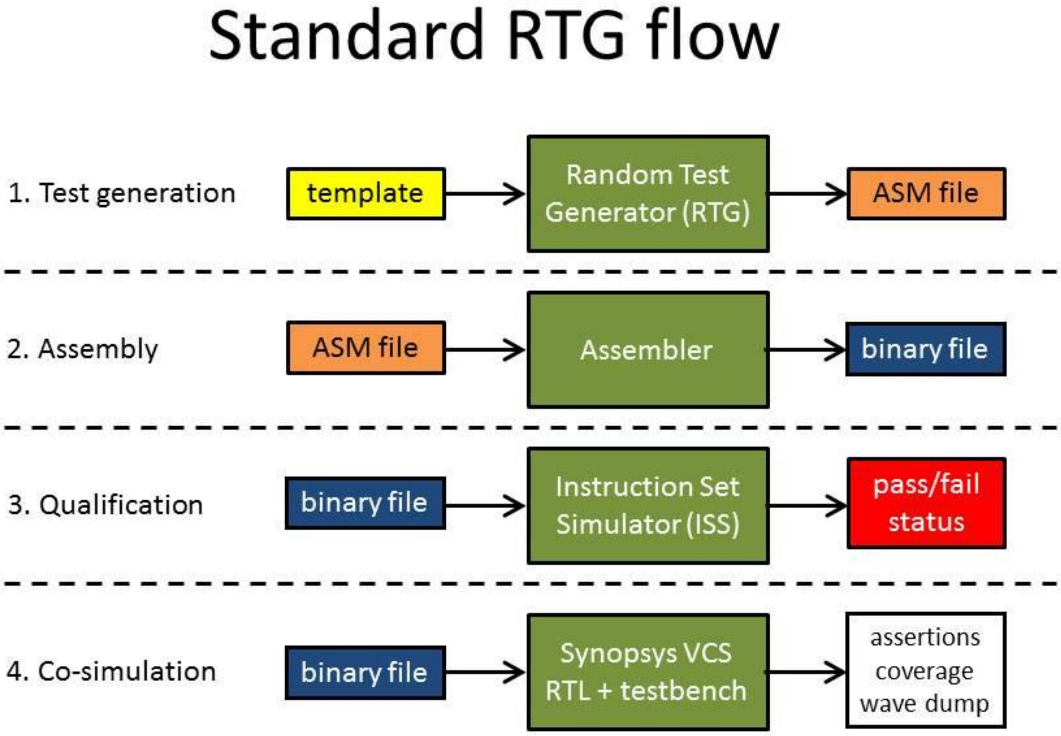

Figure 1: Standard tool flow for randomly generated tests in RTL co-simulation

The standard random test generation flow is based on the premise that each tool is self-contained and autonomous. All communication among stages is file-based. The first stage is to run the RTG using a template file as its input, and the output is an assembly language file. The template directs the RTG to generate tests that focus on particular features of the architecture. For instance, a template could constrain the RTG to generate instruction sequences that contain only floating-point instructions. Reproducibility is essential: if the same template and the same random number seed are given to the RTG, then it must generate exactly the same output file.

The second stage is to run the assembler, which produces a binary image file from the assembly language file input. The third stage involves the stand-alone instruction set simulator (ISS), which is the golden model for the instruction set architecture (ISA) of the design. The ISS qualifies the given binary file as passing or failing. A passing grade means that the binary image can be run in RTL co-simulation using Synopsys VCS or another Verilog behavioral simulator.

The outputs of RTL co-simulation are assertion checks, optional generation of coverage data, and optional generation of wave dumps for further debugging in case of test failure. Typically the RTL and the ISS are run concurrently in co-simulation. The testbench contains bridge code that checks the results of the RTL and the ISS after every instruction is retired.

The state-based random test generation flow is based on the premise that the test generation, assembly, and qualification stages run concurrently rather than separately. The granularity of information passed from one phase to the next is a single instruction rather than a complete file. Figure 2 shows a diagram of the state-based tool flow for running randomly generated tests in RTL co-simulation.

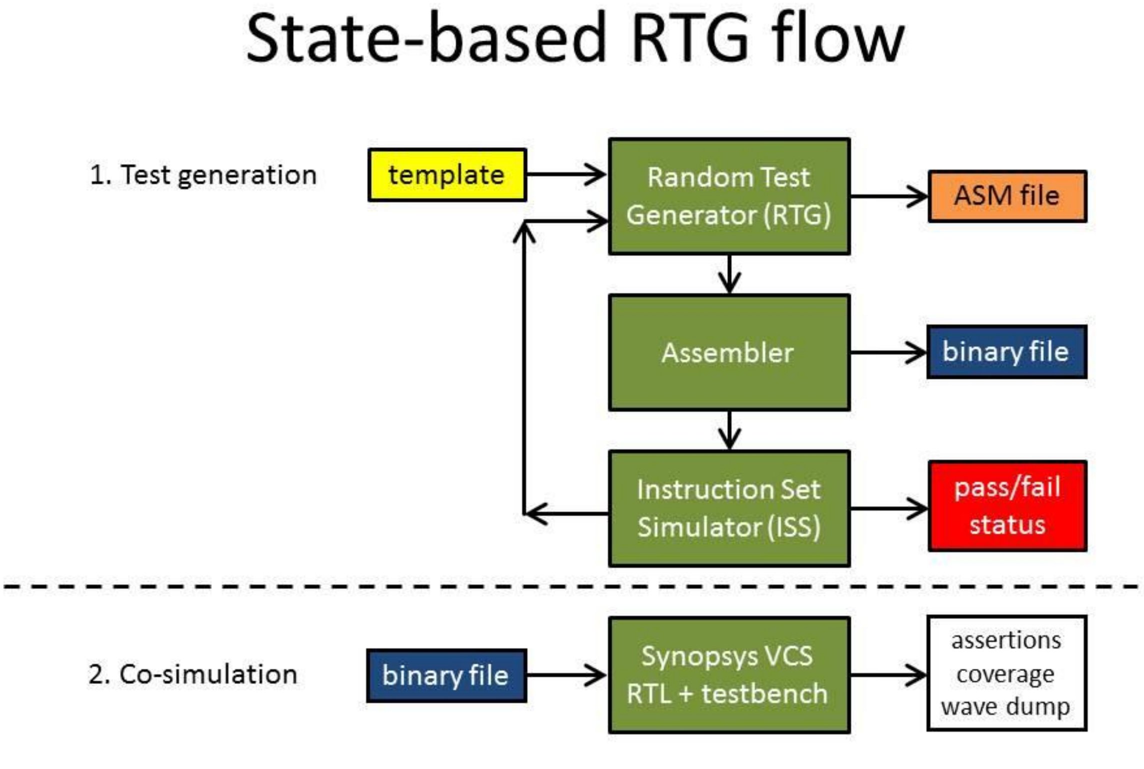

Figure 2: State-based tool flow for running randomly generated tests in RTL co-simulation

The state-based RTG flow has the following advantages over the standard flow:

- The RTG can detect whether an instruction will cause an exception.
- The RTG can insert self-checking code into the test case automatically.
- The RTG can make arbitrary test generation decisions based on the simulator state or on architectural coverage information.

As a result, the output of a state-based RTG can be much higher quality than the output from an RTG based on the standard flow. This property can reduce the number of co-simulation runs needed to close coverage and complete verification of the design. The trade-offs are that state-based RTGs can take longer to construct and they may take longer to generate test cases than standard RTGs.

Verification tool for an instruction set architecture (VISA) is a new random test generation environment that the authors built from scratch. We designed VISA to address the following requirements in pre-silicon microprocessor verification: hierarchical test specification, multiprocessing, ISA-independence, and state-based test generation.

In the rest of this paper, Section 2 shows the organization of VISA. Section 3 walks through several VISA template examples and describes how templates can access the simulator state. Section 4 discusses the speed of test generation using VISA. Section 5 provides a summary of this paper and presents conclusions, and Section 6 describes future work.

2. VISA organization

Nearly all RTGs are constructed using a combination of languages. The front-end of an RTG is usually a commonly available scripting language such as Perl, Python, Ruby, Tcl, or JavaScript. The front-end parses the input template. The back-end of the RTG is usually written in C or C++, and it is usually compiled separately and linked to the front-end to maximize the speed of test generation. The scripting language layer is attached to the C/C++ layer using a wrapper generator such as SWIG [1] to generate the proper bindings automatically.

We implemented the front-end of VISA in Ruby. Ruby, like other scripting languages, is interpreted rather than compiled and it does type-checking at run time instead of at compile time. Ruby makes it easy to write code in an object-oriented style because it supports classes, inheritance, virtual method invocation, polymorphism, and data encapsulation.

Syntactically, Ruby has an unusual look and feel because it lets the programmer avoid the use of most punctuation characters. For example, in C++, it is possible to have the following statement:

```
y = x->first()->second()->third();
```

In Ruby, the equivalent statement would be:

```
y = x.first.second.third
```

The ability to make Ruby programs terse is a blessing as well as a challenge for programmers who read Ruby code because `third` can reference either a method or a variable.

Figure 3 displays the overall organization of VISA components.

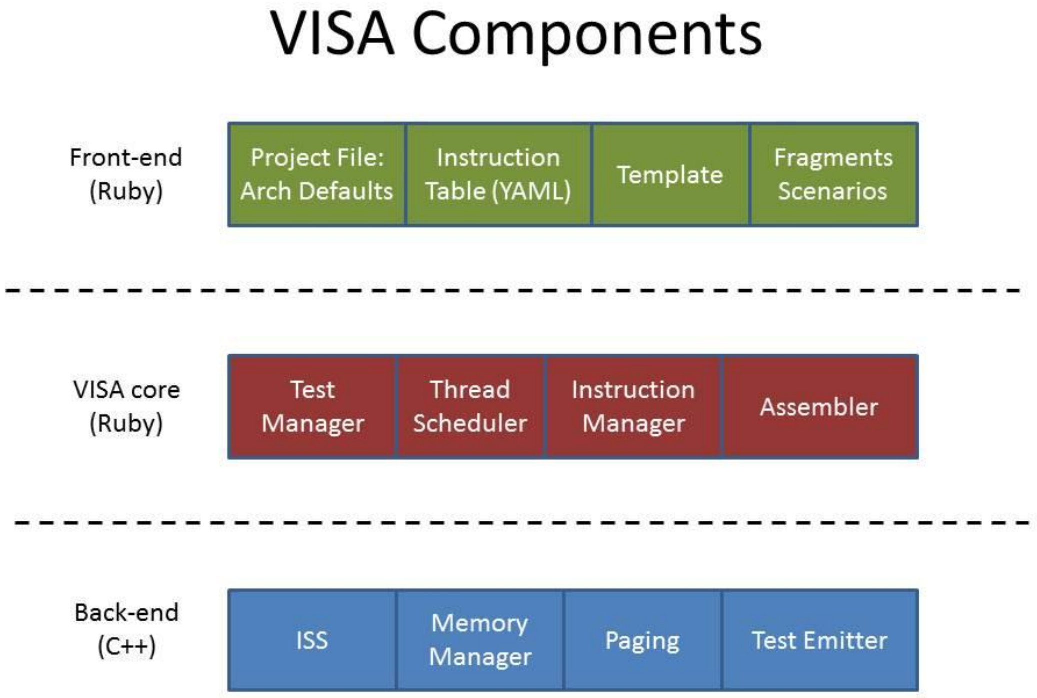

Figure 3: Organization of VISA components

Front-end components:

- The project file is loaded automatically when VISA starts executing. It contains a list of default values for the selected architecture: sizes and set associativity of all caches used in the design (L1 instruction, L1 data, L2, and optionally L3), a list of general-purpose registers (GPRs) that can be assigned randomly generated values, paging parameters, and memory allocators and placers.
- VISA's instruction table is written in YAML, a language-independent text file format [2]. YAML parsers are readily available for Ruby and other scripting languages. The instruction manager creates Ruby classes for each instruction from the YAML description.
- We considered a variety of approaches for template files. One option was to use a YAML file to specify weights or probabilities of particular instructions or instruction sequences. The disadvantage of this approach is the lack of customization. If customization is not possible in the template, then features need to be added directly to the RTG itself. We find that users are happier when they can customize their template code directly rather than waiting for a new release of the tool from the tool developers. For these reasons, we decided to use Ruby as the template language for VISA. Template code in VISA is executed directly by the Ruby interpreter. Templates can be arbitrarily complex and they can

access all components of VISA. Templates can also include other templates. We illustrate how hierarchical templates are organized in Section 3.

- Fragment and scenario generators are library modules accessed by the template. A fragment is a contiguous sequence of instructions that targets an aspect of the design that may be difficult or inefficient to hit by completely random chance. For instance, to test division by zero, the template invokes a fragment that first clears a register value to 0, then subsequently issues a divide instruction. A scenario involves the generation of instructions or fragments that are not necessarily contiguous. An example of a scenario is false sharing between two threads. The scenario generator for false sharing involves the allocation of adjacent, non-overlapping memory blocks that lie within a shared resource (i.e., a cache line) and having each thread repeatedly load from and store to memory locations in these blocks.

VISA-core components:

- The test manager directs the overall flow of VISA and it implements the top-level loop.
- The thread scheduler selects the next thread to advance if VISA is creating a multi-threaded test case.
- The instruction manager allows templates to query the instruction database and select the next instruction that is generated.
- The assembler converts instructions into bytecodes and loads them into the simulator memory.

Back-end components:

- The instruction set simulator is the golden model for the selected ISA.
- The memory manager handles operations on memory regions: creation, deletion, and extension.
- The paging manager manages page tables. It allows templates to create various mappings between virtual and physical addresses, including aliases. It also works with the memory manager to allocate memory regions in which to place the page tables.
- The test emitter is invoked after all instructions are generated. It emits the test case in the requested format. The test emitter can output either to an ASM file or a binary file or both (see Figure 2).

2.1 Running VISA

VISA takes the following arguments on the command line:

- Seed value
- Number of threads to generate
- Architecture name
- Name of ISS module
- Name of test emitter module
- Name of project file

- Name of template file
- Name of output file

VISA takes the following actions at start-up:

- It loads the dynamic modules: C++ shared object (.so) files and Ruby libraries.
- It finds the top-level test class.
- It creates an instance of the top-level test class. This action causes the initialize method of the class to be invoked. The initialize method can allocate memory regions or page tables and then assign static data to these regions. Static data are then loaded into the simulator's memory.

The test manager then takes the following actions:

- It creates an array of Ruby fibers (one per test thread). Ruby fibers are primitives for implementing lightweight cooperative concurrency. A running fiber continues running until it explicitly blocks itself by executing `Fiber.yield`. A blocked fiber waits until it is resumed [3].
- It starts each fiber by invoking the main method of the top-level test class. The state object is passed into the thread's main method for thread identification. Generally, a fiber will block itself after generating one instruction.
- It executes VISA's main loop.
- It invokes the test emitter to produce the final output files.

2.2 Main loop

The main loop executes from within the test manager. It has the following steps:

1. **Scheduling.** The test manager calls the thread-scheduling algorithm to pick the next thread to run. The test manager then resumes the fiber for the selected thread. The fiber continues from the point right after it blocked itself.
2. **Computation.** Each fiber can execute arbitrary Ruby code. It can make decisions based on the simulator state. For instance, it can select immediate data for an ADD instruction that will cause the overflow bit to be set.
3. **Instruction selection and assembly.** The fiber invokes the instruction generator to pick the next instruction to generate for this thread. Then the instruction is assembled into bytecodes that are then loaded into the simulator's memory at the location indicated by the thread's program counter (PC).
4. **Simulation.** VISA invokes the ISS to execute this instruction. In case of a loop, interrupt handler, or exception handler, the simulator is stepped multiple times until the loop or handler code terminates. One of three situations happens after the simulator is stepped:

   - If the simulator goes into a halt or shut-down state unexpectedly, then there is a bug either in the template code or in VISA that needs to be fixed.
   - If the simulator goes into a halt state and this was expected, then the thread is terminated and its fiber goes into the inactive state.

- Otherwise, the fiber blocks itself.

The main loop continues until there are no active threads to run.

3. VISA template examples

This section walks through a set of examples demonstrating how to write VISA templates.

```
1  class NullTest < TestGenerator
2  # NullTest has a trivial initialize method
3    def initialize(name, threads)
4      super(name)
5    end

6  # NullTest has an empty generate method
7    def generate(state)
8    end

9  # The main method is common to all test templates
10   def main(state)
11     thread = state.get_id
12     gen InitGenerator.new(name: "init_thread#{thread}")
13     gen self
14     gen StateCheckGenerator.new("check_#{thread}")
15     gen PassedGenerator.new("passed_thread#{thread}")
16   end
17 end
```

Figure 4: NullTest template

Figure 4 contains the template code for a Ruby class called NullTest. The following is a detailed explanation of the structure of this template:

- **Lines 3, 7, and 10.** This class contains three methods: an initialize method, a generate method, and a main method. The test manager first calls the initialize method. The test manager then calls the main method once per thread.
- **Line 12.** `InitGenerator` is an ISA-specific scenario generator that creates the initial page table, allocates code memory, and generates the boot sequence.
- **Line 13.** The method `gen` is defined in `GeneratorBase`, the parent class of `TestGenerator`. The phrase `gen self` causes the generate method for its own class to be invoked.
- **Line 14.** `StateCheckGenerator` generates code that checks the state of the golden model against the state of the RTL.
- **Line 15.** `PassedGenerator` generates the code sequence that represents a passing test.

All other test templates can be derived from NullTest. Derived classes can override the initialize and generate methods; however, the main method is common to all tests, so it doesn't need to be overridden.

```
18 require "nulltest.rb"
19 class RegRegTest < NullTest
20   def initialize(name, threads, rr_count=nil)
21     super(name, threads)
22     @rr_count = rr_count || 10
23     @regreg_gen = InstGenerator.new(
24       name:         "regreg_gen",
25       instructions: { /Reg32_Reg32/ => 100 },
26       operands:     { Reg32 => RegOperandPicker.new(size:4) }
27     )
28   end

29   def generate(state)
30     @rr_count.times do
31       gen @regreg_gen
32     end
33   end
34 end
```

Figure 5: RegRegTest template

Figure 5 is a VISA template containing the RegRegTest test class. RegRegTest overrides the initialize and the generate methods from NullTest. It uses the main method from NullTest.

- **Line 19.** The RegRegTest class is derived from NullTest.
- **Line 20.** Ruby allows a variable number of arguments to be passed on method invocations. Missing arguments are assigned to default values. Here, the initialize method takes the required parameters `name` and `threads` and the optional parameter `rr_count`, which is assigned to the value `nil` (undefined) if the caller doesn't pass a value for it.
- **Line 22.** The `@` symbol is the scoping operator for instance variables. `@rr_count` takes the value of `rr_count` if it is present; otherwise, it takes the default value 10. It is standard practice in Ruby to use the logical-OR (`||`) operator within assignments to replace uninitialized values with default values.
- **Lines 23 to 27.** The Ruby class `InstGenerator` is an interface to the Instruction Manager. The template creates an object of this class and then assigns it to the instance variable `@regreg_gen`. It selects any instruction whose name matches the regular expression `/Reg32_Reg32/`. For instance, if the instruction table contained instructions with the names `ADD_Reg32_Reg32`, `SUB_Reg32_Reg32`, and `MOV_Reg32_Reg32`, then one of these instructions would be selected at random and generated.
- **Line 29.** The test manager invokes the generate method of RegRegTest once per thread.
- **Line 31.** This is the mechanism that allows templates to recursively include other generator objects (including other templates). The `gen` method dispatches the generate method

of the object referenced by @regreg_gen. The object's generate method can, in turn, invoke the generate method of other objects.

```
35 require "nulltest.rb"
36 class LoadStoreTest < NullTest
37   def initialize(name, threads, ls_count=nil, load_pct=nil)
38     super(name, threads)
39     @ls_count = ls_count || 15
40     @load_pct = load_pct || 50
41     @load_gen = Array.new
42     @store_gen = Array.new
43     threads.each do |thread|
44       dp = "data_pool#{thread}"
45       new_memory_allocation(size: 32, pool: dp)
46       @load_gen[thread] = LoadGenerator.new(
47         name: "load_gen#{thread}", pool: dp)
48       @store_gen[thread] = StoreGenerator.new(
49         name: "store_gen#{thread}", pool: dp)
50     end
51   end

52   def generate(state)
53     thread = state.get_id
54     @ls_count.times do
55       rand_value = [0..99].rand
56       if rand_value < @load_pct
57         gen @load_gen[thread]
58       else
59         gen @store_gen[thread]
60       end
61     end
62   end
63 end
```

Figure 6: LoadStoreTest template

Figure 6 is the template for LoadStoreTest.

- **Line 39.** @ls_count is the number of times to execute the generate loop.
- **Line 40.** @load_pct is the percentage probability of generating a load instruction.
- **Lines 41 and 42.** @load_gen and @store_gen are declared as arrays to allow customization on a per-thread basis.
- **Line 43.** This is the Ruby equivalent of a "for" loop with thread as the iteration variable.
- **Line 44.** #{thread} is Ruby syntax for converting the thread number into an ASCII string, which is appended to the string data_pool.

- **Line 45.** `new_memory_allocation` allocates a block of memory. A separate 32-byte block of memory is allocated for each thread.
- **Lines 46 and 47.** `LoadGenerator` is a fragment generator that generates two instructions: one to put the address of the thread's memory block into a register, and another to perform the load from this address.
- **Lines 48 and 49.** `StoreGenerator` is like `LoadGenerator` except that a store instruction is generated instead of a load instruction.
- **Line 55.** `rand_value` is assigned a randomly chosen value in the range 0 to 99.
- **Lines 56 to 60.** If `rand_value` is less than `@load_pct`, then a load fragment is emitted; otherwise, a store fragment is emitted.

```
64 require "load_store.rb"
65 require "regregtest.rb"
66 class CombinedTest < NullTest
67   def initialize(name, threads)
68     super(name, threads)
69     @regreg_test = RegRegTest.new("rrtest_gen", threads, 1)
70     @ls_test = LoadStoreTest.new(
71                  "lstest_gen", threads, 1, 75)
72   end

73   def generate(state)
74     20.times do
75       rand_value = [0..99].rand
76       if rand_value < 40
77         gen @regreg_test  # 40% reg-reg ops
78       else
79         gen @ls_test      # 45% loads and 15% stores
80       end
81     end
82   end
83 end
```

Figure 7: CombinedTest template

Figure 7 shows an example of hierarchical composability. This template defines a new test class called CombinedTest that re-uses the code from the test template classes RegRegTest and LoadStoreTest.

- **Lines 69 to 71.** The initialize method of CombinedTest sets the number of iterations in the generate method for RegRegTest to 1, the number of iterations for LoadStoreTest to 1, and the probability of generating a load to 75 per cent. Member variables `@regreg_test` and `@ls_test` are assigned to instances of the RegRegTest and LoadStoreTest classes, respectively.
- **Lines 73 to 82.** The generate method for CombinedTest is a loop that executes 20 times and randomly picks a register-register operation, a load, or a store. The probability of generating a register-register operation is 40 percent. The probability of generating a load

is 0.60 x 0.75 = 45 percent. The probability of generating a store is 0.60 x 0.25 = 15 percent (Figure 8).

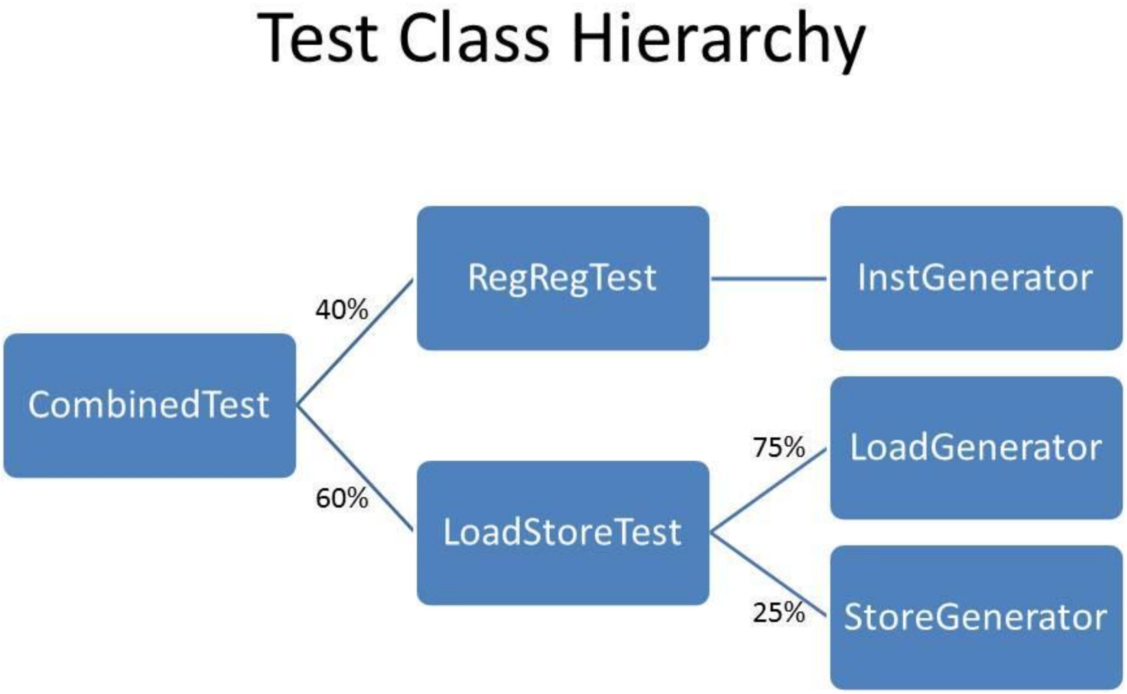

Figure 8: Hierarchy of test classes and generators in CombinedTest

In addition to `InstGenerator`, `LoadGenerator`, and `StoreGenerator`, VISA contains a number of other fragment and scenario generators that generate the following types of instruction sequences:

- Stride and region pre-fetching
- Thread synchronization
- Mutual exclusion
- Loops
- Branches
- Branch misprediction
- Sub-routines
- Self-modifying code
- Cache misses
- Exception handling
- Virtual-address aliasing

Figure 8 demonstrates that VISA's fragment generators are like Lego® bricks that can be plugged together in an unlimited number of different ways to build test classes of arbitrary complexity. We find this approach powerful and flexible. The preceding code snippets show how default parameters from test classes can be overridden when they are instantiated into other test classes.

3.1 Template access to simulator state

Templates can access the state of the simulation through an application programmer interface (API) that provides the following bindings:

- `state.get_reg8(reg_id)`
- `state.get_reg16(reg_id)`
- `state.get_reg32(reg_id)`
- `state.get_reg64(reg_id)`
- `state.set_reg8(reg_id, byte)`
- `state.set_reg16(reg_id, word)`
- `state.set_reg32(reg_id, dword)`
- `state.set_reg64(reg_id, qword)`
- `$test_manager.get_mem_state.read_mem32(address)`
- `$test_manager.get_mem_state.write_mem32(address, dword)`
- `$simulator.step_sim(state, $test_manager.get_mem_state, instruction, page_table)`

The `state` object represents the state of a thread; each thread contains a separate set of GPRs. The `state` object is passed into the `generate` method for each test template (see line 73 for an example). Identifiers that begin with a dollar sign (`$test_manager` and `$simulator`) indicate that they are globally scoped. There is only one instance of both the test manager and the simulator.

4. Performance analysis

One possible drawback to coding in Ruby is its speed of execution. Anecdotal evidence suggests that the performance penalty for Ruby is 10x to 100x compared with compiled C++ code [4]. Our own measurements demonstrated a 40x speed-up when a Ruby routine was rewritten in C++. Nevertheless, using Ruby as the implementation language for VISA does not create a bottleneck for pre-silicon verification using co-simulation with VCS. Figure 9 is a histogram that illustrates test-generation times as a percentage of co-simulation time. We generated 225 different test cases using VISA and ran each one in co-simulation. Test cases are grouped as follows:

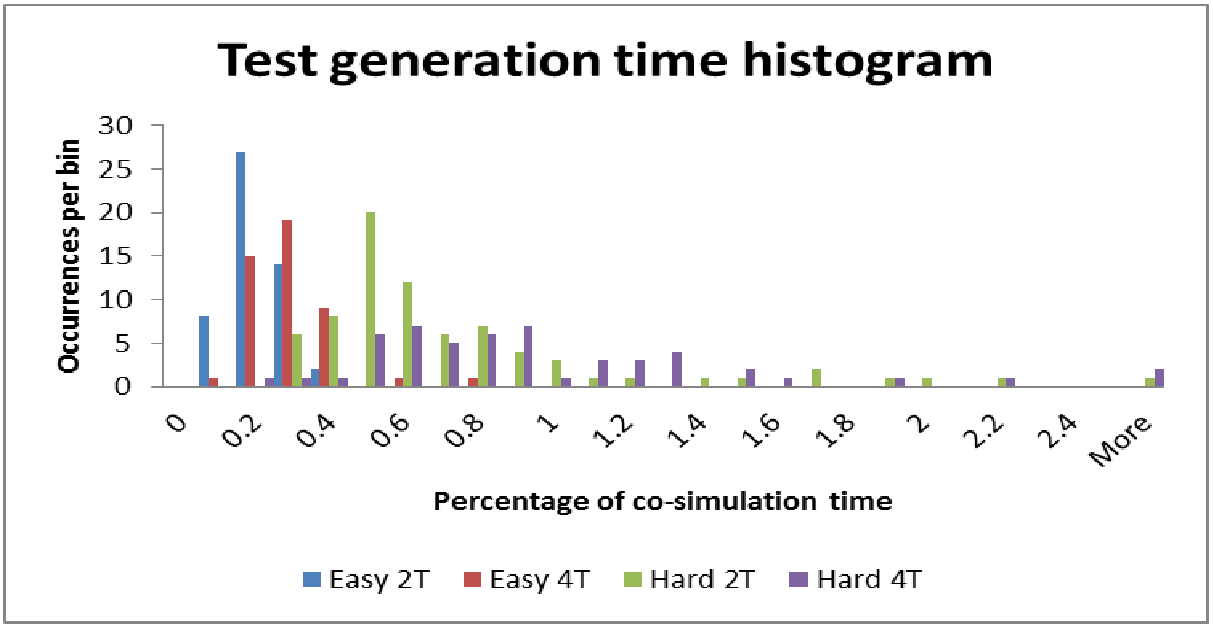

Figure 9: Histogram of test-generation times reported as percentages of co-simulation run times

- Easy 2T (blue bars): tests using two threads generated by simple templates
- Easy 4T (red bars): tests using four threads generated by simple templates
- Hard 2T (green bars): tests using two threads generated by complex templates
- Hard 4T (purple bars): tests using four threads generated by complex templates

Simple templates have little interaction among threads, whereas complex templates involve interactions among threads (for example, multiprocessor coherence and synchronization).

Test simulation times are sorted into bins with width 0.1% (the left-most bin represents the range 0.0% to 0.1%, the next bin is the range 0.1% to 0.2%, and so on). The most populous bin is the range 0.1% to 0.2%, which contains the blue bar representing Easy 2T and the red bar representing Easy 4T. There is a green spike for Hard 2T in the range 0.4% to 0.5%. The complexity of the template has more influence over test-generation time than does the number of threads.

For the majority of test cases, test-generation time is less than 1% of co-simulation time, even for the Hard 2T and Hard 4T test cases. There are outliers for some of the Hard 2T and Hard 4T test cases in which test-generation time was 3.7% of co-simulation time, but only three instances of 225 took more than 2.5% of co-simulation time. We conclude that the slowdown due to the Ruby implementation is a reasonable trade-off for pre-silicon verification.

5. Summary and Conclusions

In this paper we presented VISA, a new random test generation environment crafted in Ruby and C++. We were motivated by the following design decisions:

- We embedded the golden model ISS into VISA. This tight coupling between instruction generation and execution makes it possible for templates to query the simulator state to make informed decisions regarding subsequent instructions to generate.
- The difficulty of creating simple yet effective input templates is a common source of frustration for many verification engineers. We addressed this issue in VISA by making templates hierarchically composable. In essence, templates can include other templates directly. RTGs often have thousands of knobs (input parameters that can be set by the template). The default values for each knob can be declared in a common template that is included by all other templates.
- We chose Ruby because it encourages rapid prototyping and makes it easy to move functionality into C++ to speed up the rate of test generation. While there is a significant performance penalty for running interpreted code compared with compiled code, the speed of executing our RTL in co-simulation using VCS is 100x slower than the speed of test generation for the majority of test cases. At the moment, we are not compelled to spend a lot of effort to rewrite Ruby code into C++ because the pay-off is relatively small. This trade-off may change if we decide to use VISA for post-silicon validation.

Our experiences with Ruby have been generally positive. Only one member of our development team (the technical lead) had any significant prior experience with the language. The technical lead created the foundation of VISA and also created check-in qualification scripts to enforce a consistent coding style. The other team members who were Ruby newbies never learned or advocated for a competing coding style. We conclude that it is sufficient to have just one language expert on a development project such as this one who can serve as the evangelist for the other team members.

6. Future work

We are still very early in the initial implementation phase of VISA. So far, we have created templates that target the following use-cases:

- Multi-threaded coherency: false sharing, true sharing, ordered data accesses across threads, spin loops/synchronization, prefetcher stimulus, probes, victims
- Paging: multiple page tables (shared across threads), virtual aliasing, physical aliasing, run-time page-table modifications (setting the page-present bit in the fault handler), dynamic page-table switching, all memory types/page sizes/attributes
- Hypervisor-guest: virtual aliasing, physical aliasing, dynamic page-table switching

VISA has not been involved in a complete microprocessor development cycle at this point, and we have no information on how much this approach benefits coverage closure. We intend to present these results in a future publication.

In principle, it is possible to run VISA and co-simulation concurrently, as outlined in Figure 10. The challenge is that both VISA and VCS contain their own main loops. One solution is to use a client-server paradigm in which VISA runs as the client process and co-simulation (VCS) runs as the server process. VISA attaches to a client stub that sends remote procedure calls (RPCs) to the server stub. The API could be extended to allow VISA access to any state information in the VCS server process, including micro-architectural state and/or coverage data.

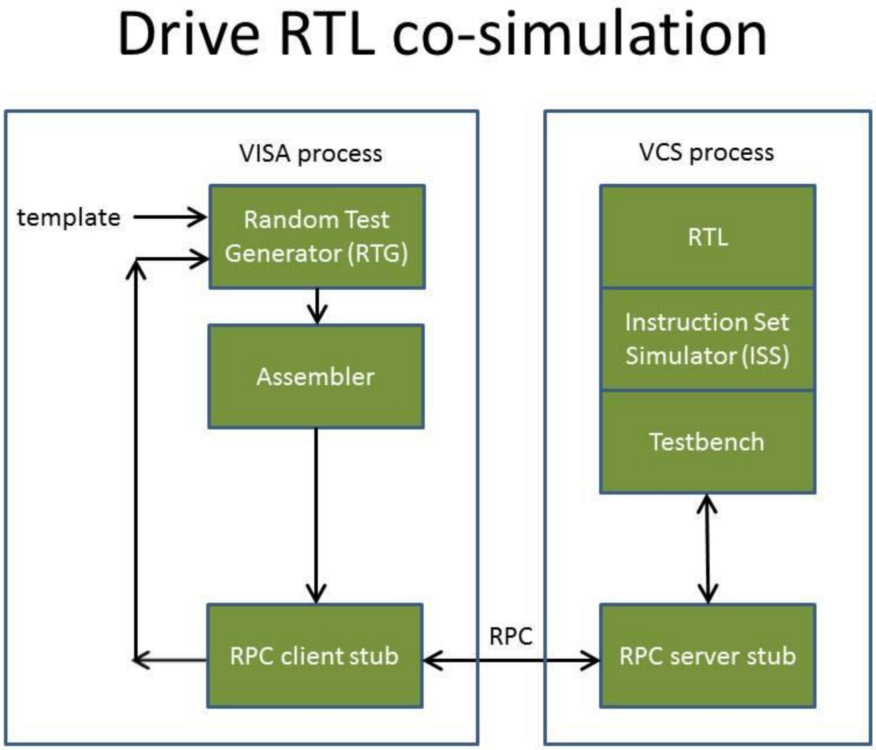

Figure 10: Using VISA to drive RTL co-simulation using client-server RPC

Running VISA and VCS concurrently in this manner is conceptually possible, but it has not been realized yet. If there is sufficient motivation to go down this path, then this approach may be the subject of a future publication.

7. References

[1] David M. Beazley. SWIG: An Easy-to-Use Tool for Integrating Scripting Languages with C and C++. Presented at the 4th Annual Tcl/Tk Workshop, Monterey, Calif. July 6-10, 1996. http://www.swig.org/papers/Tcl96/tcl96.html

[2] Clark C. Evans. YAML: YAML Ain't Markup Language. YAML is a human friendly data serialization standard for all programming languages. http://yaml.org

[3] Ilya Grigorik. Fibers & Cooperative Scheduling in Ruby. Posted on the web on May 13, 2009. http://www.igvita.com/2009/05/13/fibers-cooperative-scheduling-in-ruby/

[4] The Computer Language Benchmarks Game. Updated December 29, 2012. http://benchmarksgame.alioth.debian.org/u32/benchmark.php