Applying Constrained-Random Verification to Microprocessors

Jason C. Chen, Synopsys Inc. - December 10, 2007

Constrained-random verification (CRV) offers a highly effective way to deal with the challenges of microprocessor verification. These verification challenges are overwhelming for many reasons: complex instruction sets, multiple pipeline stages, in-order or out-of-order execution strategies, instruction parallelism, fixed- and floating-point scalar/vector operations, and other features that create a seemingly never-ending list of corner cases to exercise. The time required to create traditional directed tests has become unreasonable.

Language features such as the SystemVerilog random sequence generator allow you to create instruction sequences randomly to improve stimulus quality based on a structured set of rules and scenarios. Such random-sequence generation schemes are procedural, however, and do not take full advantage of object-based randomization using constraints.

This article proposes an object-oriented solution for processor verification challenges. The solution covers both a top-down stimulus planning process and a bottom-up implementation solution using SystemVerilog and commercially available base classes (such as those in Synopsys's Verification Methodology Manual, VMM). The description that follows covers the most important parts of this solution and uses a processor supporting the MIPS-I instruction set as an example design under test (DUT).

Stimulus strategy

As with most designs, simple random stimulus is not sufficient to verify a processor fully. Pure random instructions rarely create useful stimulus for the processor to target important design functionalities such as branches/jumps and exceptions. The CRV approach provides the ability to create useful stimulus but can only do so when the stimulus-generation infrastructure is built with enough intelligence about the processor instruction set architecture and state. Top-down planning is needed to create this infrastructure.

The main stimulus for the DUT is a program trace that can be considered as a collection of one or more instruction scenarios. In Figure 1, for example, Scenario 0 can be generic boot code for the processor with an exception handler. Scenario 1 can be a set of instructions to program the processor's internal configuration registers with dynamic contents to set up hardware watchpoints. The other scenarios can consist of one or more instructions with load/store, arithmetic, and branch operations, possibly including nested branch loops. Exception conditions can be introduced at random inside these scenarios.
1. Building blocks of a program trace from operations, instructions and scenarios.

Error conditions or exceptions must be planned for early. From a stimulus-generation perspective, planning for a processor's exception conditions must cover the occurrence of any one specific exception cause as well as the exception's occurrence probability. Moreover, the test plan should provide multiple exception conditions simultaneously to test the DUT's exception priority and handling. Because these considerations can affect the modeling of the properties and constraints between transaction objects, planning must begin early.

Objected-oriented stimulus design

SystemVerilog supports object-oriented-based data abstraction. Class objects contain information (properties) and actions (methods) that work on the properties. SystemVerilog's randomization is also built in an object framework. For processor verification, three levels of transaction abstraction &#8212; operations, instructions, and instruction scenarios &#8212; are modeled as classes. These classes are implemented in a bottom-up manner because it is essential to create the lower-level building blocks before you can raise the level of abstraction to stimulus descriptions (as verification engineers typically deal with stimulus).

A transaction is modeled as a SystemVerilog class with three major components:

- Properties—typically fields of a transaction and other supporting information that describe what the transaction is. For example, a transaction might be an ADD operation with two source registers, and the result is placed into a destination register.
- Constraints—code describing the relationships between properties.
- Methods—functions and tasks defining how transaction-level actions are performed. You can use a task to display a transaction in its assembly syntax (ADD R10, R3, R5), for example, and you can use a function to pack the same transaction into its binary representation (000000_00011_00101_01010_00000_100000).

Properties

The MIPS-I instruction set has four functional classes: no operation, load and store, computational and control. The operation class encapsulates the operation kind, the list of operands, and other properties. A "kind" property distinguishes the operation this object represents. It is an enumerated type that has a full listing of the supported opcodes as defined by the instruction set architecture (Figure 2).
2. Instruction set architecture.

In addition to the discriminant property "kind," it is useful to create another random property to describe the functional class of the operation, such as the load/store and control (branch/jump) classes. You can use this property when you need constraints for a group of operations or certain decisions are to be made across operations of the same functional class.

Similarly, you can use SystemVerilog constraints to implement the encoding rules for operations that make use of a function field (as some MIPS operations do). Sometimes, however, the transaction fields alone are not descriptive enough. Consider, for example, the MIPS branch-on-equal operation (BEQ): "If the contents of general register rs and rt are equal, the branch is taken and the program execution jumps to a branch label; the jump address is computed to the program counter (PC) calculated as the current PC plus a sign extended immediate value, or PC = PC + sext (imm)." The imm field is nothing more than a computed program counter offset.

Properties can be added to the opcode class to build good descriptions of branch operations:

- "LABEL" is added to the enumerated operation type, kind_e.
- A "label_suffix" property is added to the opcode class.
- "From" and "to" properties are added.

Each branch label takes the form LABEL_. You can think of the label_suffix property, an integer, as the line number in the program trace listing. For example, LABEL_005 is the 5th line in the sample program trace listing in Figure 3.
3. Sampling Program Trace Listing with Line Numbers.

Using line numbers meets the requirement that labels used in a legal program trace shall be unique. With the sample program trace listing in Figure 3, the "from" and "to" properties take on the values of 3 and 5, respectively, for the BEQ operation jumps from line 3 (from=3) to line 5 (to=5), if the branch is taken in program execution. The relative program counter offset (the "imm" field of the operation encoding) can be calculated from these two integral numbers. Figure 4 shows the opcode class with the added properties.

4. Additional "LABEL" kind and other operation class properties to support branches.

Today's processors support different types of exceptions to enable software to handle conditions such as illegal opcodes and watchpoints. As a requirement derived from top-down stimulus planning, the operation class supports the ability to introduce illegal opcodes. Like the LABEL kind, an operation kind called ILLEGAL is added to the enumerated property, kind. When the operation kind is randomized to ILLEGAL, a random and unassigned opcode value is then used to cause an operation to be illegal for exception testing.

Constraints

The second component of the transaction class is the constraints, which describe relationships between the properties—specifically, the two opcode objects. As an example, you need to translate processor rules such as the following MIPS specifications into SystemVerilog constraints:

- [R1] Memory load and store operations are in slot 0 only. If not, an exception is to be detected.
- [R2] Return from Exception (ERET) must be in slot 0. If not, an exception is to be detected.
- [R3] Return from Exception (ERET) is in slot 0 and must be paired with NOP in slot 1. If not, the hardware behavior is undefined.
- [R5] Writing to the same scalar register in both operations of the same instruction is disallowed. If it is attempted, the hardware behavior is undefined.

Figure 5 shows an implementation. Keep in mind that you can independently modify each SystemVerilog constraint block. For a finer resolution of control by test writers, consider implementing these rules in multiple constraint blocks.

5. Constraints describing relationships between Instruction properties.
If the constraint blocks in Figure 5 are left modified, Rules 1, 2, 3 and 5 will always be obeyed. Because these rules are implemented in separate constraint blocks, they can be individually controlled, if needed. For example, if you modify (turn off) the constraint block “slot0_only_good”, after a few randomize calls, the op[1].kind value will likely get randomized to one of the memory load and store operations (e.g., LW, SB, ). In this case, Rule 1 is violated and the condition causes an exception.

**Class Methods**

**Class methods**

The third component of the transaction class is the set of class methods. Two examples in the opcode class are the methods for displaying and packing an instruction: `opcode::psdisplay()` and `opcode::byte_pack()`. Figure 6 shows the body of the two class methods for the instruction class.

[Click here for Fig. 6](#)

**6. Implementation of standard vmm_data methods in the Instruction class.**

Now consider the instruction scenario base class. Instead of using an automatically generated scenario class as the base class for all user-defined scenarios, it is common to create another class—the instruction scenario base class—to describe the rules and best practices needed by all user-defined scenario extensions. This methodology promotes reuse through object inheritance.

Although some instruction sequences are not legal according to the processor's software tool chain requirements, you still have to verify the processor hardware's operation with the sequences. Some other instruction sequences may be legal program traces but trigger well-defined processor behaviors that need to be verified at the hardware level. If load/store accesses to data memory are not aligned, for example, an exception is triggered.

For these and other reasons, sequences of pure random instructions are unlikely to create useful programs. In addition to constraining the verification scenarios to create random but interesting sequences of instructions, you may want all user-defined scenarios to share some useful methods. You can encapsulate these methods in the common instruction scenario base class.

In the common instruction scenario base class you can implement relationships between instruction objects as constraints. For example, you can describe the memory alignment rule using constraints. As part of your planning for exceptions, however, you can turn off this constraint so that instances of misaligned memory addresses are used with the memory load and store operations at random times when the scenarios are randomized.

**Branch Scenario Considerations**

Branches are challenging for automatic instruction generation. The simple program trace shown in Figure 7, for example, has one forward BEQ branch and one backward BNE branch. Random values written to the processor's 32-bit registers are extremely unlikely to result in R1 equaling R2. Thus,
the first BEQ instruction on Line #2 would likely fall through and miss the branch condition evaluation logic. Similarly, R3 would almost never equal R4, so program execution would be stuck in the loop from lines 12 to 8 for a very long time.

Click here for Fig. 7

7. Sample program trace with forward and backward branches.

Fortunately, you can deal with backward and forward branches using constrained scenarios. With constraints, you place restrictions on a sequence of operations so that the probability of a forward branch becomes reasonable and endless loops in backward branches are avoided.

You can greatly increase the probability of a forward branch by initializing the comparing register operands just before the branch operation. For example, line 1 of the example scenario in Figure 8 adds a small positive or negative number (or zero) to R2 and assigns the sum to R1 (R1 = R2 + {-2:2}). The probability of R1 equaling R2 is now 20 percent.

Click here for Fig. 8

8. Idea to increase the taken/not taken probabilities for forward branches.

In terms of constraints, you want the operation preceding a forward branch to be an ADDI (add immediate) operation with the same operands and a small immediate value. Figure 9 shows an example.

Click here for Fig. 9

9. Sample of an ADDI (add immediate) operation.

Similarly, you can handle backward branches by treating them as loop scenarios. In the example in Figure 10, if the immediate value in line 7 is -1, the values of R3 and R4 are equal when compared in line 12. The BNE operation thus falls through and the branch is not taken. If the immediate value in line 7 is less than -1, line 11 brings the register values closer by 1. The branch in line 12 is taken for the next loop iteration until R3 equals R4. Then the branch falls through, exiting the loop. This scenario therefore avoids absurdly long loops while easily creating all conditions between simple fall-through branches to loops with several iterations.
10. *Idea to avoid endless loops for backward branches.*

In terms of constraints, you want the operation preceding a backward branch to be an ADDI operation with the same operands and a small negative value. Increment this operand (which acts as a loop index) by 1 inside the loop, just before the branch operation. Make sure that these two registers are not modified elsewhere inside the loop. In other words, none of the other destination registers can be the same operands for the branch operation. Consider the example shown in *Figure 11.*

11. *Example of a branch operation with different operands.*

The processor testbench also needs to include many other considerations for boundary conditions to prevent cases such as a backward branch "BGT R1, R2, LABEL_X" that is always taken if R2 has the smallest possible number.

While the stimulus-generation strategy centers on the constrained-random approach, it is also important to support directed random or directed stimulus. One class method defined in the typical common scenario class enables you to load a directed scenario from a file containing a pre-assembled program trace. This method has proven useful for providing directed stimuli, such as leveraged tests from the processor's software team.

**Stimulus generation**

Both test scenarios and the testbench's scenario-generator component must be controllable so that you can easily generate stimuli with varying levels of randomness. You can use SystemVerilog to describe the constraints on the list of items in an instruction scenario, for example. The items are described as a dynamic array of instruction objects. Array constraints using the SystemVerilog *foreach* construct are extremely useful to specify scenario-related constraints.

The first type of scenario is the constrained-random scenario. To focus on the arithmetic operations in a long sequence, for example, you can implement a constrained-random scenario that constrains all operations to the computational kind (i.e., without branches, load and stores, etc.).

The second type of scenario, directed random, is also useful for processors, since it is hard to turn up numerical bugs for rounding and overflow conditions with pure random numbers as inputs to arithmetic operations. For example, you can preload part of the data memory with special values such as walking patterns of 0s and 1s, close to 0, close to minimum and maximum numbers, etc. Then you can design a directed-random scenario to read these special values from memory before
executing any other arithmetic instruction stream.

Finally, you need directed scenarios to cover specific functionalities. Specifically, you can implement a scenario class to read a pre-assembled program as a directed scenario.

A scenario generator can work with any of these three types of scenarios. Given a set of scenario objects, a scenario generator randomly selects and randomizes one scenario. The generator repeats this process until reaching a user-specified condition. >p>

**Summary**

The CRV approach outlined here offers an effective way to overcome the limitations of traditional directed tests for verifying microprocessors by taking advantage of object-based randomization using constraints. This approach includes both a top-down stimulus planning process and a bottom-up implementation solution using SystemVerilog. Using this approach can achieve better verification coverage and ultimately save time by supporting testbench reuse.

**About the Author:**
*Jason C. Chen*, is a Staff Design Consultant within Synopsys Professional Services. He has a Bachelor of Electrical Engineering degree from McGill University, Montreal, Canada. He can be contacted at: jasonc@synopsys.com