Automatic Test Program Generation: A Case Study

Article *in* IEEE Design and Test of Computers · April 2004

DOI: 10.1109/MDT.2004.1277902 · Source: IEEE Xplore

4 authors:

Fulvio Corno
Politecnico di Torino
332 PUBLICATIONS **3,571** CITATIONS
SEE PROFILE

Ernesto Sanchez
Politecnico di Torino
152 PUBLICATIONS **861** CITATIONS
SEE PROFILE

Matteo Sonza Reorda
Politecnico di Torino
568 PUBLICATIONS **6,673** CITATIONS
SEE PROFILE

Giovanni Squillero
Politecnico di Torino
256 PUBLICATIONS **2,098** CITATIONS
SEE PROFILE

Automatic Test Program Generation: A Case Study

Fulvio Corno, Ernesto Sánchez,
Matteo Sonza Reorda, and Giovanni Squillero
Politecnico di Torino

Editor's note:
Comprehensive coverage measurement should guide an effective testbench generation approach. Today, feedback from coverage to test generation often requires manual work; it is desirable to implement a framework that automates this feedback process. The authors propose a genetic-algorithm-based evolution framework for testbench generation. It enables small test programs to evolve and effectively capture target corner cases based on the feedback from coverage measurement.
—Li-C. Wang, University of California, Santa Barbara

Design validation is a critical step in the development of present-day microprocessors, and some authors suggest that up to 60% of the design cost is attributable to this activity.[1] Of the numerous activities performed in different stages of the design flow and at different levels of abstraction, this article focuses on simulation-based design validation performed at the behavioral register-transfer level.

Researchers have published formal methods for complex microprocessor designs that had remarkable results.[2-4] However, almost all these techniques suffer from severe drawbacks or need considerable human efforts to handle the entire design of today's processors. Therefore, industry uses formal methods mainly for the validation of a single component or when boundary conditions allow the constraint of the validation task.

Designers typically write assertions inside hardware description language (HDL) models and run extensive simulations to increase confidence in device correctness. Simulation results can be recorded and used for checking local optimizations and running regression tests. Simulation results can also be useful in comparing the HDL model against higher-level references or instruction set simulators.

Microprocessor validation has become more difficult since the adoption of pipelined architectures, mainly because you can't evaluate the behavior of a pipelined microprocessor by considering one instruction (and its operands) at a time; a pipeline's behavior depends on a sequence of instructions and all their operands. The simultaneous execution of multiple instructions in superscalar architectures leads to additional difficulties.[3] Thus, the literature includes several investigations of the functional validation of pipelined architectures. But, because it is necessary to check all the possible interactions among instructions and operands inside the pipeline, the task is not often trivial. Utamaphethai, Blanton, and Shen developed a method for generating very effective instruction sequences for validating the branch prediction mechanism of the PowerPC 604.[5] However, this methodology is not easily applicable to general designs, and it exploits a deep knowledge of the target processor. Differently, Heath and Durbha showed that it is possible to model a pipelined processor with a high-level behavioral HDL description.[6] The authors fitted the model into a moderately sized field-programmable gate array and exploited it for various analyses, including testability. The approach requires skilled experts to perform a large amount of manual work, and its effectiveness depends on how the model captures features; the engineers' opinions can easily bias the model.

Bose and Abraham also review the presilicon validation of microprocessors, suggesting the use of models—such as those using a finite-state-machine-level characterization or input parametric behavior—to reduce the test generation complexity without significant coverage loss.[1]

Shen and Abraham propose an interesting methodology to synthesize a self-test program.[7] Although originally devised to tackle testability, the approach does not rely on any prior fault model and is applicable to func-

0740-7475/04/$20.00 © 2004 IEEE Copublished by the IEEE CS and the IEEE CASS

tional validation as well. It generates a sequence of instructions enumerating all the combinations of the operations and systematically selects operands. However, users must determine the heuristics to assign values to each instruction operand to achieve the desired goal, which might not be a trivial task. Similarly, Kranitis et al. propose a method to generate test programs for microprocessors.[8] However, this method is also mainly suited to manual implementation (it requires a deep knowledge of the processor architecture), and its effectiveness when applied to a complex pipelined architecture remains undemonstrated.

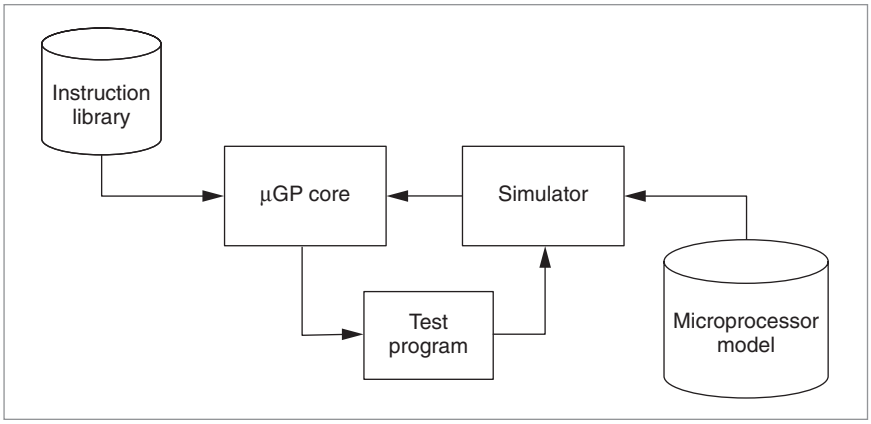

Figure 1. μGP system architecture.

Here, we propose a simulation-based methodology different from these pure or biased pseudorandom methods. Our approach calls for the generation of a set of test programs and optimization using the feedback information from a simulator able to evaluate them with respect to a given coverage metric. This method first requires encoding the syntax of the target assembly language in a very compact format exploitable by an instruction randomizer or other code generators. We use this to generate a set of valid (syntactically correct) assembly programs for use as an initial seed. Then, an evolutionary algorithm that we call μGP automatically optimizes the test set by tuning and modifying the assembly programs. Thanks to the adopted internal representation, μGP fully exploits subroutines and software traps in the generated code.

As a case study, this article considers the Sparc V8,[9] a relatively complex microprocessor with a five-stage pipelined architecture. Results show that our approach can generate an effective test set more efficiently—with a lower computational effort and producing a shorter test set—than a random approach. Some solutions in the automatically generated code appear comparable to the ones that only skilled engineers are capable of finding.

The use of such a method can dramatically help designers and engineers: Instead of checking massive random simulations for differences with respect to the correct model, validation experts can let the automatic test case generator work for the proper time and eventually examine the test set it produces. The proposed method requires very limited human intervention (just to describe the syntax of the target assembly language), and is useful for generating test programs targeted at maximizing a given metric or at covering corner cases. For the experiments reported here, we adopted RTL-instantiated statement coverage, an extended version of statement coverage.

Proposed approach

We have exploited the key idea of combining an evolutionary core and a simulator for different purposes. Earlier, we built a suitable test program by concatenating several predefined macros that we carefully developed in advance and then used a genetic algorithm to choose the optimal values for macro parameters.[10] We first applied the approach to generating test programs for manufacturing testing, showing that this method attained high testable-fault coverage on a simple microprocessor core. However, the use of fixed macros prevents the stimulation of several interactions among the instructions inside a pipeline. The method presented here does not pose such limitations.

Earlier, we exploited a preliminary version of μGP to reach the complete statement coverage of an RTL model of the i8051, a simple microcontroller without a pipeline. We also targeted the i8051 in a study where we used μGP to attain high fault coverage for the stuck-at fault model.[11] Since then, we have completely redesigned the evolutionary core to improve its efficiency, increase its flexibility, and reduce syntax constraints.[12] We also developed a self-adaptation mechanism able to internally tune parameters to their optimal values[13] and presented preliminary results on the use of μGP for the validation of DLX/pII,[14] a simple academic microprocessor.

Figure 1 shows the architecture of the proposed system, which describes the microprocessor assembly language in an instruction library. The μGP core uses a

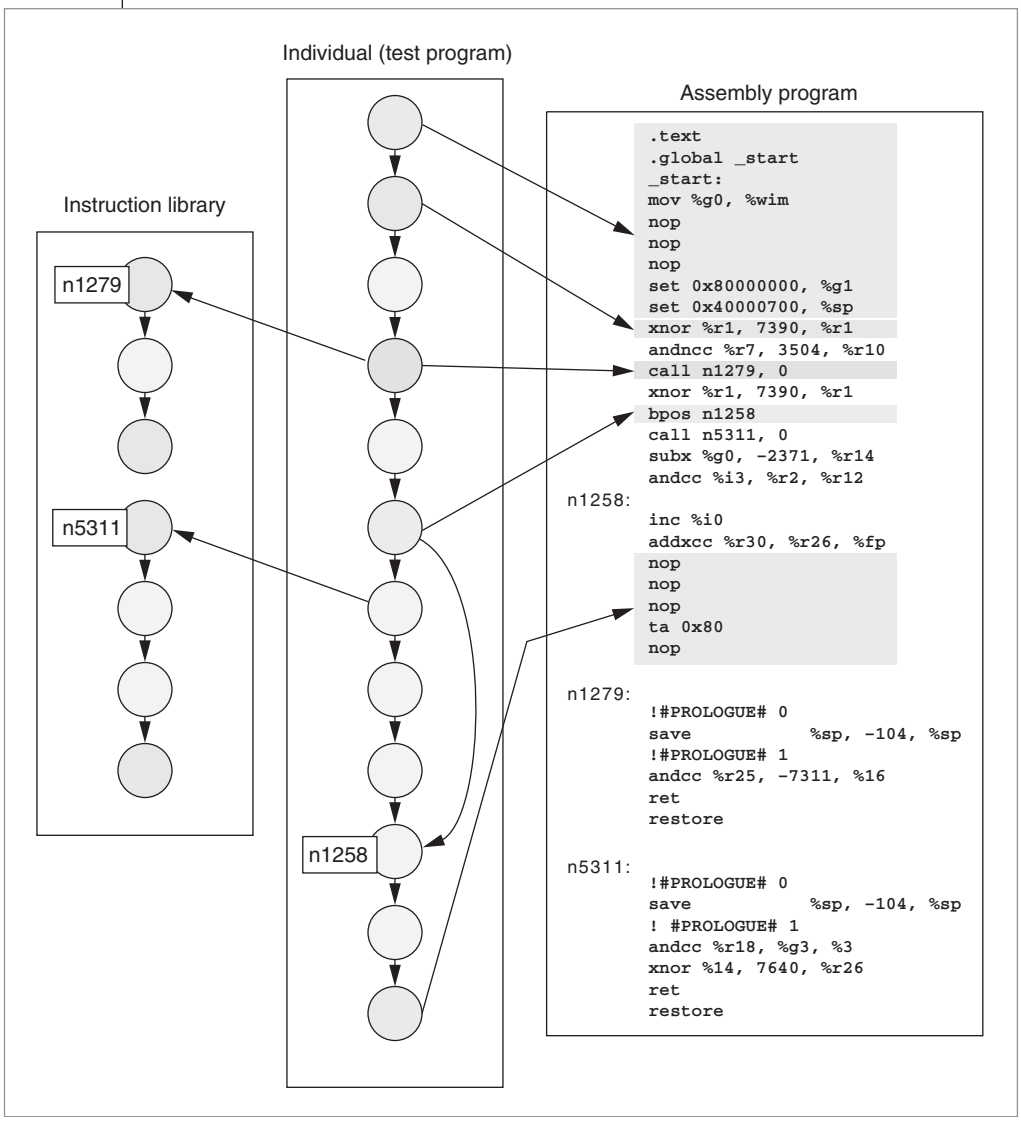

Figure 2. Test program representation.

genetic algorithm to cultivate a population of individuals, where each individual is a program. Then, our system simulates the execution of such programs with an external tool, the simulator, which evaluates the target validation metric and also drives the optimization process. The loose coupling between the instruction library and the generator enables the use of this approach with different instruction sets, formalisms, and metrics.

The instruction library describes the assembly language syntax, listing each instruction with the correct operands. It encodes all syntactically correct instructions, for each instruction listing the type of valid operands (for example, a numeric value used as an immediate value, a register together with its addressing mode, or a label representing the target of a jump). The same assembly instruction can appear several times in the instruction library in association with different operand types.

We internally represent programs as directed acyclic graphs (DAGs) modeling the program's control flow. We permit the consideration of structured primitives, such as procedures, by organizing the DAG as a collection of sub-DAGs of different types—main program, procedures, traps, and so on—called sections. DAGs consist of different types of nodes, which encode the various instruction classes, such as sequential operations, jumps, and branches. µGP maps each DAG node to a valid assembly statement, according to the instruction library. The adopted internal representation guarantees that critical situations such as endless loops and invalid instructions never occur, while not ruling out any legal combination of instructions (such as, for example, jumps, calls, traps, and so on).

Figure 2 shows a test program where the main body calls two different subroutines: n1279 and n5311 (µGP generates all labels automatically, when needed). The n1258 refers to another node in the main section of the program; in this case, the target of a branch.

The test program has an internal representation consisting of a DAG with three sub-DAGs in two different sections: The former contains the main body, while the latter contains the two distinct subroutines. Corno and Squillero offer a more detailed discussion about sections in an earlier work.[12]

In the DAG, nodes represent instructions (apart from the first and the last node, which correspond to prologue and epilogue fragments). Figure 3 shows the mapping of

a single node to an assembly line in the test program. The node contains a pointer to a specific instruction inside the instruction library (an unsigned sum with three registers as operands) and the actual values of its parameters (r25, r18, and r10).

The μGP core implements a genetic algorithm whose goal is to search for an individual—a test program—with the best value of the adopted metric. For this purpose, the algorithm evolves a population of μ individuals in discrete steps, called generations. The effectiveness of a program in terms of the adopted metric determines its probability of survival or, more exactly, the mean number of descendants that it generates. We can set this *fitness* measure to the value of the adopted metric attained by the program. Alternatively, we can have it correspond to a more complex function, permitting us to guide the algorithm in the optimization process.

Figure 4 shows the pseudocode of the adopted evolutionary algorithm. In each generation, the algorithm generates λ new individuals. After creating λ new individuals, the algorithm selects the best μ programs in the new population of λ + μ for survival. The algorithm then chooses parents, randomly selecting a given number of individuals and picking the best ones among them via tournament selection. In each generation, the algorithm generates offspring by applying different operators (mutation and crossover), fully self-adapting the strength of operators. Corno and Squillero present the full details of the evolutionary algorithm in an earlier work.[12]

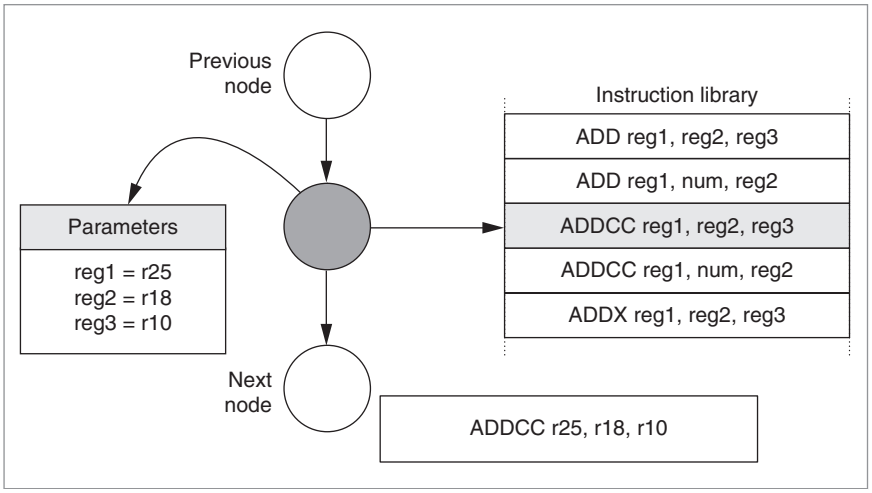

Figure 3. Internal representation (DAG node) of the instruction ADDCC r25, r18, r10. The node's structure contains a pointer to the instruction library and required parameter values. μGP uses this data to map the node to the instruction.

```
evolution() {
        P ← μ random individuals;
        while not stopping conditions {
            repeat λ times {
                A ← select parents in P
                O ← generate offspring from A
                P ← P ∪ O
            }
            sort P on individual fitness
            keep the best μ individuals in P and discard the others
        }
}
```

Figure 4. Pseudocode for μGP.

Leon2 microprocessor

The Leon2 is a synthesizable VHDL model of a 32-bit processor conforming to the IEEE Std 1754 (Sparc V8) architecture. Specifically designed for embedded applications, this processor is highly configurable, and it is available under the GNU Lesser General Public License, allowing free and unlimited use in both research and commercial applications (http://www.gnu.org/copyleft/lesser.html). Although of moderate size, the Leon2 displays several characteristics of modern processor designs.

First, the Leon2 is fully synthesizable with most synthesis tools and is implementable on both FPGAs and ASICs; it is simulatable with all VHDL-87-compliant simulators. The source files amount to about 18,000 VHDL lines, and the synthesized implementation consists of about 35,000 equivalent gates (not including RAM). The model includes separate instruction and data caches, a hardware multiplier and divider, an interrupt controller, a debug support unit with trace buffer, two 24-bit timers, two UART (universal asynchronous receiver/transmitter) ports, power-down function, a watchdog, a 16-bit I/O port, and a flexible memory controller. It is easy to add new modules using the on-chip Amba AHB/APB (advanced high-performance bus/advanced peripheral bus).

The Leon2 model does not include a floating-point unit (FPU) but provides a direct interface to the Meiko

FPU core, and a general interface to connect other FPUs. The general interface provides a generic coprocessor interface to allow for custom mathematical coprocessors. The processor implements a Sparc-V8-compliant integer unit with a single instruction issue pipeline of the following five stages:[9]

- *Instruction fetch*. If the instruction cache is enabled, the fetch unit fetches an instruction from the instruction cache. Otherwise, the cache forwards the fetch to the memory controller. The instruction is valid at the end of this stage and is latched inside the integer unit.
- *Decode*. This stage decodes the instruction and reads its operands. Operands can come from the register file or from internal data bypasses. The branch unit generates call and branch target addresses in this stage.
- *Execute*. In this stage, the integer unit performs arithmetic logic unit (ALU), logical, and shift operations. Address generation occurs in this stage for memory operations (loads and stores) and for the JMPL and RETT instructions.
- *Memory*. Here, the processor accesses the data cache. For cache reads, the data will be valid by the end of this stage, at which point it is aligned as appropriate. At this time, the processor writes store data read out in the execute stage to the data cache.
- *Write*. The write-back unit writes results of any ALU, logical, shift, or cache read operations to the register file.

The Leon2 implements the Sparc *register window* architecture, claimed to allow a significant reduction in the number of required load or store instructions over that of other RISCs, particularly for large application programs. In most cases, the Leon2 can conclude the execution of one instruction per clock cycle. The processor can insert pipeline stall cycles after certain instructions to solve data hazards.

The Leon2 can work in either of two modes: supervisor or user. In supervisor mode, the processor can execute any instruction, including privileged (supervisor-only) instructions; it can also access special (protected) registers. In user mode, an attempt to execute a privileged instruction or access a special register causes a trap to supervisor software.

In the configuration adopted for the experiments, the Leon2 does not include watchdogs, external Peripheral Component Interconnect buses, or UARTs. The integer unit uses the 16×16 integer multiplier and the radix-2 divider. We included caches in the test configuration but excluded them from the validation metric.

Experimental evaluation

We have implemented a prototype μGP in about 5,000 lines of C code, employing Mentor Graphics' ModelSim version 5.7a to simulate the design and to calculate statement coverage.

We also developed the instruction library for the Sparc V8 processor; this library includes about 500 syntactical descriptions of the possible instructions and their operands. Exploiting the idea of sections,[12] μGP can generate valid subroutines and software traps. The latter feature allows executing code in supervisor mode, validating the privileged instructions and protection mechanisms. Library development required about two working days for a programmer.

The validation metric adopted here is the RTL-instantiated statement coverage: the percentage of executed RTL statements over the total when simulating the execution of a given test program. Here, "instantiated" means that we calculate the metric against the elaborated design, rather than on the source description.

Many consider instantiated statement coverage a required starting point for simulation-based processes, and most commercial simulators currently support it. Although more complex and insightful metrics are well known, we selected this one because of its plainness, simplicity in analyzing results, and extensive tool support. However, the methodology we present here can easily exploit other simulator-supported metrics.

After showing the general results, this section describes three fragments of RTL code handling uncommon events. A pseudorandom approach does not easily cover such cases, which would normally require special attention from designers. Experiments show how μGP can automatically and effectively handle these cases.

General results

Our experiments compared the effort required to generate a test set (set of test programs) able to reach complete RTL-instantiated statement coverage and the quality of this coverage. Table 1 shows these results. The approach column reports the methodology name; the simulated column shows the number of instructions simulated during the generation process; and the final column shows the number of instructions in the programs composing the resulting test set.

We compared μGP with an in-house-developed instruction randomizer, exploiting the same internal representation devised for μGP; both methodologies exploit the assembly language description stored in the

instruction library. For all these approaches, we devised test sets able to cover 2,590 out of 5,767 instantiated statements. A deeper analysis enabled pruning statements not coverable in the adopted environment. For example, the pruning eliminated all statements that are executed when the Leon2 receives an external interrupt request because the environment does not generate any interrupt requests. Moreover, we did not use (or synthesize) some hardware, such as certain types of multipliers, because of the configuration adopted. As a result, only 2,590 instantiated statements appear truly coverable and thus all methodologies provide a test set attaining complete (100%) statement coverage.

The instruction randomizer randomly generated a test set by generating programs of about 200 instructions and keeping only programs that covered previously uncovered statements. We selected this program length as optimal after running experiments with µGP. In µGP, the self-adaptation mechanism sets the µGP parameters.

As a result, µGP devised a test set attaining 100% coverage, evaluating fewer than 5 million instructions. The instruction randomizer required the simulation of 8.4 million instructions. Moreover, µGP's test set consisted of about 2,400 instructions, whereas the instruction randomizer's test set contained about 5,300 instructions. This supports the claim that our proposed method is more effective than a purely random one.

The fitness function we used is a direct measure of the test program's attained coverage. These experiments used neither floating-point unit nor coprocessor. However, to test the decoding of mathematical instructions and registers, the µGP-generated code emulates both a floating-point unit and a coprocessor with software interrupts. We then wrote interrupt handlers and set up the interrupt table for traps fp_exception (0x08) and cp_exception (0x28).

We also used µGP to devise a minimal test set. To do so, we slightly modified the fitness function, making µGP calculate the length of the test program (in number of instructions) multiplied by a small constant. We then had µGP subtract this number from the attained coverage. These alterations artificially increased the probability of survival for the descents of individuals that were short. Simultaneously optimizing for both goals—completeness of coverage and shortness—required more iterations, but the resulting test set was only 907 instructions long and still attained 100% coverage. Significantly, to devise such a compact set of programs, µGP evaluated 7.3 million instructions, which is still less than the number required by the instruction randomizer.

Table 1. Result summary.

	No. of instructions	
Approach	**Simulated (millions)**	**Final (thousands)**
Instruction randomizer	8.4	5.3
µGP	4.9	2.4
µGP with minimal test set	7.3	0.907

```
if ((divi.signed and r.neg and r.zero) = '1') and (divi.op1 = Zero) then
v.ovf := '0';
end if;
```

Figure 5. VHDL fragment that clears an overflow flag.

Analysis of special cases

We also worked on other interesting cases, showing how µGP behaved in terms of its ability to cover a rarely occurring corner case, to produce a highly optimized test program, and to discover clever techniques for uncovering specific statements.

Divide unit. The Leon2 divide unit can perform signed and unsigned integer division between 64- and 32-bit values. Let *M* be –2,147,483,648, the smallest integer number adopting a 32-bit two's-complement representation, that is, a single 1 followed by thirty-one 0s in binary format. The integer division is a valid operation yielding the value *M* and generating no overflow. Because of the divide unit's implementation, this operation triggers the overflow flag, and it is necessary to explicitly clear this flag, using the code in Figure 5. This is not a bug and the resulting behavior is correct, but it is a striking example of a corner case.

Obviously, to validate the divide unit and reach complete instantiated-statement coverage, it is sufficient to simulate a limited number of specifically defined, handwritten test cases. This approach, nevertheless, would rely on the engineer's knowledge and could be biased by his beliefs.

Disregarding designer test cases, another method to validate the unit is to perform all divisions. However, this approach would require simulating more than operations and would therefore cause overly long simulations of the RTL model. Another possibility is to rely on pseudorandom simulations, generating small programs that perform random divisions until you reach a certain degree of confidence. In this case (unlike the first), µGP can automatically determine the correct operation and

```
        ble n1639
n1639:
        ldsb [%r16], %r16
```

Figure 6. μGP-optimized test case.

the correct values to cover the statement.

This example shows the potential of the evolutionary approach, which optimizes test cases in a process analogous to local search. Small mutations and crossover enable an effective exploration of the search space.

Memory blocking error. Two consecutive load instructions—where the second instruction uses the data loaded by the first—lead to a blocking error, causing a data_access_exception trap (0x09). Both the instruction randomizer and μGP can smoothly generate this situation. However, μGP generates the trap in a rather peculiar way, shown in Figure 6. This particular example is for a branch-on-less-or-equal (BLE) instruction immediately followed by an instruction-load-signed-byte (LDSB) instruction where both source and destination registers are r16.

As a result, these few lines stress both the data access exception logic and the control transfer logic. In fact, the processor tries to execute two consecutive load instructions. In this case, the second instruction uses the data loaded by the first in register r16, and the first instruction is in the delay slot of a conditional branch.

This case shows the effectiveness of μGP in optimizing test cases and providing an extremely compact test set.

Register window overflow/underflow. The Leon2 implements a configurable register window architecture, with two to 32 windows, according to the Sparc standard. To fully validate the register window logic, it is necessary to trigger at least a window_overflow trap (0x05) and a window_underflow trap (0x06).

As in the previous example, it would be easy to write a test case by hand once the mechanism and its description are known: It is sufficient to call a number of nested subroutines greater than the number of register windows or to execute a recursive subroutine. Similarly, an unmatched Restore instruction can cause a window underflow. Nevertheless, the experiment's purpose is to automatically generate the test program to automatically validate the register window logic.

We prevent μGP from generating nested subroutines, and in the present version it cannot produce recursive code. Moreover, because all generated programs are syntactically correct, μGP cannot generate an unmatched Restore. A possible workaround would be to explicitly add the capability for generating enough nested subroutines to trigger window overflow. However, these solutions are specific to this case, and they do not differ from the handwritten test programs.

Remarkably, μGP managed to generate both exceptions without nested subroutines or an unmatched Restore. Instead, the generated test program exploited software traps: When the processor is running in supervisor mode, it patches the window invalid mask (WIM). Then, a single Call or Restore instruction can trigger a register window overflow or underflow. The system discovered this solution without any hints from designers.

This case, as does the first one, shows the potential of the evolutionary approach: μGP can find interesting solutions that even designers could hardly have imagined.

Currently, we are analyzing new metrics, such as branch coverage or expression coverage. We are evaluating the effect of targeting different coverage metrics with respect to validation and test.

We are also trying to use the hardware performance counters of modern microprocessors to generate test programs to target specific microarchitectural events. In this new approach, μGP evaluates candidate test programs that run directly on the target microprocessor: Programs are not simulated, but rather executed. The fast evaluation will let us target complex designs, such as the Intel Pentium 4.

We are also developing a system that allows μGP to optimize an existing set of test programs. ■

Acknowledgment

This work has been partially supported by Intel through the grant for GP Based Test Program Generation.

References

1. P. Bose and J.A. Abraham, "Performance and Functional Verification of Microprocessors," *Proc. 13th Int'l Conf. on VLSI Design* (VLSI 00), IEEE CS Press, 2000, pp. 58-63.
2. N.A. Harman, "Verifying a Simple Pipelined Microprocessor Using Maude," Lecture Notes in Computer Science 2267, Springer-Verlag, 2001, pp. 128-142.
3. D. Van Campenhout, T.N. Mudge and J.P. Hayes, "High-Level Test Generation for Design Verification of Pipelined Microprocessors," *Proc. 36th Design Automation Conf.* (DAC 99), ACM Press, 1999, pp. 185-188.

4. M.N. Velev and R.E. Bryant, "Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exception, and Branch Prediction," *Proc. 37th Design Automation Conf.* (DAC 00), ACM Press, 2000, pp. 112-117.
5. N. Utamaphethai, R.D. Blanton, and J.P. Shen, "Superscalar Processor Validation at the Microarchitecture Level," *Proc. 12th IEEE Int'l Conf. VLSI Design* (VLSI 99), IEEE CS Press, 1999, pp. 300-305.
6. J.R. Heath and S. Durbha, "Methodology for Synthesis, Testing, and Verification of Pipelined Architecture Processors from Behavioral-Level-Only HDL Code and a Case Study Example," *Proc. IEEE Southeast Conf.*, IEEE Press, 2001, pp. 143-149.
7. J. Shen and J.A. Abraham, "Native Mode Functional Test Generation for Processors with Applications to Self-Test and Design Validation," *Proc. IEEE Int'l Test Conf.* (ITC 98), IEEE CS Press, 1998, pp. 990-999.
8. N. Kranitis et al., "Application and Analysis of RT-Level Software-Based Self-Testing for Embedded Processor Cores," *Proc. Int'l Test Conf.,* IEEE CS Press, 2003, pp. 431-440.
9. *The SPARC Architecture Manual Version 8,* SPARC Int'l, 1992; http://www.sparc.com/standards/V8.pdf**.**
10. F. Corno et al., "On the Test of Microprocessor IP Cores," *Proc. IEEE Design, Automation and Test in Europe* (DATE 01), IEEE CS Press, 2001, pp. 209-213.
11. F. Corno et al., "Fully Automatic Test Program Generation for Microprocessor Cores," *Proc. IEEE Design, Automation and Test in Europe* (DATE 03), IEEE CS Press, 2003, pp. 1006-1011.
12. F. Corno and G. Squillero, "An Enhanced Framework for Microprocessor Test-Program Generation," *Genetic Programming: 6th European Conf., EuroGP 2003*, Lecture Notes in Computer Science 2610, Springer-Verlag, 2003, pp. 307-315.
13. F. Corno, M. Sonza Reorda, and G. Squillero, "Automatic Test Program Generation for Pipelined Processors," *Proc. ACM Symp. Applied Computing* (SAC 03), ACM Press, 2003, pp. 736-740.
14. F. Corno, G. Squillero, and M. Sonza Reorda, "Code Generation for Functional Validation of Pipelined Microprocessors," *Proc. IEEE European Test Workshop* (ETW 03), IEEE CS Press, 2003, pp. 113-118.

Fulvio Corno is an associate professor at the Politecnico di Torino. His research interests include the test of digital systems, high-level design in automotive electronics, fault injection, and genetic algorithms. Corno has an MS in electronic engineering and a PhD in computer science from the Politecnico di Torino. He is a member of the IEEE.

Ernesto Sánchez is a PhD student at the Politecnico di Torino. His research interests include test techniques for advanced processors and evolutionary algorithms. Sánchez has a degree in electronic engineering from Universidad Javeriana, Colombia. He is a member of AJE Asociación de Profesionales Javerianos en Europa.

Matteo Sonza Reorda is a full professor in the Department of Computer Science at the Politecnico di Torino. His research interests include testing and fault-tolerant design of electronic systems. Sonza Reorda has an MS in electronic engineering and a PhD in computer science from the Politecnico di Torino. He is the chair of the committee on test program generation for the 2004 Conference on Design, Automation and Test in Europe. He is a member of the IEEE.

Giovanni Squillero is an assistant professor at the Politecnico di Torino. His research interests include combining evolutionary techniques with verification, testing, and the fault-tolerant design of electronic systems. Squillero has an MS and a PhD in computer science engineering from the Politecnico di Torino. He organizes the special sessions on evolutionary design automation for the Congress of Evolutionary Computation and the EvoNET Workshop on Evolutionary Hardware Optimization Techniques. He is a member of the IEEE.

■ For questions and comments about this article, contact Giovanni Squillero, Politecnico di Torino, Dip. Automatica e Informatica, Corso Duca degli Abruzzi 24, 10129 Torino, Italy; giovanni.squillero@polito.it.