Evolutionary Test Program Induction for Microprocessor Design Verification

Fulvio Corno, Gianluca Cumani, Matteo Sonza Reorda, Giovanni Squillero

Politecnico di Torino
Dipartimento di Automatica e Informatica
http://www.cad.polito.it/

Abstract*

Design verification is a crucial step in the design of any electronic device. Particularly when microprocessor cores are considered, devising appropriate test cases may be a difficult task. This paper presents a methodology able to automatically induce a test program for maximizing a given verification metric. The methodology is based on an evolutionary paradigm and exploits a syntactical description of microprocessor assembly language and an RT-level functional model. Experimental results show the effectiveness of the approach.

1. Introduction

Design verification is an essential activity in the design of any electronic device. Concerning microprocessors, it has been maintained that about a third of the cost of developing a new product is devoted to hardware debugging and testing [1]. Reviewing all design verification aspects would deserve a very long discussion, but it is possible to distinguish between two classes of approaches: *formal* and *simulation-based*. Formal methods try to verify the correctness of a system by using mathematical proofs, whereas simulation-based design verification tries to uncover design errors by detecting a circuit faulty behavior when deterministic or pseudo-random tests are applied. Formal methods implicitly consider all possible behaviors of the models representing the system and its specification, and the accuracy and completeness of the system and specification models, as well as required computation resources, are a fundamental limitation. On the contrary simulation-based methods do not suffer from the same constraints, but can only consider a limited range of behaviors and will never achieve 100% confidence of correctness.

Microprocessor and microcontroller complexity usually prevents the straightforward usage of formal or semi-formal techniques [2] or equivalence checking [3]. Exact approaches have been successfully exploited on specific portions or on simplified high-level models, but they are hardly applicable to complete RT-level descriptions. In addition, the verification process normally requires a deep knowledge of the processor architecture. Common pseudo-random strategies cannot be easily exploited since only correct programs can internally perform meaningful operations. As a consequence, manufacturers commonly rely on simulation-based methods where test cases are meticulously written by hand.

However, hand-written test cases can be only exploited as a first line of defense against bugs, since they focus on basic functionalities and important but rarely-occurring corner cases. During the whole design process, exhaustive or nearly-exhaustive tests are often necessary, but the effort required to manually generate them may be practically unworkable.

Today, advances in simulation and emulation technology enabled the use of other sources of test stimuli such as existing application and system software [1]. Additionally, to increase test productivity sophisticated test-generation systems have been proposed [4] [5]. Nevertheless, although all these approaches can significantly increase design productivity, they are still biased towards corner cases, far from being fully automated and not broadly exploitable.

This paper presents a new methodology for generating a test case for validating a microprocessor, where the test case is an assembly program able to maximize a predefined verification metric. Design verification of on-chip peripherals, such as timers or counters is not considered here.

The proposed approach exploits an evolutionary paradigm called *genetic programming* (GP) for generating the test program. The algorithm relies on a syntactical description of the assembly language implemented by the processor and is able to induce test cases for efficiently maximizing the verification metric without human intervention. At the end of the process, the test

* This work has been partially supported by Center of Excellence on Multimedia Radio communications (CERCOM) of Politecnico di Torino and by Agenzia Spaziale Italiana.

program can be simulated and allows designers to identify those parts of the description that the tool failed to validate, and therefore possibly require a more detailed analysis.

Evolutionary algorithms have been already successfully exploited both for validation [6] and test [7]. They demonstrated the ability to tackle large designs with few limitations. Both [6] and [7] exploited *genetic algorithms* (GAs) to cultivate sequences of input vectors (i.e., variable-length bit strings). The proposed approach aims at generating syntactically correct *test programs*, and GAs are not exploitable.

The proposed approach was tested on the i8051 microcontroller using *statement coverage* as a verification metric. The i8051, despite its relatively old age, is still one of the most popular microprocessors and can be considered a good example of a small microcontroller. However, the methodology is applicable to more complex designs, like pipelined microprocessors, and can easily exploit different design verification metrics, such as branch, conditional or path coverage.

Next Section introduces simulation-based design verification, and illustrates the proposed evolutionary approach. Section 3 details the case study, while Section 4 shows experimental results. Section 5 concludes the paper.

2. Simulation-Based Design Verification

Given an RT-level description of a microprocessor, simulation-based verification requires a test program and a tool able to simulate its execution. The goal is to uncover all design errors, and the effectiveness of the test program is usually evaluated with regards to some metric.

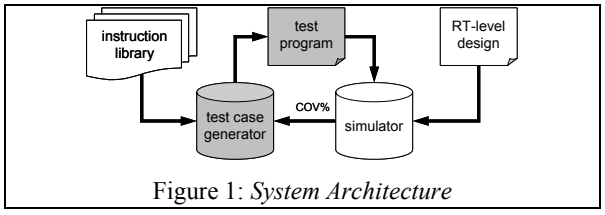

Figure 1: *System Architecture*

As reminded before, the verification metric exploited in this paper is the *statement coverage*. To avoid any confusion, in the following the term *instruction* will denote an instruction in an assembly program, while the term *statement* will refer to a statement in an RT-level description. The verification metric exploited in this paper measures the percentage of executed RT-level statements over the total.

While statement coverage is one of the simplest metrics, it can be considered as a required starting point for any design verification process. Statement coverage analysis ensures that no part of the design missed functional test during simulation, as well as reducing simulation effort from "over-verification" or redundant testing. Moreover, use of coverage analysis provides an easy and objective way of measuring simulation effectiveness to ensure that all bugs would be exposed with the minimum amount of effort. Indeed, most CAD vendors have recently added code-coverage features to their simulators.

Designers are used to write their own test programs to check the basic functionalities and critical corner cases. Devising test cases is a difficult, time-consuming task. In addition, while writing a test program requires a deep knowledge on the microprocessor architecture, the designer is not the ideal verification engineer since he can be biased by his expectation, failing to check for some errors. Hand-written test cases are definitely required, but are not sufficient in the verification process.

The simplest method to obtain an assembly test program is probably to compile a high-level routine. This approach relies on a compiler or cross-compiler, but this requirement may be easily met. Despite their effortlessness, compiled problem-specific algorithms are not the best solution. They are severely inadequate to uncover design errors: due to the intrinsic nature of the algorithms or of compiler strategies, they are seldom able to execute all statements in a description testing all functionalities.

A better strategy is to generate random assembly programs. This approach is likely to cover more statements in the description, but is less straightforward. A random generated program may easily contain illegal operations, such as division by zero, or endless loops. However, the effort required to generate a syntactically correct assembly source is still moderate. The main drawback of this method is that a random program will hardly cover all corner cases, hence resulting in low statement coverage. In order to obtain a sufficient coverage a large number of long programs are needed, resulting in overlong simulation times.

This paper presents a new approach for devising a test program, based on an evolutionary algorithm. The induction of the test case is fully automated and only requires a syntactical description of the assembly implemented by the microprocessor. The resulting program is reasonably short and maximizes the target verification metric.

The overall architecture is shown in Figure 1. The microprocessor assembly language is described in an *instruction library*, and the *test case generator* gener-

ates efficient test programs exploiting it. The execution of each assembly program is simulated with an external tool, and the corresponding statement coverage is used to drive the optimization process.

Next sections better detail the approach.

2.1. Genetic Programming for Test Program

Genetic Programming (GP) was defined as a domain-independent problem-solving approach in which computer programs are evolved to solve, or approximately solve, problems [8]. GP addresses one of the more desired goals of computer science: creating, in an automated way, computer programs able to solve problems.

Traditional GP induced programs are mathematical functions that, after being evaluated, yield a specific result. The pioneering ideas of generating *real* (Turing complete) programs date back to [9]. More recently [10] [11] suggested to directly evolve programs in machine-code form for completely removing the inefficiency in interpreting trees. A genome compiler has been proposed in [12], which transforms standard GP trees into machine code before evaluation.

This paper exploits a versatile GP-like approach for inducing assembly programs presented in [13]. The methodology exploits a directed acyclic graph (DAG) for representing the *flow* of the program (Figure 2). The DAG is built with four kinds of nodes: prologue (followed by 1 child), epilogue (followed by no children), sequential instruction (followed by 1 child), and branch (followed by 2 children).

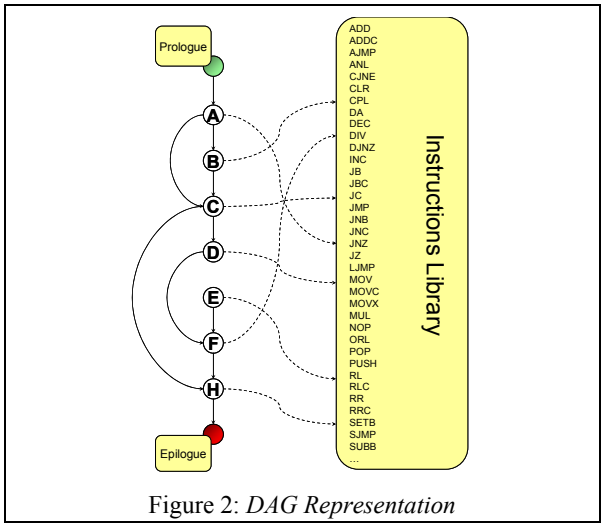

Figure 2: *DAG Representation*

- The **prologue** and **epilogue** nodes are always present and represent required operations, such as initializations. They depend both on the processor and on the operating environment, and they may be empty. The prologue has no parent node, while the epilogue has no children. These nodes may never be removed from the program, nor changed.
- **Sequential-instruction** nodes represent common operations, such as arithmetic or logic ones (e.g., node **B**). They are always followed by exactly one child. The number of parameters changes from instruction to instruction, following assembly specification. *Unconditional* branches are considered sequential, since execution flow does not split (e.g., node **D**).
- **Conditional-branch** nodes are translated to assembly-level conditional-branch instructions (e.g., node **A**). All common assembly languages implement some *jump-if-condition* mechanisms. All conditional branches implemented in the target assembly languages are included in the library.

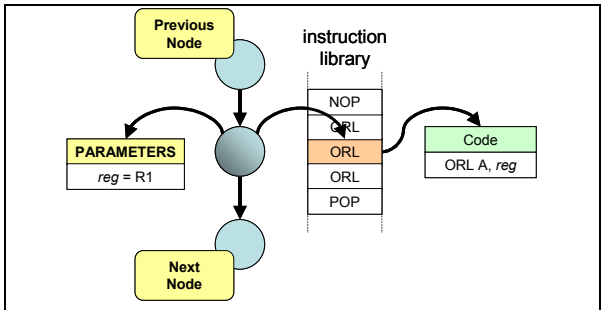

Figure 3: *Sequential Node*

Each node contains a pointer inside the instruction library and, when needed, its parameters (i.e., operand values or register specifications). For instance, Figure 3 shows a sequential node that will be translated into a "*ORL A, R1*" intruction, i.e., a bite-wise OR between accumulator and register R1. The instruction library may contain different entries corresponding to the same instruction. For instance, the entry referring to "*ORL* A, *addr*", where the *addr* parameter is the data RAM address, is different from the entry "*ORL* A, *reg*" where the *reg* parameter is a register used either as a direct or indirect specification.

The DAG is always translated to a syntactically correct, loop-less assembly program, although it is not possible to infer its semantic meaning. An induced program may perform operations on any register and any memory locations, and this exceptional freedom is essential to generate test programs.

The library approach developed in [13] enables to exploit the genetic core and the DAG structure with different microcontrollers or microprocessors that not

only implement different instruction sets, but also use different formalisms and conventions. Indeed, the method has been already successfully tested with a DLX, an academic microprocessor implementing a 5-stage pipeline [14] and a SPARC [15].

Adopted DAG representation prevents backward branches, either conditional or unconditional. This characteristic guarantees program termination, since no endless loop may be implemented, but introduces a small reduction in semantic power.

2.2. Test Program Induction

Test programs are induced by mutating the DAG topology and by mutating parameters inside DAG nodes. Both kinds of modifications are embedded in an evolutionary algorithm implementing a (μ+λ) strategy.

In more details, a population of μ individuals is cultivated, each individual representing a test program. In each step, λ new individuals are generated by mutating existing ones, parents are selected using tournament selection with tournament size τ. After creating new λ individuals, the best μ programs in the population of (μ+λ) are selected for surviving. The initial population is generated creating μ empty programs (only prologue and epilogue) and then applying i_m consecutive random mutations to each.

Three mutation operators are implemented and are chosen with equal probability:

- **Add node:** a new node is inserted into the DAG. The new node can be either a sequential instruction or a conditional branch. In both cases, the instruction referred by the node is randomly chosen. If the inserted node is a branch, either unconditional or conditional, one of the subsequent nodes is randomly chosen as the destination. Remarkably, when an unconditional branch is inserted, some nodes in the DAG may become unreachable (e.g., node **E** in Figure 2).
- **Remove node:** an existing internal node (except prologue or epilogue) is removed from the DAG. If the removed node was the target of one or more branch, parents' edges are updated.
- **Modify node:** all parameters of an existing internal node are randomly changed. Parameters include immediate values and register specifications.

The evolution process iterates until population reaches a *steady state* condition, i.e., no improvements are recorded for S_g generations.

2.3. Test Program Evaluation

Individuals are evaluated by simulation. The DAG is first translated into a syntactically correct assembly program and assembled to machine code. Then the execution of the test case is simulated on the RT-level description of microcontroller, gathering verification metric figures. The final value is considered as the *fitness* value of the individual, i.e., the extent to which it is able to produce offspring in the environment.

Fitness values are used to select λ parents for generating new offspring through a tournament of size τ (i.e., τ individuals are randomly selected and the fittest one is picked). Moreover, fitness values are used to deterministically select the best μ individuals out of the (μ+λ) ones at the end of each evolution step.

Test program evaluation does not consider the internal structure of the microprocessor, nor it includes *hints* for increasing the coverage based on designers' knowledge.

3. Case Study: the i8051 Microcontroller

The proposed approach was tested on the i8051 microcontroller using the *statement coverage* as a verification metric.

Despite its relatively old age, the i8051 is one of the most popular microcontrollers in use today, and many derivative microcontrollers are based on it. The i8051 is an 8-bit microprocessor originally designed in the 80's by Intel that has gained great popularity since its introduction. Its standard form includes several on-chip peripherals, including timers, counters, and UART's, plus 4 Kbytes of on-chip program memory and 128 bytes of data memory, making single-chip implementations possible.

The i8051 memory architecture includes 128 bytes of data memory that are accessible directly by its instructions. A 32-byte segment of this 128-byte memory block is bit addressable by a subset of the i8051 instructions, namely the bit-instructions. External data memory of up to 64 Kbytes is accessible by a special "*MOVX*" instruction. Up to 4 Kbytes of program instructions can be stored in the internal memory of the i8051, or the i8051 can be configured to use up to 64 Kbytes of external program memory. The majority of the i8051's instructions are executed within 12 clock cycles.

3.1. Instruction Library

The i8051 instructions range from 0-operand ones, like *"DIV AB"* (divide accumulator A by B) where all operands are implicit, to 3-operand ones, like the "*CJNE Op1, Op2, RelAddr"* (compare Op1 with Op2 and jump if they are not equal). The i8051 allows 5 different addressing types: *immediate*, *direct*, *indirect*, *external direct* and *code indirect*. As in many CISC,

registers are not orthogonal to the instructions and addressing modes.

The instruction library for the i8051 consists in 81 entries. Prologue, epilogue, 66 sequential operations and 13 conditional branches. All instructions related to subroutine call and interrupt call were not considered for the work described here, and the corresponding blocks in the description were not considered during statement coverage calculation.

4. Experimental Results

A prototype of the proposed approach has been developed in ANSI C language in about 1,600 lines of code. The prototype exploits *Modelsim* v5.5a by Model Technology for simulating the design and getting coverage figures.

The methodology was tested on a synthesizable RT-level implementation of the i8051 core consisting in about 7,500 VHDL lines (the corresponding gate-level netlist is about 12K gates). An external data RAM of 2 Kbytes was connected to the i8051, while no external program memory was used.

NAME	MODULE	LINES	STMS
CTR	Processor core	5,206	2,121
ALU	Arithmetic Logic Unit	429	226
DEC	Decoder	270	220
XRM	External SRAM interface	77	11

Table 1: *i8051 RT-level description*

Four main blocks can be found in the RT-level description (Table 1): the processor core control logic (CTR), the arithmetic and logic unit (ALU), the instruction decoder (DEC), and the external SRAM interface (XRM). The CTR is described behaviorally as a sequential logic block; the ALU is described behaviorally as a combinational logic block; the DEC is described as a data-flow implementing a combinational logic block; the XRM models an external SRAM. For each block Table 1 reports the total number of VHDL lines [LINES] and the number of statements [STMS].

Inducing a test program with the proposed GP required about 10,000 generations, corresponding to the evaluation of about 100,000 programs. The experiments employed about 12 hours of CPU time on a SPARC ULTRA Workstation at 400MHz with 2GB of RAM. Table 2 shows the parameters adopted.

To assess the efficiency of the proposed method, the induced test program was compared with 5 programs devised with 3 different methodologies: compiled problem-specific algorithms; random test programs; and exhaustive functional test case. Table 3 compares the different programs in term of required program ROM bytes [SIZE] and instructions executed by the program [INST]. Statement coverage figures are reported in column [TOTAL]. Statistics are also detailed for the 4 blocks [ALU], [CTR], [DEC] and [XRM].

PAR	MEANINGS	VALUE
μ	Population size	5
λ	Offspring size	10
τ	Tournament size (selective pressure)	2
i_m	Initial mutations	100
S_g	Steady state	500

Table 2: *GP parameters*

The two problem-specific algorithms are *Fibonacci* and *int2bin*. The former calculates the Fibonacci series, while the latter converts an integer to a binary representation. As expected, the coverage figure is quite low. Looking at the decoder, it may be easily inferred that only a subset of the instruction set is used. Both programs execute loops (the number of executed instructions is higher than the number of stored ones). Neither program accesses the external data RAM.

Two different random test programs were considered. The former, *Random (size = GP)*, was devised using the same effort as the GP. 100,000 random programs of approximately 500 instructions were generated and the best one was chosen. The comparison allows evaluating the effectiveness of the evolutionary core in driving the search process. The latter, *Random (size = 4K)*, was devised generating 100,000 random programs that filled almost all program ROM space and selecting the best one. It is included here to allow an estimation of the best result attainable with the random approach disregarding efficiency.

Finally, *TestAll*, an exhaustive functional test program, is considered. The test case was devised by the microprocessor designer, it is relatively long and includes several loops. It tests all possible instructions, although it is not able to check all possible corner cases in the implementation. For instance, “DIV AB” when A is less or equal than B was not taken into account.

The GP induced test program got the highest statement coverage figure with the smallest size and the lowest run time. Execution of the test case is fast, since there are no loops by construction. Remarkably, there are only 7 lines in the circuit description (1 in the ALU and 6 in the CTR) that the GP is not able to cover. These lines are executed when a conditional branch, such as *DJNZ* or *CJMP* jumps to a location preceding the current one, and these backward jumps cannot be generated by the current DAG representation.

5. Conclusions

This paper presents a methodology able to automatically induce an assembly test program for a microcontroller. The methodology is based on an evolutionary paradigm, called *genetic programming*, and exploits the syntactical description of the language. The generated test case is able to efficiently maximize a given verification metric.

A prototype of the proposed approach has been developed in ANSI C language; the method has been evaluated on a synthesizable RT-level implementation of the i8051 microprocessor using statement coverage as verification metric. Induced test programs consistently outperformed test cases devised with alternative methodologies.

Experimental results show the efficiency of the methodology. Devising a test case able to reach the complete statement coverage is a difficult task, even on a small microcontroller like the i8051. Reported data show that random programs can hardly test all corner cases and also long, carefully designed hand-made test cases may not be exhaustive. The automatically induced test program, conversely, was able to cover almost all (99.7%) of the statements in the description.

Current work is targeted to apply the proposed approach to more complex processors and overcome the semantic limitations introduced by the DAG constrain. A current implementation already includes limited subroutine call and backward branch support, and its effectiveness is presently being tested.

6. References

[1] J. Kumar, "Prototyping the M68060 for concurrent verification", *IEEE Design & Test*, Vol. 14, No. 1, 1997, pp. 34–41

[2] M. Yoeli, *Formal Verification of Hardware Design*, IEEE Computer Society Press, 1990.

[3] S.-Y. Huang, K.-T. Cheng, *Formal Equivalence Checking and Design Debugging,* Kluwer, 1998

[4] A. K. Chandra et al., "AVPGEN - a test generator for architecture verification", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 3, 1995, pp. 188–200

[5] A. Aharon et al. "Verification of the IBM RISC System/6000 by dynamic biased pseudo-random test program generator", *IBM Systems Journal*, 1991, pp. 527–538

[6] F. Corno, A. Manzone, A. Pincetti, M. Sonza Reorda, G. Squillero, "Automatic Test Bench Generation for Validation of RT-level Descriptions: an Industrial Experience," *DATE2000: IEEE Design, Automation and Test in Europe*, 2000, pp. 385-389

[7] F. Corno, M. Sonza Reorda, G. Squillero, "High-Level Observability for Effective High-Level ATPG," *VTS2000: 18th IEEE VLSI Test Symposium*, 2000, pp. 411-416

[8] J. R. Koza, "Genetic programming", *Encyclopedia of Computer Science and Technology*, vol. 39, Marcel-Dekker, 1998, pp. 29-43

[9] R. M. Friedberg, "A Learning Machine: Part (I)", *IBM Journal of Research and Development*, vol. 2, n. 1, 1958, pp 2-13

[10] P. Nordin, "A compiling genetic programming system that directly manipulates the machine code," *Advances in Genetic Programming*, 1994, pp. 311-331

[11] P. Nordin W. Banzhaf, "Evolving Turing-complete programs for a register machine with self-modifying code", *Genetic Algorithms: Proceedings of the 6th International Conference*, 1995, pp. 318-325

[12] A. Fukunaga, A. Stechert, D. Mutz, "A genome compiler for high performance genetic programming", *Genetic Programming 1998: Proceedings of the 3rd Annual Conference*, 1998, pp. 86-94

[13] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero, "Efficient Machine-Code Test-Program Induction", *CEC2002: Congress on Evolutionary Computation*, 2002, pp. 1486-1491

[14] D. A. Patterson and J. L. Hennessy, *Computer Architecture - A Quantitative Approach, (second edition)*, Morgan Kaufmann, 1996

[15] SPARC International, *The SPARC Architecture Manual (Version 8)*, Prentice Hall, 1992

NAME	PROGRAM		STATEMENT COVERAGE				
	SIZE	INST	ALU	CTR	DEC	XRM	TOTAL
Fibonacci	324	1,176	49.6%	30.2%	62.7%	81.8%	34.7%
int2bin	81	572	49.6%	21.3%	56.4%	81.8%	27.1%
Random (size = GP)	648	334	86.7%	87.6%	93.2%	100.0%	88.2%
Random (size = 4K)	4,044	2,356	96.5%	94.6%	97.7%	100.0%	95.0%
TestAll (exhaustive)	2,834	52,953	95.1%	99.4%	100.0%	100.0%	99.1%
GP Induced	469	228	99.6%	99.7%	100.0%	100.0%	99.7%

Table 3: *Experimental Results*