Automatic Test Program Generation from RT-level microprocessor descriptions

F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero
Politecnico di Torino
Dipartimento di Automatica e Informatica
Torino, Italy
http://www.cad.polito.it/

Abstract

The paper addresses the issue of microprocessor and microcontroller testing, and follows an approach based on the generation of a test program. The proposed method relies on two phases: in the first, a library of code fragments (named macros) is generated by hand based on the knowledge of the instruction set, only. In the second phase, an optimization algorithm is run to suitably select macros and values for their parameters. The algorithm only relies on RT-level information, and exploits a suitable RT-level fault model to guide the test program generation. A major advantage of the proposed approach lies in the fact that it does not require any knowledge about the low level implementation of the processor. Experimental results gathered on an i8051 model using a prototypical implementation of the approach show that it is able to generate test programs whose gate-level fault coverage is higher than the one obtained by comparable gate-level ATPG tools, while the computational effort and the length of the generated test program are similar. The method is thus suitable to be applied during the incoming inspection test phase performed on small processors, and for developing implementation-independent test suites for soft IP cores.

1. Introduction

Microprocessors and microcontrollers are known to be major challenges in the test arena, due to their complexity and heterogeneity. Techniques for microprocessor testing can be first divided in two groups, depending on whether implementation information are available (for microprocessor producers) or not (when users implement producer-independent incoming inspection test). In the latter case, only high-level functional information are available, and test solutions can not rely on any knowledge about the real implementation of the device. A similar situation arises when soft IP cores are designed, and suitable input sequences are required, able to test them no matter the technology re-mapping and the environment the core is embedded in.

In both the above cases, any Design for Testability technique can hardly be considered, and an effective solution is to devise a test program to be executed by the microprocessor itself. Its behavior must be monitored, and possible mismatches signal the existence of one or more faults inside the processor.

Traditionally, the test of a microprocessor has been performed by resorting to functional approaches based on exciting all the functions and resources described in its data-sheets [3]. This approach involves a high amount of manual work performed by skilled programmers, and does not provide any quantitative measure about the attained Fault Coverage (FC). Recently, Dey et al. proposed a deterministic method named DEFUSE [4] to generate test programs able to reach a good Fault Coverage on the ALU of a microprocessor, and to compact the result. The approach is very effective with combinationally testable parts (e.g., simple ALUs), but shows some limitation when hard-to-test sequential modules, such as Control Units, are addressed. Another approach has been proposed by Batcher and Papachristou [5] that is based on generating random sequences of instructions, but it requires the insertion of additional hardware in the microprocessor under test. Recently, Sheen et al. proposed a technique where the processor itself generates test at run-time by self-modifying code [6]. On the other hand, Utamaphethai et al. showed a method for generating instruction sequences for validating the branch prediction mechanism of the PowerPC604 [7]. Generated sequences are very effective, but the methodology exploits a deep knowledge of the processor and is not straightforward to be applied on general designs.

In this paper we propose a new approach which does not require any knowledge about low-level implementation of the processor: only an RT-level description is required. The method partly stems from the ideas already introduced in [2], but thanks to the adoption of an effective RT-level fault model [1], any reference to the gate-level netlist is avoided. The proposed method requires a limited amount of manual

work aimed at developing a library of *macros*, that are able to excite all the functions of the processor and to make the effects of possible faults observable. A macro is required for every machine-level instruction; each macro is composed of few instructions, aimed at activating the target instruction with some generic operand values representing the macro parameters, and to propagate to an observable memory position the results of its execution. The complexity of the work for developing these macros and the required skills are much lower than for the approaches based on functional testing, such as [3]; in fact, our approach just requires the development of one macro for every machine-level instruction according to a simple pre-defined skeleton for every group of instructions, and does not involve the extraction of complex graphs describing the relationships among resources, as in [3]. The final test program is composed of a proper sequence of macros taken from this library, each activated with proper values for its parameters (i.e., the operands of the composing instructions). This phase is accomplished by resorting to a Genetic Algorithm which exploits an RT-level Fault Simulator to evaluate the generated solutions. Experimental results supporting the effectiveness of the method are reported for a core of the Intel 8051 microcontroller using a prototypical implementation of our algorithm. A synthesizable VHDL RT-level description of the microprocessor is used. Final figures show that the test program generated by the tool has a higher effectiveness (in terms of attained gate-level fault coverage) than the one generated by the gate-level test program generation method introduced in [2], and that the required computational effort is comparable between the two approaches.

The paper is organized as follows. Section 2 outlines some basic concepts about the adopted test generation strategy as well as about the adopted RT-level fault model. Section 3 presents an overview of the test program generation approach we propose. Section 4 reports some preliminary experimental results assessing the effectiveness of our approach, and Section 5 draws some conclusions.

2. Test Strategy

Test sequence generation for microprocessors necessarily requires the knowledge of the processor instruction set and instruction format, since only correct programs can internally perform meaningful operations. A solution for this problem was proposed in [2] with the usage of *macros*: a short sequence of instructions aiming at creating a suitable framework for testing the part of control unit and data-path affected by a given instruction (or group of instructions).

The purpose of macros is to execute all the possible instructions and to make observable the complete result of each instruction, which also includes any flag that is possibly affected by the instruction itself.

```
MOV AX, K1     ;load register AX with K1
MOV BX, K2     ;load register BX with K2
ADD AX, BX     ;sum BX to AX
MOV RW, AX     ;write AX to RW
MOV RW2, PSW   ;write status register to RW
```

Figure 1: pseudo-code of the macro for the ADD instruction.

As an example, Figure 1 reports the code (for sake of readability we use a pseudo-assembly language) for the macro concerning the addition instruction between registers using `K1` and `K2` as parameters. RW and RW2 are two easily observable memory locations.

Macros are stored in a library. A test program is a collection of macros. An optimization algorithm (see Section 3) is proposed here to select the most suitable ones from the library, and to define the values of their parameters.

The effectiveness of test program generation for microprocessors RT-level descriptions strongly depends on the adopted RT-level fault model. We selected the *RT-Level single-bit stuck-at* fault model [1] that shows a good correlation with gate-level stuck-at faults.

An RT-level single-bit stuck-at fault is defined as a single-bit stuck-at in the effect of an RT-level assignment operation: when a fault is present, the affected object (signal or variable target of an assignment statement) loads the correct value, except for one bit that remains stuck to 0 or 1. The effect VHDL statement is the statement *corresponding* to the fault. The faults are *single* and *permanent*: only one fault is inserted at a time and the fault effect is present during the whole simulation. Other assignments of the same signal are assumed to be fault-free, since stuck-at faults on the same signal but on different statements are considered different.

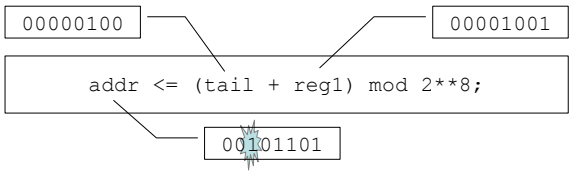

Figure 2: RT-level single bit stuck-at fault example.

Figure 2 shows the example of a RT-level single bit stuck-at fault. The fault affects the third bit of the assignment operation, and modifies the result of the expression, after it has been computed and before it is assigned to the target signal.

Better correlation with the gate-level fault model is obtained with the application of some *Fault Collapsing rules* able to partially eliminate the RT-level faults that do not correspond to any gate-level fault after synthesis.

Fault Collapsing rules are required because during synthesis the RT-level VHDL description is optimized in order to create an efficient gate-level design. The optimization process analyzes the VHDL description and simplifies all logic eliminating redundancies. As an example, assignments of constant values to a signal or variable are eliminated during optimization process.

More details about the Fault Collapsing rules can be found in [1].

3. Test Program Generation

To perform Test Program Generation starting from the analysis of the VHDL description, we must select the best macros and the values of their parameters in order to create a program able to detect the highest number of faults.

The environment we propose, whose architecture is shown in Figure 3, is composed of:

- *Fault Manager,* that analyzes the VHDL description and creates a Fault List, according to the *RT-Level single-bit stuck-at* fault model and using the *Fault Collapsing* rules introduced above;
- a *Core* that, using a set of heuristics (i.e., greedy, hill climber and evolutionary algorithms), selects the most suitable macros and the values for their parameters to create the test program;
- a *Fault Injector* that, interacting with the *Fault Simulator*, injects the faults on the microprocessor RT-level description and evaluates the effectiveness of the macros created by the *Core*.

After generating the Fault List the faults are injected during the simulation whenever the corresponding statement is executed. All the faults corresponding to a statement which has been executed at least once by the test program are labeled as *executed*. Once a fault is executed, it is also excited, if the corresponding bit assumes a value in the fault-free system which is the opposite of the stuck-at one. Finally, when a fault produces at least one difference in the output behavior of the processor (in terms of produced and observable results) it is marked as *detected*.

As we work on a microcontroller description, we can group faults in two classes:

- detectable independently from macro operands;
- detectable only using a specific set of macro operands.

In this paper, the faults that belong to the first class are called *control-dependent* faults and the ones belonging to the second class are called *data-dependent* faults. Based on our experience most of the *control-dependent* faults are located in the Control Unit and in the Instruction Decoder, where the systems decides *how* to elaborate the instruction data. Instead, most of the *data-dependent* faults are located in the data path (e.g., in the Arithmetic and Logical Unit).

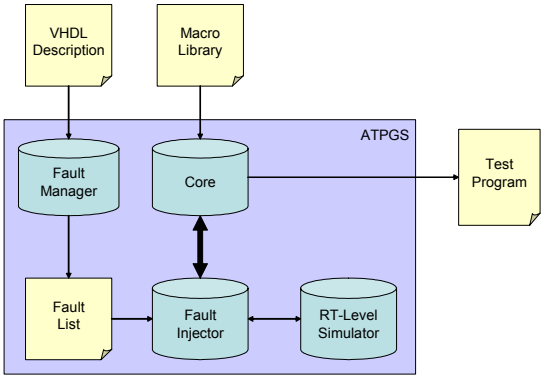

Figure 3: Test Program Generation Architecture.

3.1. Algorithm

The algorithm we propose is based on two phases:

- *control-dependent* fault detection phase
- *data-dependent* fault detection phase.

The detection of *control-dependent* faults is based on the correct selection of the operative code and the addressing mode. The detection of these faults depends on which instructions (i.e., which macros) have been executed by the microprocessor, independently from a specific set of data to be used as macro parameters. For this reason, a first phase is activated, which aims at maximizing the number of detected *control-depending* faults. The pseudo-code of this phase is reported in Figure 4.

At each iteration, the procedure `select_best_macro` simulates each macro in the library with random operands. By means of this procedure the VHDL statements executed during the fault-free simulation of each macro are identified. The macro **`M`** that maximizes the number of executed faults

is selected. The selected macro is then fault simulated and, if at least one new fault is detected, it is added to the final test program. The macro **M** is also marked as used to avoid being selected again in this phase.

```
do
{  (M,O) = select_best_macro();
   F = compute_detected_faults(M,O);
   if( F is not empty )
      add M(O) to the test program;
   drop_faults(F);
   remove M from selectable_macros;
} while(stopping_condition is false)
```

Figure 4: Control-dependent fault detection phase pseudo-code.

When all the macros of the library have been selected and fault simulated, all the macros become selectable again and the second phase starts.

The goal of the second phase is to detect *data-dependent* faults. The coverage of these faults depends on the arguments of each instruction (i.e., macro operands) executed by the microprocessor. The pseudo-code of this phase is reported in Figure 5.

```
do
{  M = select_best_macro();
   do
   {  O = select_operands(M);  /*hill climber*/
      F = compute_detected_faults(M, O);
      if( F is not empty )
      {  add M(O) to the test program;
         drop_faults(F); }
         A = compute_activated_faults(M, O);
         do
         {  Ft = select_fault(A);
            O = optimize_the_operands(M, Ft);
            if( Ft is detected )
            {  add M(O) to the test program;
               fault_dropping(M, 0); }
         }while( A is not empty );
   }while( stopping_condition() == FALSE );
   remove M from selectable_macro;
}while( selectable_macro is not empty );
```

Figure 5: Data-dependent fault detection phase pseudo-code.

As in the first phase, at each iteration the instructions executed by each macro of the library are first identified via fault-free simulation. The macro **M**, that maximizes the number of executed faults is selected.

A *hill-climbing* algorithm is then activated, whose goal is to find the values for the macro operands (**O**) that maximize the number of faults excited by the macro. At the beginning a set of random operands $\mathbf{O_{max}}$ is created and the number of faults $\mathbf{N_{max}}$ activated by the macro $\mathbf{M(O_{max})}$ is computed. At each iteration a new set of operands $\mathbf{O_{new}}$ is created applying local transformations (i.e., changing some bit values) to $\mathbf{O_{max}}$. If the number of faults $\mathbf{N_{new}}$ activated by the macro $\mathbf{M(O_{new})}$ is higher than $\mathbf{N_{max}}$, $\mathbf{O_{max}}$ and $\mathbf{N_{max}}$ are substituted by $\mathbf{O_{new}}$ and $\mathbf{N_{new}}$, and a new iteration starts. The *hill-climber* runs until the number of activated faults reaches a given threshold, or the maximum number of iterations has been reached.

For each fault $\mathbf{F_t}$ activated by the selected macro, a Genetic Algorithm, detailed in next section, is then executed, whose goal is to find the values for the macro operands (**O**) that detect the target fault.

If $\mathbf{F_t}$ is detected, the macro is added to the final test program and a fault dropping phase is activated; otherwise, the fault is discarded, to avoid being considered again with this macro.

When all the activated faults have been detected or discarded, the algorithm returns to the *hill-climber* in order to try to activate others faults.

The stopping condition is true when either the Fault Coverage reaches a given threshold, or the maximum number of iterations has been reached.

When the stopping condition is reached, the selected macro is marked as used, all the faults discarded are reinserted in the Fault List, and a new iteration starts.

The *data-dependent* faults detection phase ends when either the Fault Coverage reaches a given threshold, or all the macros of the library have been selected.

3.2. Genetic Algorithm

Once a macro has been selected from the library, a fault simulation is performed. For each fault excited by the selected macro, a Genetic Algorithm (GA) is then activated.

The goal of the GA is to identify the best values for the parameters of the selected macro in order to detect the target fault. The algorithm chooses the values for immediate operands, and those to be written in the registers or memory cells used by the target instruction.

The number of operands and their length (in bits) depend on the macro. A standard steady-state Genetic Algorithm is exploited, whose main characteristics are summarized in the following:

- chromosomes are bit strings corresponding to the concatenated operands; their length is function of the macro;
- the mutation operator randomly selects a bit in the chromosome, and complements it;
- the cross-over operator is the standard one-cut crossover;

- chromosomes are selected using a linearized fitness function and a roulette wheel mechanism.

The fitness function of a chromosome measures how far the macro **M**, created with the chromosome parameters **O**, is able to propagate the target fault $\mathbf{F_t}$ effects. More precisely, it is the maximum number of differences caused by the fault during the execution of the macro.

$$Fitness(M, O, F_t) = \underset{v \in S}{MAX} \sum_{objects} different\_bits(object)$$

where *different_bits* counts the number of bits having a different value in the fault-free and faulty system for any VHDL objects (i.e., signal and variable). The fitness function calculates the sum of differences at every clock cycle of any execution of the macro and takes the maximum.

The algorithm is stopped when the target fault is detected or a steady state is reached, i.e., when a given number of generations have elapsed without detecting the target fault.

4. Experimental Results

In order to test the effectiveness of the proposed technique we implemented it in a tool called *Automatic Test Program Generation System* (ATPGS). ATPGS amounts to about 11,000 lines of C code including an in-house developed RT-Level Fault Simulator based on a commercial VHDL Simulator (ModelSim 5.5a by Mentor Graphics).

The system has been evaluated on a synthetizable VHDL description of the Intel 8051 microcontroller, containing the core system without peripherals, whose main characteristics are summarized in Table 1.

Primary inputs	41
Primary outputs	45
VHDL lines	13,583
Processes	6
Procedures	29
RT-level faults	15,387
Gates	12,134
Flip flops	1,325
Gate-level faults	28,792

Table 1: 8051 description characteristics.

The Fault Simulator is able to simulate the entire 8051 while it executes the program stored in the embedded ROM, injecting RT-level single bit stuck-at faults in VHDL code. ATPGS is able to modify the program stored in the 8051 ROM without recompiling the entire VHDL code but interacting with the simulator in order to modify at runtime the map of the ROM. A library of 115 macros is exploited, each composed of a number of instructions that ranges from 3 to 6.

The experiments have been performed on a Sun Enterprise 250 running at 400 MHz and equipped with 2 GBytes of RAM.

Parameter	Value
Number of individuals in the population	25
Number of new individuals at each generation	25
Maximum number of generations without improvements	10
Crossover probability	0.7
Mutation probability	0.3

Table 2: Genetic Algorithm Parameters.

The values we used for the Genetic Algorithm parameters are reported in Table 2.

For the pourpose of the experiments, the RT-level ATPG was first run, with the goal of maximizing the Fault Coverage based on the RT-level fault model and a test program was obtained. For comparison porposes, a second set of experiments was then performed: the RT-level description of the 8051 was synthesized and fault simulated at gate level. Using this description, we compared the results obtained by the RT-level ATPG (Table 3) with the Fault Coverage obtained by the gate-level ATPG described in [2] (Table 4). The whole procedure adopted for the experiments is outlined in Figure 6.

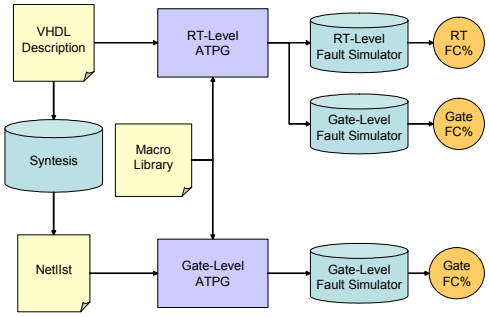

Figure 6: Experimental setup for comparison proposes.

The reported results show the proposed technique provides Fault Coverage figures higher than the gate-level ones, with a slight increase in the length of the final test program (in terms of number of instructions).

The RT-level ATPG works on a Fault List created analyzing the VHDL description and reduced by the Fault Manager applying the Fault Collapsing rules.

RT-level faults [#]	15,387
Executed faults [#]	13,364
Excited faults [#]	12,263
Detected faults [#]	12,122
Test Program Instructions [#]	883
RT-level Fault Coverage [%]	78.78
Gate-level Fault Coverage [%]	89.47

Table 3: Test Program generation from RT-level description.

Gate-level faults [#]	28,792
Detected faults [#]	25,759
Test Program Instructions [#]	624
Gate-level Fault Coverage [%]	85.19

Table 4: Test Program generation from gate-level description.

In the 8051 Fault List above 40% of the faults are eliminated by the usage of the rules; this happens because many internal parameters, especially in the Instruction Decoder and in the Control Unit, are constants. Before the proposed method is evaluated in terms of required computational effort, it must be first emphasized that in the current implementation of the tool the RT-level Fault Simulation is performed exploiting a commercial VHDL simulator. The interaction with it is necessarily loose, and therefore slow. However, if the method were integrated in the code of the simulator, a much higher efficiency would be attained. For this reason, to evaluate the required computational effort, we adopted as a parameter the number of 8051 instructions simulated by ATPGS during the test program generation phase. This number is equal to about two million instructions, and roughly corresponds to the number of instructions simulated by the gate-level ATPG described in [2].

5. Conclusions

We introduced a new method for generating test programs for microprocessors and microcontrollers. The main novelty of the proposed approach lies in the fact that it only relies on the RT-level description of the device, and does not exploit any knowledge about lower-level implementation details. The method requires the availability of a small library of macros, whose development should be performed by hand, based on the mere knowledge of the instruction set. An optimization algorithm is outlined for selecting the minimal subset of macros, and their parameters. The algorithm entirely works on the RT-level description, exploiting a suitable RT-level fault model.

Experimental results gathered on the Intel 8051 microcontroller using a prototypical implementation of the method show that the generated test program attains higher fault coverage figures (in terms of gate-level stuck-at faults) than the test program generated starting from the gate-level description, with a comparable computational effort, thus demonstrating the practical viability of the approach.

6. References

[1] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero, "An RT-level Fault Model with High Gate Level Correlation", *IEEE International High Level Design Validation Workshop*, 2000, pp. 3-8

[2] F. Corno, M. Sonza Reorda, G. Squillero, M. Violante, "On the Test of Microprocessor IP Cores", *DATE*, IEEE Design, Automation & Test in Europe Conference, 2001, pp. 209-213

[3] S. Thatte, J. Abraham, "Test Generation for Microprocessors", *IEEE Trans. on Computers*, Vol. C-29, June 1980, pp. 429-441

[4] L. Chen, S. Dey, "DEFUSE: A Deterministic Functional Self-Test Methodology for Processors", *IEEE VLSI Test Symposium*, 2000, pp. 255-262

[5] K. Batcher, C. Papachristou, "Instruction Randomization Self Test For Processor Cores", *IEEE VLSI Test Symposium*, 1999, pp. 34-40

[6] C.A. Papachristou, F. Martin, M. Nourani, "Microprocessor Based Testing for Core-Based System on Chip", *ACM/IEEE Design Automation Conf.*, 1999, pp. 586-591

[7] T.M. Niermann, W.-T. Cheng, J.H. Patel, "PROOFS: A Fast, Memory-Efficient Sequential Circuit Fault Simulator", *IEEE Trans. on CAD/ICAS*, Vol. 11, No. 2, February 1992, pp. 198-207

[8] J. Shen, J. Abraham, D. Baker, T. Hurson, M. Kinkade, "Functional verification of the Equator MAP1000 microprocessor", *36th Design Automation Conference*, 1999, pp. 169 -174

[9] N. Utamaphethai, R.D. Blanton and J.P. Shen, "Superscalar Processor Validation at the Microarchitecture Level", *12th IEEE International Conference on VLSI Design*, 1999, pp. 300-305