Using Genetic Algorithms to Accelerate Automatic Software Generation for Microprocessor Functional Testing

Renato Hentschke[1], Antônio C. S. Beck[1], Júlio C.B. Mattos[1], Luigi Carro[1,2], Marcelo Lubaszewski[1,2], Ricardo Reis[1]

[1] Renato Hentschke, Antônio C. S. Beck, Júlio C.B. Mattos, Ricardo Reis, Informatics Institute, Federal University of Rio Grande do Sul - UFRGS, Brazil, e-mail:(renato ,caco, julius,reis)@inf.ufrgs.br
[2] Luigi Carro, Marcelo Lubaszewski, Electrical Engineering Dept., Federal University of Rio Grande do Sul - UFRGS, Brazil, e-mail:(carro, luba)@eletro.ufrgs.br

***Abstract*— This paper presents an algorithm for automatic generation of programs for the functional testing of stack microprocessors. The algorithm is based on an evolutionary approach using genetic search. The paper presents the whole methodology for functional testing of this processor, since the development of program macros to the details of the genetic algorithm. The main novelty of the proposed method is the acceleration of the genetic search by multiple fault injection in some parts of the algorithm. Experimental results show that a fault coverage of 84,35% can be achieved using traditional fault simulation. Using multiple fault injection, the same coverage can be obtained in half of the simulation time**.

***Index Terms*—Test, Functional Testing, Genetic Algorithms.**

I. Introduction

Systems-on-chip are increasing in importance as the applications are getting more complex and requiring more computational power. The SoCs traditional methodology is based on IP cores reuse. Several modules, as microprocessors, are taken from third party developers for fast design and rapid time-to-market. However, this new scenario requires new test approaches.

This paper tackles the test of microprocessor cores. In general, a SoC will contain one or more of this kind of IP core embedded into the system. There are three traditional approaches for the microprocessor test: scan chains, BIST and functional testing. The scan chains approach is most commonly used due to its simplicity, but implies a lot of restrictions and increased cost for testing. First, an expensive tester with significant amount of memory for inputs and signatures storage is required. Second, the scan chains imply performance degradation of the core. But most importantly, scan chains need large test time and do not allow at-speed testing. The BIST approach is very common as well. It allows at-speed test but suffers from large test time and extra hardware. Functional testing does not suffer from any disadvantage mentioned above, but introduces other challenges.

Functional testing requires the development of a test program that is able to excite as much as possible of the processor units. This test method is pretty much suited if the processor architecture and organization is known by the programmer. However, this is not the case of SoC design, in which cores are taken from third parties. Then, automatic test program generation becomes very attractive to overcome this problem. Several works addressed this problem, such as [1][2][3]. Most of them are based on genetic search, since this kind of algorithm is able to find hardly used parts of the search space due to pseudo-random crossover and random mutation operands. Also, genetic algorithms have an evolutionary behavior, keeping the characteristics of the best generated programs and exploring new ones, until a global minimum is found.

The methodology presented in this paper is similar to [1]. First, to ease the controllability and observability of test results, the only access point needed for this approach is a link to the RAM memory. In any SoC that uses a microprocessor and a RAM memory there will be already input and output ports to access the RAM memory from other cores. The only new hardware that should be included (if not available yet) is a path to the circuit pads. An area of the memory should be reserved for the test program, while another area will be used to store a trace information of the execution. That information will indicate whether the processor passed or not the test.

The generation of the test program is pseudo-automatic. First thing that should be developed is a library of macros. One macro should be developed to test each instruction. Basically a macro consists in loading the operands to registers or to the stack (in the case of a stack processor), operate and write the result back to the memory. In the case of a jump instruction, a different value should be written in the memory in case of fail or success. This information will be used to evaluate whether a fault was detected or not. The step of creating the macros library should be done manually. However, this step is very easy and does not need any special knowledge of the microprocessor architecture. The program generation for microprocessor testing will be fully automatically. A fault simulator should be used to evaluate the fault coverage obtained during the generation process.

Compared to past works that use Genetic Algorithms for test program generation, this approach presents a new method for acceleration of the generation process. The major weakness of a genetic search is a high CPU time needed for cost function, which calls a fault simulator. Basically, this kind of simulation will introduce a stuck-at fault in each node of the circuit and run one simulation for each possible fault. In other words, in a circuit with 1500 nodes, the simulation will be repeated 1500 times, and for each time a cost function is called. Moreover, a genetic search needs several cost functions calls in order to achieve an acceptable convergence.

This work provides details for the whole proposed technique and a case study with the FemtoJava microprocessor [4]. Femtojava is a stack-based processor that executes Java bytecodes. The macros developer, in our case study, does not know any detail of the processor architecture. In addition, the genetic algorithm is totally transparent to the kind of microprocessor used. It is important to reinforce that the methodology does not require knowledge of the internal organization of the microprocessor.

This paper is organized as follows. First we introduce the macros generation process and how it works for FemtoJava microprocessor. After, in section 3, we introduce the genetic search mechanism. In section 4, we provide details of CACO-PS, the general purpose simulator used for the fault simulation. Section 5 provides details of the proposed approach to reduce the simulation time. Section 6 gives some preliminary results and section 7 provides conclusions and future work.

II. MACROS DEVELOPMENT

This section briefly describes how to develop software macros for our functional testing approach. The macros methodology is taken from [1], where more details are given. In this section we show the ideas behind the macros and detail its implementation for the FemtoJava microprocessor.

The basic idea of the macro library is to register a trace of the program functionality in the memory so that errors can be observable outside the microprocessor. There is a macro to test each instruction of the microprocessor. The macro should write something in the memory if the instruction has succeeded or any other thing if the instruction has failed.

Let us consider an arithmetic instruction as an initial example. The first two instructions of the macro should load operands for execution. After, the arithmetic operation should be called. If there is an error on any part of the instruction execution, the result should be wrong. In this case, for error detection, we can write only the result of the operation in the memory. If the arithmetic operation generated flags, they should be written in the memory as well.

Other kinds of instructions, such as "ifs", "gotos" etc, should be dealt with somehow difference. Next subsections provide details on how all the macros were generated for the FemtoJava processor.

A. FemtoJava Processor

FemtoJava Microcontroller [4] is a stack-based microcontroller to run Java bytecode. The major characteristics of this device are reduced bytecode instructions set, Harvard architecture, and small size.

FemtoJava implements an execution engine for Java in hardware through a stack machine compatible with Java Virtual Machine (JVM) specification. Since FemtoJava is a stack processor, all data transfers pass though the stack.

There are basically six types of instructions in the processor: arithmetic/logic (iadd, ixor, isub, etc.), stack manipulation (dup, pop, etc.), jump (ifge, if_icmple, goto, etc.), transfers (putstatic, sipush, etc.), sub-routine (invokestatic, return, ireturn), local variables (iload, iinc, etc.).

B. Aritmetic/Logic Macros

As detailed above, this is the most intuitive set of macros. The Femtojava processor does not store flags, so they do not need to be saved. The arithmetic macros are built as follows.

Example	Explanation
sipush x **sipush y** **iadd** **putstatic a**	1. load first operator to the stack 2. load second operator to the stack 3. perform the operation 4. write top of the stack (result) in the memory

C. Stack Macros

These instructions do some kind of manipulation on the global stack. There are instructions for simple duplication of the stack top (dup), others for complex duplications and triplications (dup_x1, dup2, dup2_x1, etc...) and an instruction for the elimination of the stack top (pop). The macros should detect if the stack has been adequately written. We developed the following templates for these instructions:

Example	Explanation
sipush x **sipush y** **swap** **if_icmpeq 5** **bipush 1** **putstatic a** **putstatic b**	1. push one (or two) number(s) randomly generated to the stack 2. perform the operation 3. if the result is correct goto 5 4. write error in memory 5. write stack top to the memory

D. Jump Macros

There are two kinds of jumps: unconditional and conditional. The following guidelines are developed for an unconditional jump:

Example	Explanation
goto 5 **bipush 1** **putstatic a** **bipush 2** **putstatic b**	1. goto 4 2. write error in memory 3. goto next instruction 4. write ok in memory

For a conditional jump we developed the template:

Example	Explanation
sipush x **sipush x** **if_icmplt 5** **bipush 1** **putstatic a** **bipush 2** **putstatic b**	1. push two numbers randomly generated to the memory 2. if (equal, greater, ...) then goto 4 3. write number1 in memory and goto next instruction 4. write number2 in memory

Note that, for both unconditional and conditional jump, a different number will be written in memory if the instruction fails or not.

E. Transfer Macros

The transfer instructions moves data from memory to the stack and from the stack to memory. It is very easy to test these instructions. The macro guideline is below:

Example	Explanation
sipush x **putstatic a**	1. push a number randomly generated to the stack 2. move the number to the memory

F. Sub-routineInstructions

To test this kind of instruction, some part of the program

memory should be reserved for subroutines. Figure 1 shows the final configuration of the program memory for all the set of macros developed. The potential problem with this kind of macros is that they introduce "jumps" for distant parts of the program memory. In the presence of a failure, the program counter can get lost and the execution get stuck by an infinite loop. To overcome this, special features can be introduced in the processor simulator in order to abort wrong execution. One possibility, that we implemented in ours, is time-out.

The FemtoJava microprocessor has three sub-routine instructions: invokestatic, return and ireturn. The first is the sub-routine call. The others are void return or return a value. To test them, we introduced two routines in the program memory. The two subroutines are detailed below:

Subroutine 1	
Code	**Explanation**
sipush x **putstatic a** **return**	1. push a random number to the stack 2. write the number in the memory 3. return

Subroutine 2	
Code	**Explanation**
sipush x **putstatic a** **return**	1. push a random number to the stack 2. write the number in the memory 3. return the number

To use both sub-routines, we created two macros:

Macro 1	
Code	**Explanation**
invokestatic sub1 **putstatic a**	1. Call Subroutine1 2. write a number to the memory (if there is a lost sequence in the program execution, this number will not be in the memory)

Macro 2	
Code	**Explanation**
invokestatic sub2 **putstatic a**	1. Call Subroutine2 2. write return value to the memory

G. Local Variables

FemtoJava supports local variables within a sub-routine. For that, replaced the flat program we had before by a subroutine called main. Also, the processor supports a predetermined number of local variables. For simplicity, we fixed to 5 variables. These variables will be stored in a fixed area of the global stack. There are two kinds of local variable instructions: load/store and arithmetic. The methodology for macro construction of these instructions is the same as seen previously.

The final configuration of the program memory is shown in figure 1.

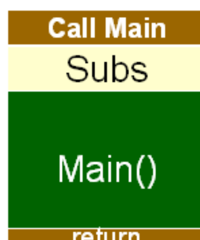

Fig.1 Final configuration of the RAM memory

III. Program Generation

A program will be composed of a sequence of macros from the library. Each macro, as seen in section 2, is independent of the next, as each represents an isolated and complete computation. So, now, using the macros library, we discuss how to generate the program for functional testing automatically.

The search mechanism implemented is a genetic algorithm. Each individual in the population will be a complete test program. The first step in the genetic algorithm is the generation of random initial programs. The user of our tool may specify the number of macros contained in the initial test programs. All selections are made randomly. After that, several iterations are made. In each iteration we execute several genetic operators. At the end, the best individual is selected as best test program. The steps bellow show the basics of our genetic algorithm.

step 1: Evaluation of all individuals (Fault coverage of each program)

step 2: Identification of the elite group

step 3: Crossover

step 4: Mutation

step 5: If not finished, go to step 1

Step 1 is accomplished by our fault simulator (section 4). Basically, the simulator returns the fault coverage of each individual in the population. This value is used to classify the individuals, to select the elite group and to select the partners for the crossover function.

Step 2 should identify the best individuals, using the fault coverage given by step 1. The elite groups are individuals that cannot be changed by any genetic operator. They can be used as crossover partners but they will not be eliminated.

Step 3 is the crossover operator. Each individual that is not in the elite group will select a partner to cross the characteristics. After crossing, the old individual is eliminated. The crossover operator is detailed in section 3.1.

Step 4 is the mutation operator. In fact, there are 4 mutation operators defined in our genetic algorithm. All of them are detailed in section 3.2. A mutation will only happen in non-elite individuals.

Step 5 is simply the stop criterion of the algorithm, which is a fixed number of iterations.

In each iteration, there is an evaluation of all population to create a group that we call "elite". The elite is composed of the best individuals of each generation. The size of this group may be determined by the user.

A. Crossover Operator

The crossover objective is to propagate the best characteristics to the new generations of individuals. All individuals that are not in the elite group may be the "male" of the crossover operator. The "female" will be chosen based on a roulette technique, where the probability of choosing an individual is proportional to the fault coverage achieved by that individual. All individuals may get a chance to participate, but the worst ones will have reduced probability. After the process, the male is eliminated to keep the population with the same number of individuals.

The process is very simple and is illustrated by figure 2. Basically each individual represents a program, which is a

sequence of macros. The chromosome will be exactly this sequence. The crossover operator selects a split point in both chromosomes. The new individual will be one of these possibilities, with 50% probability for each: the first part of the father's chromosome plus the second part of the mother's chromosome or the first part of the mother's chromosome and second part of the father's chromosome.

There is a probability for the crossover operator to be called for each operation. In our experiments we used a 96% probability.

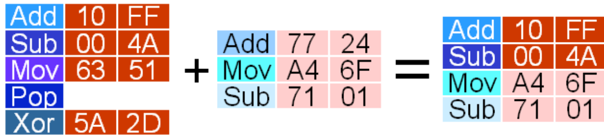

Fig. 2 Crossover Operator

B. Mutation Operator

The mutation is responsible for introducing new characteristics to the population, for a higher variability of macros combinations and operands. Only non-elite individuals are allowed as operands of a mutation. There are basically four mutation operands. Each has a different probability to be called: insert random macro, delete random macro, change operands of all macros, shuffle the macros order. For our experiments, we used 60% probability for shuffle, 82% for insert, 40% for delete, 80% for operand change. Each probability is independent of the other. It is possible, for example, that all mutations are executed. Observe, however, that the insert probability is higher than the delete one, which indicates a tendency of increasing the size of the programs.

Shuffle mutation

Figure 3 illustrates this method. Basically the order of the macros is shuffled. Another order may achieve a transition that was not achieved in the past. If a new configuration is found and it is good enough to put the individual in the elite group, the new characteristic will be saved. Otherwise, a new shuffle is implemented. However, shuffling may not vary pretty much the fault coverage.

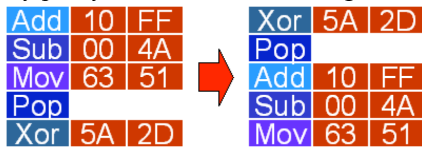

Fig. 3 Shuffle Mutation

3.2.2 Insert Macro Mutation

Figure 4 illustrates this method. Basically a new macro is inserted in the end of the program, with random operands. This mutation is very important for the increase of fault coverage of a given program.

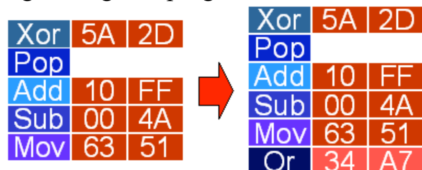

Fig. 4 Insert Macro Mutation

3.2.3 Delete Macro Mutation

Figure 5 illustrates this method. Basically a randomly selected macro is eliminated. This mutation is useful to reduce program sizes.

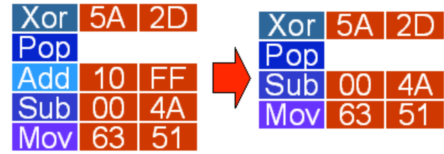

Fig. 5 Delete Macro Mutation

Operand Change Mutation

Figure 6 illustrates this method. Basically all the operands of all macros are changed randomly. The values of the operands are very important to the fault coverage.

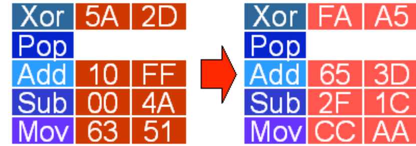

Fig. 6 Operand Change Mutation

IV. Fault Simulator

The cost function of the proposed genetic algorithm is simply a call of the fault simulator to evaluate the fault coverage of the program. CACO-PS [5], a compiled-code cycle-accurate power simulator, was extended to support stuck-at fault injection.

The first step is to run the program normally as storing the final configuration of the RAM memory of a correct execution. After that, for each possible stuck-at fault of the circuit, the simulation is repeated. If the stuck-at fault implies in a erroneous final memory configuration, that fault is detected, otherwise it is not.

The simulator supports multiple abstraction levels to describe each component. The FemtoJava processor description that we worked on is basically RTL. Components like ALU and control are not detailed. The fault simulator may only interact with I/O of this blocks. Because of that, there is a reduced number of possible stuck-at faults achieved by our microprocessor description (1700 in this case). By traditional fault simulation, 1700 possible faults with result in 1700 simulations to find the fault coverage of a single program. This consumes too much CPU time, so that running times of our genetic algorithm can last more than a day. To overcome this problem, one possibility is running the simulator in parallel, as done by [1]. We experimented a new approach, that is explained in details in section 4.1. Basically injects mutliple faults in the same CPU.

A. Mutiple Fault Injection

Our idea is to modify the genetic algorithm so that an imprecise fault coverage evaluation can be used without harming the search. Basically, there are two uses for the cost function in our genetic algorithm. First is to evaluate the individuals to classify them by the cost criterion, so that the roulette for crossover can be made and the elite group can be selected. The second use is to return the fault coverage of the winner. Obviously that we do not want that an imprecise fault coverage is returned for the user. But an imprecise fault coverage can be used for internal comparisons of the genetic algorithm, since all individuals use the same

algorithm for fault coverage evaluation.

By that, we implemented an optimistic fault coverage estimation based on multiple faults. Basically, each time that the simulator runs it injects two (could be more) faults simultaneously. If the RAM memory is not correct, it is assumed that both faults where covered by the test program. If the RAM is correct, it is assumed that both faults were not detected. It is a very rare situation that a fault will correct the other fault, so the last assumption tends to be true. The problem of impreciseness is related to the first assumption.

To escape from the situation of faults masking other faults, we tried to inject multiple faults in distant nodes of the circuit. Basically, the fault simulator has a vector data structure to store the nodes of the processor. The multiple faults were injected in distant positions of the vector. Each signal in the processor is composed by a certain amount of bits. In addition of getting distant signals, we pass through the different bits of the signals differently (from the first to the last bit or the vice-versa) in each new genetic generation. To implement that, we select a different random seed at each iteration and, each time the simulator is called, the random seed is reinitialized.

The modified genetic algorithm is described below.

step 1: Evaluation of all individuals (is done with unprecise estimation)

step 2: Reevaluation of the best one to be displayed (true evaluation)

step 3: Identification of the elite group (using the unprecise measure done in step1).

step 4: Crossover

step 5: Mutation

step 6: If not finished, go to step 1

This technique provides n times speedup, depending only on the number n of multiple injected faults. By some experiments, we observed that the multiple fault injection does not degrade the quality of final solution neither the convergence of the process (see section 5). However, some limitations of our experiments must be highlighted. First, the description of our architecture is at the RTL level. The internal blocks, such as ALU, registers and others are not detailed. Only the inputs and outputs are considered "wires", so that they can be stuck-at or not. Internal wires are not considered for fault injection. That limitation may point the obtained results are not correct. On the other hand, increasing the number of wires may be good for the effectiveness of the technique. If there were more wires, it would be easier to find non-correlated wire, so that multiple injection could be more effective.

V. Experimental Results

We made some experiments to check the effectiveness of the proposed technique. The experiments were made based on two sets of macros.

The first set of macros contains only few macros, excluding jumps, sub-routine and any other operation that modifies the program counter. This limitation was done because on previous versions of the fault simulator the program could get lost by an error on the PC signal, for example. This was overcome by limiting the number of executed cycles to the number of cycles of the execution without injected faults.

Initially, we tried to start from very small programs (using 1 to 3 macros on the initial population) and let the genetic algorithm find the appropriate size for the program. The generation of the initial population, which is random, found a fault coverage of 61% (in the best individual). The genetic algorithm finished with a fault coverage of 73,4%. The cost variation curve is shown in figure 6.

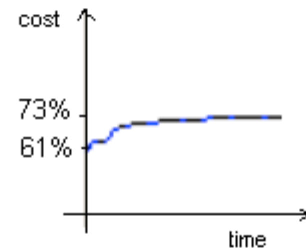

Fig. 6 Fault coverage variation for a genetic algorithm execution with reduced macros set

The second try was to start with larger programs in the initial population and see what else the genetic algorithm could do to improve the fault coverage. The initial population started in 72,1% and the genetic search found 73,4%, the same as in the last experiment. The results show that the genetic algorithm is able only to do few improvements on the fault coverage. We believe that this happens due to the high level description of the architecture. A genetic algorithm would be able to find better configurations if there were more faults modeled in the architecture.

After the first set of experiments and after we fixed our fault simulator to accept event jump macros, we ran new experiments. First, we started from a population of 1 to 3 macros randomly chosen. The starting fault coverage was 76,6%. At the end, the genetic algorithm found a fault coverage of 84,35%.

To verify our multiple fault injection approach, we ran the same simulation, under exactly the same conditions, and we achieved the same fault coverage on half of the time! Figure 7 shows the real fault coverage variation of the multiple faults approach, showing that the process converge to better configurations.

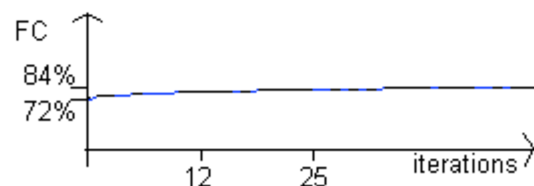

Fig. 7 – Real fault coverage variation for a genetic algorithm execution with full macros set and multiple faults

VI. Conclusions and Future Work

This paper describes a genetic algorithm to be used as test program generator for functional testing of microprocessor. By the experiments run over the Femtojava processor, we observed that it is pretty much relevant the size of the macros library for the test program generation. In the initial experiments, we tested a restricted set of macros and

achieved a maximum fault coverage of 73,4%. Using a full set of macros, we achieved 84,35%.

The main conclusion of this work is that multiple fault injection can be used to accelerate the genetic algorithm without harming its search capability.

Some improvements can be done to increase the effectiveness of the technique. First, a study of the processor architecture could help to select the best partners for multiple fault injection. Second, the RAM memory could be inspected for a number of different words. If there is just one wrong word, then we could assume that one of the faults was propagated to the memory, increasing the preciseness of the multiple injection.

VII. References

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

[2] [2] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero, "Fully Automatic Test Program Generation for Microprocessor Cores", IEEE Design, Automation & Test in Europe, 2003, pp. 1530-1591

[3] [3] N. Kranitis, G. Xenoulis, D. Gizopoulos, A. Paschalis, Y. Zorian, “Low-Cost Software-Based Self of RISC Processor Cores”, IEEE Design, Automation & Test in Europe, 2003, pp. 1530-1591

[4] [4] S. A. Ito; L. Carro; R. P. Jacobi; Making Java Work for Microcontroller Applications. Design & Test of Computers, IEEE, Volume: 18, Issue: 5 , Sept.-Oct. 2001, pp. 100 –110

[5] [5] A.C.S. Beck F., J.C.B. Mattos, F.R. Wagner, L. Carro, “CACO-PS: A General Purpose Cycle-Accurate Configurable Power-Simulator”, 16th Brazilian Symp. Integrated Circuit Design (SBCCI 2003), Sep. 2003