On the Automatic Generation of Software-Based Self-Test Programs for Functional Test and Diagnosis of VLIW Processors

Davide Sabena, Luca Sterpone, and Matteo Sonza Reorda

Dipartimento di Automatica e Informatica, Politecnico di Torino, Torino, Italy

{davide.sabena,luca.sterpone,matteo.sonzareorda}@polito.it

Abstract[1]**.** Software-Based Self-Test (SBST) approaches have shown to be an effective solution to detect permanent faults, both at the end of the production process, and during the operational phase. However, when Very Long Instruction Word (VLIW) processors are addressed these techniques require some optimization steps in order to properly exploit the parallelism intrinsic in these architectures. In this chapter we present a new method that, starting from previously known algorithms, automatically generates an effective test program able to still reach high fault coverage on the VLIW processor under test, while minimizing the test duration and the test code size. Moreover, using this method, a set of small SBST programs can be generated aimed at the diagnosis of the VLIW processor. Experimental results gathered on a case study show the effectiveness of the proposed approach.

Keywords: SBST, VLIW processor, Fault Simulation, Fault Diagnosis.

1 Introduction

The continuous scaling in the semiconductor fabrication process combined with the progressive growth of the integrated circuits operation frequency pushes processor cores to face more difficult testability problems. Furthermore, several phenomena such as metal migration or aging become more likely, thus increasing the occurrence of permanent faults in the generic system, in particular during the circuit operational phase. For these reasons, in order to provide high fault coverage with acceptable costs, new test solutions are being investigated and evaluated (e.g., in terms of silicon area overhead, required test infrastructure and test time).

Software-Base Self-Test (SBST) has been demonstrated to be a promising and effective approach for the test of processors and processor-based systems [1]. The SBST main idea is to generate test programs to be executed by the processor under test, able to fully stimulate the processor itself or other components belonging to the

[1] This work has been partially supported by the European Commission through the LoRelei project (PIRES-GA-2011-29521).

system, and to detect possible faults by looking at the produced results. The SBST technique does not require any additional hardware; therefore, the whole test cost is reduced and no performance penalty is introduced. Moreover, the SBST technique allows at-speed testing and can be easily used even for on-line test purposes. Hence, processor and System on Chip (SoC) testing approaches are increasingly adopting SBST techniques, often in combination with other approaches.

Correct identification of the most common defective parts in a SoC helps to characterize the technological process. The localization of a fault allows to effectively direct physical investigation of the underlying defects [2]. Moreover, a good diagnosis capability is fundamental for the devices containing self-repair skills. On the other side, it is well known that the complexity of diagnostic test generation is much higher than that of detection-oriented test generation [3]. Among the various diagnosis techniques, the Software-Based Diagnosis (SBD) methodology has turned out to be a suitable solution for processor cores embedded in SoCs [2][3].

Today, several applications demand for high performance while exposing a considerable amount of Instruction Level Parallelism (ILP), such as Digital Signal Processing [4]: among the various microprocessor architectures, Very Long Instruction Word (VLIW) processors have been demonstrated to be extremely attractive for such kinds of applications. Nowadays, several products for embedded applications adopt VLIW processors; therefore, the problem of testing them is increasingly relevant.

A major difference of VLIW processors with respect to traditional superscalar processors is the instruction format. Several VLIW instructions, named *micro-instructions*, are grouped into one large macro-instruction (also called *bundle*) where all micro-instructions within the bundle are executed in parallel computational units; each one is independent and referred to as *Computational Domain*. The operation scheduling performed by VLIW architectures is executed at compile time; therefore, the compiler is responsible for allocating the execution of each instruction to a specific Functional Unit (FU).

Due to these characteristics, VLIW processors are suitable for safety-critical systems adopted in mission-critical applications such as space, automotive or rail-transport fields which require computationally intensive functionalities combined with low power consumption. For example, the processor Tilera TILE64TM, composed of several VLIW cores, is used to efficiently perform image analysis on-board a Mars rover in support of autonomous scientific activities [5][6].

Few previously developed SBST approaches may be found in the literature in order to properly test VLIW processors against permanent faults; more in particular, part of them rely on suitable instructions belonging to the original processors instruction set to apply the test patterns previously generated by automated test pattern generation (ATPG) tools, which particularly focus on internal components [7]. These methods present some drawbacks: first of all, transforming the test patterns generated by the ATPG into test programs is not always straightforward; secondly, the resulting test programs are not optimized, especially in terms of test duration; finally, the attainable fault coverage is rarely as high as it may be required.

VLIW processors include a register file having some characteristics (in particular, the fact that it can be accessed from different domains) that make it different than the

one in other processors. In [8] we focused on this component and proposed a solution, based on a SBST approach, which resulted to be quite effective.

Considering the diagnosis problem in VLIW processor, in the literature there is only a preliminary work aimed at the localization of permanent defects inside VLIW components, and the provided solution is a combination of several self-test techniques (SBST and BIST) [9].

In this chapter we focus on the generation of effective SBST test programs for VLIW processors, characterized by minimal size, minimal duration and maximum fault coverage. The proposed method starts from existing SBST test programs developed for the different FUs embedded into most processors (e.g., ALUs, adders, multipliers and memory units). Although the characteristics of FUs used within a VLIW processor are similar to those used in traditional processors, generating optimized code to effectively test these units is not a trivial task: our test generation procedure addresses the several units embedded into distinct parallel computational domains, thus taking into consideration the inherently parallel architecture of VLIW processors. Another goal of our work was the development of a general approach that could lead to the automatic generation of the test program for a VLIW processor, once the test code for testing each unit is available, and the processor configuration is known. The architecture of a VLIW processor does not include any custom hardware module, but rather a combination of common Functional Units. Our solution allows test program generation and optimization to be performed autonomously, while automatically exploiting the VLIW characteristics, without any further manual effort. The proposed method allows to generate highly optimized test programs which exploit most of the VLIW processor features and are aimed at minimizing the test time and the test program size. Besides, the method does not require the usage of any ATPG tool, since it is fully functional. Finally, without any additional effort, it is possible to exploit the test programs developed during the proposed flow to perform fault diagnosis and thus identify the faulty unit among the most relevant modules of the considered VLIW processors.

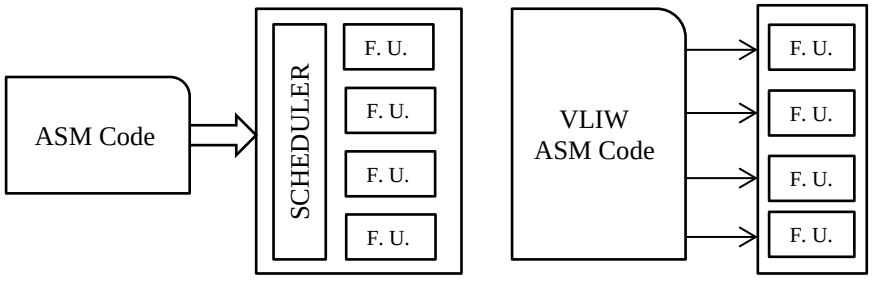
The main contribution of this chapter is the description of the first technique able to completely automate the generation of effective Software-Based Self-Testing programs for VLIW processors, while guaranteeing that the resulting programs are optimal in terms of duration and size. Exploiting this automatic method, test programs having some diagnostic properties can also be generated. The proposed method has been evaluated on a VLIW platform based on the Delft University ρ-VEX VLIW processor [10][11] which supports most of the features of industrial VLIW architectures. The results we achieved clearly demonstrate the effectiveness of our approach. Considering the generation of the optimized test programs, clock cycles have been reduced by approximately 54% with respect to the original test programs, while the size of the optimized test program decreased by approximately 58%. When the diagnosis capabilities are considered, given a generic fault in the VLIW processor under test, we are able to distinguish it uniquely in the 2.78% of the cases; moreover, in 79.15% of cases we are able to identify the faulty module containing the fault itself, while in the remaining cases we are able to narrow down the set of candidate faulty modules to 2 modules (54.52%) or to 3 modules (38.81%).

The chapter is organized as follows. Section 2 gives an overview of the VLIW architecture. Section 3 describes the related work on Software-Based Self-Test techniques specifically oriented to VLIW processors, while Section 4 explains in detail the proposed method. Experimental results on the selected case study and their analysis are presented in Section 5. Finally, conclusions and future work are described in Section 6.

2 VLIW Architecture Summary

The main characteristic of a VLIW processor is the fact that all the operations are executed by parallel *Computational Domains*, each one characterized by its own Functional Units. Besides, the scheduling is totally static, since compile tools preliminary define it at compile time. As illustrated in Fig. 1, the assembly code for a VLIW processor is drastically different from the point of view of the machine code with respect to a superscalar processor: several instructions are grouped together in a single macro-instruction (named *Bundle*) and for each instruction there are some information items that allow to assign its execution to a specific Computational Domain. Consequently, in a VLIW processor there isn't any hardware instruction scheduler, and the tasks typically performed by this component are done by the compiler. The power consumption is thus reduced and the silicon area decreases if compared to traditional superscalar processors. Furthermore, the Instruction Level Parallelism (ILP) can be adequately exploited (at least in the case of data intensive applications) since a good compiler is able to decide which instructions can be executed in parallel by checking the entire program at compile time [8].

A generic VLIW processor parametric architecture may have a variable number of functional units (FUs), so that different options, such as the number and type of functional units, the number of multi-ported registers (i.e., the size of the register file), the width of the memory buses and the type of different accessible FUs, can be modified depending on the application requirements [4].



(a) Superscalar Architecture (b) VLIW Architecture

Fig. 1. Architectural differences between a superscalar and a VLIW CPU.

All the characteristics of a specific VLIW processor are grouped together and are listed in the so called *VLIW manifest*. The manifest specifies the number of computational domains, the number and type of the Functional Units embedded into each computational domain, the size and access mode of the register file and any other feature that must be taken into account when developing the code for the processor.

3 Related Work

Methodologies that require an external tester to perform the test are infeasible without the use of very expensive Automatic Test Equipments (ATEs); however the increasing gap between maximum ATE frequencies and SoC operating frequencies makes external at-speed testing problematic and expensive; at-speed testing is needed because of failures detectable only when the test is performed at the device operating frequency. Moreover, external test often involves long time and significant efforts to introduce the required hardware and may be characterized by long test application times [12]. While ATEs use external resources to perform testing task, BIST involves internal hardware resources: additional hardware and software are integrated into the circuit to allow it to perform self-testing. The usage of BIST leads to lower the cost of the complete test as well as the test time, maintaining or improving the fault coverage, at the cost of additional silicon area [8].

SBST techniques represent a special solution for on-chip testing [12], since they adopt existing processor resources and instructions to perform self-testing without any intrusiveness. The main advantage of the SBST methodology is that it uses only the processor functionality and instruction set for both test pattern application and output data evaluation, and thus does not introduce any hardware overhead in the design. However, software-based self-test methods may require very long programs to achieve high fault coverage of the device under test, and require ad-hoc techniques for generating suitable test programs [1][12]. Several papers are available in the literature related to methods for the functional self-test of processors, but only few of them refer to the test of Very Long Instruction Word (VLIW) processors [8][13][14][15].

In [8] we proposed a new SBST algorithm oriented to the test of the Register File of a generic VLIW processor; that paper highlights the particular structure of the register file belonging to a VLIW processor, that presents a particular structure since it is shared by all the computational domains of the processor; in particular, the proposed algorithm is able to efficiently test the complex cross-bar switch embedded into the component. Another technique able to obtain a good diagnostic resolution with a low hardware overhead is proposed in [14]; this technique combines scan and SBST and it is oriented to the test of VLIW processors. The specific characteristic of that approach is the ability to detect faults inside the processor functional units, obtained by loading the same test patterns directly to the test registers of all the computational domains. The proper functionality of each domain is tested by comparing the test response of all domains, which should be the same than in the fault-free case. This solution involves a hardware overhead of about 6% and requires that the processor run in self-test mode.

Similar to test approaches, several Software-Based Diagnosis (SBD) methods applied to processors have been recently developed. In [2] a new cost-effective approach is presented: the approach is based on the automatic generation of a diagnostic test set using an existing post-production test set; the authors propose to improve that set using an evolutionary method. In [9] the authors present a new diagnostic method for VLIW processors, based on scan-based BIST and SBST, aimed at a good diagnostic resolution with low hardware overhead. Software-based BIST is introduced for a fast diagnosis of the Computational Domains of the processor. This is an initial work in the field and it is based on the use of several existing self-test techniques; moreover, it is based on a specific VLIW processor and requires the introduction of several hardware test module in the considered processor.

4 The Proposed Method

In this chapter we describe a new method that allows the automatic generation of an optimized SBST program for a generic VLIW processor, once its specific configuration is known. The proposed method is composed of two main steps, denoted as *Fragmentation* and *Customization*; moreover, we propose two different flows specifically oriented to test and diagnosis, respectively. Considering the test flow, step C.1 is characterized by *Selection* and *Scheduling*; considering the diagnosis flow, step C.2 is characterized by *Classification* and *Equivalence Check* (Fig. 2); hereafter, the detailed description of each of these steps will be provided.

The only two requirements for the global generation flow are the manifest of the VLIW processor under test, containing all the features of the processor itself, and a library containing a set of programs able to autonomously test the different modules within the processor. The library is a collection of generic SBST programs taken from the literature [8][12][16][17][18]: it contains some functional test code able to test the most relevant Functional Units of a generic VLIW processor. The codes stored into the library are purely functional (i.e., do not require any Design for Testability feature) and are completely independent of any physical implementation of the Functional Unit they refer; these codes are described with a pseudo-code based on C language. The mapping process of these codes to the specific architecture under test is performed by the second step of the proposed method (i.e., the Customization step).

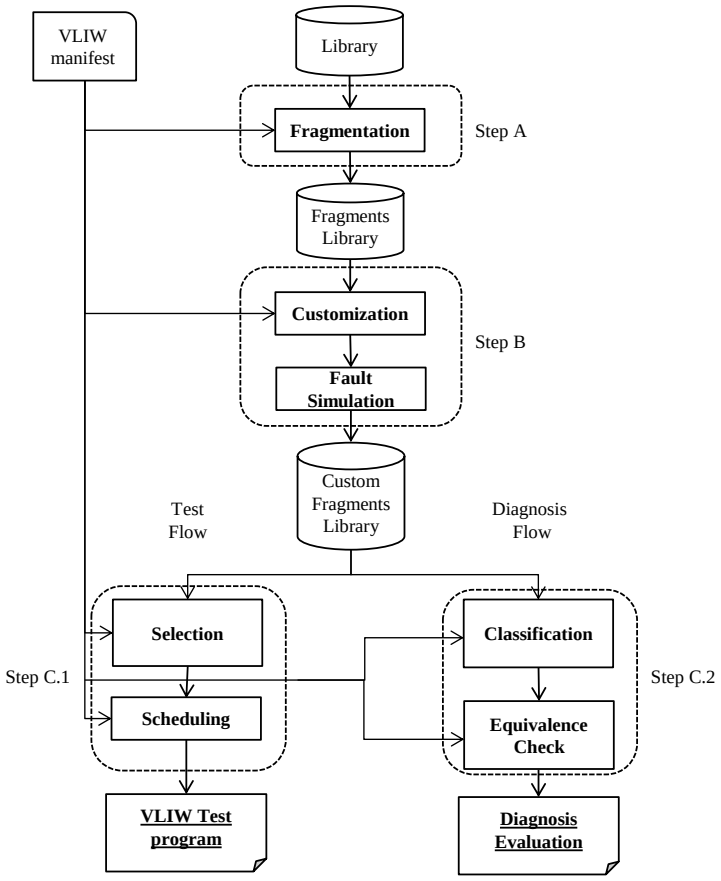


Fig. 2. The flow of the proposed test and diagnosis method.

4.1 Fragmentation

The goal of the Fragmentation phase is the minimization of the number of test operations in order to generate optimized and efficient test programs. Two main tasks are performed by the Fragmentation phase: the first is the selection from the library of the test programs needed to test the VLIW processor under test, ignoring those which refer to Functional Units that are not belonging to the processor itself. The second task performed by this step is the fragmentation of each selected test program into a set of smaller pieces of code, named *Fragments*, containing few test operations and the other instructions needed to perform an independent test. The generation of a fragment is done by building it around a single instruction, and includes some preliminary instructions required to correctly perform it and to forward the results into observable locations [2][19]; the description of a Fragment is performed through some architecture-independent code. On the other hand, a test program is typically composed of a set of test operations enclosed in a loop; a series of short test programs are

generated by simply separating the test operations using the Loop Unrolling technique, as shown into the pseudo-code of Fig. 3.

The code is then optimized by executing the Fragmentation phase, which exploits the fact that a VLIW processor is composed of parallel computational domains that execute several operations in parallel, as described in Section 2. Due to this feature, when a SBST program is executed with the purpose of testing a selected unit, at the same time several operations can also be executed on other parallel units. In Fig. 4 an example of this concept is shown, where it is possible to notice that by applying the SBST program for the test of the VLIW register file [8] several faults related to the Functional Units (e.g., the adders and the MEM unit) are also covered. The main idea behind test program fragmentation is to divide the original programs in atomic test units in order to effectively evaluate each one of them; multiple fault coverage is therefore avoided and the test code can be optimized in terms of test time and used resources. Once the Fragmentation phase is completed, a new library called *Fragments Library* is obtained, that contains the set of architecture-independent Fragments.

```
1. for each cycle C of the loop L {
   1.1.   S = set of performed operations;
   1.2.   PI = input pattern applied to S into the cycle C;
   1.3.   R = expected results performing S using PI as
          input pattern;
   1.4.   GENERATE_NEW_FRAGMENT (PI, S, R);
2. }
```

Fig. 3. The pseudo-code of the Fragmentation phase.

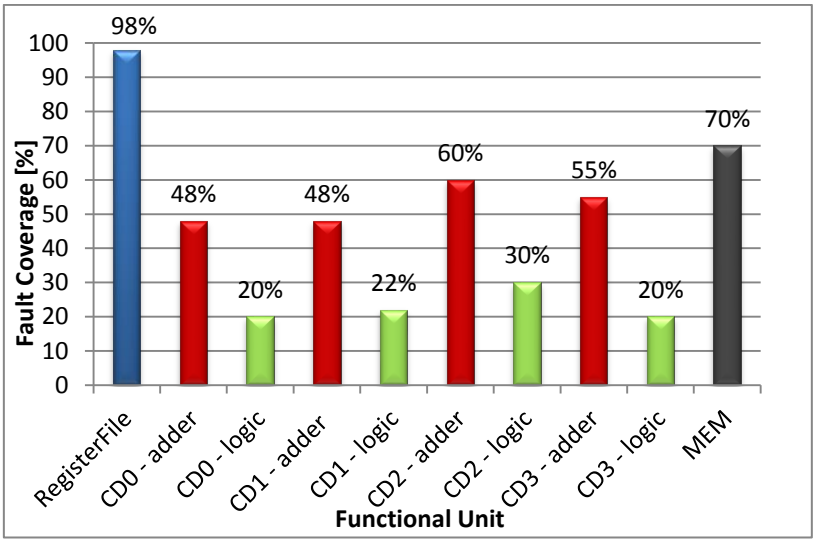


Fig. 4. The Fault Coverage of the test program for the Register File with respect to faults in the other modules of the processor.

4.2 Customization

The translation of the generic architecture-independent test programs into the VLIW code is managed by the Customization step, which uses the Instruction Set Architecture (ISA) of the considered processor. In detail, starting from the VLIW manifest and from the Fragments Library, the method translates each generic Fragment into a *Custom Fragment* that can be executed by the processors under test. A Custom Fragment is defined as a set of instructions related to the ISA of the processor under test that performs several operations in order to test the addressed Functional Unit. In Table 1 an example of the Customization process is reported, where the code of a Fragment before and after the Customization phase appears. The example is based on a multiplication instruction, and the produced result is saved into the memory. As the reader can notice, at the beginning the code is a generic ISA-independent code, while after the Customization step, a VLIW code is generated, exploiting the ρ-VEX processor ISA [10][11].

Table 1. Example of the translation performed by the customizer.

Before Customization
`R = mul (All 0's, All 0's);` `Store(R , memory);`
After Customization

`;;----Macro-instruction 1----`
`CD0 : mov R1 = 0;`
`CD1 : mov R2 = 0;`
`;;----Macro-instruction 2----`
`CD0 : mul R3 = R1, R2;`
`;;----Macro-instruction 3----`
`CD0 : stw 4[R7] = R3;  //R7 is the stack pointer`
`;;----------`

The Customization phase performs two relevant tasks: the definition of the resources needed to execute the code (such as the memory area required and the registers) and the introduction of the information, inside the code, that assign the execution of an instruction to a defined VLIW Computational Domain. In Table 1, it is reported an example of this translation, where `CDx` is the Computational Domain in charge of executing the addressed instruction.

The translation of each Fragment is performed independently from the others; furthermore, one architecture-independent Fragment can be translated into several architecture-dependent Fragments, following the features listed in the VLIW manifest, such as the type of functional units contained in each Computational Domain: for example, if in the considered VLIW processor there are 4 adder units, one for each of the 4 Computational Domains, the generic Fragment related to the test of an adder is translated into 4 architecture-dependent Fragments, one for each adder unit embedded into the Computational Domains. When the Customization phase is terminated, each architecture-dependent Fragment is fault simulated in order to compute a detailed list

of faults covered by the specific test program considering all the resources of the VLIW processor. Finally, a library called *Custom Fragments Library* is obtained: it contains all the architecture-dependent Fragments used to test the processor under test and the list of faults covered by each of them. As shown in Fig. 2, the fault lists associated to each Custom Fragment are also used for the diagnosis flow, as we will explain in Section 4.4.

4.3 Selection and Scheduling

During this phase two important processes are performed: the selection of the Custom Fragments, according to the objective to be achieved, and the merge of these in order to obtain a compact and efficient test program.

Considering the Selection step, the Custom Fragments are selected by an algorithm which implements two alternative rules depending on the user requirements. The first rule is based on the selection of the minimum number of Custom Fragments that allow to reach the maximum coverage with respect to all resources of the processor under test. In this way several Custom Fragments are not selected since the faults covered by these Fragments are already covered by other fragments previously selected. The pseudo-code of this algorithm is shown in Fig. 5.

```
1. FL = Fault List of the considered processor;
2. CFL = Custom Fragments Library;
3. SFL = Selected Fragments List;
4. while ( CFL is not empty AND found) {
   4.1.  select Fragment F that allows to maximize
         the coverage of FL;
   4.2.  if (F exists){
            • put F into SFL;
            • remove F from CFL;
            • found = TRUE;
   4.3.  } else
            • found = FALSE;
5. }
```

Fig. 5. The pseudo-code of the algorithm for the selection of the Custom Fragments.

The second rule is based on optimizing the number of resources used by the selected Custom Fragments. The maximal number of usable resources, in terms of registers and memory words, can be specified by the user. On the basis of these constraints, the algorithm selects the Custom Fragments that allow to reach the maximum coverage without using more resources than those specified. In this way the method is able to generate test programs depending on the final requirements: for example, if the final goal is to generate test programs for on-line testing, with the use of this algorithm we are able to generate test codes that exploit only a limited set of registers and memory words.

At the end of the Selection phase, the selected Custom Fragments enter the Scheduling phase: this process is responsible for the integration of the Custom Fragments, in order to obtain an optimized and efficient final test program. To reach this goal the scheduler optimizes and merges the codes contained into the Custom Fragments exploiting the VLIW features; in particular, it compacts the test programs aiming at maximizing the ILP of the processor. To perform the merge operation two techniques are defined and adopted; considering two or more Custom Fragments, the former is based on the exploitation of the common input pattern belonging to different instructions: in this case it is not required to define two instances of the same input data to perform the test instructions; an example of this operations is shown in Table 2, where two Custom Fragments, related to the test of the adder units embedded into the Computational Domain 0 and 1, are merged into a single test program. In this way the ILP is better exploited and the number of macro-instructions required is less than the sum of the macro-instructions of the two Fragments. The latter technique is based on the maximization of the ILP of the VLIW architecture: starting from the code of the selected Custom Fragments, the macro-instructions of these codes are merged together in order to maximize the parallel operations executed by the code.

Table 2. Example of the optimization operations performed by the scheduler

Custom Fragment A	**Custom Fragment B**
;;--Macro-instruction A1 CD0 : mov R1 = 0; CD1 : mov R2 = 0; *;;--Macro-instruction A2* CD0 : add R8 = R1, R2; *;;--Macro-instruction A3* CD0 : stw 0[R1] = R8; *;;----------*	*;;--Macro-instruction B1* CD0 : mov R1 = 0; CD1 : mov R2 = 0; *;;--Macro-instruction B2* CD1 : add R9 = R1, R2; *;;--Macro-instruction B3* CD0 : stw 0[R1] = R9; *;;----------

Final Test Program F

```
;;-- Macro-instruction F1
CD0 : mov R1 = 0;
CD1 : mov R2 = 0;
;;-- Macro-instruction F2
CD0 : add R8 = R1, R2; //tests the adder of CD0
CD1 : add R9 = R1, R2; //tests the adder of CD1
;;-- Macro-instruction F3
CD0 : stw 0[R7] = R8;  //R7 is the stack pointer
;;-- Macro-instruction F4
CD0 : stw 4[R7] = R9;  //R7 is the stack pointer
;;----------
```

The goal of this scheduling technique is to generate the macro-instructions of the final test program, thus reducing the whole test time. Three analysis steps are required to acquire the necessary information with respect to each Custom Fragment: the resources required by the code, such as the registers, the memory words and the Func-

tional Units exploited; the temporal characteristics, defined as the number of clock cycles where the resources mentioned above are employed in the execution of the code; finally, the data dependences between the instructions belonging to the Custom Fragments. These pieces of information are used to create the final test program, according to the features of the VLIW processor described in the VLIW manifest. In order to do this, the scheduler uses three structures: the first is an activity frame schedule that is used to schedule the execution of the Custom Fragments into the Computational Domains: an example of this is reported in Fig. 6, where the chart representation of the activity frame schedule of the code listed in Table 2 is reported, consisting of two Custom Fragments, called A and B, each composed of three macro-instructions called A-1, A-2, A-3 and B-1, B-2, B-3, respectively. The second structure needed to create the final test program is a graph structure, where the dependences between the instructions composing the program are saved; in Fig. 7 is reported the graph structure related to the simple example shown in Table 2. Finally, the last structure is a graph containing the information about the resources, such as registers and memory word, used by the final test program for each clock cycle. At the end of this step, the final test program is generated.

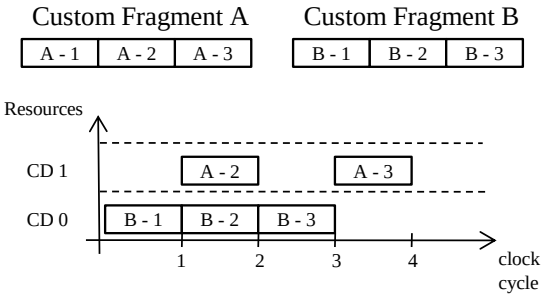


Fig. 6. The chart representation of the activity frame schedule.

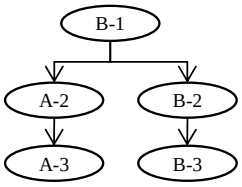


Fig. 7. The graph structure for the instruction dependence.

4.4 Classification and Equivalence Check

In some situations, diagnosis is required, which means that the goal becomes the identification of the fault existing in the unit under test. For example, diagnosis is crucial in the ramp-up phase of a new product, when the yield of the production process is expected to grow thanks to the tuning of the process (which requires knowing where the faults are) [20]. Another typical scenario where diagnosis is crucial is when sys-

tem reconfiguration can be performed after a fault is detected, e.g., thanks to the adoption of a programmable architecture: in this case diagnosis is crucial to identify (once a fault is detected during the operational phase) the partition containing the fault, so that the system can be reconfigured and the partition can be substituted by a fault-free one [21].

Given the importance of diagnosis, we performed a preliminary analysis about the diagnostic power of the test programs generated by our method, and we made some considerations aimed at improving their diagnosis capabilities.

First of all, we will define the notation to be used and the steps of the diagnosis method; then, we will report some experimental figures (in Section 5.2) about the diagnostic capabilities of the test programs generated by the proposed method.

Notation. Let us call $F = \{f_0, f_1, \ldots, f_{n-1}\}$ the set of n faults that can affect the Unit Under Test (UUT). Each of these faults causes the UUT to produce a given output behavior b when a given sequence of input stimuli is applied; let b_i denote the output behavior produced by fault f_i, and b_g the output behavior of the fault-free circuit. Clearly, $b_i = b_g$ for all undetected faults f_i. In the literature (and in practice) the output behavior can be observed (for the purpose of diagnosis) resorting to two different criteria:

- Criterion #1: the output behavior of a fault is simply the sequence of time instants in which the fault is detected. Therefore, according to this criterion $b_i = b_j$ iff the two faults f_i and f_j are detected in the same time instants.
- Criterion #2: the output behavior of a fault is the sequence of output values produced by the fault. Therefore, according to this criterion $b_i = b_j$ iff the two faults f_i and f_j always produce the same output values.

For the purpose of this paper we will consider a criterion which is a mix of criterion #1 and criterion #2. In particular, we will classify faults according to an output behavior corresponding to the set of values produced by the program at the end of its execution. Therefore, according to this criterion $b_i = b_j$ iff the two faults f_i and f_j produce the same output values in memory at the end of their execution.

A given pair of faults (f_i, f_j) is said to be *distinguished* by a given sequence of input stimuli I iff $b_i \neq b_j$. Otherwise, they are said to be *equivalent wrt I*. All faults that are equivalent wrt to a give sequence of input stimuli I are said to belong to the same *Equivalence Class wrt I*. A detected fault f_i is said to be *fully diagnosed* by a sequence of input stimuli I iff any couple of faults (f_i, f_j) including f_i is distinguished by I. Since two faults f_i, f_j can never be distinguished if they are functionally equivalent, the number of fully diagnosed faults in a circuit is typically rather low.

Several possible metrics can be adopted to measure the diagnostic capabilities of a sequence of input stimuli I [22]. A popular one is the so-called *diagnostic resolution*, or DR(I), which corresponds to the fraction of all pairs of detected faults that are distinguished by I.

When diagnosis is used in reconfigurable system for identifying the partition including the fault, the precision required is lower: in fact, the final goal in this case is to be able to distinguish all pairs of faults belonging to different partitions, while distinguishing pairs of faults belonging to the same partitions is not of interest. Hence, in

this case a different definition of the diagnostic resolution can be introduced, based on a given partition of the circuit elements among P partitions. Assuming that the generic fault f_i is associated to the partition p_i, we will only consider those pairs of faults (f_i, f_j) such that $p_i \neq p_j$ and define the *partition-oriented diagnostic resolution* of a given sequence of input stimuli *I*, or PRDR(I), as the fraction of all pairs of detected faults belonging to different partitions that are distinguished by *I*.

Method. Considering the Diagnosis flow, shown in Fig. 2 Step C.2, there are two main steps necessary to acquire the diagnostic data.

First of all the fault lists associated to each Custom Fragment, and generated through fault simulation (Fig 2, Step B) are analyzed and compared: the goals of this analysis are (1) the classification of each fault, belonging to the VLIW processor under test, in the class of distinguished faults and equivalent faults, respectively, and (2) the creation of the equivalence classes, according to the notation described in the previous paragraph.

The second step is the analysis and the classification of the equivalence classes; for each of them, the classification is based on the number of partitions that have at least one fault in the considered equivalence class; the composition of the partition defines the granularity of the diagnosis and it is managed by the final user, according to the chosen diagnosis goal.

At the end of these two steps, using the obtained data and given a fault in the considered VLIW processor, we will be able to either uniquely identify it (if the fault is distinguished), or to identify the partition (one or more) containing the fault itself and the equivalent faults (if the fault has one or more equivalent).

5 Experimental Results

In this section, we present the experimental results, both for the optimized generation of the SBST program and for the diagnosis evaluation; the ρ-VEX VLIW processor has been used as a case study (Fig. 8).

The ρ-VEX is a VLIW processor released by researchers from Delft University of Technology [10][11]. Among its main features, the most important advantage is the possibility of reconfiguring the pipeline according to the user need. The pipeline, in the standard configuration, is composed of four stages: fetch, decode, execute and write-back. Following the VLIW architecture principles, the decode, execute and write-back stages are divided into four Computational Domains (CD). The fetch unit is in charge of fetching a VLIW macro-instruction from the attached instruction memory; then, it splits the considered macro-instruction into several (according to the processor configuration) micro-instructions; finally, these are passed in parallel to the decode unit. In the decoding stage two main tasks are executed: firstly, the operations are performed, and secondly the registers used as operands are fetched from the general purpose register file (the GR module of Fig. 8) and from the branch management register file (the BR module of Fig. 8). The micro-operations are then forwarded to

the parallel execution units, that in this case are ALUs (1 ALU for each CD) and MULs (2 MULS, embedded in the second and in the third CD).

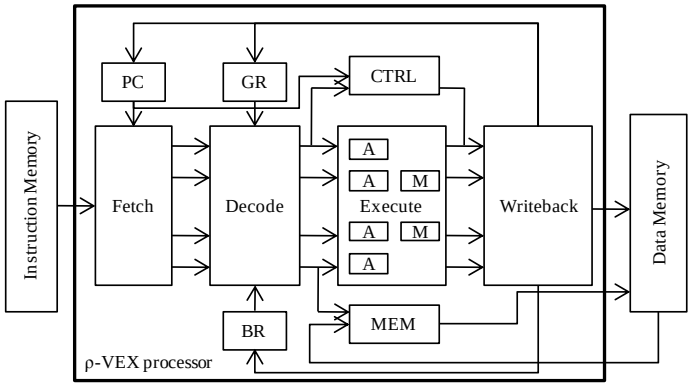


Fig. 8. The ρ-VEX VLIW processor [10][11].

In order to perform the stuck-at fault simulation experiments, we synthesized and implemented the ρ-VEX processor using a standard ASIC gate library. In total the number of faults is 387,290. The assembly code generated following the described method has been inserted into the instruction memory; then, a fault simulation experiment has been performed. Moreover, we wrote a prototypical tool (composed of about 3K lines of C++ code) implementing the proposed methods.

First of all, we have selected 6 SBST programs [8][12] [16][17][18] from the literature for testing the Functional Units embedded in the processor: each of them has been encoded in architecture-independent pseudo-code and has been inserted in the starting library. At the end of the fragmentation step we obtained a Fragments Library composed of 520 architecture-independent Fragments, while at the end of the Customization step the Custom Fragments Library was composed of 989 Custom Fragments.

5.1 Optimized SBST Program Generation Results

Using the technique for the maximum coverage with the minimum number of Fragments, 768 Custom Fragments have been selected and subjected to the scheduling step. At the end, we obtained the final test program for the test of the ρ-VEX processor: the generation time was approximately 40 hours, of which about 95% used for the fault simulation of the Custom Fragment. Computational time has been evaluated on a workstation with an Intel Xeon Processor E5450. We compared the test program generated by our approach with a test program consisting in several literature-based test programs simply queued in a unique test program, without performing any selection or scheduling steps, therefore adopting a realistic test estimation of what can be achieved with previously developed test algorithms without any optimization method. In order to fairly evaluate the two solutions, the original test programs have been ap-

plied using the loop-unrolling technique, as it is common for any VLIW application. In Table 3 we compare the obtained results.

As the reader can notice, while the coverage remains at the 98%, the number of clock cycles and the size of the test program generated with the proposed method decreased significantly. This is due to two causes: the former is that not all the Custom Fragments are chosen in the selection step; in fact the maximum coverage is reached with about 78% of the Custom Fragments. This comes from the fact that some fragments are aimed at detecting faults in some unit, which were already covered by Fragments targeted at other units. The latter is related to the scheduling step, that optimizes the code compacting the instructions, exploiting the VLIW features, and parallelizing as much as possible the execution of the Custom Fragments; consequently, the amount of clock cycles required by the final test program, is about 54% less than in the test program obtained using previously developed test programs without any selection or scheduling improvements.

It is also worth mentioning that the proposed method was able to reduce by about 58% the size of the test code. In Table 4 the achieved coverage for the relevant units of the ρ-VEX processor are reported.

Table 3. Optimized SBST program generation: obtained results.

Test Program	Clock cycle [#]	Fault Coverage	Size [KB]
Original Test Programs	18,540	98.2%	3,894
Proposed method	8,447	98.2%	1,612

Table 4. Details of the achieved fault coverage.

ρ-VEX Components		Faults [#]	Fault coverage
Fetch		2,156	99.2%
Decode		269,196	98.1%
Execute	4 ALU	75,554	98.3%
	2 MUL	37,244	98.6%
	MEM	1,730	97.2%
Writeback		1,420	98.1%
Total		387,290	98.2%

5.2 Diagnosis Evaluation Results

First of all we wrote a C++ program able to compare the fault lists generated by the Fault Simulation step (Section 4.2); the goal of this program is the detection of the number of distinguished faults and the classification of the undistinguished faults, i.e., the equivalent faults, in two categories: the first is composed of the faults which are equivalent and belonging to the same partition, while the second is composed of the

faults belonging to different partitions. For this purpose, we divided the ρ-VEX processor in 10 partitions: the fetch unit, the decode unit, the general-purpose register file, the branch-management register file, the write-back unit, and one for each Computational Domains (i.e., 4) in which the functional units are embedded.

Then, we run this program using two different sets of fault lists: the first contains only the fault lists associated to the Custom Fragments selected by the Selection step (Fig. 2, Step C.1) of the optimized generation of the SBST program, which are 78% of the total; the second set, instead, contains the fault lists of all the Custom Fragments generated by the Customization step. In Table 5 the results of these two experiments are reported.

As it is possible to notice, the set of all fault lists (set 2) allows to increment the number of distinguished faults and the number of the equivalent faults belonging to the same partition. Consequently, considering the results of Table 5, given a fault in the ρ-VEX processor, in about 82% of the cases we are able to identify the partition affected by the fault itself.

In Table 6 the evaluation of the Equivalence Classes, generated when all the fault lists of the all Custom Fragments are considered (fault lists set 2), is shown; the purpose of this evaluation is the classification of each equivalence class, based on the number of partitions with at least one fault in the considered equivalence class. As reported in Table 6, about 93% of the equivalence classes are composed of faults belonging to the same partition. In the other cases, as reported in the graph of Fig. 9, most of the classes are composed of equivalent faults belonging to two (54.52%) or three (38.81%) different partitions.

Table 5. Faults classification: diagnosis point of view.

Faults lists set	**Distinguished Faults**	**Equivalent Faults**		
		SAME partition	**DIFFERENT partitions**	**TOTAL**
1 – Optimized Test	1.13%	63.29%	35.59%	98.87%
2 - All	2.78%	79.15%	18.07%	97.22%

Table 6. Equivalence classes evaluation.

Partition [#]	**E.C. [#]**	**E.C. [%]**	**Faults Category**
1	14,319	92.90 %	Equivalent – SAME partition
2	597	54.52 %	Equivalent – DIFFERENT partition
3	425	38.81 %	
4	42	3.84 %	
5	21	1.92 %	
6	9	0.82 %	
7	0	0.00 %	
8	1	0.09 %	
9	0	0.00 %	
10	0	0.00 %	

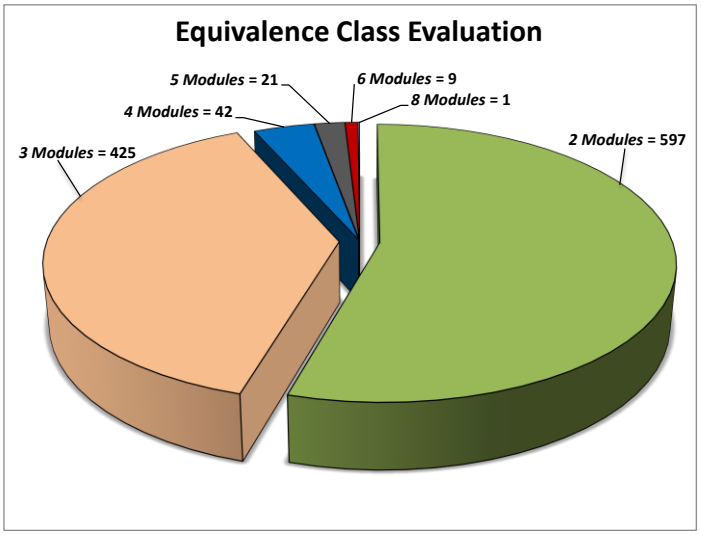


Fig. 9. The classification of the equivalent classes calculated using all the available faults lists.

6 Conclusions and Future Work

In this chapter we presented the first method able to automatically generate optimized Software-Based Self-Test programs for VLIW processors. The obtained results, with respect to the selected case study, clearly demonstrate the efficiency of our method, that allows to reduce significantly both the number of clock cycles and the required memory resources with respect to the plain application of previous methods. Moreover, it is also possible to exploit the proposed method to obtain a set of small SBST programs useful for the diagnosis of the considered VLIW processor.

As future work we plan to better evaluate the performance of the proposed solution with the use of another VLIW model with different Functional Units; moreover, we plan to generate small optimized SBST programs that can be specifically used for on-line testing and able to improve the diagnosis capabilities.

References

1. M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda, “Microprocessor software-based self-testing,” IEEE Design & Test of Computers, vol. 2, No. 3, pp. 4-19, May-June 2010.
2. P. Bernardi, E. Sànchez, M. Schillaci, G. Squillero, and M. Sonza Reorda, “An Effective Technique for Minimizing the Cost of Processor Software-Based Diagnosis in SoCs,” Design, Automation and Test in Europe, DATE '06, vol. 1, pp. 1–6, March 2006.
3. L. Chen and S. Dey, “Software-Based Diagnosis for Processors,” Design Automation Conference 2002, pp. 259–262, 2002.
4. J.A. Fisher, P. Faraboschi, and C. Young, “Embedded computing: a VLIW approach to architecture, compilers and tools,” Morgan Kaufmann, 2004.
5. B. Bornstein, T. Estlin, B. Clement, and P. Springer, “Using a multicore processor for rover autonomous science,” IEEE Aerospace Conference, pp. 1 –9, March 2011.
6. Tilera Corporation, “Multicore Development Environment User Guide,” Doc #UG201 Release 1.2, February 2008.
7. M. Beardo, F. Bruschi, F. Ferrandi, and D.Sciuto, “An approach to functional testing of VLIW architectures,” IEEE High-Level Design Validation and Test Workshop, pp. 29-33, 2000.
8. D.Sabena, M. Sonza Reorda, and L. Sterpone, “A new SBST algorithm for testing the register file of VLIW processors,” IEEE International Conference on Design, Automation & Test in Europe (DATE), pp. 412-417, March 2012.
9. M. Ulbricht, M. Schölzer, T. Koal, and H. T. Vierhaus, “A New Hierarchical Built-In Self-Test with On-Chip Diagnosis for VLIW Processors,” 2011 IEEE 14th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), pp. 143–146, April, 2011.
10. S. Wong, F. Anjam, and F. Nadeem, “Dynamically reconfigurable register file for a softcore VLIW processor,” IEEE International Conference on Design, Automation and Test in Europe (DATE), pp. 962 – 972, March 2010.
11. S. Wong, T. Van As, and G. Brown, “ρ-VEX: a reconfigurable and extensible softcore VLIW processor,” International Conference on ICECE Technology, pp. 369 – 372, December, 2010.
12. N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, “Software-based self-testing of embedded processors,” IEEE Transactions on Computers, vol. 54, No. 4, pp. 461-475, April 2005.
13. T. Koal and H.T. Vierhaus, “A software-based self-test and hardware reconfiguration solution for VLIW processors,” IEEE Symposium on Design and Diagnostic of Electronic Circuits and Systems (DDECS), pp. 40 – 43, April, 2010.
14. M. Ulbricht, M. Scholzel, T. Koal, and H.T. Vierhaus, “A new hierarchical built-in self-test with on-chip diagnosis for VLIW processors,” IEEE Symposium on Design and Diagnostic of Electronic Circuits and Systems (DDECS), pp. 143 – 146, April 2011.
15. A. Pillai, W. Zhang, and D. Kagaris, “Detecting VLIW hard errors cost-effectively through a software-based approach,” Advanced Information Networking and Applications Workshops, pp. 811-815, 2007.
16. D. Gizopoulos, M. Psarakis, M. Hatzimihail, M. Maniatakos, A. Paschalis, A. Raghunathan, and S. Ravi, “Systematic software-based self-test for pipelined processors,” IEEE Transaction on Very Large Scale Integration (VLSI) Systems, vol. 16, No. 11, pp. 1441 – 1453, November 2008.

17. A. Paschalis, D. Gizopoulos, N. Kranitis, M. Psarakis, and Y. Zorian, “Deterministic software-based self-testing of embedded processor cores,” IEEE International Conference on Design, Automation and Test in Europe (DATE), pp. 92-96, 2001.
18. N. Kranitis, D. Gizopoulos, A. Paschalis, and M. Psarakis, “Instruction-based self-testing of processor cores,” IEEE VLSI Test Symposium, pp. 223-228, 2002.
19. E. Sanchez, M. Sonza Reorda, and G. Squillero, “On the transformation of manufacturing test sets into on-line test sets for microprocessor,” IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pp. 494-502, October 2005.
20. P. Bernardi, E. Sánchez, M. Schillaci, G. Squillero, and M. Sonza Reorda, “An Effective Technique for the Automatic Generation of Diagnosis-Oriented Programs for Processor Cores,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Volume: 27 , Issue: 3, pp. 570 – 574, 2008.
21. M. Koester, W. S. Luk, J. Hagemeyer, M. Porrmann, and U. Rückert, “Design Optimizations for Tiled Partially Reconfigurable Systems,” Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, vol. 19, no. 6, pp. 1048 – 1061, 2011.
22. P.G.Ryan et al., “Fault dictionary compression and equivalence class computation for sequential circuits,” in Proc. IEEE Int. Conf. Comput.-Aided Des., pp. 508–511, 1993.