Effective Software-Based Self-Test Strategies for On-Line Periodic Testing of Embedded Processors

Antonis Paschalis
Department of Informatics & Telecommunications
University of Athens, Greece
paschali@di.uoa.gr

Dimitris Gizopoulos
Department of Informatics
University of Piraeus, Greece
dgizop@unipi.gr

Abstract

Software-based self-test (SBST) strategies are particularly useful for periodic testing of deeply embedded processors in low-cost embedded systems that do not require immediate detection of errors and cannot afford the well-known hardware, software, or time redundancy mechanisms.

In this paper, first, we identify the stringent characteristics of an SBST test program to be suitable for on-line periodic testing. Then, we introduce a new SBST methodology with a new classification scheme for processor components. After that, we analyze the self-test routine code styles for the three more effective test pattern generation (TPG) strategies in order to select the most effective self-test routine for on-line periodic testing of a component under test. Finally, we demonstrate the effectiveness of the proposed SBST methodology for on-line periodic testing by presenting experimental results for a RISC pipeline processor.

1 Introduction

The problem of testing embedded processors in high-complexity embedded systems is becoming more and more challenging at any level of the system life cycle. After manufacturing testing the embedded system is placed to its natural environment where operational faults may appear. Cosmic rays, alpha particles, electromagnetic interference, and power glitches are some of the main reasons for operational faults appearance. Operational faults are classified to *permanent faults*, which exist indefinitely, *intermittent faults*, that appear at regular time intervals and *transient faults* that appear irregularly and last for short time [1]. On-line testing aims at detecting and/or correcting these operational faults by means of *concurrent* and *non-concurrent* testing strategies.

Concurrent on-line test strategies are used to detect all kinds of operational faults within small time frame (low fault detection latency) while keeping the system in normal operation. These strategies utilize hardware redundancy techniques like duplication with compare, watchdog, and self-checking design [1], [2]. However, when large increase in silicon area is not acceptable, time or software redundancy techniques provide an alternative for on-line testing. These techniques are based on duplicating program statements, executing programs repeatedly [3] or implementing signature monitoring.

Non-concurrent on-line test strategies are particularly useful for periodic testing which assures system reliability. *On-line periodic testing* is useful in critical applications of embedded systems when it is combined with a concurrent test scheme to provide a comprehensive on-line testing strategy. Besides, in several non-critical low-cost applications of embedded systems there is no need for immediate detection of errors and thus no need for hardware, software or time redundancy mechanisms that increase significantly the system cost. In this case on-line periodic testing is also particularly useful since it detects permanent faults and intermittent faults with fairly large duration (when test is applied periodically).

The on-line periodic testing techniques are usually based on *hardware-based self-test* (HBST) techniques, like built-in self-test (BIST), that provide excellent test quality as they achieve the at-speed testing goal with high fault coverage. However, in the case of high performance, low area and low power consumption embedded processors, the application of HBST techniques is limited and sometimes prohibited since such embedded processors cannot tolerate performance degradation, high hardware overhead and increased power consumption.

Recently, the use of low-cost *software-based self-test* (SBST) techniques for on-line periodic testing of embedded processors has been proposed in [4] as effective alternatives to HBST techniques. The SBST techniques are non-intrusive in nature as they use the processor instruction set to perform self-testing. The key concept of software-based self-test is the generation of an efficient test program that leads to high fault coverage.

The processor executes periodically an efficient test program residing in the memory system (e.g. in a flash memory) at its actual speed (at-speed testing) and very small area, performance or power consumption overheads are induced for the embedded system. In the case of on-line periodic testing, the efficient test program must satisfy the following requirements: small memory footprint, small execution time and low power consumption [4]. In addition, there is a demand for low development cost of the efficient test program particular necessary for the case of low-cost applications of embedded systems.

SBST techniques *functional* in nature that use randomized instruction sequences have been proposed in [5]-[7]. Such techniques have low test development cost due to their high abstraction level, but they also have the drawback of achieving immediate to high fault coverage using a large number of instruction sequences. Thus, the derived test program has large size and requires excessive test execution time. In addition, long fault simulation time is required for fault grading. Therefore, these techniques are not suitable to on-line periodic testing.

SBST techniques *structural* in nature, targeting processor components have been proposed in [8]-[10] as promising techniques for efficient testing of a processor deeply embedded in an embedded system. Based on a divide-and-conquer approach, first, processor components and their corresponding component operations are identified. Then, for every component under test (CUT) within the processor and for every operation of the CUT, test patterns are generated targeting structural faults. After that, the test patterns are transformed to self-test routines (consisting of processor instruction sequences) which are used to apply test patterns to the inputs of the CUT and collect test responses from the outputs of the CUT. All self-test routines together constitute a

1530-1591/04 $20.00 (c) 2004 IEEE

test program with stringent requirements in code size, data size, and execution time in order to be suitable to on-line periodic testing of the embedded processor.

The test patterns are derived by following the three more effective test pattern generation (TPG) strategies. The first TPG strategy is based on deterministic automatic test pattern generation (ATPG) and is usually applied to combinational components, where instruction-imposed constraint ATPG is feasible. The second TPG strategy is based on pseudorandom TPG and is applied to combinational components with irregular structure, where instruction-imposed constraints can be taken into consideration. Both TPG strategies are low gate-level strategies, since they require the knowledge of the gate-level structure of the embedded processor. The third TPG strategy is based on regular deterministic TPG [9]-[10] that exploits the inherent regularity of the most critical to test processor components like arithmetic and logic components, shifters, comparators, multiplexers, registers and register files which usually constitute the vast majority of processor components. In many cases acceptable fault coverage is derived after testing only these components. This TPG strategy is a high-level strategy since the derived test patterns are independent on gate-level implementation and constitute test sets of constant or linear size.

In this paper, first, we identify the stringent characteristics of an SBST test program for on-line periodic testing. Then, we introduce a new SBST methodology for on-line periodic testing that includes a new classification scheme for processor components. This scheme is well suitable for the systematic selection of the convenient TPG strategy for test pattern derivation, as well as, the systematic transformation of test patterns to a self-test routine. After that, we analyze the self-test routine code styles for the three TPG strategies with respect to code size, data size and execution time characteristics. Such an analysis is needed to select the most effective self-test routine for on-line periodic testing according to specific processor component test pattern characteristics. Finally, we demonstrate the effectiveness of our SBST methodology for on-line periodic testing by presenting experimental results for a pipeline RISC processor of MIPS architecture.

2 On-Line Periodic Test Program Characteristics

On-line periodic testing is performed in-field while the processor operates at its normal operational environment. The processor executes the efficient SBST program residing in the memory system (e.g. in a flash memory) at its actual speed (at-speed testing) under the supervision of the operating system.

Test program execution may be initiated during system startup or shutdown thus ensuring system normal operation with respect to permanent faults, but it imposes large fault detection latency. Alternatively, the operating system scheduler may identify idle cycles and issue test program execution or test program may be executed at regular time intervals with the aid of programmable timers found in the system. In these cases, the SBST program for on-line periodic testing is another process that has to compete with user processes for system resources, CPU cycles and memory.

To alleviate the system operation overhead, a SBST program should run in the minimum possible number of CPU clock cycles. An ideal period for test program execution should be the *quantum time cycle* assuming an operating system with round robin scheduling. Typical values of quantum times used in embedded applications are in the range of a few hundreds of milliseconds (msec). Although it is possible to have test program execution span over more than one quantum time, this will lead to further system operation overhead due to larger context switch overheads.

In case that the operating system scheduler identifies idle cycles and issues test program execution, *fault detection latency* depends on the test program execution time. The test program execution time must be as short as possible and less than a quantum time cycle in order to reduce fault detection latency.

In case that test program is executed at regular time intervals with the aid of programmable timers, fault detection latency depends on the time interval between two consecutive test program executions, as well as, the test program execution time. The time interval is specified as a tradeoff between user program performance and fault detection latency with capability of the system to detect intermittent faults. Also, the test program execution time must be as short as possible and less than a quantum time cycle in order to reduce fault detection latency.

Therefore, the main characteristic of an SBST test program to be suitable for on-line periodic testing is *the shortest possible test execution time which must be less than a quantum time cycle*.

The test program execution time can be generally described by the following equation [11]:

```
CPU-execution-time = clock-cycle-time ×
   ( CPU-clock-cycles +
     pipeline-stall-cycles +
     memory-stall-cycles )
```

The existence of pipeline stalls and memory stalls increase test program execution cycles and must be avoided if it is possible.

Pipeline stalls should be avoided by constructing test programs which do not cause unresolved data hazards. Control hazards are usually avoided in architectures that implement the *branch delay slot* resolution, like MIPS, by proper instruction placement in the delay slot. However, pipeline stalls are unavoidable when *branch prediction* is used to handle branch conditions.

Test programs with big memory footprint (code and data) take more time to run due to increased number of memory stalls. Additionally, such a test program may force user programs to be unloaded from cache memory. When user program resumes, it will experience cache misses which will affect its performance. Memory stalls are reduced when test programs take advantage of *temporal* and *spatial* locality.

A common application for on-line periodic testing is mobile applications where power consumption is of great importance. A study by Intel [12] shows that 33% of a notebook system power is consumed in the CPU with a 2-level cache hierarchy system. In the CPU about 20%-30% of power is consumed in the cache system and about 30% is consumed in clock circuitry. Considering the data transfers from external memory in case of a cache miss the power consumed in the overall memory system increases furthermore. The processor has to stall when a cache miss occurs. Extra energy has to be consumed in driving clock-trees and pulling up and down the external bus between the on-chip cache and external memory. Therefore, reduction of memory stalls also reduces power consumption during on-line periodic testing.

Consequently an SBST program for on-line periodic testing must have the following stringent characteristics:

- ❑ The shortest possible test execution time which must be less than a quantum time cycle.
- ❑ Small code without unresolved data hazards and with as much as possible compact loops that take advantage of temporal locality and sequentially executed instructions that take advantage of spatial locality.
- ❑ Small data structured in arrays that take advantage of spatial locality.

3 SBST Methodology for on-line periodic testing

The introduced here SBST methodology for on-line periodic testing of embedded processors consists of three phases as follows:

Phase A: Identification of component operations and processor components *with relevant multiplexers*, as well as, instructions that excite component operations and instructions (or instruction sequences) for controlling or observing processor registers.

Phase B: Classification of processor components in classes with the same properties and component prioritization for test development. This new classification scheme is well suitable for the systematic selection of a convenient TPG strategy for test pattern derivation, as well as, the systematic transformation of test patterns to a self-test routine.

Phase C: Development of self-test routines based on specific self-test routine code styles for the three TPG strategies with respect to the stringent characteristics for on-line periodic testing.

These phases are explained in more details in the following subsections.

3.1 Information extraction

The starting point of our SBST methodology is the instruction format derived from the processor instruction set architecture (ISA) and the low register transfer level RTL description of the processor micro-operations derived from the RTL description of the processor.

Based on this information first we identify the component operations and the processor components with specific inputs and outputs that carry out these component operations. At this stage we identify possible multiplexers appearing in the inputs or the outputs of the processor components. Then, we map component/multiplexer inputs and outputs to internal temporary registers for multi-cycle datapaths or pipeline register fields for pipelined datapaths.

Then, we identify the instructions that carry out a specific operation and excite the pertinent processor component with the corresponding multiplexer(s) and register(s), if they exist.

Finally, we identify appropriate instruction(s) to control the values of processor component/multiplexer inputs or the corresponding registers; this is the controllability part of the methodology. Also, we identify instruction(s) to ensure propagation of the processor component/ multiplexer outputs or the corresponding registers to processor primary outputs; this is the observability part of the methodology. Both the control and observe processes can be performed using single processor instructions or an instruction sequence.

3.2 Component classification and test priority

From the information extracted in Phase A we classify the processor components in the following three classes:

Visible components (VC)

The components of the embedded processor whose inputs and outputs are visible to Assembly language programmer. For these components there is at least one instruction or instruction sequence that controls their inputs or the corresponding registers. Also, there is at least one instruction or instruction sequence that ensure propagation of their outputs or the corresponding registers to processor primary outputs. These components are further classified in three sub-classes according to the type of their inputs and outputs (data or addresses).

Data visible components (D-VC): The inputs of these components receive data test patterns that can be stored: (a) in fields of an instruction with immediate addressing mode, (b) in the register file with an instruction with register addressing mode, or (c) in the data memory. The outputs of these components produce data responses that can be stored: (a) in the register file, (b) in the data memory, or (c) in a data register directly connected to register file or data memory. Such components are: *ALUs, shifters, multipliers, dividers, special data registers, the data fields of pipeline registers and the register file*. In this class also belong the *multiplexers* at the inputs or the outputs of these components. The data visible components have the highest test priority since they have the highest testability and dominates the processor area. These components are suitable for on-line periodic testing and in many cases their testing results in acceptable fault coverage.

Address visible components (A-VC): The inputs and outputs of these components receive addresses of the memory system. The values of these addresses depend on the memory positions where the instructions or the data will store. Thus, these components become visible with convenient storing of instructions or data in the memory system. Such components usually appear inside the *instruction fetch unit* and the data memory controller (i.e. the *memory address register*). In this class also belong special address registers and the address fields of pipeline registers, as well as, the *multiplexers* at the inputs or the outputs of these components. These components are not suitable for on-line periodic testing, since they require a lot of distributed memory references and the derived self-test routines can not take advantage of temporal and spatial locality which reduces the cache miss overhead. These components occupy a very small part of processor area and are partially tested as a side-effect of testing the D-VCs. The A-VCs are tested after the D-VCs only in case that the fault coverage is not acceptable.

Mixed (address-data) visible components (M-VC): These components have inputs and/or outputs of both types (address or data) which can become visible in a way mentioned above. For example, in this sub-class the *adder* used for implementation of PC-relative addressing is included. These components have the same characteristics with address visible components.

Partially visible components (PVC)

The components of the embedded processor that generate control signals and are usually implemented as *finite state machines*. Since the control outputs of these components affect the operation of visible components these components can be considered as partially visible to Assembly language programmer. Such a component is the processor control unit. These components have medium testability. To test such components we adopt simple high level functional tests like the application of all instruction opcodes for the case of testing the processor control logic, as well as, the application of instructions that achieve the most possible RTL code coverage for the case of testing a specific finite state machine. These components occupy a very small part of processor area and may be suitable for on-line periodic testing.

Hidden components (HC)

The components of the embedded processor that are added in a processor architecture usually to increase its *performance*, but they are not visible to the assembly language programmer. These components include a small portion of pipeline registers (except address and data fields belonging to visible components), pipeline control units (e.g. forwarding unit, hazard detection unit), pipeline multiplexers, branch prediction mechanism and other performance increasing components related to *instruction level parallelism* (ILP) techniques. Besides, registers, multiplexers and control logic to handle interrupts and exceptions are included in hidden components. We remark that hidden components, especially those used for data pipelining, are sufficiently tested as a side-effect of testing the D-VCs.

3.3 Self-test routine development

The self-test routine development starts with test pattern derivation and continues with the transformation of test patterns to self-test routines which satisfies the test program requirements mentioned above for on-line periodic testing. Test patterns and the corresponding self-test routine code styles are derived according to the following three effective TPG strategies.

Deterministic ATPG based TPG strategy

The first TPG strategy is based on deterministic automatic test pattern generation (ATPG) and is usually applied to combinational D-VCs, where instruction-imposed constraint ATPG provided by commercial tools is feasible. This strategy is a low gate-level strategy, since it requires the knowledge of the gate-level structure of the embedded processor.

The ATPG based test patterns are transformed to effective self-test routines with two alternative ways: (a) the test patterns are transformed to instructions supporting immediate addressing, or (b) the test patterns are stored in memory system and a loop-based self-test routine fetches these from the memory system and applies to CUT.

Let us assume that the instruction *function* with register addressing carries out the specific operation "function" and excite the pertinent D-VC with the corresponding multiplexer(s) and register(s), if they exist.

The self-test routine code style that generates a small number of *n* test patterns for D-VCs with two inputs X and Y by using instructions with immediate addressing mode, is shown in Figure 1 (following MIPS Assembly language in all figures).

```
li $s0, pattern_X_1;
li $s1, pattern_Y_1;
function $s2 $s0, $s1;
jal compaction_routine_address;
          ......
li $s0, pattern_X_n;
li $s1, pattern_Y_n;
function $s2 $s0, $s1;
jal compaction_routine_address;

li $s3, signature_address;
sw $s2, (signature_displacement) ($s3);
```

Figure 1: ATPG based code style with immediate instructions

Test patterns are loaded in registers using the `li` (load immediate) pseudo-instruction, which the assembler decomposes to instructions `lui` and `ori` without transferring data from memory. After test pattern application, test responses are compacted by using a compaction routine, usually a software MISR routine with negligible aliasing to avoid transferring of data to memory that imposes data cache miss. At the end the final signature is unloaded to data memory at the address `signature_address+signature_displacement` for error identification.

This self-test routine has the following characteristics:

□ The code size depends linearly on the number of test patterns. If the number of test patterns is small enough, the code size is small, as well.

□ Code without unresolved data hazards (assuming that the processor supports forwarding) with sequentially executed instructions that take advantage of spatial locality.

□ There is a high instruction miss rate alleviating by the spatial locality and depending linearly on the number of test patterns.

□ No load data memory reference and only one store data memory reference impose no data cache miss.

Alternatively, the self-test routine code style that generates a small number of *n* test patterns for D-VCs with two inputs X and Y by using data fetching from the memory system and a loop based code, is shown in Figure 2.

```
   li $s3, first_pattern_address;
   addi $s4, $zero, number_of_test_patterns
   add $t0, $zero, $zero;
test_pattern_loop:
   lw $s0, 0($s3);                 # pattern_X
   addiu $s3, $s3, 0x0004;         # for 32-bit data
   lw $s1, 0($s3);                 # pattern_Y
   addiu $s3, $s3, 0x0004;
   function $s2 $s0, $s1;
   jal compaction_routine_address;
   addiu $t0, $t0, 0x0001;
   bne $s4, $t0, test_pattern_loop;

   li $s5, signature_address;
   sw $s2, (signature_displacement) ($s5);
```

Figure 2: ATPG based code style with data fetching

We assume that the first test pattern is in data memory address first_pattern_address, as well as, the number of test patterns for both inputs is number_of_test_patterns. All test patterns are loaded, then are applied to CUT and afterwards test responses are compacted. At the end the final signature is unloaded to data memory for error identification.

This self-test routine has the following characteristics:

□ The code size is small and independent of the number of test patterns.

□ Code without unresolved data hazards (assuming that the processor supports forwarding) with compact loop that takes advantage of temporal locality.

□ There is low instruction miss rate alleviating by the spatial locality and depending linearly on the number of test patterns.

□ There is high data miss rate alleviating by the spatial locality and depending linearly on the number of test patterns.

Both alternative ways of effective self-test routines are used in practice for on-line periodic testing since they have short execution time, assuming that the number of ATPG based test patterns is small. The former has high instruction miss rate while the latter has high data miss rate. The selection is mainly based on test routine execution time and depends on the clock cycles per instruction (CPI) of the pertinent instructions and especially of instruction `lw`.

Pseudorandom based TPG strategy

The second TPG strategy is based on pseudorandom TPG and is also usually applied to combinational D-VCs with irregular structure, where instruction-imposed constraints can be taking into consideration. The pseudorandom test patterns are transformed to a loop-based software LFSR self-test routine. This strategy also is a low gate-level strategy, since it requires the knowledge of the gate-level structure of the embedded processor.

Let us assume an instruction *function* with register addressing that excite the pertinent D-VC as previously.

The self-test routine code style that generates a large number of pseudo-random test patterns for D-VCs with two inputs X and Y by using a loop based code, is shown in Figure 3.

```
   li $s3, seed;
   li $s4, polynomial;
   addi $s5, $zero, number_of_test_patterns;
   add $t0, $zero, $zero;
test_pattern_loop:
   # LFSR generation for $s0; # pattern_X
   # LFSR generation for $s1; # pattern_Y
   function $s2 $s0, $s1;
   addiu $t0, $t0, 0x0001;
   jal compaction_routine_address;
   bne $s5, $t0, test_pattern_loop;
   li $s6, signature_address;
   sw $s2, (signature_displacement) ($s6);
```

Figure 3: Pseudorandom based code style

We assume that the number of test patterns for both inputs is number_of_test_patterns and seed and polynomial are used by the software implemented LFSRs. All pseudo-random generated test patterns are applied to CUT and afterwards test responses are compacted. At the end the final signature is unloaded to data memory for error identification.

This self-test routine has the following characteristics:

- The code size is small and independent of the number of test patterns.
- Code without unresolved data hazards (assuming that the processor supports forwarding) with compact loop that takes advantage of temporal locality.
- There is low instruction miss rate alleviating by the spatial locality and depending linearly on the number of test patterns.
- No load data memory reference and only one store data memory reference impose no data cache miss.

Pseudorandom-based self-test routines are used when a D-VC with irregular structure is considered for on line periodic testing. Such test routines usually have large execution time since processor components are random-pattern resistant and thus a large number of test patterns must be applied to reach acceptable fault coverage.

Regular Deterministic based TPG strategy

The third TPG strategy is based on regular determinist TPG [9]-[10] that exploits the inherent regularity of the most critical to test processor components like arithmetic and logic components, shifters, comparators, multiplexers, registers and register files which usually constitute the vast majority of processor components. In many cases acceptable fault coverage is derived after testing only these components. This TPG strategy is a high-level strategy since the derived test patterns are independent on gate-level implementation and constitute test sets of constant or linear size. The regular deterministic based test patterns are transformed to self-test routines with two alternative ways: (a) the test patterns are transformed to instructions supporting immediate addressing mode in case that the test set size is small enough, or (b) the test patterns are transformed to a loop-based self-test routine with an initial value, a final value and a specific function to generate the regular transition from the one value to the other.

In case that the number of regular deterministic test patterns is small enough, we use the processor instruction set with immediate addressing mode to generate and apply test patterns as it is shown in Figure 1. This is also the case for the register file, where all registers of the register file must receive two patterns. In order to avoid stores in data memory the testing of register file is done in two phases, as follows. In the first phase we test the one half of the register file by using registers of the other half for compaction, while in the second phase we do the opposite.

Otherwise, let us assume an instruction *function* with register addressing that excite the pertinent D-VC as previous.

The self-test routine code style that generates a small number of regular deterministic test patterns for D-VCs with two inputs X and Y by using a loop based code, is shown in Figure 4.

```
   li $s0, initial_value_X;          # initiate X
   li $s3, initial_value_Y;
   li $s4, final_value_X;
   li $s5, final_value_Y;
   add $s1, $s3, $zero;              # initiate Y
test_pattern_loop:
   function $s2 $s0, $s1;
   # generate next Y pattern in $s1;
   jal compaction_routine_address;
   bne $s5, $s1, test_pattern_loop;
   add $s1, $s3, $zero;             # initiate Y
   # generate next X pattern in $s0;
   bne $s4, $s0, test_pattern_loop;
   li $s6, signature_address;
   sw $s2, (signature_displacement) ($s6);
```

Figure 4: Regular deterministic based loop code style

We have assumed that for every value of input X all values of input Y are applied to CUT and afterwards test responses are compacted. In some cases the final value is exactly the initial value and some instructions are deleted. At the end the final signature is unloaded to data memory for error identification.

This self-test routine has the following characteristics:

- The code size is small and independent of the number of test patterns.
- Code without unresolved data hazards (assuming that the processor supports forwarding) with compact loop that takes advantage of temporal locality.
- There is low instruction miss rate alleviating by the spatial locality and depending linearly on the number of test patterns.
- No load data memory reference and only one store data memory reference impose no data cache miss.

We remark that in any case `nop` instructions are inserted accordingly when forwarding is not supported. Also, similar self-test routines can be derived without any effort if we consider an instruction *function_I* with immediate addressing to carry out the specific operation “function”.

TPG strategy applicability for on-line periodic testing

Comparing the applicability of the three TPG strategies for on-line periodic testing in order to select the most effective one for a specific number of test patterns, we conclude that:

- The deterministic ATPG based TPG strategy is applicable to combinational D-VCs in case that the number of test patterns is small enough.
- The pseudo-random based TPG strategy is usually applicable to combinational D-VC with irregular structure in case that the larger number of test patterns leads to affordable execution time.
- The regular deterministic TPG strategy is applicable to combinational or sequential D-VCs with inherent regularity which dominate the processor area and cover the vast majority of faults.

4 Experimental results

A 32-bit embedded RISC processor core of MIPS architecture named Plasma that implements 3-stage pipeline with forwarding is used to demonstrate the effectiveness of the proposed SBST methodology for on-line periodic testing [13]. The Plasma core was enhanced with a fast parallel multiplier [14] and was synthesized with area optimization at *26,080* gates targeting a 0.35um technology library. The design runs at a clock frequency of *57* MHz. Mentor Graphics suite was used for VHDL synthesis, functional and fault simulation (Leonardo, ModelSim and FlexTest products, respectively).

The CUTs with the highest priority for on-line periodic testing are the D-VCs (parallel multiplier, serial divider, register file, shifter and ALU), the PVC (control logic) and the memory controller which is 73% D-VC (memory data register and data multiplexers), 23% A-VC (memory address register) and 4% PVC (special control). The D-VCs dominate the processor area (92%).

We have applied to all CUTs the most effective TPG strategies with low development cost and we have achieved an acceptable high fault coverage of *95.6*%. We have used regular deterministic code style (RegD) with loops (L) or immediate type instructions (I) for all D-VCs except the shifter where we use ATPG deterministic code style (AtpgD) with immediate type instructions (I). Also we have used functional tests (FT) for the control logic. For every self-test routine a final signature is derived after compaction of all responses by using a shared software MISR routine of 8 words. At the end of periodic testing 7 signatures, one for every CUT, are unloaded to data memory for fault detection.

Component gate count and classification, self-test program statistics (code style, program size in words, CPU clock cycles and data memory references – loads and stores), along with the achieved single stuck-at fault coverage and the percentage of the processor overall fault coverage which is missing from each of the CUTs, are presented in specific columns of Table 1, respectively.

The derived self-test program for on-line periodic testing has the required stringent characteristics:

- A very small code of only 808 words without pipeline stalls that takes advantage of temporal locality and spatial locality.
- A small number of only 87 memory data references that imposes a small number of data cache misses. The only CUT that requires 80 loads and stores for test application is the memory controller.
- A very short CPU execution time of *9,905* clock cycles. Assuming an average instruction/data cache miss rate of 5% and a miss penalty of 20 clock cycles, the test execution time is less than 11.000 clock cycles or less than 200 usec which is much less than a quantum time cycle.

5 Conclusions

We have shown that the introduced here SBST methodology for embedded processors results in very effective SBST strategies for on-line periodic testing. Both types of permanent and intermittent faults are detected by a small embedded test program with test execution time much less than a quantum time cycle. SBST for on-line periodic testing can be applied to improve reliability of low-cost embedded systems based on embedded processors where hardware, software or time redundancy can not be applied due to their excessive cost in terms of silicon area and execution time.

References

[1] H. Al-Assad, B. T. Murray, J. P. Hayes, *"Online BIST for Embedded Systems"*, in IEEE Design & Test of Computers, vol.15, no.4, Oct.-Dec. 1998, pp.17-24.

[2] M. Nicolaidis, Y. Zorian, *"On-line Testing for VLSI – A Compendium of approaches"*, in Journal of Electronic Testing: Theory and Applications, Vol. 12, No. 1-2, 1998, pp 7-20

[3] N. Oh, E. J. McCluskey, *"Error Detection by Selective Procedure Call Duplication for Low Energy Consumption"*, in IEEE Trans. on Reliability, Vol. 51, No. 4 December 2002 pp.392-402

[4] G. Xenoulis, D.Gizopoulos, N. Kranitis, A.Paschalis, *"Low-Cost On-Line Software-Based Self-Testing for Embedded Processor Cores"*, in Proc. of IEEE International On-Line Testing Symposium 2003, pp. 149-154.

[5] J. Shen, J. Abraham, *"Native mode functional test generation for processors with applications to self-test and design validation"*, in Proc. of IEEE International Test Conference 1998, pp. 990-999.

[6] K. Batcher, C. Papachristou, *"Instruction randomization self test for processor cores"*, in Proc. of the VLSI Test Symposium 1999, pp. 34-40.

[7] P. Parvathala, K. Maneparambil, W. Lindsay, *"FRITS – A Microprocessor Functional BIST Method"*, in Proc. of the IEEE International Test Conference 2002, pp. 590-598.

[8] Li Chen, S. Dey, *"Software-Based Self-Testing Methodology for Processor Cores"*, IEEE Transactions on CAD of Integrated Circuits and Systems, vo.20, no.3, pp. 369-380, March 2001.

[9] N. Kranitis, D. Gizopoulos, A. Paschalis, Y. Zorian, *"Instruction-Based Self-Testing of Processor Cores"*, in Proc. of the IEEE VLSI Test Symposium 2002, pp. 223-228.

[10] N. Kranitis, G. Xenoulis, A. Paschalis, D. Gizopoulos, Y. Zorian, *"Application and Analysis of RT-Level Software-Based Self-Testing for Embedded Processor Cores"*, in Proc. of IEEE International Test Conference 2003.

[11] J. Hennessy, D. Patterson, *"Computer Architecture A Quantitative Approach"*, MKP, 1996.

[12] Intel Corporation, Mobile Power Guidelines 2000, Dec 11, 1998.

[13] Plasma CPU Model. http://www.opencores.org/projects/mips

[14] J. Phil, E. Sand, Arithmetic Module Generator for High Performance VLSI Designs. http://www.fysel.ntnu.no/modgen

Component	Gate Count (gates)	Classification	Code Style	Size (words)	CPU Clock Cycles	Data Refer.	FC (%)	Miss. FC (%)
Parallel Mul. – Serial Div.	11,601	D-VC	RegD (L + I)	68	6,848	2	96.3	1.8
Register File	9,905	D-VC	RegD (I)	278	1,302	1	97.8	0.7
Memory controller	1,119	73% D-VC	RegD (I)	113	357	81	90.3	0.3
Shifter	682	D-VC	AtpgD (I)	195	571	1	99.9	0.0
ALU	491	D-VC	RegD (L + I)	61	582	1	96.8	0.1
Control Logic	230	PVC	FT	85	245	1	89.3	0.1
Pipeline	885	HC					98.4	0.0
Remaining	1,167	39% D-VC		8			63.1	1.4
Total	26,080	92% D-VC		808	9,905	87	95.6	4.4

Table 1: Component gate count and classification, self-test program statistics and fault coverage of MIPS Plasma for on-line periodic testing.