Software Based Self-Test of Register Files in RISC Processor Cores using March Algorithms

Matthieu Tuna, Mounir Benabdenbi
Laboratoire LIP6, Dept ASIM
University of Pierre et Marie Curie, Paris VI
{matthieu.tuna, mounir.benabdenbi}@lip6.fr

Abstract

Beside external test and hardware Built-In Self-Test techniques, a non-intrusive low-cost strategy has been developed: Software-Based Self-Test. SBST component-oriented approach for microprocessor allows to reach, with small self-test programs, a high fault coverage. In this context, this paper focuses on the test of the Register File of RISC processor cores. This component has the structure of a small memory. Then, using the appropriate March tests, 100% fault coverage for numerous fault models can be reached. This method was applied to test the Register File of a MIPS microprocessor. Results show that the method allows to obtain small-sized self-test programs while the use of March tests ensures a high test quality.

1 Introduction

With the advent of Systems on a Chip (SoCs), design methodologies are mainly driven by the Time-To-Market, and therefore are more and more based on the use of predesigned Intellectual Property (IP) cores. IP cores such as microprocessors, peripherals, or dedicated blocks are bind together to make complex SoCs. This deep integration has consequences for the test process. Poor test accessibility for external testers increase test application time, huge memory is required to store test data volume. Moreover at-speed testing is becoming a very challenging task since the gap between the tester's frequencies and SoC frequencies is getting larger.

A solution to alleviate the role of the Automated Test Equipment (ATE) can be find in hardware Built-In Self-Test (BIST) techniques. Internal hardware test generators and test response analyzers are used to generate and apply test patterns at the speed of the circuit. Unfortunately, for random-pattern resistant structures like microprocessor, extensive use of test point insertion is needed to improve initial low fault coverage. This extra added circuitry may result in significant performance degradation.

In the other hand, with SoC paradigm, alternative solution like Software-Based Self-Testing (SBST) methodologies re-emerge for embedded processors test. Software-Based Self-Testing consists on the execution of a program by an embedded microprocessor. Thus, functional internal resources are used to vehicle test patterns targeting structural faults. The self-test program is initially downloaded into the internal instruction memory thanks to a low-cost external equipment. Then, the microprocessor executes the self-test program, computes and stores self-test responses in the data memory. Finally, the external equipment brings back those self-test responses for evaluation.

Different methodologies have been proposed for the development of the self-test program. Previous works like [1], [2] and [3], based on the microprocessor Instruction Set Architecture (ISA), have a low structural fault coverage since their approaches are functional. In [4] the authors proposed a structural component-oriented approach. High fault coverage is reached, but the resulting self-test program is large since it is generated on-chip and applies pseudo-random patterns. Another component-oriented approach has been presented in [5], where small specific self-test routines, based on compact loops of instructions, are developed for each component. Moreover, a classification of all microprocessor components is done in order to prioritize the development of the self-test program of each component. High priority is given for components with a great silicon area (high percentage of the total fault coverage) and directly accessible by the ISA. A contrario, low priority is given to small components with few access. Having a hierarchy for the components to be tested reduces the total test development time.

The approach in [5] allots high priority to microprocessor key components like the Register File . Therefore, special attention must be paid to achieve a high test quality for high priority components. This paper focuses on the software-based self-test of the Register File (RF) and describes a methodology to develop small self-test programs

Algorithm	Algorithm sequence	Faults detected			
		SAF	AF	TF	SCF
MATS	$\{\updownarrow(w0);\updownarrow(r0,w1);\updownarrow(r1)\}$	All	Some		
MATS+	$\{\updownarrow(w0);\uparrow(r0,w1);\downarrow(r1,w0)\}$	All	All		
MATS++	$\{\updownarrow(w0);\uparrow(r0,w1);\downarrow(r1,w0,r0)\}$	All	All	All	
MARCH X	$\{\updownarrow(w0);\uparrow(r0,w1);\uparrow(r1,w0);\updownarrow(r0)\}$	All	All	All	
MARCH C-	$\{\updownarrow(w0);\uparrow(r0,w1);\uparrow(r1,w0);\downarrow(r0,w1);\downarrow(r1,w0);\updownarrow(r0)\}$	All	All	All	All

Table 1. Different Fault Model detected by March Algorithms.

while keeping a high fault coverage. This paper points out the fact that the physical structure of an RF is the same as a little Static Random Access Memory (SRAM). The outcome is that appropriate fault models for the RF must be taken into account. Thus, these fault models should be the same as any memory. Insight of that, we decided to create a methodology relying on March tests to develop self-test programs for RFs.

The Software-Based Self-Test of RISC microprocessor RF was first tackled in [6], however the authors do not consider it as a memory: the RF used is synthesized in a logic-gates netlist. Therefore, only the stuck-at fault model was targeted. Using only this fault model leads to consider only 50% of all the faults occurring in the RF component [7].

This paper is organized as follows. Section two presents the physical structure of a register file, some appropriate fault models and a recap of March tests. The third section describes a methodology for self-test program development based on the use of March-based algorithms applied to RFs. Several March tests were implemented, and experimental results are presented in the section four. Finally the fifth section concludes this paper.

2 Targeted Fault Models and Tests for RF

Appropriate tests for relevant fault models ensure detection of physical defects leading to high quality test. Therefore, the purpose of this section is to present the physical structure of an RF in order to show relevant fault model and a selection of appropriate tests.

A RISC-architecture processor uses a large number of general-purpose registers (GPRs). Those GPRs are grouped together in one component : the Register File (RF). De facto, the Register File of the processor is one of the largest components and has an inherent regularity. Most RFs of pipelined RISC processor cores have three ports, one for writing, two for reading. If we consider a 32-bit architecture microprocessor, the general case is an RF including 32 registers of 32-bits wide. Therefore, this RF contains 1024 memory elements accessed for a read or a write through a 32-bit word. In order to shrink size and consumption, the RF is designed as a multi-port memory [8]. During the layout generation of a processor core, the RF is integrated as a pre-designed macro-cell, from a full-custom library, or generated by CAD-tools (in case of standard-cell layout). Usually, memory generator tools can provide a variety of SRAM-memories such as FIFOs, ROMs, Memory Caches, and Register Files. As a memory device the RF includes three major parts: the memory array, the address decoder and the read/write logic. The memory array is typically made of memory cells based on the 6-transistor SRAM-cell with two bit lines per write port and one bit line per read port [9].

Thus, the RF can have the same physical defects as a memory. Lot of works have been done in the memory testing field to create fault model based on error caused by physical defect. We succinctly present some RAM fault models [10] used in this paper. The *Stuck-At Fault* (SAF) occurs when the logic value of a cell is always 0 (SA0) or always 1 (SA1). An *Address Decoder Fault* (AF) corresponds to read or write in another cell than the expected one . A *Transition Fault* (TF) occurs when a cell fails to transit from 0 to 1 or from 1 to 0 when attempting to write a value. The *State Coupling Fault* (SCF) occurs when the coupling cell *j* is in a given state *y* that forces the coupled cell *i* into state *x*.

In order to detect those RAM faults, well-known March tests have been developed. The Table 1 presents several March-based algorithms and corresponding detected faults. As we say, the RF is only accessible using a word (or register), in the special case of Word-Oriented Memories (WOMs), the read and write operations involve reading and writing a word of data called *Data Background* (DB). For instance, the word-oriented MATS algorithm becomes $\{\updownarrow(\mathrm{w}a);\updownarrow(\mathrm{r}a,\mathrm{w}\bar{a});\updownarrow(\mathrm{r}\bar{a})\}$ where a is the *Data Background* and $\bar{a}$ the *Inverted Data Background*. Fault occurring on a single cell (e.g. SAF) can still be detected using any DB and the corresponding Inverted DB. However, faults involving two cells (e.g. CF) introduce the problem of detecting faults between two cells at one address (i.e. the same word). In [11], Van De Goor divides faults between memory cells into inter-word faults and intra-word faults. The only way

to detect the latter kind of faults is to execute the March test several times using different DB. In [7] a method to construct the DBs for state CFs has been given. The table 2 presents the 6 DBs for a 32-bit word memory. These 6 DBs will be the ones used to test the RF.

Number	Data Background	
	normal	inverted
1	0x00000000	0xFFFFFFFF
2	0x0000FFFF	0xFFFF0000
3	0x00FF00FF	0xFF00FF00
4	0x0F0F0F0F	0xF0F0F0F0
5	0x33333333	0xCCCCCCCC
6	0x55555555	0xAAAAAAAA

Table 2. Different DB for state CF.

3 RF self-test program implementing March algorithms

The main goal of this section is to present the self-test program development enabling the test of the RF component. The self-test program must implement a March Algorithm in order to detect any fault detailed in the previous section. Globally, the testing process consists in filling the entire RF (in increasing or decreasing order) with a *data background* (DB) and reading in the RF (also in increasing or decreasing order) the expected *data background*. The first part of this section presents the way to apply the different operations of a March Algorithm such as *w<DB>, r<DB>*, and how to increase or decrease the sequence order. This section highlights the fact that the program needs some registers to store two kinds of informations. The first kind of informations, needed by the March operations, is what we call the persistent informations (typically the data backgrounds). The second kind of informations are those needed for the reduction of the self-test program size (use of variables, counters, addresses, function arguments). Both kinds of informations are stored in a set of registers called the *reserved area* which does not undergo the test. The first part of this section details the program corresponding to the execution of March elements. It also describes what kind of informations need to be stored in the *reserved area*. The second part of this section exhibits a solution to test the overall RF, including this *reserved area*.

3.1 Write, Read and sequence order

In the following we consider a Reduced Instruction Set Computing (RISC) architecture: the MIPS architecture. All MIPS instructions are 32-bits in length, the Register File has 32 registers named from $0 for the register number 0 to $31 for the register number 31. The Instruction Set Architecture (ISA) of a MIPS-architecture processor is divided into three different coding formats: R, I and J. Since only R and I-Format address the registers of the RF, we present only this two format (see figure 1).

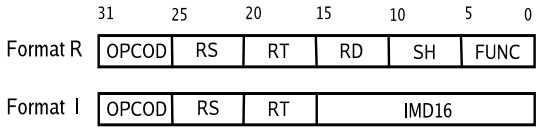

Figure 1. R-Format and I-Format of the MIPS ISA

The I-Format is used for load and store operations (lw/sw), for arithmetic and logic operations using an immediate operand and for conditional short jumps (branch if). The R-Format is used for instructions using two source registers (Rs/Rt) and one destination result register (Rd).

3.1.1 WRITE operation

The *w<DB>* operation means put (write) the DB into a targeted word of the memory. In our case this operation consists in loading into the targeted register the DB value. The I and R format provide three different ways to handle this. The DB to be written may have three sources: it can be encoded in the instruction, it can be stored in the memory or it can be stored in a register of the RF component.

From instruction The I-Format allows to directly store an Immediate operand of 16 bits in the instruction. As the DB is 32 bits wide it had to be cut into two 16-bit chunks. Therefore, to write a DB in a register, at least two I-Format instructions are necessary. A sequence implementing this kind of write operation is given below:

```
lui $5, Immd1
ori $5, $5, Immd2
```

The DB is written into the register number 5. The *Immd1* and *Immd2* correspond respectively at the 16-MSB and the 16-LSB of the DB. The cost of March write operation with such method is 2 cycles for execution and 2 words in instruction memory.

From memory Only one instruction allows to load data from memory: lw $Rd, Immd($Rs). The word stored in the memory at the address ($Rs+Immd) is written in the register pointed out by $Rd. Using this instruction implies two assumptions. First, the DB is assumed to be stored in the memory. In the case of CF detection, all the DBs can be

stored in the memory. Second, the address of the DB is assumed to be in the register pointed out by Rs. Since this register cannot at the same time hold this kind of information and undergo the test, we assume that this register is placed in the reserved area of the RF. The cost in cycles depends on the number of cycles needed to access the data memory. In terms of memory storage, the operation requires the use of one instruction per write and a set of DBs.

From register The last method uses an R-Format instruction like the *or* instruction: or $Reg, $Ref, $Ref. This instruction copies $Ref into $Reg. The advantage of this method is its promptness since one cycle is needed to achieve the write operation of $Reg. Nevertheless, before the use of this kind of R-Format instruction, $Ref register must be initialized by another method like both described above.

3.1.2 READ operation

A $r<DB>$ operation means to read the word stored at a targeted address, this word should be equal to the expected one (DB). Thus, the read operation includes two distinct steps: a read and a comparison. The comparison step can be done by the microprocessor itself or by the external tester. In both cases, it is mandatory to write at least one response in the memory to be then processed by the outside test environment. There are several ways to achieve this March read operation:

Store All One potential method is to store all the read words of the RF in the embedded memory. Using a store word instruction, each word of the RF is written in the memory. The comparison is done by the outside tester, to know if the RF is fault free or not. As a load instruction, a register is needed to hold the address where to store the value. Same cause, same consequence this register can not undergo the test and is put in the reserved area. This method is very useful for diagnosis purpose, since after each read operation the memory contains an *image* of the RF. The evident drawback of this method is the memory consumption.

Compute a Signature Another method to implement the March read operation is to use for the comparison a pre-computed signature stored in a register located in the reserved-area. This method is similar to hardware MISR techniques. For that, we have to initialize the reserved-area register with a seed and use a sequence of instructions to compute the signature. To avoid aliasing, this sequence must be carefully chosen since all the registers read contain the same value. The last step consists in comparing the computed signature with the expected one. The advantage of this method compared to the first one is that only one word (the computed signature) have to be stored in the memory. However, the sequence of instructions to compute the signature is very time consuming (in terms of cycle).

Read and Compare In the MIPS microprocessor ISA, an instruction allows the comparison between two registers. One register contains the DB value and is used as a reference, whereas the other register is the tested one. The reference register is stored in the reserved area of the RF. The instruction *Branch If not Equal* is used like this: bne $Ref, $Reg, label. $Ref is the reference register, $Reg the tested one, and if $Ref is not equal to $Reg the program jump to the label. At the address corresponding to the label, we can find the part of the program managing the failing cases. The advantage of this method is quiet obvious, just one instruction is needed to read and compare, and moreover the algorithm can be stopped at the first error.

3.1.3 Increasing/Decreasing order sequence

In March algorithms, read and write operations are done after increasing/decreasing the cell address. In our case, the address corresponds to the destination register number. Below an example of the implementation of the March element $\uparrow$(r0, w1) is given, where $30 and $31 are registers located in the reserved area and containing respectively a DB and the corresponding inverted DB:

```
        .
        .
        .
bne  $5, $30, fail
or   $5, $31, $31
bne  $6, $30, fail
or   $6, $31, $31
bne  $7, $30, fail
or   $7, $31, $31
bne  $8, $30, fail
or   $8, $31, $31
        .
        .
        .
```

3.2 Divided RF and CF detection

We have to take into account that the resulting self-test program must be small. Use of functions as well as loops in the program allows a drastic reduction of code size. Consequently, some registers had to be reserved for that purpose in order to contain variables, counters, arguments, or addresses. Furthermore, as seen before, almost all instructions used for the write operation, or the read operation require the use of some extra registers containing references

and addresses. Thus, to allow a flexibility for the self-test program development we have to reserve a part of the Register File. As a result, the entire RF can not undergo the test in one time. A solution, is to divide in two parts the Register File. One part (called part-B) is used for what we called the *reserved area*, where variables, counters, addresses are stored. Whereas, the other part (called part-A) undergo the test. At the end of the test of the part-A, we switch these two parts to test the part-B not tested yet, and the part-A becomes the *reserved area*. Thus, the test of the entire RF is done in two steps. The figure 2 depicts the bipartition of the RF.

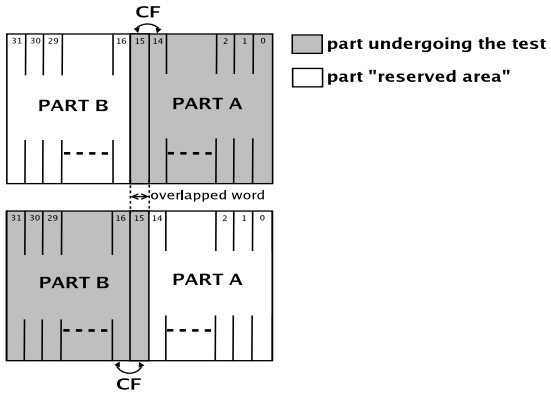

Figure 2. The Register File division

Some March Algorithms detect Coupling Faults. For instance, a March C- can detect in the RF a coupling fault between a cell *i* in the register \$X and a cell *j* in the register \$Y. If \$X and \$Y belong to the same RF part, the coupling fault can be detected, in the other hand, if \$X belongs to part A and \$Y belongs to part B, since no test is performed between these two cells the coupling fault cannot be detected. Thus testing the RF in two steps may, at first sight, not allow the detection of coupling faults between cells not located in the same part. Nonetheless, if we take into account the topology of the RF memory cell-array, we can restrict the couple of cells involved in CFs to neighbor cells[12]. The figure 3 shows the two-dimensional memory cell-array of the RF. A cell can be implied in a Coupling Fault with at most eight potentially neighbor cells. The outcome is that in case of a Word Oriented Memory, inter-word CFs can only occur between the two neighbor words C-1 and C, or the two neighbor words C and C+1, as shown in the figure 3. As a result, to enable the capability to detect Coupling Fault in the RF with our partitioning strategy, it is sufficient to have between the two parts of the RF at least one overlapped word. The figure 2 shows that in the first test step, the CF between the register 14 and 15 can be revealed whereas during the second step of the test, the CF between the register 15 and 16 can be detected.

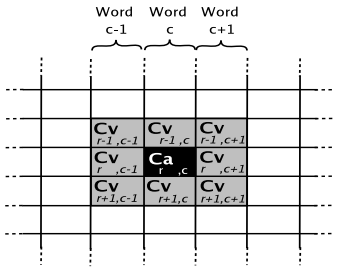

Figure 3. Cell neighborhood

4 Experimental Results

To validate the method and make the corresponding measurements we use a simulation platform running under a complete SystemC framework. The simulation chosen is cycle-accurate and bit-accurate, in order to measure the exact number of clock cycles needed for the execution of the self-test program, as well as its exact size in the memory. The microprocessor used is a five-stage pipelined MIPS R3000 with a Register File including 32 registers 32-bits wide. Several March tests were implemented leading to different self-test programs. The Table 3 shows the results for each self-test program. The size is given in terms of 32-bit words and the test time is given in clock cycles. To see detected faults by each algorithm refer to Table1.

Algorithm	Test Program Size (32-bits words)	Clock Cycles
MATS	187	148
MATS+	220	183
MATS++	252	247
MARCH X	252	215
MARCH C-	479	2,178

Table 3. Different March Algorithms.

The outcome of these results is that the proposed development methodology leads to small-sized self-test programs, from 187 words for the MATS algorithm to 479 words for the MARCH C-. A quick glance at the clock cycles column allows one to highlight a striking difference between MARCH C- and other March test. Let's consider two sets of algorithms: {MATS, MATS+, MATS++, MARCH X} in one hand, and {MARCH C-} in the other hand. The first set needs very few clock cycles to complete the tests whereas the second set is much longer. This difference is due to the Coupling Fault detection capability of the second set. In this case, the test must be repeated for each DB in

order to detect intra-word state CFs. In fact, this second set contains all March algorithms able to detect the state Coupling Faults. This latter set enhances the test quality but multiply test time. In this paper only state CFs have been targeted, nevertheless, the use of other appropriate DBs sequences allow the detection of other CF subclasses (e.g. inversion CF, disturb CF, idempotent CF)[11].

Another comment on the results obtained for MATS++ and MARCH X algorithms must be done. Both programs size are equal since they have the same number of write and read operations. Thus, we can expect to have the same test time execution. Nevertheless, a difference in favor of MARCH X appears. This is due to the fact that MATS++ last March element is a read-write-read operation. More exactly the last write-read operation in the same march element leads to a *Read After Write* data hazard in the five-stage pipeline of the microprocessor inducing some stalls, which improves the number of cycles.

According to Software-Based Self Test strategies, the program size is the key-point to reduce the global test application time. Indeed, the self-test program is first downloaded into the embedded memory from the external test equipment, and then, is executed at the SoC operating frequency. The key issue is that the download phase usually predominates the overall test application time. In [6] the self-test program needs 927 clock cycles to be completed with a size of 425 words and only Stuck-At-Faults are targeted. In our approach, the MATS++ self-test program is 1.7 times smaller and requires 4.3 times less cycles while ensuring the detection of all Stuck-At-Fault, Address Fault and Transition Fault. Since the program size is the key-point to lower the test application time, using the MARCH C- (or a March test detecting CF) improves the test quality while the overall test application time is not significantly increased. This is particularly true if we consider a high ratio between SoC operation frequency and tester's frequency.

5 Conclusion

Using a component-oriented Software-Based Self-Test approach, this paper describes the self-test program development of a high-priority component: the Register File of RISC processor cores. A methodology for self-test program development, fully suitable to apply any March test to Register Files, was presented. The main novelty of this article is the use of March algorithms since RF is designed as a memory. The self-test program development do not require any net-list or sequential ATPG since March test guarantee 100% fault coverage of detected fault model.

The proposed methodology enables a great flexibility in the program development and leads to small-sized self-test programs while the use of March tests ensures a high test quality.

References

[1] J. Shen and J. A. Abraham. Native Mode Functional Test Generation For Processors with Applications to Self-Test and Design Validation. In *Proceedings IEEE International Test Conference (ITC)*, Washington, DC, October 1998.

[2] Ken Batcher and Christos Papachristou. Instruction Randomization Self Test for Processor Cores. In *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 34–40, Dana Point, CA, April 1999.

[3] F. Corno, M. Sonza Reorda, G. Squillero, and M. Violante. On the Test of Microprocessor IP Cores. In *Proceedings Design, Automation, and Test in Europe (DATE)*, Munich, Germany, March 2001.

[4] L. Chen and S. Dey. Software-Based Self Testing Methodology for Processor Cores. *IEEE Transactions on Computer-Aided Design*, March 2001.

[5] N. Kranitis, G. Xenoulis, D. Gizopoulos, A. Paschalis, and Y. Zorian. Low-Cost Software-Based Self-Testing of RISC Processor Cores. In *Proceedings Design, Automation, and Test in Europe (DATE)*, Munich, Germany, March 2003.

[6] N. Kranitis, G.Xenoulis, D. Gizopoulos, A. Paschalis, and Y. Zorian. Software-Based Self-Testing of Large Register Banks in RISC Processor Cores. In *LATW'2003 - IEEE Latin-American Test Workshop*, February 2003.

[7] R. Dekker and F. Beenker and L. Thijssen. A Realistic Fault Model and Test Algorithms for Static Random Access Memories. In *IEEE Transactions on Computer-Aided Design*, volume 9(6), pages 567–572. IEEE Press, June 1990.

[8] J. M. Rabaey et al. *Digital Integrated Circuits*. Prentice-Hall, 2003.

[9] Richard D. Jolly. A 9-ns, 1.4-Gigabyte/s, 17-Ported CMOS Register File. In *IEEE Journal of Solid-State Circuits*, volume 26(10), pages 1407–1412. IEEE Press, October 1991.

[10] A. J. van de Goor. *Testing Semiconductor Memories: Theory and Practice*. John Wiley & Sons, Chichester, England, 1991.

[11] A.J. van de Goor, I.B.S. Tlili. March Tests for Word-Oriented Memories. In *Proceedings Design, Automation, and Test in Europe (DATE)*, page 501, 1998.

[12] M. Nicolaidis, V.C. Alves, and H. Bederr. Testing Complex Couplings in Multi-Port Memories. In *IEEE Transactions on VLSI Systems*, volume 3(1), pages 59–71. IEEE Press, March 1995.