Deterministic Software-Based Self-Testing of Embedded Processor Cores

A. PASCHALIS [1] D. GIZOPOULOS [2] N. KRANITIS [3] M. PSARAKIS [3] Y. ZORIAN [4]

[1] Department of Informatics, University of Athens, Greece
paschali@di.uoa.gr

[2] Department of Informatics, University of Piraeus, Greece
dgizop@unipi.gr

[3] II&T, NCSR "Demokritos", Athens, Greece
{nkran | mpsarak}@iit.demokritos.gr

[4] LogicVision, San Jose, CA, USA
zorian@logicvision.com

Abstract

A deterministic software-based self-testing methodology for processor cores is introduced that efficiently tests the processor datapath modules without any modification of the processor structure. It provides a guaranteed high fault coverage without repetitive fault simulation experiments which is necessary in pseudorandom software-based processor self-testing approaches. Test generation and output analysis are performed by utilizing the processor functional modules like accumulators (arithmetic part of ALU) and shifters (if they exist) through processor instructions. No extra hardware is required and there is no performance degradation.

1. INTRODUCTION

Complex and highly functional Systems-on-Chip (SOCs) have become a reality due to recent advances of deep sub-micron technology. SOC designs are based on embedded cores: reusable complex functional blocks also called virtual components or intellectual property (IP) blocks. The majority of SOC designs are built around embedded RISC processor cores or digital signal processor cores that provide the SOC with significant processing capabilities. In addition, special purpose cores, for functions such as graphics, audio/video, are embedded processors, using microcode implemented in predefined macros (ROM, PLA) or using code stored in embedded RAM blocks.

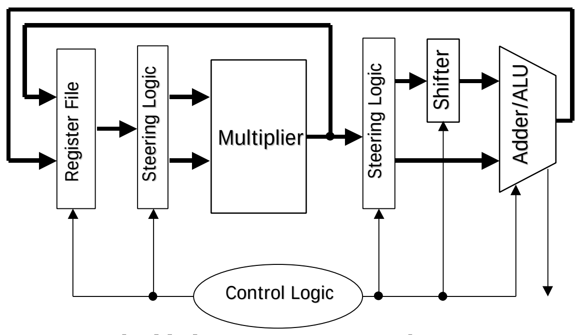

Figure 1: Embedded Processor Datapath

A typical structure of an embedded processor datapath is shown in Figure 1. Arithmetic and logic operations in the datapath are implemented by functional modules like multipliers, adders/subtracters, Arithmetic Logic Units (ALUs) and shifters. Register files store the results of the operations while steering logic and control logic determine the flow of data inside the datapath [1].

In an SOC design, a large number of complex arithmetic and logic functional modules are deeply embedded in datapaths of embedded processor cores which are further embedded in the overall SOC. Due to this design style, embedded processor cores have significant testability problems. Built-In Self-Test (BIST) [2] has been shown to be an excellent solution to these problems not only for embedded processor cores but also for the other important class of embedded cores i.e. memories.

The main advantage of self-testing methodologies is that they provide actual at-speed testing of the chip which is a very difficult target to achieve with external tester technology due to the rapidly increasing gap between operating frequencies of SOC designs and external tester frequencies. Additionally, the use of self-test methodologies drives down the overall test cost of the SOC while it also provides better IP protection than classical scan-based external testing techniques.

Self-test methodologies for embedded processor cores have the advantage that they can be based on the processor's instruction set and thus require no hardware and/or performance overhead in the design. This is due to the fact that both test generation and response compaction is performed by processor instructions without the need to add extra self-test hardware. Such self-testing approaches have been recently proposed in the literature [3]-[9].

The processor self-testing approaches of [3]-[9] rely on the use of pseudorandom instruction sequences [3], [4], pseudorandom operations and operands [5], [6], while scan chains can be used for the application of test patterns [7], [8]. In [9] the concept of self-test signatures is introduced when LFSR-like pseudorandom test generation is performed by software programs.

The efficiency of the pseudorandom software-based self-testing approaches presented in the literature, depends both on the internal structure of the functional modules of the processor and the width of the processor word. For example a 16-bit and a 32-bit ALU require different seeds and polynomials to achieve high fault coverage in an LFSR-based self-test methodology. When, in this case, the pseudorandom patterns are generated by software a different routine must be written in each case and extensive fault simulation experiments must be performed to measure the seed/polynomial pair efficiency.

Additionally, pseudorandom self-testing requires the application of large number of tests to the processor core. In the case that LFSR operation is emulated by a software routine [9], the generation of a new pseudorandom test pattern requires not only one instruction but at least a bitwise XOR and a shift operation. Thus, a pseudorandom software-based self-test approach requires a large number of clock cycles for the execution of the self-test routines.

The alternative to pseudorandom self-testing is usually the insertion of scan chains for the application of a test set generated by an Automatic Test Pattern Generator (ATPG). In the case of embedded processors this alternative is not usually applied due to the fact that scan insertion significantly impacts the processor performance.

In this paper we present a methodology for developing deterministic as opposed to pseudorandom software-based self-test routines for embedded processor cores. The self-test routines for the processor functional modules are based on deterministic test sets we developed in our previous papers, as well as, in newly developed test sets. In the case of multiplier-accumulator testing, processor datapath functional modules like multipliers, adders, and subtracters are efficiently tested by small deterministic test sets based on repetitive-patterns shown in our previous papers [10]-[13]. In this paper we present self-test routines that generates these test-sets by utilizing existing adders (or subtracters) instead of fixed-length (8-bit) binary counters, proposed so far. We have proven in the past both theoretically and experimentally that the efficiency of these test sets does not depend either on the internal architecture of the functional module or the width of the datapath word. For example, the same test set of repetitive test patterns can be used to achieve very high fault coverage for any size of a standard array multiplier [10], Booth encoded [11] or tree multiplier [12]. In the case of both ALU and shifter testing, we present self-test routines that generate new small deterministic test sets by utilizing existing ALU and shifter performing simple arithmetic, logic, or rotate operations.

According to our methodology, compact self-test routines (small code size and smaller number of clock cycles for routine execution compared to pseurorandom-based self-testing) are derived for the processor functional modules. Additionally, the efficiency of the proposed deterministic self-testing routines does not need to be re-evaluated each time the datapath width or the architecture of a functional module of the processor changes. This flexibility of the proposed methodology is particularly useful in our days since soft (HDL) versions of embedded processors are widely used after proper modifications.

The processor instructions utilized in the proposed self-testing routines are simple, classical instructions of register and immediate addressing types supported by any Instruction Set Architecture of the embedded processor cores like move, add/subtract, logical functions (AND, OR, XOR, XNOR), multiply, compare, branch/jump with/without conditions and with/without link.

In the case of simpler accumulator-based processor architecture without a large set of general purpose registers, the routines can be tailored to apply the same set of deterministic test patterns to the processor functional modules. Furthermore, in the case of a processor architecture (or DSP) with more than one functional modules of the same type (multiplier, adder, ALU, shifter) the self-test routines can be easily tailored so that they provide very high for coverage of all modules of the same type. Therefore, the proposed self-test routines can be easily executed by any processor architecture, utilizing the respective instructions of its instruction set.

2. DETERMINISTIC PROCESSOR SELF-TESTING

In this section we present our deterministic software-based self-testing methodology. For each individual of the processor datapath's functional modules or for pairs of modules we, first, adopt a deterministic test strategy such that no modifications of the datapath structure are required, and then, we show the relevant assembly code for an example processor core. The assembly code we use in our presentation is based on [1] but similar code can be written in any different type of embedded processor since the set of instructions we use is met in any instruction set architecture of contemporary processors.

2.1 Multiplier-Accumulator Testing

When a multiplier exists in the processor's datapath, which is the case in most recent RISC-based embedded processors, the pair of the multiplier and accumulator functional modules is tested together. If accumulation is performed by the addition/subtraction operation of an ALU then the arithmetic part of the ALU is tested with the multiplier while the logic part of it is tested with a separate self-testing routine as we will show in the next subsection.

We adopt the determinist test strategy described in [10], [11], [12], for multiplier-accumulator pairs. As it was proven theoreticaly and experimentaly in these papers very high fault coverage is achieved for a $N \times M$ multiplier when its operands X and Y receive a set of 4-bit repetitive test patterns of the form:

$$X = X_{N-1} \ldots X_1 X_0 = (c_7\, c_6\, c_5\, c_4)\, (c_7\, c_6\, c_5\, c_4) \ldots (c_7\, c_6\, c_5\, c_4)$$
$$Y = Y_{M-1} \ldots Y_1 Y_0 = (c_3\, c_2\, c_1\, c_0)\, (c_3\, c_2\, c_1\, c_0) \ldots (c_3\, c_2\, c_1\, c_0)$$

When operand sizes N and M are not multiples of 4, the applied test patterns are truncated. The total number of test patterns applied to the multiplier-accumulator pair are 256 (=2^8, all combinations of the 8 bits $c_7\, c_6 \ldots c_1\, c_0$), or in some cases a subset of them. The achieved fault coverage for *any* operand size N, M and *any* multiplier architecture (carry-save array, carry-propagate array, Wallace-tree, Booth encoded) is higher than 99% [10], [11], [12].

In our experiments for the justification of the proposed self-test methodology, we consider unsigned long multiplication, where the result is stored in two registers of the register file. In the case that the deterministic self-testing methodology is applied to fixed-width processor datapaths (for example in an embedded DSP) solutions like the ones presented in [6] or [13] can be used.

According to the proposed self-test methodology for the multiplier-accumulator pair, Test Pattern Generation is performed by the existing processor ALU which operates in the addition mode as an accumulator (or of course by a separate accumulator if such exists) by using two registers R1 and R2 of the register file. The two multiplier operands *X* and *Y* are stored in registers R1 and R2, respectively during the execution of the self-testing routine. The deterministic test set of 256 test vectors [10], [11], [12] is generated by a software self-test routine as follows:

- R1 and R2 are initialized to zero.
- R1 receives 16 test vectors consisting of 4-bit repetitive patterns from 0000 to 1111. This is done by the addition R1=R1+R8, where R8 contains the constant value of 4-bit repetitive pattern 0001.
- For every different test vector of R1, the R2 register also receives the same 16 different test vectors consisting of 4-bit repetitive patterns from 0000 to 1111. This is also done by the addition R2=R2+R8.

The result of the multiplication R1×R2 is stored in two registers R3 and R4 of the register file where R4 holds the most significant word of the product and R3 holds the least significant word of the product (R4:R3 = R1 × R2).

Output data evaluation (compaction) is performed by the ALU operating in addition mode as accumulator utilizing two registers R5, R6. Register R6 contains the final signature after the execution of the self-test routine. The details of the response compaction using an accumulator is well known that affect the compaction quality (low aliasing). In our experiments the following sequence of additions are performed for output data compaction with negligible aliasing problems [6].

- Initially, R6 is set to zero.
- For every test vector of R1, R2, three additions are performed:

  R5 = R4 + R3 *(the 2 words of the product are added)*

  R6 = R6 + R5 *(add the previous signature)*

  R6 = R6 + 0 + carry *(add the carry generated)*

After performing these three additions for all 256 test vectors the adder of the ALU (arithmetic part of the ALU) is completely tested. This has been extensively verified for various adder architectures in the accumulator.

The self-test code (for a 32-bit embedded processor [1]) that generates at the inputs of the multiplier and the accumulator the deterministic test set of 256 test vectors is the following (constants are shown in hexadecimal):

```
      MOV R1, 00000000        ; R1 = 0
      MOV R2, 00000000        ; R2 = 0
      MOV R6, 00000000        ; R6 = 0
      MOV R7, signature       ; declare signature
      MOV R8, 11111111        ; 4-bit incr. by 1
loop: UMUL R4,R3,R2,R1        ; multiply
      ADD R5,R4,R3            ; add product words
      ADD R6,R6,R5            ; compaction with 1's
      ADC R6,R6,0             ; complement addition
      ADD R2,R2,R8            ; increment R2
      CMP R2, 11111110        ; R2 final value ?
      BNE loop                ; branch, if not
      MOV R2, 00000000        ; R2 = 0
      ADD R1,R1,R8            ; increment R1
      CMP R1, 11111110        ; R1 final value ?
      BNE loop                ; branch, if not
      CMP R6,R7               ; check signature
      BNE test_fail_routine
      JMP test_pass_routine
```

The fault-free signature is calculated in advance by a simple logic simulation of the self-test code (the final signature is the result of the accumulation of the 256 multiplication results). Also, there are two routines that terminate the multiplier-accumulator test process. The first routine runs when a fault has been detected (test_fail_routine) while the latter runs when no fault has been detected (test_pass_routine). The implementation of these routines directly depends on how the embedded processor core is used in the SOC and which is the general test strategy, which is applied in the SOC as a whole.

The derived code achieves after compaction, the results shown in the following Table for various architectures.

WORD LENGTH	ARCHITECTURE	FAULT COVERAGE
32 bits	CSA – RCA	99.5%
32 bits	CSA – CLA	99.3%
32 bits	BWM – RCA	99.3%
32 bits	BWM – CLA	99.2%

where CSA=carry save array multiplier, BWM=Booth encoded tree (Wallace) multiplier, RCA=ripple-carry adder and CLA=carry lookahead adder.

In the case of the Booth encoded Wallace multiplier R1 receives test vectors consisting of 5-bit repetitive patterns and R2 receives test vectors consisting of 3-bit repetitive patterns which provides the best fault coverage [12]

The self-test code has the following advantages:

- It does not depend on the functional implementation of the multiplier and the adder (see Table above).
- It does not depend on the processor word width. We present 32-bit experiments since this is the typical word length today.

- It is well suited to any processor core without requiring any modification of its datapath structure.

2.2 ALU Testing

As we mentioned earlier, the arithmetic part of the processor's ALU is tested with the multiplier which is the most usual situation in embedded processor cores.

In this subsection we deal with the testability of the logic part of the ALU taking into account two different ALU designs which are usually met in embedded processors.

According to the first approach of the ALU design, the logic unit that implements the logical operations is designed separately from the adder of the ALU. The outputs of both modules are connected to the ALU output through a two-way multiplexer and selection between arithmetic and logical operations is achieved by the multiplexer select signal M [14]. In this case we can use any adder in the ALU without restrictions. Also, we can design a logic unit that implements the necessary logic operations. In this paper we consider the logic unit that implements all 16 two-operand logical operations [14]. Four function select signals S3, S2, S1, S0 specify one out of the 16 operations. The adder of the ALU as well as the with the one part of the multiplexer of the ALU is tested during multiplier-accumulator testing, as we mentioned earlier. The remaining part of the ALU, the logic unit with the other part of the multiplexer is tested during ALU testing presented in the self-test code of this subsection.

According to the second ALU design, the module that implements the logical operations is designed in conjunction with a carry lookahead adder (74x181) [14]. In this case all possible 16 logical operations are implemented, as well as, 16 complex arithmetic operations. The part of the ALU that implements the addition i.e. carry lookahead generator and the sum logic are tested during multiplier-accumulator testing. The remaining part of the ALU, that is, modules M1, M2 that implements the logic operations are tested during ALU testing.

We adopt a deterministic test strategy that tests the "logic" part of the ALU by performing the logical operations: AND, OR, EX-OR, EX-NOR and NOT. This test strategy is general enough and it can be applied to various ALU designs including the two ALU designs mentioned above. In case that the logic unit design is simpler the logical operations EX-NOR and NOT may not be necessary.

Test pattern generation is performed by putting the four possible combinations of values all 0's and all 1's in two registers R1 and R2 of the register file. Also, in register R3 we put the EX-OR of the values of R1 and R2. Operands X and Y of the ALU are stored in registers R1 and R2, respectively. This way, a deterministic test set is generated as follows:

- R1 and R2 receive all four possible input combinations of all 0's and all 1's.
- R3 is the EX-OR of R1 and R2.
- For every value of R1 and R2, the following logical operations are performed:
  (a) R4 = R1 AND R2 **(b)** R5 = R1 OR R2 **(c)** R6 = R5 XOR R4 **(d)** R7 = R1 XOR R2 **(e)** R8 = R1 XNOR R2 *(may not be necessary)* **(f)** R9 = NOT R3 *(may not be necessary)*
  and the following comparisons are performed:
  (a) R3 with R6 **(b)** R7 with R3 **(c)** R8 with R9 which detect a possible fault.

The code (for a 32-bit embedded processor) that generates the test vectors at the inputs of the ALU follows:

```
          MOV R1, 00000000    ; R1 = 0
          MOV R2, 00000000    ; R2 = 0
          MOV R3, 00000000    ; R3 = 0
          BL subrout          ; branch and link
          MOV R1, FFFFFFFF    ; R1 = all 1's
          MOV R2, FFFFFFFF    ; R2 = all 1's
          BL subrout          ; branch and link
          MOV R1, 00000000    ; R1 = 0
          MOV R3, FFFFFFFF    ; R3 = all 1's
          BL subrout          ; branch and link
          MOV R1, FFFFFFFF    ; R1 = all 1's
          MOV R2, 00000000    ; R2 = 0
          BL subrout          ; branch and link
          JMP test_pass_routine
subrout:  AND R4,R2,R1        ; AND
          OR R5,R2,R1         ; OR
          XOR R6,R5,R4        ; EX-OR
          CMP R6,R3           ; R3 = R6 ?
          BNE test_fail_routine
          XOR R7,R1,R2        ; EX-OR
          CMP R7,R3           ; R3 = R7 ?
          BNE test_fail_routine
          XNOR R8,R1,R2       ; EX-NOR
          NOT R9,R3           ; NOT
          CMP R9,R8           ; R9 = R8 ?
          BNE test_fail_routine
          RET                 ; PC = link register
```

The self-test code of this section along with the self-test code for the multiplier-accumulator pair achieves, the results shown in the following Table for various architectures of the processor ALU.

WORD LENGTH	ARCHITECTURE	FAULT COVERAGE
16 bits	SEP – RCA	100.0%
16 bits	SEP – CLA	99.9%
16 bits	COM - CLA	99.9%
32 bits	SEP – RCA	100.0%
32 bits	SEP – CLA	99.9%
32 bits	COM - CLA	99.9%

where SEP=separate arithmetic/logical part of the ALU, COM=combined arithmetic/logical part of the ALU, RCA=ripple-carry adder and CLA=carry lookahead adder.

No compaction is performed due to the capability of fault detection with simple comparisons and thus there are no aliasing problems. In the case that the processor is not equipped with a multiplier, the arithmetic ALU part can be tested by itself by a separate self-testing routine similar to the one presented in the previous subsection for the multiplier-accumulator pair.

2.3 Shifter Testing

We consider the testability of a barrel shifter that performs the *n*-bits right rotate (ROR) operation. Other operations can be tested similarly. The outputs of the shifter drive directly the one of the two-operand inputs of the ALU [1].

We adopt a deterministic test strategy that tests the shifter by performing the ROR operation $2n+3$ times starting from value 1, where n is the word length. We perform the ROR operation $2n+3$ times instead of $2n$ in order to avoid aliasing problems. The output of the shifter is connected to the input of the shifter through register R1 which is used as data register. For every value of R1 two ROR operations are carried out: 0 shift positions and 2^n-1 shift positions. Register R2 is used as a counter and contains the number of ROR operations and thus counts from 0 to $2n+3$. Register R3 is used as a control register and contains the number of shift positions, that is, 0 or 2^n-1.

The shifter itself performs test pattern generation and output data evaluation. The respective code follows:

```
      MOV R1, 00000001       ; R1 = 1
      MOV R2, 00000000       ; R2 = 0
      MOV R3, 00000000       ; R3 = 0
      MOV R4, signature      ; declare signature
loop: MOV R1,R1,ROR R3       ; R1 shifted by R3
      EOR R3,R3,FFFFFFFF     ; R3 = NOT R3
      ADD R2,R2,00000001     ; increment R2
      CMP R2,00000043        ; R2 = 2x64 = 67 ?
      BNE loop               ; branch, if not
      CMP R1,R4              ; check signature
      BNE test_fail_routine
      JMP test_pass_routine
```

The self-test code achieves 100% fault coverage for 8-bit, 16-bit and 32-bit barrel shifter widths. Similar code can be derived for such simpler shifter designs and for more complex ones.

2.4 Case Study

Apart from experimenting with the various modules presented in the previous sections, we have evaluated the effectiveness of the proposed methodology in a classical embedded controller architecture, Intel 8051.

The flow we followed is based on a synthesizable VHDL model of the controller. The synthesized circuit includes one 8x8 carry-save array multiplier and a number of 8-bit addition and subtraction units which number depends on the particular application. We performed synthesis using Synopsys Design Compiler and fault simulation using Cadence Verifault. A self-test routine consisting of about 190 bytes of code provides a post-compaction fault coverage of 99.4% for the arithmetic modules of the core when three add/subtract units are used.

Currently, we are evaluating the efficiency of the proposed methodology to the other parts of the processor (register files, control part) and also the application of the methodology to various processor architectures (RISC-based, accumulator based, with or without multiplier unit).

3. Conclusions

The proposed deterministic software-based self-testing architecture for embedded processor cores compares favorably with earlier pseudorandom based architectures since it does not require extensive fault simulations to estimate its effectiveness and it requires smaller test application time and power consumption due to its deterministic nature. It can be applied to any word length and any internal architecture of the arithmetic modules of the processor.

References

[1] ARM9TDMI, *Technical Reference Manual*, Nov. 1998.

[2] V.D.Agrawal et al., "*BIST for Digital Integrated Circuits*", AT&T Tech. Journal, March 1994, pp. 30.

[3] J.Shen, J.Abraham, "*Native mode functional test generation for processors with applications to self-test and design validation*", ITC 1998, pp. 990-999.

[4] K.Batcher, C.Papachristou, "*Instruction randomization self test for processor cores*", VTS 1999, pp. 34 – 40.

[5] K.Radecka, J.Rajski, J. Tyszer, "*Arithmetic built-in self-test for DSP cores,*" IEEE Trans. on CAD, vol.16, no.11, Nov. 1997, pp. 1358–1369.

[6] J.Rajski, J.Tyszer, "*Arithmetic Built-In Self-Test for Embedded Systems*", Prentice Hall, 1997.

[7] S.Hellebrand, H.-J.Wunderlich, "*Mixed-mode BIST using embedded processors*", ITC 1996, pp. 195 – 204.

[8] R. Dorsch and H.-J. Wunderlich, "*Accumulator based deterministic BIST*", ITC 1998, pp. 412 – 421.

[9] L.Chen, S.Dey, P.Sanchez, K.Sekar, Y.Chen, "*Embedded Hardware and Software Self-Testing Methodologies for Processor Cores*", DAC 2000, pp 625-630.

[10] D.Gizopoulos, A.Paschalis, Y.Zorian, "*An Effective Built-In Self-Test Scheme for Array Multipliers*", IEEE Trans. Computers, vol. 48, no. 9, pp. 936-950, September 1999.

[11] D.Gizopoulos, A.Paschalis, Y.Zorian, "*An Effective Built-In Self-Test Scheme for Booth Multipliers*", IEEE Design & Test of Computers, vol. 15, no. 3, pp. 105-111, July-September 1998.

[12] A.Paschalis, M.Psarakis, D.Gizopoulos, N.Kranitis, Y.Zorian, "An *Effective BIST Architecture for Fast Multiplier Cores*", DATE 1999, pp. 117-121.

[13] D.Gizopoulos, A.Paschalis, Y.Zorian, "*An Effective BIST Scheme for Datapaths*", ITC 1996, pp. 76-85.

[14] J.P. Hayes, "*Computer Architecture and Organization*", McGraw Hill, 1998.