Systematic Test Program Generation for SoC Testing Using Embedded Processor

M. H. Tehranipour, M. Nourani
Center for Integrated Circuits & Systems
The Univ. of Texas at Dallas
Richardson, TX 75083
{mht021000,nourani}@utdallas.edu

S. M. Fakhraie, A. Afzali-Kusha
VLSI Circuits and Systems Laboratory
The Univ. of Tehran
Tehran 14399, Iran
{fakhraii,afzali}@ut.ac.ir

Abstract

Embedded processors are now widely used in system-on-chips. The computational power of such processors and their ease of access to/from other embedded cores can be utilized to test SoCs. This paper presents a software-based testing of embedded cores in a system chip using the embedded processor. We present a methodology to systematically generate test programs that test the processor and other cores in system chip. The method requires almost no overhead but provides great flexibility in terms of structural/fault coverage, test mechanism and future reuse.

1 Introduction

Embedded test integrates multiple disciplines: DFT features, BIST pattern sources and sinks, precision and high-speed timing for at-speed test, test support for many different core types (logic, memory, processors, and analog), and capabilities for diagnosis and debug. With embedded test, the on-chip test data generation reduces the volume of external interaction which can be customized per core type. Also, the on-chip test and diagnostic data compression reduces ATE data logging and the ATE speed requirements. More importantly, the on-chip test program generation can potentially achieve true or near at-speed test that can scale to match the process performance [1] [2]. The problem with external test pattern based strategies is that they typically require large test data volume and need complex dynamic test control protocols to get stimulus to cores, receive responses from cores and set up the necessary control to execute the test on a core. A promising strategy that minimizes such requirements is BIST. All known benefits of BIST come from the fact that BIST embeds test into the circuit under test. It generates and evaluates the test patterns on-chip. Hence, it does not require porting of test vectors from the core creator to the integrator and then to the fabricator [1] [3] [4].

The nature of a core may also have an impact on the internal test strategy. For example, almost all memories today tend to use BIST. Hence, providers of embedded memory cores typically incorporate BIST wrappers in their memory core design. Frequently, test designers today use self-testing methodology for processor cores [5] [6] [7]. This will be a software-based test providing test procedures (to generate deterministic and random data) by a processor core for testing itself or other cores [8] [9] [10].

Self-testing methodology in a system-chip by running test programs using a programmable core has several potential benefits including, at-speed testing assuming the program are short, low DFT overhead due to elimination of dedicated test circuitry and better power and thermal management during testing. This self-test strategy is referred to as embedded-software-based self-test [9].

1.1 Motivation and Contribution

Embedded processors can be utilized to run software routines for test pattern generation and response evaluation [6] [8]. However, in almost all works reported in the literature, generation of test program is somewhat ad-hoc. In this work, we propose a software-based test mechanism for testing system chips. The main contribution of this paper is a systematic approach to generate the Test Programs (TPs) and utilize the instruction set of embedded processor to test a system chip including the processor itself. Due to protecting IP by core vendors, we assume only the instruction set and limited architectural/test information of the cores are available.

In our approach, first the processor tests itself by reading its self-test program from memory. Then, the tested processor tests the other embedded cores. A direct downloading mechanism, e.g. DMA (Direct Memory Access), is used to store test programs in Program Memory (PM) before the test session begins. All of the test programs to test processor and other cores such as memories are stored in PM. The testing process is effectively done at-speed or near at-speed since the entire testing process is done inside the system. Additionally, the test controller is almost non-existent because the processor uses its normal-mode controller (i.e. instruction decoder) to execute the test program. More importantly, since the test mechanism is specified in software the method is very flexible for: a) soft and hard cores, b) future change and reuse and c) incorporating various type of cores that may use different test methodologies (e.g. BIST, scan, etc.) or even different fault models (e.g. stuck-at, functional delay, etc.).

The rest of this paper is organized as follows. Section 2 describes the software-based system chip methodology. In Section 3 developing test program is described. Test evaluation framework is described in Section 4. Experimental results are presented in Section 5. Finally, concluding remarks are in Section 6.

0-7803-7761-3/03/$17.00 ©2003 IEEE

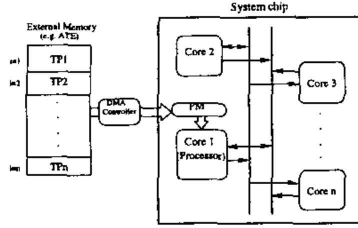

Figure 1. System chip software based testing.

2 Software-Based System Chip Testing

Accessing cores for testing is a challeging problem. Test access mechanism (TAM) has been addressed in the literature. In our work, we assume all cores are accessible by the processor. We propose a general methodology for testing the embedded cores in a system chip with the aid of the processor and based on their behavioral and structural testability information. Such information reflects the relationship between test vectors and fault/structural coverage. Using the testability information, the test program is generated systematically for a SoC under test. The test patterns (random or deterministic) can be generated internally be executing a pattern generation procedure. The processor applies the generated test patterns to cores and reads the responses from the core under test for final evaluation. For testing each core, its test program is loaded into PM and executed by the processor. Figure 1 shows the general concept of a software-based test for a system chip. As shown in Figure 1, for testing $core_i$, first TP is loaded into the PM of the processor by a DMA and then processor executes the loaded TP_i. Each TP_i consists of four major procedures.

1- Pattern Generation Procedure (PGP): The PGP generates the required test patterns (random or deterministic) for applying to a core under test. For example, the behavior of an LFSR [11] can be implemented as a program for random pattern generation. **2- Test Delivery Procedure (TDP):** This procedure delivers patterns/signatures to/from a core under test. The TDP is the most critical part of each test program for minimizing test time. We will elaborate more in the next section. **3- Signature Generation Procedure (SGP):** This procedure stores the test results of executing the TDP in the memory. When necessary SGP also compresses the responses and generates a signature for comparison procedure (CMP). **4- Comparison Procedure (CMP):** It compares the fault free signatures precomputed and prestored for a given core with the generated signatures by the SGP program.

3 Developing Test Program

There are four procedures in each test program, i.e., PGP, TDP, SGP, CMP. Writing the PGP, SGP and CMP is quite straight forward. In this section, we focus on the TDP which determines the test time as well as overall coverage. Writing the TDP is done in two phases as follows.

3.1 Phase 1: System-Level Covering

In this phase, the instructions are grouped based on the cores that they can access. Figure 2 shows a small example

		System Chip Cores			
Group	Instruction	Core 1 Processor	Core 2 Memory	Core 3 SPI	Core 4 HPI
1	ADD Ri, Rj	√			
	ADD Ri, (mem)	√	√		
	MUL Ri, Rj	√			
2	MOV Ri, (mem)		√		
	MOV (mem), Ri		√		
3	MOV Ri,DXR			√	
4	MOV HPIR,Rj				√
...	...	...	...	...	...

Figure 2. Grouping instructions at system level (phase 1).

of SoC containing four cores. For simplicity, we assumed that ADD and MUL instructions provide enough coverage of processor core and therefore they are considered in the processor core column. MOV instructions to send/receive data to/from memory core are considered in memory core instruction group. MOV instructions that deal with DXR and HPIR registers are also considered for Serial Port Interface (SPI) and Host Port Interface (HPI) cores, respectively.

Note that in general there may exist several groups of instructions that use different operands, addressing modes and cores and therefore they may be considered multiple times. MOV instruction, as shown in Figure 2 is one such instruction. Also, all lines are classified as data and control lines. Data and control lines are exercised with test data and instructions, respectively. Test data is provided from memory which is preloaded by executing PGP. Instructions in TDP will exercise the control lines when they are decoded by instruction decoder. This data-control relationship can be further explored in phase 2 upon availability of structural information.

3.2 Phase 2: Core-Level Covering

In this phase, the instructions chosen in phase 1 for a specific core will be further optimized based on the structural coverage that they provide with respect to the components inside the core. In other words, instructions in each group are classified based on their actual operations. This step targets a maximum coverage of the components within each core with available instructions in its group. For example, Figure 3 shows the details of each instruction type for group 1 (to test processor core).There are two types of ADD instructions corresponding to cases that both operands are registers and one register and one memory. Each tick in Figure 3 indicates that instruction covers the component in that column. Moreover, operands depict the mechanism of applying data and control to component.

3.3 TDP Optimization

To minimize test time we need to choose the minimum subset of instructions that can cover (for the purpose of testing) all cores (at the system level) and all components (at the core level). Our formulation for phase 1 and 2 is a conventional covering problem. It is essentially the problem of choosing

			Components in Processor Core					Memory Core
Group	Type	Instruction	ALU	MAC	R1	R2	R3	
1	(1)	ADD R1,R2	√		√	√		
		ADD R1,R3	√		√		√	
		ADD R2,R3	√			√	√	
	(2)	ADD R1,(mem)	√		√			√
		ADD R2,(mem)	√			√		√
		ADD R3,(mem)	√				√	√
	(3)	MUL R1,R2		√	√	√		
		MUL R1,R3		√	√		√	
		MUL R2,R3		√		√	√	

Figure 3. Classifying instructions at core level (phase 2).

		Components in Processor Core				
Type	Instruction	ALU	MAC	R1	R2	R3
(1)	ADD R1,R3	√		√		√
(2)	ADD R2,(mem)	√			√	
(3)	MUL R1,R2		√	√	√	

Figure 4. Selected instructions after solving the covering problem.

minimum number of rows that can cover all columns of a table similar to Figure 2 and 3 [12]. There are many heuristics available based on greedy algorithm, graph theory and ILP [13]. In our work we have used a graph-based heuristic to solve it [13]. Although no optimality is guaranteed, it always achieves optimal or near optimal solution. Due to lack of space, this algorithm is not discussed here.

Note carefully that the two covering problems in Phase 1 and 2 are obviously dependent. In fact, choosing instructions in phase 2 to cover components of a specific core may affect other cores. To consider both covering tables and find a better solution, in such scenarios we include the information of phase 1 into the covering table of Phase 2. For example, in Figure 2, the instructions $ADD\ R_i$, (mem) not only affect components in the processor core but they also affect the memory core. Therefore, the memory core is added in the covering table in Phase 2 (i.e. the shaded column in Figure 3). Applying this algorithm to the covering problem of Figure 3, produces Figure 4 which uses three instructions instead of nine.

3.4 TDP Formation

After selecting a subset of instructions for each core (e.g. Figure 4) we construct the test program by including all instructions chosen in the test program. To be flexible in applying any number of test patterns to each core, we embed each instruction in a loop.

If deterministic patterns are used, the number of iterations of loops are already specified. When random patterns are used in the test program, the loop counters should be estimated. This information is often provided by core vendors or test engineers who analyzed the testability of the SoC's cores. A typical TDP procedure for processor core testing is shown in Figure 5. As shown, all loop constructs use at most three instructions. Therefore, applying test patterns to each component of the core is performed at the clock speed of the processor.

```
For (i=1 to L1){
        MOV   R1, (m1)
        MOV   R3, (m2)
        ADD   R1, R3 }
For (i=1 to L2){
        MOV   R2, (m3)
        ADD   R2, (m4)}
For (i=1 to L3){
        MOV   R1, (m5)
        MOV   R2, (m6)
        MUL   R1, R2}
```

Figure 5. Test delivery procedure.

Note carefully that Figure 5 is only a typical example. Depending on the available test analysis information for cores we can change the test program. For example, loop counters can be increased to provide higher fault coverage (e.g. cores tested by random patterns) or more instruction can be added for higher instruction coverage (e.g. testing memory write).

4 Test Evaluation Framework

In the previous section, we proposed a covering formulation to find the minimum number of instructions that cover the components (inside cores) and cores (inside system chip). By doing this, we can transfer all patterns (random or deterministic) to the core/component inputs and collect their responses. However, the overall fault coverage of SoC depends on both the coverage of individual components/core and the coverage of the entire SoC including its cores, glue logic, controller and interconnects.

Our proposed test evaluation framework is shown in Figure 6 and is used for estimation of fault coverage by applying THE test programs to SoCs. As shown, initial test program written in assembly language (.asm) is changed to machine code (.hex) by the assembler/compiler. A C program converts it into a .vec file for SYNOPSYS [14]. The .vec file and gate level HDL code of the core is applied to the fault simulator and the fault coverage is obtained. If the coverage is not satisfactory with user interaction the test program is changed. The process is repeated to obtain a satisfactory fault coverage or no more improvement is achieved. Briefly, the general guideline for modifying test program is based on : 1) increasing loop iterations for random patterns, 2) enhancing functional coverage and 3) including more instructions in the loops to stimulate more control signals.

5 Experimental Results

UTSDSP processor is used for experimentation [15]. It is compatible, at the instruction set level, with the TI's TMS320C54x DSP processor family [16]. This is a fixed-point digital signal processor designed in the University of Tehran. There are total of 187 instructions in UTS-DSP processor running at 65MHz. In this system, there are five major cores, i.e. processor, SRAM (32K 16-bit), ROM (2K 16-bit), Serial Port Interface (SPI) and Host Port Interface (HPI) cores.

Table 1 shows the architectural and test specification of cores in UTS-DSP and the statistic of the procedures used as

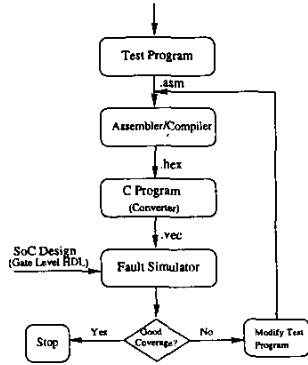

Figure 6. Test evaluation framework.

Table 1. Test specification of cores in UTS-DSP.

Cores	*Test Method*	*Procedure Size [Byte]*				*# of Inst. in TDP*
		PGP	TDP	SGP	CMP	
Processor	Random	21	286	27	16	67
RAM	Deterministic	10	53	–	29	6
ROM	Deterministic	8	19	27	16	5
SPI	Random	21	22	27	16	8
HPI	Random	21	35	27	16	11

test programs. Note that only RAM and ROM were tested using deterministic set of patterns, i.e. 9N March algorithm for RAM [17] [18]. The "-" in SGP column indicates that using March algorithm we do not store the memory-read results. Instead, we compare read and write data on fly [11]. For ROM core, we assumed all of its content will be read and verified.

The final test evaluation of the UTS-DSP system is tabulated in Table 2. Specifically, the overall execution time of the test programs in mili-second and the fault coverage percentage (FC%) for each core are given in the last two columns. Note carefully that the fault coverage reported here is obtained using the test evaluation framework of Figure 6 for embedded cores one by one. The results are identical to the fault coverage of non-embedded core testing because our software test methodology successfully delivers/analyzes the test patterns/signatures to/from cores. We need to point out, however, that the overall fault coverage, achieved after enhancing programs, was about 86%. This is mainly due to difficulty in testing the SPI/HPI controllers and the wide interconnects among the cores.

6 Conclusion

In this paper, we presented a methodology for systematic generation of test programs for SoC software-based testing.

Table 2. Experimental results.

Cores	*TP Size [Byte]*	*# of Patterns*	*Test Time [ms]*	*FC [%]*
Processor	350	1320	2.54	95.6
RAM	92	10	81.00	100
ROM	62	2014	0.12	100
SPI	86	95	0.28	86.1
HPI	99	120	0.31	81.3

For a given processor and its instruction set, the SoC architectural information and test specification of cores are used to develop test procedures. These procedures are combined and iteratively executed to test SoC's cores and cores' components at the system and core levels, respectively. The generic covering formulation helps to minimize the test time while it guarantees structural coverage of all cores and components.

References

[1] Y. Zorian, S. Dey and M. Rodgers, "Test of Future System-on-Chip," in Proc. *Int. Conf. on Computer-Aided Design (ICCAD'00)*, pp. 392-398, 2000.

[2] Y. Zorian, "System-Chip Test Strategies," in Proc. *Design Automation Conf. (DAC'98)*, pp. 752-757, 1998.

[3] J. Rajski and J. Tyszer, "Modular Logic Built-In Self-Test for IP Cores," in Proc. *Int. Test Conf. (ITC'98)*, pp. 313-321, 1998.

[4] S. Hellebrand, H. Wunderlich and A. Hertwig, "Mixed-Mode BIST Using Embedded Processors," in Proc. *Int. Test Conf. (ITC'96)*, pp. 195-204, 1996.

[5] K. Batcher and C. Papachristou, "Instruction Randomization Telf Test for Processor Cores," in Proc. *VLSI Test Symp. (VTS'99)*, pp. 34-40, 1999.

[6] R. Rajsuman, "Testing a System-on-a-Chip with Embedded Microprocessor," in Proc. *Int. Test Conf. (ITC'99)*, pp. 499-508, 1999.

[7] L. Chen, S. Dey, P. Sanchez, K. Sekar and Y. Chen, "Embedded Hardware and Software Self-Testing Methodologies for Processor Cores," in Proc. *Design Automation Conf. (DAC'98)*, pp. 625-630, 2000.

[8] A. Paschalis, D. Gizopoulos, N. Kranitis, M. Psarakis and Y. Zorian, "Deterministic Software-Based Self-Testing of Embedded Processor Cores," in Proc. *Design, Automation, and Test in Europe (DATE'01)*, pp. 92-96, 2001.

[9] K. Radeka, J. Rajski and J. Tyszer, "Arithmetic Built-In Self-Test for DSP Cores," IEEE Trans. On CAD/ICAS, vol. *16*, no. *11*, pp. *1358-1368*, 1997.

[10] C. Papachristou, F. Martin and M. Nourani, "Microprocessor Based Testing for Core-Based System on Chip," in Proc. *Design Automation Conference*, pp. 586-591, 1999.

[11] M. Bushnell and V. Agrawal, *Essentials of Electronic Testing*, Kluwer, 2000.

[12] E. McCluskey, *Logic Design Principles*, Prentice Hall, 1986.

[13] G. DeMicheli, *Synthesis and Optimization of Digital Circuits*, McGraw Hill, 1994.

[14] Synopsys Design Analyzer, "User Manual for SYNOPSYS Toolset version 2000.06-1," Synopsys Inc., 2001.

[15] P. Riahi, M. Movahedin, M. Fayyazi and A. Ghalambor-Dezfoli, "UTS-DSP IC Core," in Proc. *Iranian Conf. on Electrical Engineering (ICEE'99)*, pp. 71-79, 1999.

[16] *TMS320C54x CPU and Instruction Set Reference Guide*, Texas Instrument, Inc., 1996.

[17] R. Dekker, F. Beenker and L. Thijssen, "Fault Modeling and Test Algorithm Development for SRAMs," in Proc. *Int. Test Conf. (ITC'88)*, pp. 343-351, 1988.

[18] M. Tehranipour, Z. Navabi and M. Fakhraie, "An Efficient BIST Method For Testing of Embedded SRAMs," in Proc. *Int. Symp. on Circuits and Systems (ISCAS'01)*, vol. 5, pp. 73-76, 2001.