DEFUSE: A Deterministic Functional Self-Test Methodology for Processors

Li Chen and Sujit Dey
Department of Electrical and Computer Engineering
University of California, San Diego
lichen, dey@ece.ucsd.edu

Abstract

At-speed testing is becoming increasingly difficult with external testers as the speed of microprocessors approaches the GHz range. One solution to this problem is built-in self-test. However, due to their reliance on random patterns, current logic BIST techniques are not able to deal with large designs without adding high test overhead. In this paper, we propose a functional self-test technique that is deterministic in nature. By targeting the structural test need of manageable components with the aid of processor functionality, this technique has the fault coverage advantage of deterministic structural testing and the at-speed advantage of functional testing. Most importantly, by relieving testers from test application, it enables at-speed testing of GHz processors with low speed testers. We have demonstrated our methodology on a simple accumulator-based microprocessor. The results show that with the proposed technique, we are able to apply high-quality at-speed tests with no test overhead.

Keywords – At-speed testing, self-test, microprocessor, instructions, structural testing.

1. Introduction

As the speed of microprocessors approaches the GHz range, at-speed testing is becoming increasingly difficult with external testers. According to the 1997 Semiconductor Industry Association Roadmap [1], if the current testing techniques are to be continued, the test equipment cost can rise towards $20 million. Moreover, due to the inherent inaccuracy of testers, at-speed testing of high-speed IC's will result in an unacceptably high yield loss of 48% by 2012. To ensure the economic viability of the industry to manufacture high-performance IC's, radically new testing techniques are needed. One solution to the problem of at-speed testing is built-in self-test [2], as it alleviates the burden on external testers. However, while memory BIST techniques have been successful for self-testing of embedded memory, current logic BIST techniques remain mostly impractical due to low fault coverage, coupled with large area and performance overheads. A primary reason for the relative success of memory BIST is due to the deterministic nature of the memory tests facilitated by the regular structure of memory components, as opposed to logic BIST's reliance on random patterns.

While manufacturing test of processors has been extensively addressed over the last several years, only recently has self-testing been in focus. In an earlier work, Bieker et al. developed techniques for efficient compilation of self-test programs for embedded processors [3]. But they left the responsibility for generating the self-test programs to the test engineers. More recently, Shen et al., and Batcher et al. have proposed techniques for functional self-testing of processors [4][5]. Both approaches rely on generating and applying random instruction sequences to the processor core. In [6][7][8][9], the processor functionality has been used for on-chip test generation and test response compaction. In [6] and [7], random operations and operands are generated and applied to test the ALUs of DSP cores. In [8] and [9], the processor is used to generate random test patterns, and scan chains are used to apply the test patterns.

Recognizing the reason behind the success of memory BIST techniques, our goal in this work is to develop a processor self-test methodology that uses random test patterns, but in a more controlled manner, where the targets of the random tests are determined *a priori*. Secondly, to obtain high fault coverage of manufacturing defects, our technique targets structural faults directly, as opposed to functional faults targeted by techniques using random instructions [4][5]. Thirdly, we use the functionality of the processor not only to generate test patterns on-chip (as in [6][7][8][9]), but also to apply the test patterns at the clock speed of the processor, thereby achieving at-speed self-test. In short, we propose a new processor self-testing methodology, DEFUSE, that combines the fault-coverage advantage of deterministic structural testing with the at-speed advantage of functional testing.

We propose a divide-and-conquer approach. The pre-test step includes the generation of realizable component tests and the encapsulation of component tests into self-test signatures. The self-testing step includes the on-chip test application and response collection by using the functionality of the processor under test. At component

This work is supported by MARCO/DARPA Gigascale Silicon Research Center (GSRC).

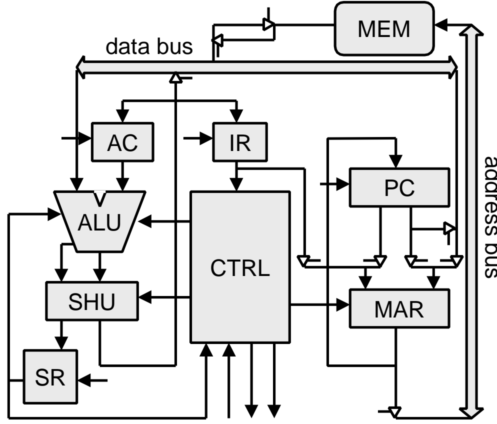


Figure 1. Processor example: PARWAN

level, tests are targeted at structural faults. At processor level, the functionality of the processor is used to apply the structural tests to each component at-speed. The structural component tests are delivered by processor instructions. Therefore, they must obey the constraints imposed by the instruction set.

In the following two sections, we will describe the two steps included in our self-test technique: (1) component test preparation and (2) on-chip self-test. We will use a simple accumulator-based microprocessor named PARWAN [10] (Figure 1) to illustrate our methodology. PARWAN includes the following components: Arithmetic Logic Unit (ALU), Accumulator Unit (AC), Controller (CTRL), Instruction Register Unit (IR), Program Counter Unit (PC), Memory Address Register Unit (MAR), Shifter Unit (SHU), and Status Register Unit (SR).

The remaining of the paper is organized as follows. Section 4 describes the software framework used for evaluating the fault coverage of self-test programs. Section 5 presents the experimental results in terms of processor fault coverage. The conclusion is drawn in Section 6.

2. Component test preparation

During component test preparation, we develop tests for individual components of the processor, such as the ALU, the SHU, and the PC. Structural faults are targeted during component test generation. Component tests can either be stored or generated on-chip. If stored tests are used, component tests are loaded directly to the processor memory before the test. If tests are to be generated on-chip, we characterize the test need of the component by a self-test signature, which includes the seed (*S*) and the configuration of a pseudo random number generator (*C*), as well as the number of test patterns to be generated (*N*). The self-test signatures, instead of actual tests, are loaded to the processor memory before the test. The self-test signatures can be expanded on-chip into test sets using a pseudo random number generation program. Multiple self-test signatures may be used for one component, if it is necessary for increasing the fault coverage. The purpose of using self-test signatures is to reduce the time for loading the tests and the memory requirement for storing all test patterns at the same time. Notice that both self-test signatures and stored test sets can be loaded to processor memory using a low speed external tester prior to the application of tests.

One of the challenges of the proposed method lies in the generation of realizable component tests. In the next two sections, we will describe the constraints on component tests imposed by the processor instruction set and the methods for modeling these constraints during test generation.

2.1 Instruction-imposed constraints

Since the delivery of component tests relies on processor instructions, it is impossible to deliver some test patterns. Thus, component tests need to obey certain constraints imposed by the processor instruction set, such as validity of input values and dependency between inputs. For example, while most data inputs are constraint-free, control inputs often come with many constrains since they are determined by instruction opcodes.

We will next use one component of the PARWAN processor, SHU, to illustrate the types of constraints imposed by the instruction set.

A block diagram of SHU is shown in Figure 2. The input signals include `data_in`, `in_flag`, and the shifting signals from the controller. `in_flag` include 4 bits, `v`, `c`, `z`, and `n`, which denote overflow, carry, zero, and negative, respectively. The shifting signals includes two bits, `asl` and `asr`, which denote arithmetic-shift-left and arithmetic-shift-right.

The constraints imposed by the processor instruction set can be divided into two types, depending on whether the constraint is timing-related. We define constraints which can be specified in a single time frame as *spatial constraints*, and constraints spanning over several time frames as *temporal constraints*.

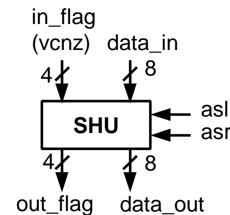


Figure 2. SHU

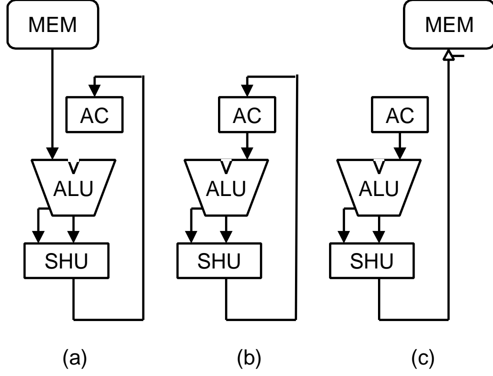


Figure 3. Hardware paths involved in testing the SHU (a) loading data into AC, (b) shifting, and (c) storing AC content to memory

For SHU, the spatial constraints imposed by the processor instruction set include the following:

1. `asl` and `asr` cannot be both 1,
2. `z` and `n` must be consistent with `data_in`, and
3. `v = xor(c, sign_bit(data_in))`.

The temporal constraints on SHU are imposed by the sequence of instructions that apply tests to SHU. The instruction sequence includes three steps. The first step is to store the data to be shifted into the accumulator (AC). This can either be done by the load accumulator instruction or any instruction that stores its result in AC. The second step is to use a shift instruction to shift the data stored in AC. The shift result cannot be stored directly to memory and has to be stored in AC temporarily. The third step is to store the AC content to memory for later analysis. The hardware paths in these three steps are shown in Figure 3. Notice that it involves three passes through SHU. Due to the possibility of aliasing, a test set with full coverage on SHU might leave certain faults undetected, if applied to SHU using instructions. To prepare tests not susceptible to the aliasing problem, temporal constraints need to be modeled during component test generation.

Tupuri et al. and Vishakantaiah et al. have addressed the issue of constraint test generation in [11][12][13]. They have proposed a methodology in systematically extracting component constraints from the processor HDL description. The extracted constraints are used in generating component test vectors. The component test vectors can later be translated to processor level test vectors, which have to be applied by external testers. Such constraint extraction method, if made available, can be used in the self-test scheme proposed in this paper. However, as Wohl et al. have pointed out in [14], certain architectural constraints cannot be detected at circuit level. Therefore, the set of constraints extracted from the structural description of the processor is only a subset of the constraints imposed by the instruction set.

2.2 Constraint modeling in test generation

Having described the spatial and temporal constraints imposed by the processor instruction set, we will now describe how these constraints can be modeled during component test preparation.

If component tests are generated by ATPG, spatial constraints can be specified during test generation with the aid of the ATPG tool. For instance, constraint "`asl` and `asr` cannot be both 1" can be specified to the ATPG tool we are using as follows [15]:

```
ADD ATPG FUNCTION f1 AND asl asr
ADD ATPG CONSTRAINT 0 f1
```

The first statement defines a new variable, `f1 = and(asl, asr)`. The second statement restricts the value of `f1` to 0. As a result, `asl` and `asr` can never be both 1.

As an alternative, spatial constraint can be specified with a virtual constraint circuit proposed in [11].

If random tests are used for components, random patterns can only be used on independent inputs. In the case of SHU, these would be `data_in` and `c`. Inputs such as `z`, `n`, and `v` can be derived from these inputs. It is inconvenient to assign random patterns to instruction-related signals, such as the shifting signals. Therefore, they are fixed when random patterns are applied to other inputs. The fixed value of the instruction-related signals may be changed if it helps to improve the component fault coverage.

For example, the shifting signals (`asl` and `asr`) can be fixed to 10, 01, or 00 while random patterns are applied to `data_in` and `c`. This corresponds to three SHU modes: left shift, right shift, and no shift. Random tests can be developed for each mode separately, until an acceptable fault coverage is achieved. As a result of component test preparation, a self-test signature is generated for each SHU mode, as shown in Table 1. Multiple self-test signatures can be used for each mode, if it is necessary for improving the fault coverage.

The temporal constraints of SHU can be modeled using the three-phase sequential circuit shown in Figure 4. The three phases correspond to the three instructions for applying tests to SHU, which are loading data into AC, shifting, and storing AC content to memory (Figure 3). Notice that the data inputs and flag inputs of SHU are

Table 1. Self-test signatures for SHU

`asr, asl`	`data_in, c`	`v, z, n`
fixed to 00	($C1$, $S1$, $N1$)	derived
fixed to 01	($C2$, $S2$, $N2$)	derived
fixed to 10	($C3$, $S2$, $N3$)	derived

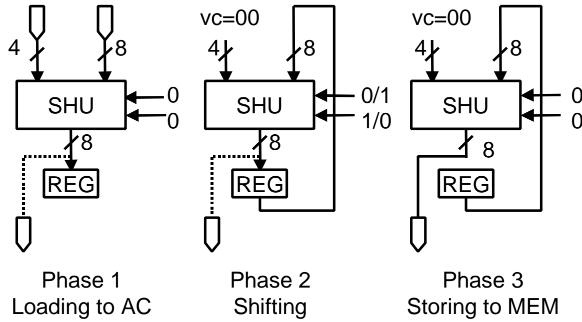


Figure 4. Circuit for modeling the temporal constraint of SHU

only connected to the primary inputs in the first phase, in which the AC content is loaded from the memory. The data outputs of SHU are only connected to the primary outputs in the third phase, in which the test response is stored to memory. The shifting signals in these two phases are set to 0's. The `v` and `c` flags are set to 0's in the second and the third steps, since neither shift instructions nor store instruction can set them to 1. At any phase, the inputs to SHU must also obey the spatial constraints we have described before.

3. On-chip self-test

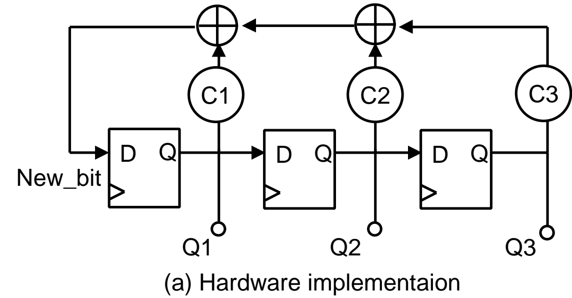
In this section, we describe the phase of the on-chip self-test, which includes the on-chip generation of component test patterns, the delivery of component tests, and the analysis of their responses using processor instructions.

3.1 On-chip test generation

If tests are chosen to be generated on-chip, we need to expand the component self-test signatures determined during component test preparation into test sets using a pseudo random number generator. Figure 5 illustrates this process. We could chose to generate the pseudo random numbers using a Linear Feedback Shift Register (LFSR), as shown in Figure 5(a). Alternatively, a more flexible and more cost-effective solution is a software LFSR, as shown in Figure 5(b). Unlike the hardware LFSR, the software LFSR is programmable and can be reused to generate any LFSR configurations without any test overhead. The configuration of the LFSR is determined by a self-test signature, which includes the characteristic polynomial of the LFSR (C), the initial state of the LFSR (S), as well as the number of test patterns to be generated (N). Multiple signatures can be used for one component, if it is necessary for improving the fault coverage.

3.2 On-chip test delivery and response collection

Since the component tests are developed under the constraints imposed by the processor instruction set, it



(a) Hardware implementaion

```
Q = S
Do N times
  begin
  AC <= Bitwise-and(C, Q);
  if AC has an odd number of 1's
    New_bit <= 1;
  else New_bit <= 0;
  Q <= New_bit:(Q shifted to right by 1)
  end
```

(b) Software implementation

Self-test signature: (C, S, N)

Figure 5. Hardware and Software implementation of LFSR

will always be possible to generate delivery programs, which can be used to apply component tests.

However, special care needs to be taken for collecting component test response. As we discussed in Section 2, on the input end, data inputs and control inputs have different controllability, therefore constraints need to be used in test preparation. Similarly, on the output end, data outputs and status outputs have different observability and should be treated differently during response collection. Status outputs can only be observed by special sequences of instructions.

Here we illustrate the propagation of status outputs with the ALU in the PARWAN processor (Figure 1). A block diagram of ALU is shown in Figure 6.

The ALU has four status outputs, `v` (overflow), `c` (carry), `z` (zero), and `n` (negative). The sequence of instructions for propagating these signals is shown in Figure 7. Instructions 0 – 2 apply a test vector to ALU. (`lda` and `sta` stands for load accumulator and store

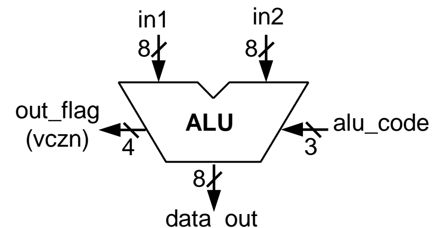


Figure 6. ALU

```
0             lda addr(y)
1             add addr(x)
2             sta data_out
3             lda 11111111
4             brav ifv
5             and 11110111
6  label ifv  brac ifc
7             and 11111011
8  label ifc  braz ifz
9             and 11111101
10 label ifz  bran ifn
11            and 11111110
12 label ifn  sta flag_out
```

Figure 7. Observing status outputs

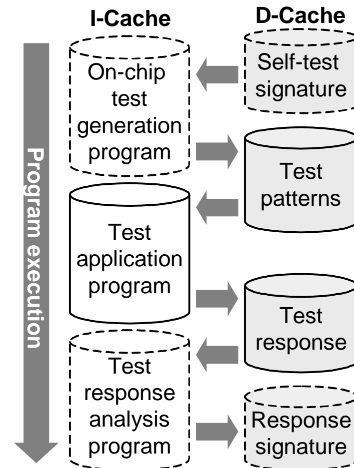


Figure 8. Self-test methodology

accumulator, respectively.) After instruction 1, the status outputs become available. Instructions 3 – 11 create an image of the status outputs in the accumulator. (`brav`, `brac`, `braz`, and `bran` are branch-if-overflow, branch-if-carry, branch-if-zero, and branch-if-negative instructions, respectively.) First, an all-one vector is loaded to the accumulator. If `v` is one, the all-one vector is left untouched. Otherwise, a zero replaces the one at the 4th bit from right. Other status bits are treated similarly. After the execution of instruction 11, an image of the status output is created in the accumulator. Instruction 12 stores this image to memory.

In general, although there are no instructions for storing the status outputs of a component directly to memory, the image of the status outputs can be created in memory by using instructions whose results depend on the status outputs. This technique can be used to observe the status outputs of any components.

In summary, Figure 8 outlines the steps involved during the phase of on-chip self-test. In the case when self-test signatures are used, an on-chip test generation program emulates a pseudo random pattern generator and expands the signatures into test patterns. The test patterns are delivered/applied to components by an on-chip test application program at the speed of the processor. The test application program also collects the test responses and saves them to memory. If desired, the test responses can be compressed into response signatures using a test response analysis program. The responses are stored into memory and can later be unloaded and analyzed by an external tester.

By targeting the structural test need of smaller and less complex components, our self-test technique, DEFUSE, has the fault coverage advantage of deterministic structural testing. Since component test application and response collection are done with instructions instead of with scan chains, it requires no area or performance overhead, and the test application is performed at-speed. Most importantly, by shifting the role of external testers from applying tests and monitoring responses to loading test programs and unloading responses, it enables at-speed testing of GHz processors with low speed testers.

4. Test evaluation framework

To evaluate the fault coverage of a test program on the processor under test, we have established the test evaluation framework shown in Figure 9. The assembler takes the test program and prepares a VHDL test bench containing the initialized instruction memory and data memory. The VHDL simulator takes the design description, runs the test bench, and captures the input signals to the processor. These are the test vectors to be applied during fault simulation. Finally, the effectiveness of the test program is evaluated by the fault coverage. Currently, we use *QuickHDL*, *Leonardo*, and *FlexTest* from Mentor Graphic for HDL simulation, synthesis, and fault simulation, respectively.

5. Experimental results

We have applied the DEFUSE methodology to PARWAN, a simple accumulator-based microprocessor with unified data memory and instruction memory [10], as shown in Figure 1. The instruction set contains 17 instructions. The synthesized version of PARWAN contains 1785 equivalent NAND gates and 53 flip-flops. The data bus is 8-bit wide, shared by both `data_in` and `data_out`. The address bus is 12-bit wide. Accesses to both buses are through tri-state buffers.

Three components of PARWAN were directly targeted during component test preparation. They are ALU, SHU, and PC. The expected test coverage is

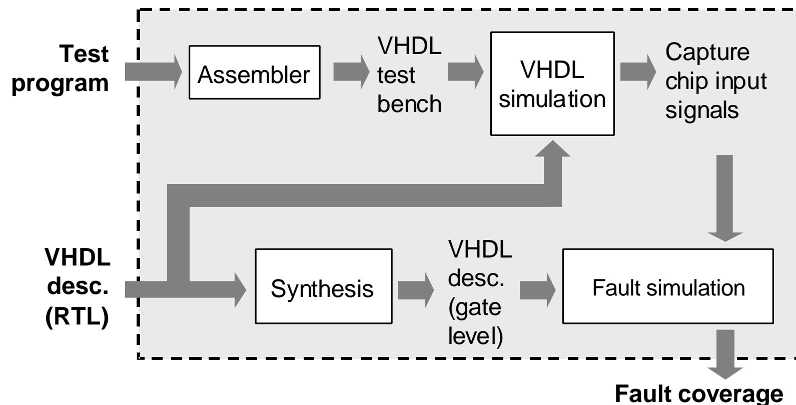


Figure 9. Test evaluation framework

98.81% for the ALU, 99.27% for the SHU, and 85.00% for the PC. We were unable to obtain full coverage for these components due to the existence of constraints imposed by the instruction set. No tests were generated for other components, as they are not easily accessible through instructions. We expect them to be tested intensively during the test for the targeted components.

Both pseudo random tests and deterministic tests were used for testing these components. For example, 205 pseudo random test patterns were used for testing the ALU. 12 deterministic test patterns were used for testing the PC. The PC was tested with *jump* instructions. Several instructions, such as the next *jump* instruction, had to be stored at *each* jump target. Thus, to reduce the memory required for storing such instructions, we chose to test the PC with deterministic tests, as it required fewer test patterns (i.e. fewer *jump* instructions).

Table 2 shows statistics on the on-chip test generation program and test application programs used for difference components (Figure 8). For each program, the number of instructions, the size of the program in bytes, and the execution time in processor cycles are given. Program sizes indicate the storage requirement of the self-test programs and the tester time needed to load these programs to the system memory.

The complete self-test program achieved an overall fault coverage of 91.42%. Notice that unlike what a conventional fault simulator assumes, DEFUSE does not require processor outputs to be monitored by an external tester during the application of self-test. The test response is collected after the test by unloading the component test response stored in memory. In general, if a conventional fault simulator is used for evaluating the fault coverage of DEFUSE, only primary outputs related to memory should be observed. This includes address outputs, data outputs, and read/write signals for the memory. In the case of PARWAN, all primary outputs are memory-related. Therefore, all outputs are observed during fault simulation.

The component fault coverages are shown in Table 3, in which the first row contains the names of the individual components (accumulator units, instruction register unit, program counter unit, memory address register unit, status register unit, arithmetic logic unit, shifter unit, control unit, datapath interface, and CPU interface). The component fault coverages are obtained from the full-processor fault simulation, not from the fault simulation on individual components. Notice that the datapath interface (DP I/F) mainly consists of buses and tri-state buffers. It is not targeted during component test preparation. Hence the fault coverage for this unit is low. The presence of tri-state buffers reduced the testability of the circuit, since error propagated to tri-state buffer enabling signals can cause bus-contention problem or floating bus problem.

Table 4 shows the comparison between DEFUSE and conventional testing techniques.

The fault coverage by sequential ATPG is unacceptably low, even after we manually identified all tri-state buffers and set constraints on their enabling signals to ensure error propagation. The ATPG constraints are set so that for any bus, at any instant, one and only one tri-state buffer is writing to the bus. Notice that even with the constraints, this behavior is only guaranteed for the fault-free circuit. The fault coverage is much lower (4.22%) if no information whatsoever is given to the test generator. The abort limit of sequential ATPG are as follows: backtrack = 30, cycle = 300, and time = 300 seconds. Sequential ATPG is able to apply at-speed test without test overhead. However, it requires a high-speed tester for applying tests and monitoring responses at-speed.

Full Scan is able to achieve modest fault coverage with an area overhead of 10.92%. Note that the area overhead is an under-estimate, since it doesn’t include the

Table 2. Statistics on the self-test programs

	Pseudo random number generation program	Test application			Total
		ALU	SHU	PC	
# instructions	46	213	243	73	575
Program size [bytes]	87	424	471	147	1129
Execution time [cycles]	87764	37686	11604	595	137649

Table 3. Component fault coverage [%]

AC	IR	PC	MAR	SR	ALU	SHU	CTRL	DP I/F	CPU I/F
99.33	98.61	89.16	97.22	98.88	98.48	94.08	88.26	71.57	97.14

Table 4. Comparison with existing techniques

	Overhead		Fault Coverage	At-speed?	External tester?
	Area	Delay			
Sequential ATPG	0	0	46.82%	Y	Y
Full Scan	10.92%	0.53%	82.18%	N	Y
LBIST	204.03%	41.40%	20.13%	Y	N
DEFUSE	0	0	91.42%	Y	N

routing overhead of the scan chains. The fault coverage is not extremely high due to the presence of tri-state buffers. No information is given to the test generator regarding the tri-state buffer enabling signals.

We also compared DEFUSE with a commercial Logic BIST tool (referred to as *LBIST* hereafter). *LBIST* first applies full scan and boundary scan to the circuit, then connects the scan chains to a BIST circuit, which contains a pattern generator (LFSR) and a signature compressor (MISR). The scan chains are connected to the LFSR via a phase shifter, which is designed to reduce the level of linear correlation between scan chains.

LBIST is unable to insert BIST structures to a circuit with bi-directional pins and possible bus contention problems. Hence, before applying *LBIST*, we manually modified the circuit description of PARWAN. The modifications include: (1) splitting all bi-directional pins into separate I/O pins, and (2) replacing all tri-state buffers with selectors. Figure 10 illustrates the structure of a selector corresponding to a pair of tri-state buffers, where *d1* and *d2* denote the data signals and *s1* and *s2* denotes the select signals. Bus contention problems are impossible with selectors. However, the outputs of the selectors are scrambled when more than one select signal is high or none of the select signal is high.

We divided the 53 flip-flops of PARWAN into 30 scan chains. *LBIST* automatically chose a 17-bit LFSR as the pattern generator. A total of 32767 vectors were generated by the LFSR. The test overhead and the fault coverage of *LBIST* are shown in Table 4. The unusually high test overhead is due to the small size of the original circuit. The fault coverage achieved by *LBIST* is disappointing. Possible reasons are: (1) the 17-bit LFSR is unable to provide the randomness required for achieving a high coverage, and (2) *LBIST* is unable to set the select signals properly, causing the selectors to block the error propagation paths.

To improve the fault coverage, we tried increasing the size of the LFSR and the number of patterns generated by the LFSR. We performed two test runs, in which the size of the LFSR was increased to 32 and 53 and the number of patterns were increased to 1,000,000. The results show no improvement in fault coverage.

It should be noted that we applied *LBIST* directly on an unmodified processor, and processors are typically random-pattern-resistant. The random-pattern-testability of a circuit can be improved with techniques such as test point insertion and the bounding of X-generators. As

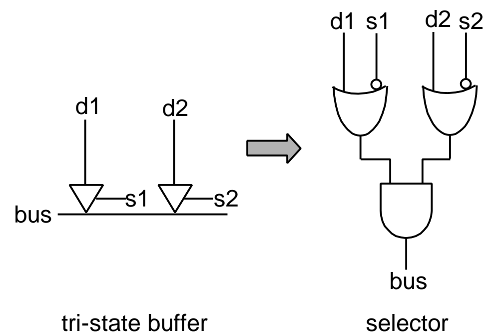


Figure 10. Converting tri-state buffers to selectors

reported in [16], if a circuit is modified to be BIST-ready (i.e., random pattern testable) with such techniques, scan-based logic BIST *can* achieve relatively high fault coverage. However, we were not able to achieve similar results with *LBIST*, as it performed no modifications to the circuit.

The proposed method, DEFUSE, is able to achieve high fault coverage without any test overhead. Most importantly, it enables at-speed testing without any requirement on the performance of external tester. For this particular example, the fault coverage achieved by DEFUSE is higher than that achieved by full-scan. This is because processor instructions are designed to set the tri-state buffer enabling signals properly without any information from the test engineer.

Notice that the execution time of the self-test program is not the tester time, since the processor under test need not to be monitored by external tester. The external tester is used for loading and unloading the self-test program and the response data. Therefore, the tester time is determined by the size of the self-test program and the size of the output vectors to unload.

To prove the effectiveness of DEFUSE on large designs, we are now in the process of applying it to an industrial microprocessor, which is a hardware implementation of the Java virtual machine instruction set. The microprocessor has an area of 381845 equivalent NAND gates. A self-test program of 1400 instructions has been applied to its floating-point unit, which has an area of 23365 equivalent NAND gates. A fault coverage of 90.5% has been achieved on the floating-point unit by this self-test program.

6. Conclusion

In conclusion, we propose a new technique that enables at-speed self-testing using the functionality of the processor under test. Structural faults are targeted during the self-test, while the functionality of the processor is used as a mean for applying tests. We have demonstrated the effectiveness of the proposed method on a simple microprocessor. The advantages of the proposed technique include enabling at-speed testing with low speed testers, as well as achieving high fault coverage without sacrificing area or performance. By breaking up a complex system into manageable pieces and targeting at individual components, we expect to apply this technique on large processors and systems in the future.

Acknowledgment

The authors would like to thank Pablo Sanchez for many inspiring discussions and numerous help with commercial CAD tools.

References

[1] *The National Technology Roadmap for Semiconductors*, Semiconductor Industry Association, 1997.

[2] V.D. Agrawal et al., "Built-in self-test for digital integrated circuits," *AT&T Technical J.*, Mar. 1994, pp. 30.

[3] U. Bieker and P. Marwedel, "Retargetable self-test program generation using constraint logic programming," *Proceedings of the 32nd Design Automation Conference*, San Francisco, California, June 1995, pp. 605 – 611.

[4] J. Shen and J. A. Abraham, "Native mode functional test generation for processors with applications to self test and design validation," *Proceedings of the International Test Conference 1998*, Washington DC, Oct. 1998, pp. 990-999.

[5] K. Batcher and C. Papachristou, "Instruction randomization self test for processor cores," *Proceedings of the 17th IEEE VLSI Test Symposium*, Dana Point, California, April 1999, pp. 34 – 40.

[6] J. Rajski and J. Tyszer, *Arithmetic Built-in Self-Test for Embedded Systems*, Prentice Hall, 1998.

[7] K. Radecka, J. Rajski, and J. Tyszser, "Arithmetic built-in self-test for DSP cores," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol.16, no.11, Nov. 1997, pp. 1358 – 69.

[8] S. Hellebrand and H.-J. Wunderlich, "Mixed-mode BIST using embedded processors," *Proceedings of the International Test Conference 1996*, Washington DC, Oct. 1996, pp. 195 – 204.

[9] R. Dorsch and H.-J. Wunderlich, "Accumulator based deterministic BIST," *Proceedings of the International Test Conference 1998*, Washington DC, Oct. 1998, pp. 412 – 421.

[10] Z. Navabi, *VHDL: Analysis and modeling of digital systems*, New York, McGraw-Hill, 1993.

[11] R. Tupuri and J. A. Abraham, "A novel functional test generation method for processors using commercial ATPG," *Proceedings of the International Test Conference 1997*, Washington DC, Nov. 1997, pp. 743 – 752.

[12] P. Vishakantaiah, J. A. Abraham, and D. G. Saab, "CHEETA: Composition of hierarchical sequential tests using ATKET," *Proceedings of the International Test Conference 1993*, Baltimore, Maryland, Oct. 1993, pp. 606 – 615.

[13] R. Tupuri, A. Krishnamachary and J. A. Abraham, "Test generation for gigahertz processors using an automatic functional constraint extractor," *Proceedings of the 36th Design Automation Conference*, New Orleans, Louisiana, June 1999, pp. 647 – 652.

[14] P. Wohl and J. Waiculauski, "Test generation for ultra-large circuits using ATPG constraints and test-pattern templates," *Proceedings of the International Test Conference 1996*, Washington DC, Oct. 1996, pp. 13 – 20.

[15] *FastScan and FlexTest Reference Manual*, V8.6_4, Mentor Graphics Corporation.

[16] G. Hetherington, T. Fryars, N. Tamarapalli, M. Kassab, A. Hassan, and J. Rajski, "Logic BIST for large industrial designs: real issues and case studies," *Proceedings of the International Test Conference 1999*, Atlantic City, New Jersey, Sept. 1999, pp. 358 – 367.