Systematic Software-Based Self-Test for Pipelined Processors

Mihalis Psarakis[1] Dimitris Gizopoulos[1] Miltiadis Hatzimihail[1]
Antonis Paschalis[2] Anand Raghunathan[3] Srivaths Ravi[3]

[1] Dept. of Informatics, University of Piraeus, Greece
{mpsarak | dgizop | mhatz}@unipi.gr

[2] Dept. of Informatics & Telecomm., University of Athens, Greece
paschali@di.uoa.gr

[3] NEC Laboratories America, Princeton, NJ, USA
{anand | sravi}@nec-labs.com

ABSTRACT

Software-based self-test (SBST) has recently emerged as an effective methodology for the manufacturing test of processors and other components in Systems-on-Chip (SoCs). By moving test related functions from external resources to the SoC's interior, in the form of test programs that the on-chip processor executes, SBST eliminates the need for high-cost testers, and enables high-quality at-speed testing. Thus far, SBST approaches have focused almost exclusively on the functional (directly programmer visible) components of the processor. In this paper, we analyze the challenges involved in testing an important component of modern processors, namely, the pipelining logic, and propose a systematic SBST methodology to address them. We first demonstrate that SBST programs that only target the functional components of the processor are insufficient to test the pipeline logic, resulting in a significant loss of fault coverage. We further identify the testability hotspots in the pipeline logic. Finally, we develop a systematic SBST methodology that enhances existing SBST programs to comprehensively test the pipeline logic. The proposed methodology is complementary to previous SBST techniques that target functional components (their results can form the input to our methodology), and can reuse the test development effort behind existing SBST programs. We applied the methodology to two complex, fully pipelined processors. Results show that our methodology provides fault coverage improvements of up to 15% (12% on average) for the entire processor, and fault coverage improvements of 22% for the pipeline logic, compared to a conventional SBST approach.

Categories and Subject Descriptors

B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance.

General Terms

Reliability, Measurement, Performance, Experimentation.

Keywords

Processor testing, Functional Testing, Software-Based Self-Test.

1. INTRODUCTION

Rising test data volumes, together with the high cost of functional testers, and the increasing gap between tester frequencies and SoC operating frequencies, have driven the adoption of various design-for-testability (DFT) techniques. For example, scan-based and built-in self-test (BIST) solutions such as [1] relax the requirements on testers and reduce the overall test cost. However, the associated hardware overheads, performance degradation, and excessive power consumption are not acceptable for carefully optimized designs like processors. Functional test still forms an integral part of the manufacturing test process for processors, *e.g.*, for performance test and speed binning.

Software-based self-test (*SBST*) is an approach that has gained increasing acceptance for testing processor and other components in SoCs. SBST moves the test functions from external testers to on-chip resources: processors, memories, and on-chip buses. A typical flow for SBST involves downloading test programs and data into on-chip memories, using a low-speed, low-cost external tester. Subsequently, these test programs are executed by the processor at actual/full speed (at-speed) and test responses are stored back in the on-chip data memory from which they can be read out. Test application is performed at the processor's normal operating conditions and frequency, and no extra power is consumed compared to the normal operation.

Several SBST approaches have been proposed in the literature [2]–[12]; a comprehensive survey of the topic can be found in [13]. Of the various SBST approaches, [2], [3], and [4] base test generation on the functional behavior of the processor. Based on a functional fault model, Shen *et al.* [2] performed functional testing on a graph-based model of the processor. Batcher *et al.* [3] exploit on-chip hardware to generate pseudorandom instruction sequences. Parvathala *et al.* [4] proposed an automated self-test methodology that generates random instruction sequences with pseudorandom data supplied by software LFSRs.

Other SBST approaches [5]–[11] are structural in nature. Most of these approaches have been applied either to low-complexity processor models (short word length, simple functional units and without pipelining), or to selected functional modules in complex processors. Corno *et al.* [5] proposed a test flow based on evolutionary algorithms and presented experimental results on an 8-bit microcontroller (8051 core). Chen and Dey [6] proposed an SBST methodology that abstracts the circuitry surrounding the ALU as constraints that can be used during test generation. This methodology was first manually applied to a simple processor core (Parwan). It was improved in [7], where a solution for automatic constraint extraction was proposed. The resulting methodology was applied to the combinational logic in the execution stage of a commercial, embedded processor (Tensilica's Xtensa). Kranitis *et al.* [8] developed a high-level, component-oriented SBST metho-dology that is based on knowledge of the ISA and the processor's register transfer level (RTL) description, and applied it to two pipelined processors—Plasma/MIPS and MIPS R3000 compatible processor. Both processors are pipelined, but they do not fully implement the mechanisms of hazard detection and data forwarding. Recent advances in SBST include [9], [10], and [11]. While the authors of [9], [11] focus their efforts on speed binning and delay fault testing issues in processors, Gurumurthy *et al.* [10] presented a technique that furthers the coverage of random test instruction sequences. Authors of [9], [10] targeted complex RISC processors such as the

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

OpenRISC 1200. However they do not provide fault coverage results for the entire processor.

In this paper, we develop an SBST methodology that focuses on the most commonly used performance enhancing mechanism in processors: *pipelining*. Using quantitative and qualitative analyses, our work shows that: (a) pipeline-related logic represents a significant portion of modern embedded processors, and (b) it is very poorly tested by SBST programs that are developed for functional components such as register files and the ALU. A testability study for pipelined processors was given in [12], where the testability problems were demonstrated and preliminary solutions were discussed. The contributions of the present paper include a systematic testability analysis of pipelined logic (address related components, hazard detection and forwarding logic) and the development of effective, generic solutions that can be applied to test the pipeline logic of *any* processor. The proposed solutions are integrated in a largely automatable SBST methodology which adapts available SBST programs and enhances them to achieve high fault coverage for the pipeline logic. The experimental results for two fully pipelined RISC processors show that the proposed solutions yield a 22% fault coverage improvement for the pipeline logic itself. Significant improvements are also seen in the overall processor's fault coverage: up to 15% fault coverage improvement (12% on average) over existing solutions for different implementations of the benchmark processors.

2. TESTING PIPELINED PROCESSORS

We evaluated the challenges of pipeline logic testing using two publicly available fully pipelined RISC processor models: miniMIPS [14] and OpenRISC 1200 [15]. Both models implement a well-known 5-stage pipeline architecture with hazard detection and data forwarding mechanisms [16], which allowed us to study the real testability problems of pipelining. While we present our analysis in the context of miniMIPS, we note that the analysis is equally valid for OpenRISC or any other pipelined processor.

2.1 Testability Problems

We identify the testability problems of a pipelined processor by first applying test routines that target only the functional components, developed according to a previous component-based SBST methodology [8]. The SBST programs of [8] were developed for two processors based on the MIPS ISA, and were easily adapted to the processors used in the current work, which are also based on the same ISA. The SBST routines of [8] achieved very high fault coverage (between 93% and 95.5%) for three different implementations of the MIPS ISA, which are, however, much simpler than the fully pipelined miniMIPS and OpenRISC processors considered in this paper. The functional component self-test routines of [8] target the following functional components of the processor: (a) *computational* components that perform arithmetic or logical operations: ALU, shifter and multiplier, (b) *interconnect* components that guide the data flow, and (c) *storage* components: register file. These test routines *do not* explicitly target any of the performance enhancing components of the pipelined processor, including the pipeline logic.

Example: Figure 1 shows the basic components of miniMIPS annotated with results from fault simulation. The numbers next to each component represent: PGC, the percentage contribution of the component to the total gate count of the processor (upper number) and CFC, the fault coverage achieved for the component by SBST programs that were targeted for functional components (lower number). The shading of the components reflects the test effectiveness of the applied SBST routines (darker colors imply higher fault coverage).

The results show that the SBST programs achieve very high fault coverage for the functional components of miniMIPS under the stuck-at fault model: 95.90% for the ALU and 99.47% for the register file. Although the functional components of miniMIPS are efficiently tested, the overall fault coverage is *only* 86.58%, because functional components form a relatively smaller part of the overall processor, unlike the case of simpler, pipelined processor implementations of [8].

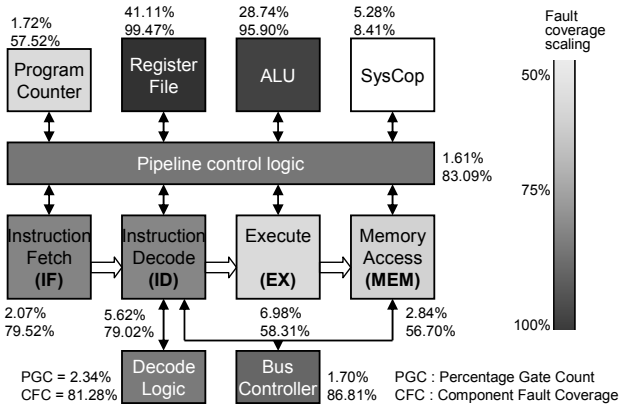

Figure 1. Fault simulation results

The components mainly responsible for this low fault coverage figures are the pipeline logic and the system co-processor. The pipeline logic consists of the pipeline registers, hazard detection logic, and hazard resolution logic including multiplexing stages for the implementation of forwarding and pipeline interlocks. The system co-processor implements exception and interrupt handling mechanisms. In this paper, we provide generic testability enhancement solutions for the pipeline-related logic.

Focusing on the pipeline-related components in all stages, we computed their gate count to be almost 20% of the total gate count of the processor. However, they remain poorly tested—the fault coverage of the entire pipeline logic is less than 72%. The IF and ID stages have a fault coverage of less that 80%, while the coverage of the EX and MEM stages is less than 60%. This severely affects the overall fault coverage, which is only 86.58%. It is obvious that the pipeline logic cannot be tested as a side effect of SBST routines that target the functional components. Enhance-ments are, therefore, necessary to detect pipeline logic faults. ■

An in-depth analysis of the undetected faults of every component of the pipeline reveals the following problems:

☒ **Testability problem 1:** *Logic carrying address-related information through the pipeline stages is poorly tested.*

Table 1 shows the address-related registers, their size in bits and their corresponding task. The Instruction Memory Address (IMA) is carried in all pipeline stages, while the Data Memory Address (DMA) is carried in the case of load/store instructions. The address information stored in the pipeline registers is also used by other components, like the bus controller, the program counter (PC) and the exception unit. Therefore, the testability of these units is also affected negatively. Each of the address-related pipeline registers is coupled with a significant amount of control logic to perform its task, which also has low testability.

Table 1: Address-related logic in the pipeline stages
(IMA: Instruction Memory Address, DMA: Data Memory Address)

Stage	Registers (bits)	Task
IF	IMA (32)	Sends *instruction address* to memory
ID	IMA (32) DM offset (16)	
EX	IMA (32) DMA (32)	Computes *branch target address* Computes *data memory address*
MEM	IMA (32)	Sends *data address* to memory

The controllability issues with address-related logic are as follows. The excitation of the faults of address registers requires the application of vectors that set all address bits to both 0 and 1, while the excitation of the faults of address computation logic coupled with the registers requires the application of a *rich* set of address values that potentially must cover the entire memory space of the processor. The only way to achieve this is to execute instructions and to access (read/write) data that are stored in many different regions of the memory space of the processor.

On the other hand, poor observability of the address-related pipeline components is due to the fact that their faults cannot be directly propagated to primary outputs (address bus) from any pipeline stage but only from the IF stage (where instruction addresses are placed on the address bus) or from the MEM stage (where data addresses are placed on the address bus).

The testability problems of the address-related logic are not only a concern for pipelined processors but also for non-pipelined ones. The problem is *accentuated* in pipelined implementations, because multiple instances of address information flow through the pipeline stages. The testability analysis in the miniMIPS processor showed that *more than* 60% *of the undetected faults of the pipeline structure are related to the address logic*.

☒ **Testability problem 2:** *The hazard detection and resolution logic (forwarding/interlocking) are poorly tested.*

The hazard detection logic and the forwarding paths are not explicitly targeted and thus, the corresponding fault coverage is low. All pipelined implementations of the MIPS ISA need to implement the forwarding paths between the following pairs of stages: EX→EX, MEM→EX, MEM→MEM. This is the case in the two fully pipelined implementations we are considering (miniMIPS and OpenRISC). For this reason, direct application of SBST programs generated for functional components results in a much smaller fault coverage in the pipeline stages EX and MEM (less than 60%) in comparison with stages IF and ID. SBST programs for functional components should, therefore, be *refined* to improve fault coverage by including instruction sequences that: (a) generate all possible resolved/unresolved hazards, (b) activate the stall condition for all pipeline stages, and (c) activate all different forwarding paths towards all pipeline stages.

It bears mentioning that the two identified testability problems are *generic* in the sense that they appear in *any* pipelined implementation of *any* ISA, and so are the solutions that we provide in the following section.

2.2 Proposed solutions

In the following subsections, we present solutions to resolve the two main testability problems of pipelined implementations.

2.2.1 Address-related components

A generic solution for the excitation (controllability) of address-related faults requires the execution of SBST code located in different regions of the *entire* memory space. One of the key test methodology challenges is to facilitate this solution during the course of fault simulation. In other words, placement of SBST code should be done *without* requiring a huge physical memory model of several gigabytes, which will adversely affect fault simulation speed. Such a solution would also ensure that during actual testing, a huge memory is not required in the loadboard.

The key idea behind the proposed solution involves partitioning of a given SBST program into multiple *code segments*. These code segments are *virtually* stored and executed from different memory regions. To demonstrate this, we consider a pipelined processor and a test environment with the following characteristics:

☑ the maximum addressable (virtual) memory space of the processor is M words;

☑ the size of the self-test program is P words;

☑ the size of the physical (actual) instruction memory is T words. It is mapped to address range $[K \ldots K+T-1]$.

The virtual memory space is divided into m different regions of size T, where $m = M / T$. These different memory regions are mapped equivalently to the actual instruction memory. For example virtual addresses $x,\ x+T,\ldots,\ x+iT,\ldots,\ 0 \le i \le m-1$, are mapped to the same physical address $(x+iT) \bmod T + K$. The test program is partitioned into r equal segments of size S, where $S = P / r$. The number of segments must be $2 \le r \le m$. If it is equal to the number of memory regions ($r = m$) then every region will contain one segment. Otherwise (if $r < m$), the segments will be stored in the memory regions $0, m/r-1, 2m/r-1, \ldots$.

Note that each program segment is *not* loaded in the start address of the corresponding memory region (an offset is added to the start address in order to calculate the absolute memory location); the consequence of this is that the program segments are not stored in consecutive memory ranges. This offset must be appropriately selected taking into consideration the instruction cache parameters (size, block placement algorithm) to avoid cache misses.

Example: Suppose the virtual memory space of the processor is 2^{30}=1G words, the size of the self-test program is 2^{14}=16K words, and the size of the actual instruction memory is 2^{24}=16M words (instruction memory is mapped to address range $[0 \ldots 2^{24}-1]$, K=0). Figure 2 depicts the partitioning of the self-test program into segments and its execution from different memory regions. The virtual address space is divided into 64 different regions of 16M words. All virtual memory regions are mapped to the same actual address range $[0 \ldots 2^{24}-1]$. The self-test program is partitioned into 4 equal segments of 4K words each (the number of program segments must be between 2 and 64). The relative addresses of the program segments are translated into absolute addresses and the segments are loaded into different memory regions. Since $r < m$, the segments are virtually loaded in regions 0, 15, 31, 47. ■

For fault simulation purposes, we implement a *memory mapping module* that implements $f(x) = (x \bmod T) + K$ and maps the virtual addresses to the actual addresses of the instruction memory. The module is placed between the processor memory interface and the instruction memory. In the case that we use this method in a fault simulation environment, we can simply implement the behavioral model of this module in HDL. In the case of a loadboard, we can use an FPGA attached to the memory interface of the processor in order to implement this module.

A similar approach is followed with the data memory: the variables of the self-test code (test data and responses) are stored in different virtual memory regions. Parameters that affect the partitioning of the variables are: the number of variables, the size of the actual data memory, and the size of the on-chip data cache.

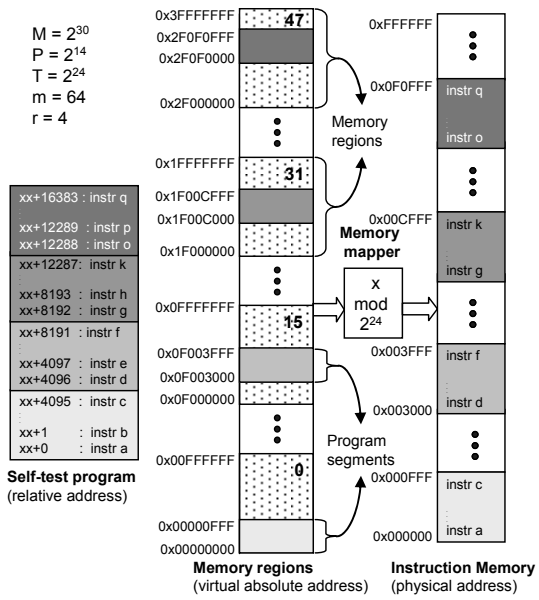

Figure 2. Self-test program segments

We next address the observability problems of address-related faults. Considering that in a manufacturing test environment the address bus of the processor is observable, the address-related faults could be propagated to it. There are two different paths through which they can be propagated to the address bus (Paths 1, 2 in Figure 3a). Path 1 shows propagation of faulty addresses due to faults in the PC and the IF stage. These addresses are propagated to the external address bus through the Bus Controller. Path 2 shows the propagation of faulty addresses due to faults in the ID and EX stages. These faulty addresses are first propagated through the branch calculation process and then again via path 1.

In some scenarios (*e.g.*, on-line testing), response capture is performed using only the data bus. Figure 3b shows two ways of propagating address-related faults towards the data bus. Path 3 shows the use of link instructions. When a jump-and-link or branch on condition and link instruction is executed, the value of the PC is stored in a return address (RA) register. Using a store instruction, the value of the RA register can be propagated to the data bus. Path 3 guarantees the propagation of faulty addresses occurring in the ID and EX stages. Path 4 uses exceptions to capture faulty addresses from EX and MEM stages and propagate them through the system co–processor to the data bus. In case of an exception, the Exception Program Counter (EPC) register of the system co–processor holds the address of the affected instruction. Performing a store instruction from the co–processor achieves propagation of the address stored in the EPC to the data bus.

2.2.2 Hazard detection and forwarding mechanisms

We propose generic solutions to improve the fault coverage of the control logic and the dataflow path of the pipeline structure. The basic concept is to apply test instruction sequences that increase the *activity* of the pipeline components: cause hazards of different types, and activate all forwarding paths multiple times with different data. We propose a method that processes *existing* SBST routines (routines for functional components guarantee the diversity of operands) and creates multiple instantiations of a given routine so as to activate the pipeline logic.

We describe the method for a *hypothetical* pipelined processor with n stages. Consider two instructions `Ia`, `Ib` such that

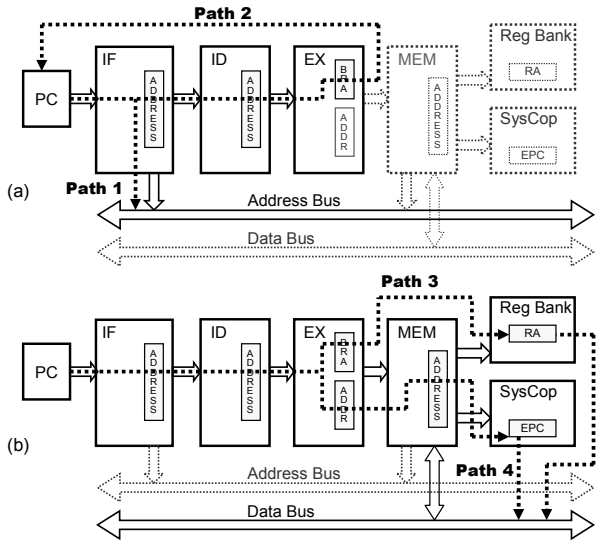

Figure 3. Propagation paths for address-related faults

- `Ia` produces its result at pipeline stage p and stores it at pipeline stage s, $1 \le p \le s \le n$.
- `Ib` reads its input data at pipeline stage d, but actually needs the input data at pipeline stage f, $1 \le d \le f \le n$.

Let us consider the following code:

```
Ia Ra,op1,op2      -- Ra ← Ia(op1,op2)
K1
...
Kc-1
Ib Rb,Ra,op3       -- Rb ← Ib(Ra,op3)
```

There is a data dependency between `Ia` and `Ib`. Between the two instructions, instructions `K1,K2,…,Kc-1` are inserted in the pipeline. Assuming that there is no hazard in the execution of `K1,K2,…,Kc-1`, then `Ib` is inserted in the pipeline c cycles after `Ia` (when $c = 1$, `Ia` and `Ib` are consecutive instructions).

Lemma 1: In the previous code the following rules are valid:

a. If $s - d < c$, no data hazard occurs

b. If $s - d \ge c$, a data hazard occurs

   1. If $p - f \ge c$, the hazard is unresolved and a pipeline stall is inserted
   2. If $p - f < c$, the hazard can be resolved by forwarding

Proof: If $s - d < c$, instruction `Ia` has stored its result before instruction `Ib` reads it, thus no data hazard occurs.

If $s - d \ge c$, `Ib` attempts to read its input data before `Ia` writes it, thus a data hazard does occur. If $p - f \ge c$, `Ia` does not produce its result before `Ib` actually needs it and thus data hazard is unresolved. Otherwise ($p - f < c$), a forwarding path is activated that forwards the intermediate result of instruction `Ia` (that has been produced at cycle p) to the cycle f of instruction `Ib`. ■

In Lemma 1, c is the distance between two instructions with data dependency. Parameter c determines, along with s and d, the number of different cases that should be applied to ensure that all different scenarios are exercised. Based on Lemma 1, we propose a method that processes an assembly routine and generates multiple *variants* – each dealing with a different case: a, b1, b2. We illustrate the method with an example.

Example: Consider the following code snippet for a 5-stage (IF, ID, EX, MEM, WB) pipelined processor:

```
lw   R1,Imm1(R0)
addi R2,R1,Imm2
```

Instruction `lw` produces its result at stage MEM (p=4) and stores it to register `R1` at stage WB (s=5). Instruction `addi` reads register data at stage ID (d=2) but actually executes the addition at stage EX (f=3). The proposed method generates the code variants shown in Figure 4 by inserting `nop` instructions and thus altering the value of c. The code variants fall within different cases of Lemma 1, thereby activating different pipeline mechanisms and for-warding paths. For example, although variants b and c fall within the same case of Lemma 1 (b2), they activate different forwarding paths: MEM→EX, MEM→ID, respectively. If the distance c in the original code fragment is 2 or greater, the method cannot generate a code variant that deals with case b1 of Lemma 1. ■

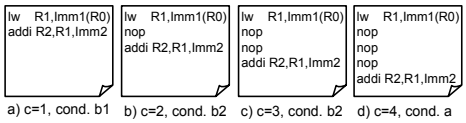

Figure 4.Test code variants to activate hazards and forwarding

3. SBST METHODOLOGY

Figure 5 presents a systematic SBST methodology that incorporates the solutions presented in Section 2.2. The methodology takes as its input existing SBST programs (which may only target functional components) and various parameters of the pipelined processor. The output of the methodology is an enhanced set of SBST programs that can achieve high fault coverage for the pipeline logic. The methodology consists of two phases: Phase 1 applies the solution for the hazard detection and forwarding mechanisms, and Phase 2 applies the solution for the address-related components.

Phase 1 processes the input SBST programs to identify *def-use* pairs, which represent all possible dependencies between the instructions (def-use analysis is used for optimization in compilers; a variable's value is *defined* when an assignment is made to it and is *used* when it appears on the right side of an assignment.) For every def-use pair, a set of code variants is constructed. In order to generate code variants, some information about the pipeline architecture is required: (i) the number of pipeline stages, (ii) for each instruction, the stages that reads, processes, produces and writes back data (parameters used in Lemma 1), and (iii) the implemented forwarding paths.

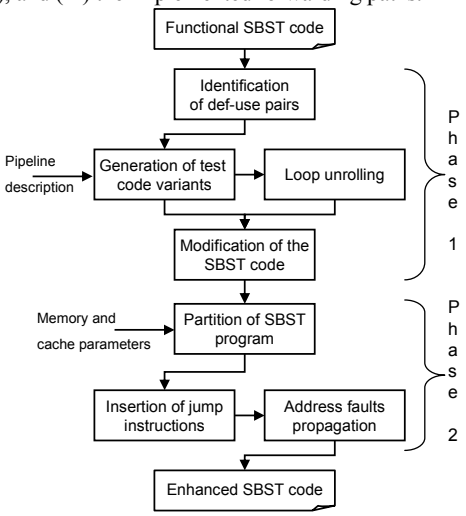

Figure 5. Proposed SBST methodology

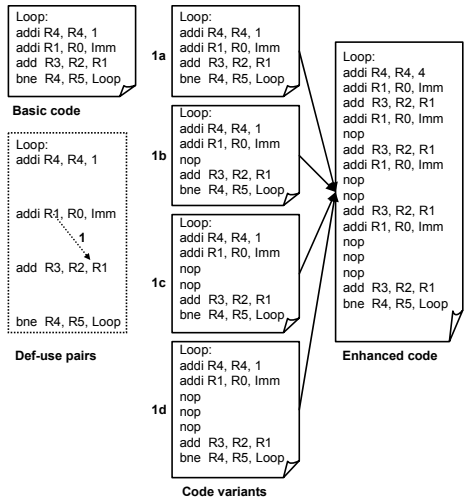

Figure 6. Generating and combining code variants

For SBST programs that have a data dependency within a loop, it may be possible to use a combination of "loop unrolling" and "loop fusion" to effectively combine all code variants. The example shown in Figure 6 depicts a loop with one data dependency, four different code variants and their fused version.

For general data dependencies, phase 1 scans the code, and generates the corresponding code variants for each def-use pair. Then, it selects one among the code variants and substitutes the original instructions with this. The selection algorithm for the code variants maintains a history table that stores which condition of Lemma 1 is activated each time. The next time, the algorithm selects a code variant that activates a condition that was least recently used in the history table. An example code snippet is shown in Figure 7, which includes three def-use pairs. For each case, the code variants are constructed. The algorithm selects the code variants 1a, 2b, and 3c to build the enhanced code.

Phase 2 partitions the SBST programs generated by phase 1 into segments. For code partitioning, memory parameters of the processor and the test environment are required: these include size of the virtual memory, size of the test program, and size of the physical instruction and data memory. After partitioning, the relative addresses of the assembly code are translated into absolute virtual addresses and the program segments are loaded in the corresponding addresses of the physical instruction memory. Next, the required jump instructions are inserted at the end of each segment in order to transfer the program execution to the next segment. Finally, phase 2 inserts the appropriate instruction sequences that guarantee the propagation of address-related faults through propagation paths 2, 3, and 4 (Figure 3). Path 2 is necessary in order to ensure the propagation of address-related faults that can not be observed through path 1 (i.e. address faults in EX stage). Paths 3 and 4 are necessary when responses can be captured *only* through the data bus, for example in on-line testing.

The proposed SBST methodology does not require a detailed (e.g., RTL or gate-level) processor model, since it is based on simple parameters of the pipeline and the memory subsystem. In the case that the required information can not be automatically extracted from a high-level description of the processor, *e.g.*, an instruction set description language (ISDL) model, it could be manually specified. Seen in this context, the proposed SBST methodology is largely automatable since the steps of both phases can be easily implemented in an automated test flow.

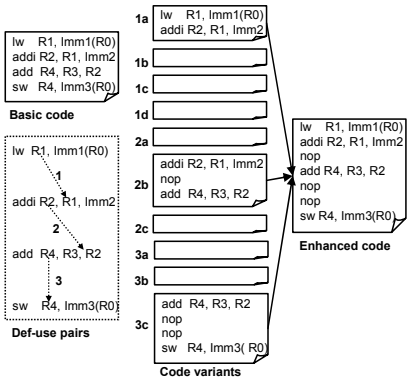

Figure 7. Selection of code variants

4. EXPERIMENTAL RESULTS

We demonstrate the applicability of the proposed methodology to two pipelined RISC processors, miniMIPS [14] and OpenRISC 1200 [15]. The processors were synthesized using a 0.18um technology library: miniMIPS runs at 100MHz and contains 32,817 gates, while OpenRISC 1200 runs at 102MHz and contains 35,657 gates. Table 2 presents statistics for the original SBST programs [8], and the enhanced SBST programs that have been generated by the proposed methodology. The results show that the proposed methodology incurs little overhead in both test program size and execution times.

Note that the difference in test execution time between the two processors is because miniMIPS does not include caches and assumes an "ideal" memory subsystem (reads and writes to/from memory last one cycle), while OpenRISC implements instruction and data caches and thus compulsory cache misses occur (even for cache hits, two cycles are required to load data from cache).

Table 3 shows the fault coverage for every component of miniMIPS. The enhanced test programs achieve a stuck-at fault coverage of more than 86% for the processor's pipeline components, which is an improvement of 22% over the original scenario. The co-processor must be addressed in an SoC-specific manner, and higher testability can be achieved. The enhanced test routines also increase the fault coverage of other components, especially those with address-related problems, like PC and Bus Controller. This improvement shows that the proposed solution for address-related faults can also benefit non-pipelined processors.

Table 2: SBST program statistics

Processor	Original SBST Progs.		Enhanced SBST Progs.	
	Size (words)	Time (cycles)	Size (words)	Time (cycles)
miniMIPS	1,177	5,990	1,565	7,162
OpenRISC	2,172	44,271	3,116	56,716

Table 3: Fault coverage for various components in miniMIPS

Component	Original FC %	Enhanced FC%
ALU	95.90	97.85
Register Bank	99.47	99.98
PC	57.52	86.32
Decode Logic	81.28	83.53
Bus Controller	86.81	93.95
SysCop	8.41	87.90
IF Stage	79.52	90.86
ID Stage	79.02	90.24
EX Stage	58.31	84.12
MEM Stage	56.70	81.87
Pipeline Control	83.09	93.64
Pipeline Total	**71.40**	**86.60**
Processor Total	**86.58**	**95.08**

Table 4 presents the total stuck-at fault coverage for the two pipelined processors. We give results for two different versions of the miniMIPS: (A) refers to the original design, which includes a fast parallel multiplier, while (B) does not include the multiplier. Since the fast parallel multiplier occupies a significant part of the processor (about 20%), removing the multiplier causes the impact of pipeline components to be more significant in (B). This experiment allowed us to quantify the benefits of testing the pipeline logic in processors that do not feature hardware multiplier units. The experimental results show that the enhanced SBST programs can achieve a total fault coverage of around 93% for the pipelined processors. Fault coverage improvements upto 15% (12% on an average) are achieved for the entire processor.

Table 4: Fault simulation results for the pipelined processors
(Improvement = (Enhanced FC – Original FC)/Original FC)

Processor	Original FC %	Enhanced FC %	Improvement (%)
miniMIPS (A)	86.58	95.08	9.81
miniMIPS (B)	81.51	94.00	15.32
OpenRISC	80.36	90.03	12.03
Average	**82.81**	**93.03**	**12.34**

5. CONCLUSIONS

In this paper, we described an SBST methodology that enhances existing SBST programs for functional components of a processor so as to target the pipeline logic. The proposed solutions are applicable to any pipelined processor and can exploit existing SBST programs. We applied the proposed methodology to two pipelined RISC processors, and were able to achieve significant fault coverage improvements.

6. REFERENCES

[1] G.Hetherington, T.Fryars, N.Tamarapalli, M.Kassab, A.Hassan, J.Rajski, "Logic BIST for Large Industrial Designs: Real Issues and Case Studies", *IEEE Intl. Test Conf.*, pp. 358 – 367, 1999.

[2] J.Shen, J.Abraham, "Native Mode Functional Test Generation for Microprocessors with Applications to Self-Test and Design Validation", *IEEE Intl. Test Conf.*, pp. 990 – 999, 1998.

[3] K.Batcher, C.Papachristou, "Instruction randomization self test for processor cores", *IEEE VLSI Test Symp.*, pp. 34 – 40, 1999.

[4] P.Parvathala, K.Maneparambil, W.Lindsay,"FRITS–A Microprocessor Functional BIST Method", *IEEE Intl. Test Conf.*, pp. 590 – 598, 2002.

[5] F.Corno, M.Sonza Reorda, G.Squillero, M.Violante, "On the Test of Microprocessor IP Cores", *Design Automation & Test in Europe*, pp.209 – 213, 2001.

[6] L.Chen, S.Dey, "Software-Based Self-Testing Methodology for Processor Cores", *IEEE Transactions on CAD*, vo.20, no.3, pp. 369 –380, March 2001.

[7] L.Chen, S.Ravi, A.Raghunathan, S.Dey, "A Scalable Software-Based Self-Test Methodology for Programmable Processors", *Design Automation Conference,* pp. 548 – 553, 2003.

[8] N.Kranitis, A.Paschalis, D.Gizopoulos, G.Xenoulis, "Software-based self-testing of embedded processors", *IEEE Transactions on Computers*, Volume 54, Issue 4, pp. 461 – 475, April 2005.

[9] C.Wen, L.-C.Wang, K.-T.Cheng, W.-T.Liu, and C.-C.Chen, "On a software-based self-test methodology and its application," *IEEE VLSI Test Symp.*, pp. 107 – 113, 2005.

[10] S.Gurumurthy, S.Vasudevan and J.Abraham, "Automated Mapping of Pre-Computed Module-Level Test Sequences to Processor Instructions", *IEEE Intl. Test Conf.*, 2005.

[11] V.Singh, M.Inoue, K.K.Saluja, and H.Fujiwara, "Instruction-based delay fault self-testing of pipelined processor cores," *IEEE Intl. Symp. on Circuits and Systems*, pp. 5686 – 5689, 2005.

[12] M.Hatzimihail, M.Psarakis, G.Xenoulis, D.Gizopoulos, A.Paschalis, "Software-Based Self-Test for Pipelined Processors: A Case Study", *IEEE Intl. Symp. on Defect and Fault Tolerance in VLSI Systems*, pp. 535 – 543, 2005.

[13] D.Gizopoulos, A.Paschalis, Y.Zorian, *Embedded Processor-Based Self-Test*, Kluwer Academic Publishers, 2004.

[14] miniMIPS CPU www.opencores.org/projects/minimips

[15] OpenRISC1200 www.opencores.org/projects/web/or1k/openrisc_1200

[16] D.A.Patterson, J.L.Hennessy, *Computer Organization and Design, The Hardware/Software Interface*, 3rd Edition, Elsevier, 2005.