A Self-Repairing Execution Unit for Microprogrammed Processors

THIS PROCESSOR DYNAMICALLY RECONFIGURES ITS INTERNAL MICROCODE TO EXECUTE EACH INSTRUCTION USING ONLY FAULT-FREE BLOCKS FROM THE EXECUTION UNIT. WORKING WITHOUT REDUNDANT OR SPARE COMPUTATIONAL BLOCKS, THIS SELF-REPAIR APPROACH PERMITS A GRACEFUL PERFORMANCE DEGRADATION.

Alfredo Benso
Silvia Chiusano
Paolo Prinetto
Politecnico di Torino, Italy

The emerging field of self-repair computing could have a major impact on deployable systems for space missions and defense applications. These systems must survive and perform at optimal functionality for long durations in unknown, harsh, and/or changing environments. Examples of such applications include outer solar system exploration, missions to comets and planets with severe environmental conditions, long-lasting space-borne surveillance platforms, defense systems, and monitoring and control of long-term nuclear waste and other hazardous environments. Self-repair computing could also greatly enrich commercial applications that require high availability and serviceability. These applications could range from biomedical devices to automotive applications.

The proposed self-repair architecture for microprogrammed processors is transparent to the user and tolerates the occurrence of multiple faults in the device's functional units. The processor achieves self-repair by letting the device dynamically reconfigure its internal microcode to execute required computations using only fault-free system units. One of the main novelties of our approach is that it does not require adding redundant or spare computational blocks to the system. The approach introduces a graceful degradation of device performance, but nevertheless lets the processor complete the requested operations even when multiple faults are present in its functional units.

Researchers have done little work in the field of self-repair computing. Most studies focus on field-programmable gate arrays.[1-4]

Microprogrammed target architecture

Our approach's target device is a vertical microprogrammed architecture that executes single-instruction, single-data (SISD) instructions. Since space applications generally use well-known and well-tested microprocessors that are usually one to three generations behind the most advanced research, we chose a very simple architecture to easily demonstrate our approach's applicability and effectiveness. Nevertheless, the same self-repair strategy can be implemented in more complex microprogrammed architectures, such as those with pipelines, branch prediction units, or speculative execution.

A microprogrammed processor basically

0272-1732/01/$10.00 © 2001 IEEE

consists of two main units—a control unit and a data path, as shown in Figure 1. These units execute the user's program, which is described at the assembly level (with macroinstructions) and usually stored in RAM located outside the target architecture. A decode unit decodes each macroinstruction. The decode unit provides the control unit with the address of the microroutine that when executed will complete the requested operation.

The control unit executes the required microinstructions by driving the correct enable signals to the data path. The control unit includes a ROM that stores the microcode, a microinstruction pointer, and a sequencer. The microinstruction pointer indicates the current microinstruction stored in ROM, whereas the sequencer computes the next microinstruction's address. The sequencer usually includes a microinstruction pointer stack, which permits microsubroutine execution. ROM stores all the microinstructions that constitute the sequence of operations to implement the device's macroinstructions (assembly level instructions).

The data path contains all blocks used to store and manipulate data, including two main blocks: the register array and the execution unit. The register array consists of all registers—status, user, and temporary, for example—that can be referred to in either the macro- or microcode. It is obviously possible to read or write the content of all registers.

The execution unit acts as the microprocessor's core because it is in charge of manipulating data. It includes all the functional blocks required during the computations, from very simple logical operations (AND, OR, and XOR) to basic arithmetic operations (addition, subtraction, and shift) and more complex ones (multiplication and so on).

Self-repair candidates

The best candidate units for implementing a self-repair architecture should be those most subject to faults. Assuming that a unit's criticality is proportional to its area, the most critical units in the proposed architecture are the register array (51.29 percent of the entire chip area) and the execution unit (31.32 percent). We obtained these area values from a VHSIC hardware description language model of the microprogrammed architecture synthesized using Synopsys tools.[5]

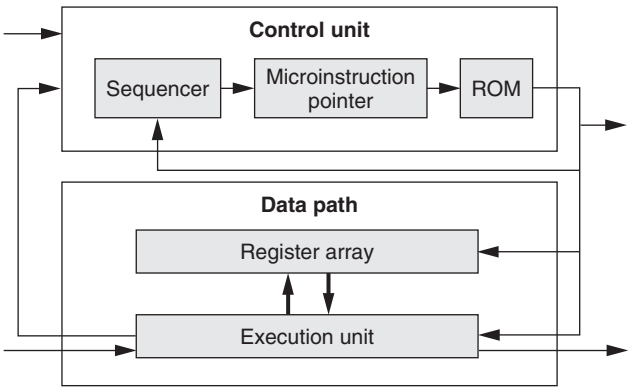

Figure 1. Basic microprogrammed architecture.

We will focus on a self-repair approach for the execution unit because the literature already includes several techniques for designing a fault-tolerant register array.[6,7] For example, these techniques include

- *Specialized data coding.* In these types of techniques, data stored into the microprocessor register array is coded using parity, Hamming, or more complex codes. Doing so permits easy detection and possibly correction of permanent or transient faults.
- *Dynamic register allocation.* These techniques dynamically reconfigure the registers addressing space to exclude faulty registers from the set of available ones.

Self-repairing execution unit

The main idea of our proposed approach is to design the control unit and data path to guarantee execution of all microinstructions, even in the presence of faulty functional blocks in the execution unit. In our approach, we use dedicated built-in self-test (BIST) architectures to provide an online status—either good or faulty—for each block in the execution unit. For each microinstruction, we define an alternative sequence of microinstructions that can execute the same operation using only fault-free units. We implemented the idea by adding a replace unit to the basic architecture, as Figure 2 (next page) shows. This unit modifies the microinstruction execution flow to reallocate functional units on the fly to avoid using faulty modules.

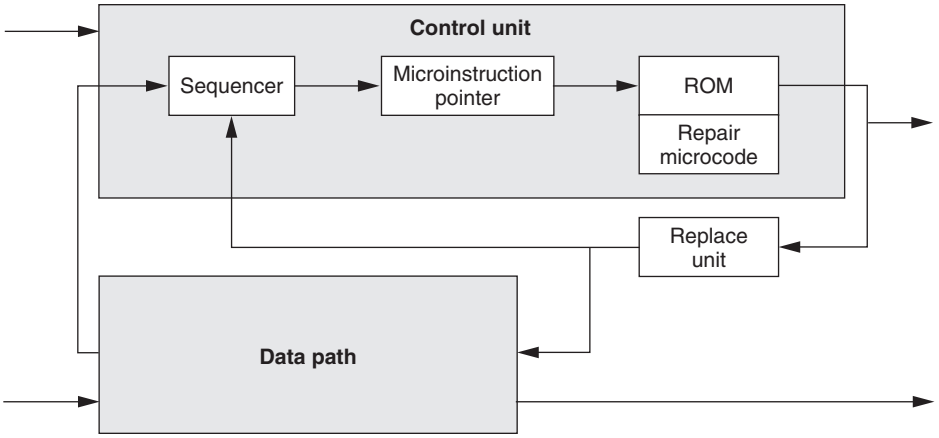

Figure 2. Basic microprogrammed architecture with replace unit.

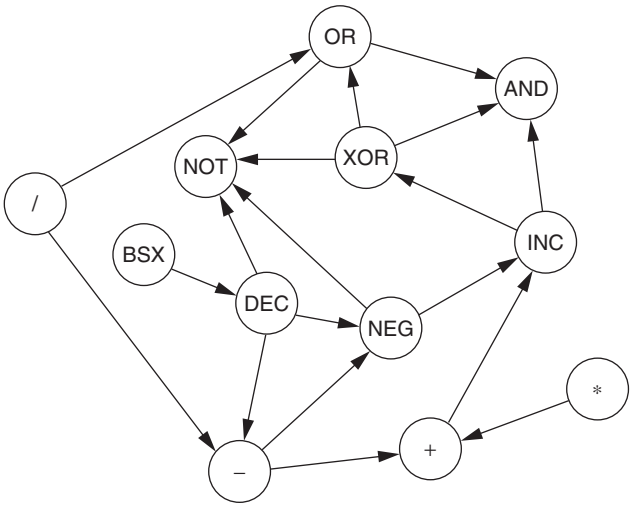

Figure 3. Replacement graph.

If the unit involved in a microinstruction's execution is faulty, the replace unit replaces the execution flow of the original microcode with the repair routine for that instruction, which is available in the repair microcode.

In particular, the replace unit substitutes the faulty microinstruction with a no-op and forces the control unit to jump to the proper repair routine. The repair microcode is therefore a set of repair routines that specify alternative execution paths for a given operation. That is, the repair routine of instruction v is a piece of code able to execute v using alternative functional units.

For example, if a multiply instruction has to be executed and no multiply units are available, the replace unit replaces the multiply operation with a repair routine that implements the multiplication as a set of addition and shift operations. The resulting microcode execution will cause a graceful degradation in system performance, but will nevertheless let the device output a correct result.

One main goal of this approach is to tolerate multiple faults in the execution unit. To address this issue, the replace unit and the repair microcode allow nested replacements of faulty microinstructions. For example, suppose that the multiplier is faulty and therefore the replace unit executes the multiplication using addition and shift operations. If the adder is faulty as well, the replace unit will replace its functionality by increment and shift operations. If the increment operation is also faulty, the replace unit will specify execution of the multiplication using EXOR, AND, OR, and shift operations, and so on.

To formalize the self-repair capability provided by the repair microcode, we defined a replacement graph. In Figure 3, each node is a microinstruction. Node v's successors are the alternative modules (or microinstructions) used to execute instruction v. To avoid deadlock, the replacement graph must be a direct acyclic graph (DAG).

To improve performance, the terminal nodes of the graph must be the simplest microinstructions—the ones that use the simplest units. This consideration stems from the following reasons:

- The simplest units are also the smallest ones, which therefore have the lowest probability of being affected by faults.

- It is possible to design fault-tolerant simple units (for example, triple-module-redundancy units) without introducing a significant area overhead. In this way, the processor can tolerate and repair a very high number of faults affecting the most complex units.

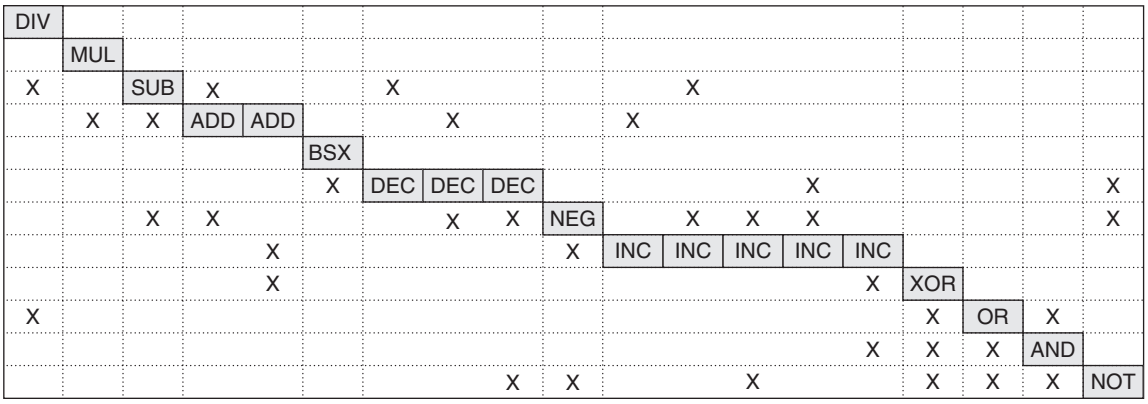

Figure 4. Enhanced repair microcode. To work around a faulty module for the microinstruction in the gray boxes, substitute the instructions indicated by the X's in the same column. If a microinstruction appears more than once, it means our approach offers more than one alternative. The complexity of the functional units involved decreases as you move to the right in this figure.

To improve the self-repair capabilities of our approach, we implemented an enhanced version of the repair microcode. This version specifies several repair routines for each faulty microinstruction, each routine using progressively simpler functional units. This enhanced approach sorts repair routines by increasing performance degradation (and, in general, by decreasing complexity of units allocated to functionally replace the operation). Using this solution, increasingly less complex units will replace a faulty unit, allowing, in general, execution of the microinstruction with replacement units that introduce the lowest possible execution time overhead.

Figure 4 shows alternative units used to replace every microinstruction. Each column corresponds to a repair solution for the microinstruction reported in the corresponding shaded cell. For example, the DEC microinstruction has three possible repair solutions: the first using the SUB microinstruction, the second using ADD and NEG (two's complement), and the third using NEG and NOT (one's complement).

Our experimental results show that this solution considerably reduces the performance degradation introduced by the replace microcode.

Testing the execution unit

Our self-repair architecture assumes that the replace unit always knows the state (good or faulty) of the execution unit's functional modules. We are currently evaluating four possible online-BIST approaches: cyclic based, incremental, arithmetic, and functional. Each provides runtime diagnostic information about the execution unit's status. We qualitatively evaluate each of these four alternatives in terms of area overhead, fault detection capabilities, and fault latency.

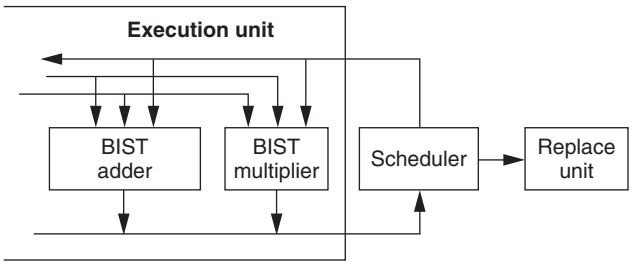

Figure 5. Block diagram of execution unit that incorporates cyclic-based BIST.

Cyclic-based test

In this cyclic-based approach, shown in Figure 5, the microprocessor incorporates dedicated BIST logic around each execution unit module to exhaustively test the module's functionality. A scheduler cyclically enables the BIST procedure for each module. During the test, the BIST procedure always marks the module under test as faulty. So if an executing microinstruction requires this module, the execution unit will instead execute the repair microcode without interrupting the processor's normal activity.

This solution obviously implies some performance degradation because the execution

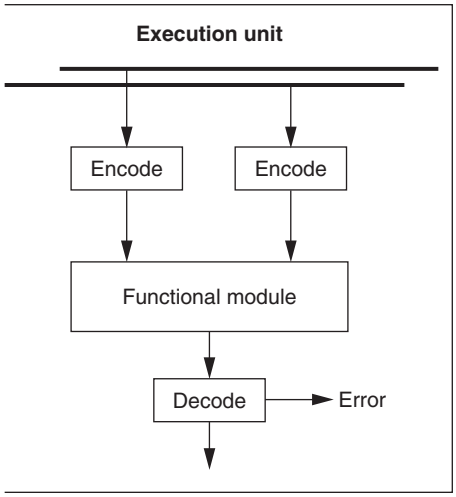

Figure 6. Block diagram of an execution unit that incorporates arithmetic-code-based BIST.

unit is always running repair microcode to replace the module under test. A possible alternative is to periodically execute the execution unit's BIST, activating it by inserting a special macroinstruction into the application code. This scheme does not degrade performance, but it does periodically interrupt execution of the user application for testing. In this case, the fault detection latency is proportional to the period of time chosen between the activation of two consecutive test sessions.

Incremental test

Because most execution unit modules are combinational logic, another solution is to perform a structural pseudorandom test. We therefore defined a BIST architecture based on linear-feedback shift registers to generate pseudorandom test patterns. The architecture also relies on multiple-input shift registers to observe and compact the module responses.

This technique exploits the fact that microinstructions do not use all the execution unit modules concurrently—there are always some unused modules. Thus, at every clock cycle, it is possible to generate and apply a new test pattern to all unused modules. In this case, fault latency is not easily predictable, because it depends on the resource allocation required by the user application.

Arithmetic test

This solution, shown in Figure 6, exploits arithmetic codes to test the execution unit's functional modules. A particular characteristic of arithmetic codes is that they are invariant with respect to certain arithmetic operations. An encoder and a decoder are used to encode the operands before a module processes them and to decode the result to verify its correctness. In this solution, the fault latency equals the time elapsed between the fault occurrence and the successive activation of the faulty module.

Functional test

This solution implements a functional test of the execution unit as a user program (that is, as a set of macroinstructions) that the user periodically executes, for example, at system start up. The test is based on a starting-small approach: It verifies functionality of the simplest modules (registers, for example) first and then tests the most complex modules, exploiting the functionality of the already verified units. Since it is a software-based functional test, this approach does not introduce any hardware overhead. The fault detection latency depends on the frequency of test activation.

Tool

To evaluate our approach's effectiveness, we implemented the Micro Repair Simulator (Mires) tool that allows

- validation and simulation of a C++ model of the self-repair architecture,
- fast prototyping of new repair routines, and
- fault injection of permanent faults into the model to evaluate self-repair capabilities and performance degradation.

The simulator loads the compiled microcode, repair microcode, and user application macrocode. We also implemented a specialized compiler for both the micro- and macrocode. Next, the simulator allows step-by-step execution of both micro- and macrocode, continuously monitoring all processor resources.

The injector simulates the fault occurrence in one of the execution unit modules. It does not inject a real fault, but simulates a faulty-unit notification from the BIST logic. In this way, it is possible to simulate the repair

process' behavior for each fault combination. It is also possible to execute a complete fault injection experiment that automatically emulates all possible combinations of faulty units that might appear in the execution unit.

Experimental results

To compute the processor's performance degradation, we executed a very simple program with only one macroinstruction: (`-10 *0 -1`). We chose a multiply operator because, when faulty, it allows the activation of several repair routines. We executed the algorithm several times, each time injecting a different combination and number of faulty units. Figure 7 shows the results for both the basic and enhanced repair micro-code. The *x*-axis shows the number of injected faults, whereas the *y*-axis shows the average ratio between the execution times for the fault-free and repaired execution units.

We computed the area overhead necessary for synthesizing a VHDL model of the proposed architecture using Synopsys tools. Table 1 shows the execution unit, ROM, the replace unit and give for each, their initial area and final areas. Final areas result from applying the basic and enhanced versions of our proposed approach.

Experimental results reported in Table 1 show that the area overhead introduced by the replace unit and the repair microcode is very low for both the basic (6.43 percent) and enhanced (6.64 percent) version of our approach.

The results achieved in this research are very promising. We believe that low-cost techniques such as the one we propose here will soon become of interest in commercial applications, where consumers demand increasingly higher levels of reliability and serviceability We are currently focusing on self-repair techniques that target commercial microprocessor cores, such as ARM cores. We are studying the implementation of mixed hardware and software techniques that guarantee integrity of both data and execution flow in cases of permanent or transient faults. MICRO

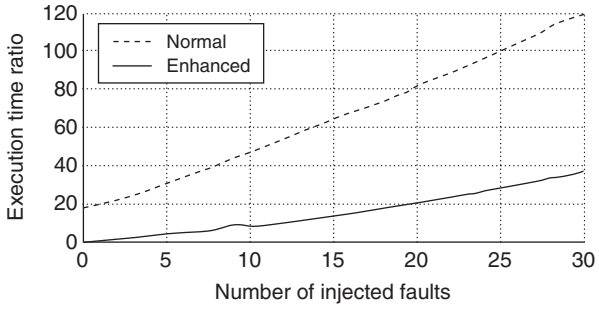

Figure 7 Performance degradation.

Table 1. Area overhead.

Approach	Size of replace unit (no. of gates)	Size of ROM (no. of gates)	Size of execution unit (no. of gates)
No self-repair capabilities	0	6,488	125,485
Basic built-in self-repair	1,222	11,455	127,779
Enhanced built-in self-repair	1,485	11,473	127,779

Acknowledgment

This work was partially supported by Istituto Superiore Mario Boella under the TestDOC: Quality and Reliability of Complex System-on-Chip project.

References

1. W. Mangione-Smith and B. Hutchings, "Configurable Computing: The Road Ahead," *Proc. Reconfigurable Architectures Workshop*, IT Press, Chicago, 1997, pp. 81-96.
2. J. Lach, W. Mangione-Smith, and M. Potkonjak, "Efficiently Supporting Fault-Tolerance in FPGAs," *Proc. ACM/SIGDA 6th Int'l Symp. Field-Programmable Gate Arrays*, ACM Press, New York, 1998, pp. 105-115.
3. M.J. Wirthlin and B. L. Hutchings, "A Dynamic Instruction Set Computer," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 99-107.
4. R. Bittner and P. Athanas, "Wormhole Run-Time Reconfiguration," *Proc. ACM/SIGDA Int'l Symp. Field Programmable Gate Arrays*, ACM Press, New York, 1997, pp. 79-85.
5. *VHDL Compiler Reference Manual*, Synopsys, Mountain View, Calif., 1994.
6. F.J. Macwilliams and N.J.A. Sloane, *The Theory of Error-Correcting Codes II*, vol. 16, North-Holland Mathematical Library, Amsterdam, 1998.

7. M. Nicolaidis and Y. Zorian, "Online Testing for VLSI—A Compendium of Approaches," *J. Electronic Testing, Theory and Applications (JETTA)*, vol. 2, nos. 1/2, Feb.-Apr., Kluwer Academic Publishers, Boston, 1998, pp. 7-20.

Alfredo Benso is a research assistant at Politecnico di Torino, Italy. His research interests include design-for-testability techniques, dependability analysis, and software-implemented hardware fault tolerance of computer-based systems. Benso received a PhD in computer engineering from the Politecnico di Torino. He is the chair of the IEEE Computer Society Test Technology Technical Council Web-based activities group.

Silvia Chiusano is a research assistant at Politecnico di Torino, Italy. Her research interests include high-level testing, design-for-testability techniques, BIST, and dependability. Chiusano received a PhD in computer engineering from Politecnico di Torino.

Paolo Prinetto is a full professor of computer engineering at Politecnico di Torino, Italy, and a joint professor at the University of Illinois at Chicago. His research interests include testing, test generation, BIST, and dependability. Prinetto received an MS in electronic engineering from Politecnico di Torino. He is a Golden Core Member of the IEEE Computer Society and the elected chair of the IEEE Computer Society Test Technology Technical Council.

Direct questions or comments about this article to Alfredo Benso, Politecnico di Torino, Corso Duca degli Abruzzi 24, 10129 Turin, Italy; benso@polito.it.