DCIS 2006

# Real Time Fault Injection Using On Chip Debug Infrastructures – A Case Study

André Fidalgo, Manuel Gericota, Gustavo Alves
Department of Electrical Engineering – ISEP
Rua Prof. Bernardino de Almeida 431
P-4200 Porto, PORTUGAL
+351.22.8340500
{anf, mgg, gca}@isep.ipp.pt

José Ferreira
Department of Electrical Engineering – FEUP
Rua Dr. Roberto Frias, s/n
P-4200-465 Porto, PORTUGAL
+351.22. 508 14 00
jmf@fe.up.pt

*Abstract*— **As electronic devices get smaller and more complex, dependability assurance is becoming fundamental for many mission critical computer based systems. This paper presents a case study on the possibility of using the on-chip debug infrastructures present in most current microprocessors to execute real time fault injection campaigns. The proposed methodology is based on a debugger customized for fault injection and consists of injecting bit-flip type faults on memory elements without modifying or halting the target application. Three different configurations are compared in terms of performance, area overhead and communication bus width. The basic debugger design is easily portable and applicable to different architectures, providing a flexible and efficient mechanism for verifying and validating fault tolerant components.**

Index Terms – **Computer Fault Tolerance, Fault Injection, On Chip Debug, Reliability**

## I. Introduction

Today, most safety-critical applications require the use of some type of computer-based device, causing their implantation to grow and expand into new areas like the automotive and biomedical fields. However, as electronic systems increase in complexity and decrease in size their correct operating behavior is becoming harder to guarantee [1]. Circuits are getting more sensitive to noise and to other factors, with the appearance of soft errors becoming a real possibility even for devices used in non-hostile environments, making dependability a necessity for a much broader area of applications. Dependable systems are designed to handle errors that originate from software or hardware faults and to recover from them, while maintaining acceptable operating conditions. The possibly destructive nature of a failure and the long error latencies impair identifying the cause of failures in field operation and in the normal time that it takes for a failure to occur. To identify and understand potential errors, it is desirable to experiment on an actual device as to better study and improve its dependability. This approach can be applied either on the development phase, where models or prototypes are used, or on the deployment phase, if faults can be deliberately injected in useful time without damaging the equipment. This experiment-based approach requires knowledge of the system architecture and behavior, and especially of the mechanisms implemented to provide tolerance to faults, errors or failures, i.e. the events leading to a service failure on microprocessor based systems [2]. Specific instruments and tools must be used to induce these hazards and monitor their effects and in the case of microprocessor systems, access to the internal resources is of utmost importance. Many of today's microprocessors provide such access through dedicated built-in debug circuitry, often designated as on-chip debug (OCD). The use of these OCD infrastructures for fault injection purposes is an efficient solution for verifying and validating fault tolerant designs. This paper describes recent research on real time fault injection targeting such devices (i.e. without halting application execution), based on the development and use of a debugger optimized for fault injection. The rest of the paper is organized as follows: the next section gives an overview of fault injection methodologies used on microprocessor systems and previous work on this area; section 3 presents the system used as a target, the fault injection oriented debugger and some proposals for enhanced fault injection support; section 4 presents the experimental results obtained so far and finally section 5 discusses these results and lays the basis for future work.

## II. Fault Injection On Microprocessors

### A. Overview

In microprocessor systems, the most common methodology to achieve dependability is the use of fault-tolerant components, both in hardware and software. The correct behavior of such components must be tested and fault injection can be used to (1) identify design or implementation faults, (2) verify & validate fault tolerance capabilities and (3) estimate how often failures will occur and evaluate the consequences of such failures.

Fault injection is normally structured in campaigns, each being composed of a series of experiments during which the target system runs (a specific application is executed) and a specific fault (or set of faults) is inserted at specific trigger conditions. The target system behavior is monitored and information is recorded as comprehensively as necessary and possible, to later understand and evaluate the effects of the inserted fault(s).

Existent microprocessor fault injection techniques are commonly classified in three broad groups, namely (1) simulation based fault injection, (2) software based fault injection (SWIFI), and (3) physical fault injection.

Simulation based fault injection is mostly used in the early phases of a design when the target system exists only in model format. This technique requires a model of the target itself, the necessary simulation tools to insert faults and adequate processing capabilities to run the simulation [3].

Software based fault injection consists of reproducing at a logical level the errors originated by physical faults using software commands already available on the target device. This allows the injection of errors on all resources accessible by software, like registers, program and data memory, most peripherals and some timers [4]. Physical fault injection is a more realistic approach in the sense that it tries to replicate real world faults. All physical techniques perform an actual fault insertion on the circuit or emulate their immediate consequences (errors) through internal or external action. Access to the circuit elements is usually performed either through specific hardware equipment [5] or using debug and test infrastructures included on the target chip [6]. Physical fault injection may also be performed without a direct connection between the fault injector and the system under test, either through laser [7], heavy-ion radiation or electromagnetic fields [8].

The hardest part of microprocessor fault injection is how to access those internal elements where faults are more probable, generally the memory elements and communication buses, without disturbing the running applications. OCD infrastructures provide access to internal resources in parallel with the target hardware and running software, being an excellent mechanism for modifying register and / or memory values (i.e. insert faults) and subsequently retrieve the data necessary for result analysis.

The OCD facilities implemented by different families of processors share some common characteristics that form a core feature set, which usually includes run-control, breakpoint support and memory and register access. Some devices include more advanced features like watchpoints, program trace and real time debugging capabilities. In general, an OCD is a combination of hardware and software on the microprocessor chip that requires some external hardware to be used, the basic requirement being some kind of communication link between the chip and the host machine. The access to the OCD infrastructure is made through an interface port usually requiring an external debugger in between.

The use of OCD infrastructures for fault injection can overcome some of the limitations present on other approaches. For instance, simulation techniques are often time-consuming and may lead to erroneous results as they are intrinsically dependant on the quality of the available model. SWIFI techniques require modifications to the running code, which in fact modifies the target system, and coverage is limited to the resources accessible by software. Most physical fault injection techniques are expensive and precise control of the instant and location of a fault is often very difficult or even impossible. In most cases, OCD fault injection techniques rely on halting the processor, either by the use of control signals or breakpoints, and subsequently modifying the targeted registers or memory locations to insert the intended faults. When available, trace capabilities provide an efficient mean to monitor fault propagation and effects.

### *B. Previous Work*

As a technological solution, a major problem with OCD is the lack of a consistent set of capabilities and a standard communications interface across processor architectures. An industry consortium has been working on the establishment of a standard for OCD, which is still on a proposal phase and is formally designated as “IEEE-ISTO 5001, The NEXUS 5001 Forum Standard for a Global Embedded Processor Debug Interface” [9]. If widely adopted, it may be possible to employ the same debugger to access the core of multiple processor architectures and to use a similar set of debugging features for all. Additionally, the feature set that this standard proposes for the higher classes of compliance provides a useful set of tools for real time fault injection in the form real time access to memory and on-the-fly program and data trace.

Experimental work has been done in our research group and in the DISCA-UPV [10] to evaluate the possibilities of executing real-time fault injection on a NEXUS compliant microprocessor. The target systems used were based on a Motorola MPC565 CPU [11], which is a commercial 32 bit microcontroller with widespread use on the automotive industry. In our case, the debugger used was an iSystems IC3000 [12] (iTracePro version) and its integrated debugging software Winidea 2005. This software allows direct control of the debugger and the use of scripts (running on the host machine) to automate the debugging tasks.

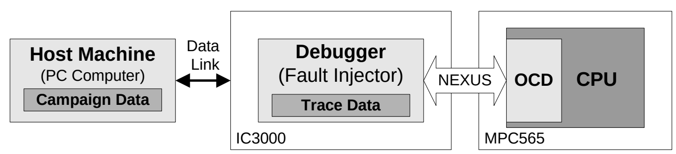


Fig. 1. Fault Injection Environment (MPC565)

The fault campaigns were manually generated and translated into Winidea scripts. A typical fault injection operation would require the microprocessor to run until the triggering condition was met, this being signaled to the host machine so that it could instruct a memory access operation (via debugger and OCD) to inject the intended fault. The obtained results confirmed most of the expected potentialities and simultaneously identified some shortcomings both in fault triggering and performance. It proved possible to insert faults in memory space without affecting the running application and then use the trace information gathered as an effective mean to analyze program flow, before and after the actual fault activation. However, as all NEXUS compliant debuggers currently communicate with the host machine through Ethernet or USB connections, and as the fault campaigns must be run on the host machine, this imposes a bottleneck on the time required for an actual memory access. This fact causes the time interval required for reading a memory cell contents and writing back a modified value to be measured in milliseconds. This delay allows the initial data to be overwritten by the application running on the target system, the magnitude of the problem depending of the running application and memory position targeted. An additional problem is the triggering of a fault. Both the described problems are not directly related with the OCD capabilities but rather with the available tools, which lack some features that, not being necessary for debug, would be very useful for fault injection. The probability of the running application overwriting the targeted cell during the fault injection process can be minimized by reducing the writing delay of the fault injection process. The triggering delay problem can be solved by adding reactive behavior to the debugger so that it can perform a write operation on the detection of a specific signal or message from the target system. This issue can be addressed by a debugger with the required capabilities and interfaces.

## III. Case Study

### A. Target System

The use of a NEXUS compliant debugger benefits from the useful features defined in this standard and increases the area of immediate applicability of the developed concepts and solutions. As neither the actual compatible CPUs nor the commercial debuggers are easily modifiable, the reported case study requires (1) an alternative microprocessor core where a compliant OCD infrastructure could be implemented and (2) a customized debugger, as specific libraries are required for each target. The OCD and the debugger itself were developed as two distinct VHDL modules, aiming to keep them simple and easily portable to maintain a high level of compatibility with different target architectures. In this way a complete proof-of-concept solution was tested and the requirements for its migration to different systems were evaluated.

The cpugenerator [13] building tool was selected to create the different microprocessor targets. It is publicly available through opencores [14] and allows the automatic creation of 4, 8, 16 or 32 bit RISC microprocessor cores, being possible to configure several parameters like bus type, interrupt support and memory configuration. The OCD version implemented on the target system is NEXUS Class 2 compliant and provides some customization features, to be compatible with different CPU configurations with only minor adjustments. It is possible to define the data bus width (input and output) and the internal FIFOs used to store data prior to its decoding or communication. These parameters are very important as they may constrain the capabilities of the OCD in terms of trace and real time access. On the other hand, the use of larger buses can significantly increase the logic overhead imposed by the OCD infrastructure. The target application for testing is a matrix_addFT program, which is a fault tolerant version of a matrix adder. The fault tolerance is achieved by duplicating each arithmetic operation and then comparing the obtained results, with any difference triggering an error detection routine. Although not as powerful as hardware fault tolerance, this solution allows for some degree of dependability without modifications to the hardware, at the cost of memory space and some performance penalty.

The NEXUS standard defines a minimum set of debugging features, the interface port and the communication protocol. The implemented features include all common OCD features plus real time access to memory. The interface with the outside world is made using the AUX port option, which provides two message data buses for OCD data input and output along with independent clock and control signals. Two additional event pins allow halting the processor and provide exact timing for watchpoint / breakpoint signaling. The communication protocol followed the NEXUS standard spec, with all mandatory messages being included and two additional optional messages added for internal register access and OCD configuration.

### B. Fault Injection Environment

The selected fault model is the one used in most common fault scenarios for microprocessor based critical systems [15] and consists of single bit-flip faults in random memory elements at also random moments during the application execution. The actual fault trigger can be any instruction occurrence of the running application, covering the entire execution time. The fault location can be any resource accessible for writing through the OCD, including memory and internal registers (real time access is only possible when targeting memory).

All experiments are structured into fault injection campaigns, each one defining a set of fault injection operations where specific fault coordinates (location x value) and trigger condition are selected. In each such operation the processor is reset and the application runs from start. Each campaign is generated by an external tool and then described as a script with the necessary messages to be sent to the OCD infrastructure, both for configuration and data collection. Initialization is performed by loading the application into memory and setting up the OCD infrastructure as required by the specific operation. The target memory value at the moment of the injection must be determined beforehand, using either the knowledge of the running application code or a prior faultless execution up to the fault triggering instant and then using the OCD to read the relevant memory cell contents. In this manner it is possible to determine the value that should be stored so that a single bit-flip is caused on the target with a single write operation. The fault trigger condition is selected from the executed application code and can be any event that triggers a watchpoint, like an instruction execution or a data access. A normal fault injection scenario consists of the NEXUS compliant target microprocessor, the debugger running the fault campaigns and the host machine being used for set up and data analysis. This is presented in Fig. 2.

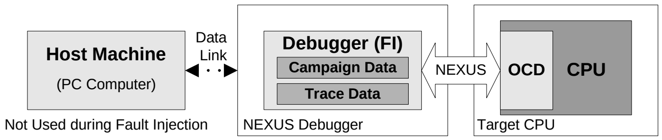


Fig. 2. Fault Injection Environment (Case Study)

The main advantage of this fault injection solution is the debugger capability to manage the entire fault injection process. Although the host machine is responsible for downloading the fault campaign data to the debugger and uploading the trace data after the fault campaign execution, the entire fault campaign is executed autonomously by the debugger. Additionally, if the target system is implemented on a FPGA device it is possible to add the debugger (and all relevant fault campaign data) as a module implemented on the same device, with the inherent advantages in terms of performance and cost.

Each fault injection operation consists of loading the debugger input memory with a series of instructions describing the steps required for its execution. After the initial set up is completed, the debugger waits for the triggering condition to be met, which will be signaled by a watchpoint hit signal or by a breakpoint hit message. When either of these events occurs the debugger sends a message to the OCD instructing it to write into the target memory position the intended faulty value. Although the debugger allows an instantaneous reaction, the actual fault insertion requires the transmission and decoding (by the OCD) of at least one complete message (the write command and data). During the entire operation the output memory records the trace messages that are sent by the OCD, to allow a subsequent program flow reconstruction and fault effect analysis. From these messages it is possible to diagnose fault effects, verifying if the fault was acknowledged by the error detection routine, and after the application runs its course it is possible to use the OCD to check if all final results are correct. All set up steps can be done with the target processor running normally, but the fault activation may only take place after this set up is performed. The program trace is not affected and operates normally before, during and after the fault injection process, reacting exactly as if a "real" fault occurred.

### *C. Debugger*

The debugger is presented in Fig. 3 and consists of a debugger core connected to two memory banks (input and output) and to a NEXUS debug port. All elements were designed to optimize the execution of fault injection operations with emphasis on execution speed.

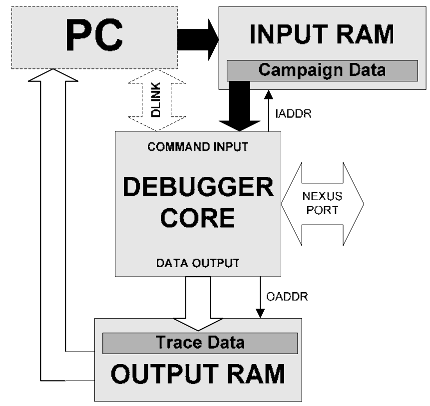


Fig. 3. Debugger

The debugger core is a simple processor type device that fetches commands from the input memory, controls execution and manages the data flow and possible error conditions. Direct control is possible through specific signals (DLINK) which may replace either the input or output memories (or both) as source of commands and destination of data. The access to the input memory, for reading purposes, is controlled by the debugger core and executed sequentially. Table I displays a list of available commands and the corresponding parameters.

Table I
Debugger Commands and Parameters

| MNEMONIC | PARAM | DESCRIPTION |
|---|---|---|
| HALT | None | Halts the target microprocessor execution and enters DEBUG mode. |
| RUN | None | Starts or resumes the target microprocessor execution. |
| RESET | None | Resets the target microprocessor. |
| DRESET | None | Resets the debugger, restarting command fetch from the initial input memory position. |
| DCONFIG | <code> | Configures the debugger according to the <code> parameter. |
| WAIT | <time> | Waits for a number of clock cycles defined by the <time> parameter. |
| WAITFOR | <event> <time> | Waits for a specific message or a watchpoint hit signal from the target OCD, during a specific period of time. The messages can be any response or trace message. |
| READRAM | <address> | Reads the contents of the memory cell at the specified address. |
| WRITERAM | <address> <data> | Writes a byte of data to the memory cell at the specified address. |
| READREG | <address> | Reads the contents of a register at the specified address. |
| WRITEREG | <address> <data> | Writes a byte of data to the register at the specified address. |

The output memory is used to store data for subsequent program flow analysis. The type of information stored can be selected by configuring the debugger and depends on the task at hand and available memory. The NEXUS port is managed by a communication controller responsible for translating commands into messages to be sent and retrieving the messages received from the OCD. The width of the data buses defines the duration of the transmission required by each message.

### D. Performance Improvements

The fault injection procedure described on the previous subsections was planed with the double objective of improving the performance and maintaining the highest level of compatibility with different target microprocessor architectures. It is possible to improve performance even further by modifying the OCD infrastructure present on the target microprocessor. Two approaches requiring modifications to the OCD were tested, namely (1) the simplification of the communication between the debugger and the OCD and (2) the migration of the reactive behavior to the OCD infrastructure itself. The first approach implies modifications to both the OCD and the debugger and consists of removing the NEXUS interface on both modules and connecting the debugger directly to the OCD interface signals, as displayed in Fig. 4.

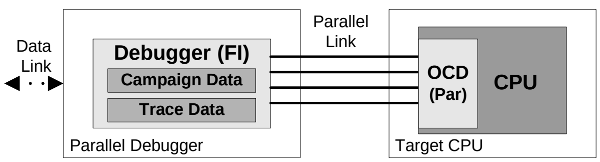


Fig. 4. Parallel Linking

In this approach the NEXUS communication manager is replaced by some glue-logic enabling parallel access to the OCD interface, with all required signals being accessible for reading (or writing) on a single clock cycle. In our case study the parallel link width varies between 34 and 106 bits, depending on implemented features and internal bus widths. The main modification is the elimination of the coding and decoding of the NEXUS messages and the inherent delay induced by those steps. However, this approach can only be applied either in simulation or using a special version of the target system and implies the loss of standardization and a considerable increase in the communications port width. The second approach is described in more detail in [16] and consists of adding an extra module to the OCD infrastructure in order to allow it to control part of the fault injection process. In this alternative the debugger and the NEXUS interface are unchanged, the differences being in the sequence of commands used for each fault injection operation as the actual triggering of the fault and memory writing operations are executed by the enhanced OCD itself.

## IV. Experimental Results

The target system, the debugger and the different memories were designed as VHDL models using the ISE 7.1i development environment [17] and simulated using the Modelsim 6.0a simulation engine. Three different OCD implementations were used with a common 32 bit CPU configuration, as summarized in Table II. The MPC565 is included for comparison purposes, the values representing the best possible configuration.

Table II
Target System Configurations

| Configuration | CLK (MHz) | MDI (bits) | MDO (bits) |
|---|---|---|---|
| OCD_Normal | 25 | 2 bits | 8 bits |
| OCD_Extended | 25 | 4 bits | 8 bits |
| OCD_Parallel | 25 | Parallel Link (72 bits) | |
| MPC565 | 40 | 2 bits | 8 bits |

The OCD_Normal and OCD_Extended configurations vary in terms of port width and on the size of the internal message buffers, with MDI being the Message Data In bus and MDO the Message Data Out bus. OCD_Normal represents a configuration equivalent to the best available for the MPC565 microprocessor and OCD_Extended represents an improved configuration for faster memory writing. The OCD_Parallel configuration replaces the NEXUS communication elements present on the OCD by synchronous access to the OCD input signals and requires a special version of the debugger. All configurations include separate ROM and RAM banks on the target system, the first for storing the program code and the later for application data. The fault campaigns were structured as follows:

- The OCD is configured once at the beginning of the campaign, with the configuration depending on the fault injection target (memory or registers). Each campaign is loaded into memory and the experiments are executed sequentially with the target CPU being RESET between experiments.
- The instruction address that triggers each fault injection is randomly generated from the actually executed ROM space and the target memory position is randomly selected from the actually used RAM space.
- The results are retrieved after all the experiments are complete and their analysis is performed externally with each experiment being diagnosed, to check if the final results are correct and if the fault was detected by the fault tolerance routine.

The simulation of about 100 fault campaigns repeated for each configuration returned the results presented in Table III. In this table inconclusive results represent experiments that had to be discarded due to incongruent trace data, and fault injection delay represents the time interval between the meeting of the trigger condition and the actual insertion of the faulty value as obtained from the simulation waveforms.

Table III
Fault Injection Results

| Configuration | Normal | Extended | Parallel |
|---|---|---|---|
| Inconclusive Results | 4% | 3% | 0% |
| Fault Injection Delay (In Clock Cycles) | 38 | 21 | 3 |

Some conclusions, relative to the fault injection process, are possible at this stage:

- It wouldn't be possible to execute the same fault campaigns (on real time) on a system using an MPC565 and a commercial controller as the reaction delay would be to high for this particular application (the total execution time is less than the interval required for injecting a single fault).

- When targeting memory in real time, some experiments return inconclusive results because the CPU writes on the memory cell being targeted before the fault is actually inserted.
- The width of the communication channel between the debugger and the OCD clearly affects the performance of the fault injection process, with the use of larger buses reducing the occurrence of inconclusive results.

The number of equivalent gates for each module and each target configuration is given by Table IV.

Table IV
Area Overhead (in Logic Gates)

| Module | Normal | Extended | Parallel |
|---|---|---|---|
| **CPU core** | 53717 | 53717 | 53717 |
| **OCD** | 17601 | 18801 | 15211 |
| **Debugger** (except RAM) | 992 | 1079 | 820 |

From the above values it is possible to confirm that a simple debugger (tasked only with fault injection campaigns management and results storage) requires comparatively little space on a programmable device.

## V. Conclusions

Dependability evaluation efforts sometimes neglect the possibilities of powerful OCD infrastructures present on the target device, even knowing that their use as a mean to execute non-intrusive real-time fault injection campaigns is often the best solution in terms of performance and capabilities. The reasons behind this are sometimes lack of appropriate tools or inadequate documentation. The diversity of methodologies, feature implementation and interface ports are also a downside. Our case study shows that the use of an optimized debugger and an OCD with real time access capabilities allows the execution of fault campaigns on the target memory space with full coverage of the application execution and used resources. The possibilities in terms of fault triggering and fault injection delay are dependent on the OCD capabilities, with communication speed being the fundamental factor. The use of larger communications ports allows faster operation and therefore minimizes the risk of the running application interfering with the process. The use of direct communication between the debugger and OCD infrastructures allows a considerable gain in performance but demands an increase in the width of the communication port that would not be acceptable for most hardware implementations. Possible middle term solutions are the increase of the OCD port size within acceptable values or the migration of some features to the inside of the OCD. Both solutions allow better performance at the cost of some additional logic overhead on the target OCD circuitry. Ongoing work is aimed at applying the proposed solutions to different target architectures and fault tolerant techniques. Simultaneously, means to further improve performance and coverage are also being studied.

## References


[1] "Coping with SEUs/SETs in microprocessors by means of low-cost solutions: A comparison study"; M. Rebaudengo, M. S. Reorda, M. Violante, B. Nicolescu, R. Velazco; IEEE Transactions on Nuclear Science, Vol 49, No 3; June 2002.

[2] "Basic concepts and taxonomy of dependable and secure computing"; A. Avizienis, J.C. Laprie, B. Randell, C. Landwehr; IEEE Transactions on Dependable and Secure Computing, Volume 1, Issue 1; Jan 2004.

[3] "Comparison and application of different VHDL-based fault injection techniques"; J. Gracia, J.C. Baraza, D. Gil, P.J. Gil; IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems; San Francisco, USA; Oct 2001.

[4] "Experimental evaluation of a COTS system for space applications"; H. Madeira, R. R. Some, F. Moreira, D. Costa, D. Rennels; International Conference on Dependable Systems and Networks; Bethesda, USA; June 2002.

[5] "Experimental Validation of High-Speed Fault-Tolerant Systems Using Physical Fault Injection"; R. J. Martínez, P. J. Gil, G. Martín, C. Pérez, J. J. Serrano; Seventh IFIP Working Conf. Dependable Computing for Critical Applications: DCCA-7; San Jose, USA; Jan. 1999.

[6] "Evaluation of the Thor Microprocessor Using Scan-chain-Based and Simulation Based Fault-Injection"; P. Folkesson, S. Svensson, J. Karlsson; 8th European Workshop on Dependable Computing (EWDC-8); Goteborg, Sweden; April 1997.

[7] "A Technique for Automated Validation of Fault Tolerant Designs Using Laser Fault Injection (LFI)"; J. R. Samson , W. A. Moreno, F. J. Falquez; 28th Annual International Symposium on Fault-Tolerant Computing; Munich, Germany; June 1998.

[8] "Comparison of physical and software-implemented fault injection techniques"; J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, G. H. Leber, IEEE Transactions on Computers, Volume 52, Issue 9; Sept. 2003.

[9] "The Nexus 5001 Forum Standard for a Global Embedded Processor Interface version 2.0"; IEEE-ISTO 5001; 2003.

[10] "INERTE: Integrated NExus-Based Real-Time Fault Injection Tool for Embedded Systems"; P. Yuste, D. de Andrés, L. Lemus , J. J. Serrano, P. J. Gil; The International Conference on Dependable Systems and Networks; San Francisco, USA; June 2003.

[11] www.freescale.com

[12] www.isystem.com/Products/Emulators/iC3000/

[13] Giovanni Ferrante, "CPUGEN 2.00", 2003.

[14] www.opencores.org

[15] "How to characterize the problem of SEU in processors & representative errors observed on flight"; R. Velazco, R. Ecoffet, F. Faure; 11th IEEE International On-Line Testing Symposium; Saint Raphael, France; July 2005.

[16] "A Modified Debugging Infrastructure to Assist Real Time Fault Injection Campaigns"; A. Fidalgo, G. Alves, J. Ferreira; 9th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (in press); Prague, Czech Republic; April 2006

[17] www.xilinx.com