

A Fault Injection Methodology and Infrastructure for Fast Single Event Upsets Emulation on Xilinx SRAM-based FPGAs

*Original*

A Fault Injection Methodology and Infrastructure for Fast Single Event Upsets Emulation on Xilinx SRAM-based FPGAs / DI CARLO, Stefano; Prinetto, Paolo Ernesto; Rolfo, D.; Trotta, P.. - ELETTRONICO. - (2014), pp. 159-164. ( 27th IEEE Defect and Fault Tolerance in VLSI and Nanotechnology Systems Symposium (DFTS) Amsterdam, NL 1-3 Oct. 2014) [10.1109/DFT.2014.6962073].

*Availability:*

This version is available at: 11583/2571947 since: 2016-10-07T17:36:56Z

*Publisher:*

IEEE

*Published*

DOI:10.1109/DFT.2014.6962073

*Terms of use:*

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

*Publisher copyright*

IEEE postprint/Author's Accepted Manuscript

©2014 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

# A Fault Injection Methodology and Infrastructure for Fast Single Event Upsets Emulation on Xilinx SRAM-based FPGAs

Stefano Di Carlo, Paolo Prinetto, Daniele Rolfo, Pascal Trotta  
Politecnico di Torino

Dipartimento di Automatica e Informatica  
Corso Duca degli Abruzzi 24, I-10129, Torino, Italy  
Email: {*FirstName.LastName*}@polito.it

**Abstract**—Modern SRAM-based Field Programmable Gate Arrays (FPGAs) are increasingly employed in safety- and mission-critical applications. However, the aggressive technology scaling is highlighting the increasing sensitivity of such devices to Single Event Upsets (SEUs) caused by external radiation events. Assessing the reliability of FPGA-based systems in the early design stages is of utmost importance, allowing design exploration of different protection alternatives.

This paper presents a Dynamic Partial Reconfiguration-based fault injection methodology implemented by an integrated infrastructure for SEUs emulation in the configuration memory of Xilinx SRAM-based FPGAs. The proposed methodology exploits the Xilinx Essential Bits technology to extremely speed-up fault injection, ensuring correct operations of the fault injection infrastructure during the whole injection process.

**Index Terms**—Fault Injection, FPGA, SEUs, SRAM-based FPGA, Essential Bits technology.

## I. INTRODUCTION

Nowadays, the reduced performance, speed and power gaps between Field Programmable Gate Arrays (FPGAs) and Application Specific Integrated Circuits (ASICs) is suggesting IC designers to choose FPGAs not only for ASIC prototyping, but also for final products, leading to lower time-to-market and limited non-recurrent engineering costs [1], [2].

However, due to their structure, SRAM-based FPGAs are highly susceptible to Single Event Upsets (SEUs), i.e., unintentional bit-flips in memory elements that can be caused by external radiations phenomena, such as collisions with alpha particles, protons, or heavy ions striking the device [3], [4]. Radiation induced Single Event Upsets (SEUs) can affect both the memory elements embedded in the design, and the configuration memory that stores the related FPGA configuration. In the former case, SEUs may alter the content of the FPGA internal memory resources employed in the design (e.g., flip-flops, distributed RAMs, or Block-RAMs), modifying processed data and/or the application control flow, thus leading to problem similar to the ones occurring in ASICs. In such cases fault injection techniques similar to the ones used when dealing with ASICs can be employed to assess the reliability of the design (e.g., scan-chains [5]). Instead,

the latter case is much more critical, since a bit-flip in an FPGA configuration memory cell may permanently alter the functionality of the implemented circuit. In particular, it can cause changes in the configuration of Look-Up Tables (LUTs), Configurable Logic Blocks (CLBs), internal hard macros (e.g., Digital Signal Processors or Block-RAMs), or routing matrices, leading to completely different circuits from the initially configured ones [6]. Furthermore, the configuration memory is very large compared to all the other elements in the device. Therefore, the probability that SEUs affect the FPGA configuration memory is high, making it a major concern when designing high reliable FPGA-based systems. Nevertheless, it has been demonstrated that, in most cases, SEUs affecting the configuration memory do not influence the design functionality [7]–[9]. Consequently, fault injection techniques are needed to deeply analyze the effect that SEUs in the configuration memory have on the actual implemented design. Fault injection can help designers to discover the weaknesses of the circuit and to take the proper countermeasures by selectively and efficiently applying the most suitable fault detection and/or fault tolerance techniques to harden the design.

To accurately assess the reliability of FPGA-based systems, designers must rely on very expensive neutrons or heavy ions beam radiation tests on actual circuit prototypes [10]. Preliminary, even less accurate, fault injection experiments in the early design stages can potentially reduce the number of design iterations, speeding up the entire design process of complex Systems on a Programmable Chip (SoPC).

In literature, several solutions tackling this problem have been proposed and developed, spanning from simulation-based methods [6] to hardware approaches [11]–[15]. The latter approaches guarantee the maximum fault injection speed when the circuit under test and the fault injection infrastructure are implemented in the same FPGA. To achieve high fault injection rates they usually exploit the Dynamic Partial Reconfiguration (DPR) feature of modern SRAM-based FPGA devices (i.e., the ability to dynamically change selected portions of a circuit, while the rest of the design is left unchanged and fully functional [16]).

Basically, for each fault injection run, the infrastructure

reads a *frame* of the configuration memory, composed of several 32-bit words. Then it modifies the *frame* according to the chosen fault model, and reconfigure the configuration memory with the faulty *frame* exploiting the Internal Configuration Access Port (ICAP) [16] of the FPGA. The internal fault injection infrastructure also provides the input vectors and reads the outputs of the system under test to verify their correctness. Finally, the faulty *frame* is restored and another bit in the same or in another frame is flipped [12], [13].

Nonetheless, to properly operate, these approaches require an in-depth knowledge of the structure and the frames addressing order of the considered FPGA. In fact, the designer must know the addresses of the configuration frames associated with the FPGA resources implementing the system under test. This information is not explicitly provided by vendors and depends on the specific FPGA model. Moreover, in some unpredictable conditions, since a bit-flip in a *frame* can affect also the information in the other frames, the fault-free configuration cannot be restored after fault injection [13]. In this case, a reconfiguration of the entire device is therefore needed to avoid fault accumulation effects [17]. Furthermore, extra care must be taken since faults injected using the integrated approaches can affect the operations of the fault injection infrastructure itself. This unintended effect can lead to the stall of the fault injection process, or to unknown erroneous faults classification results [17].

This paper presents a Dynamic Partial Reconfiguration-based fault injection methodology and infrastructure for SEUs emulation in the configuration memory of Xilinx SRAM-based FPGAs. The fault injection infrastructure and the system under test are integrated in the same device. The proposed methodology exploits the Xilinx *Essential Bits* technology [18] to:

- 1) extremely speed-up the fault injection process, as demonstrated by experiments carried out on designs of different complexity;
- 2) ensure the correct behavior of the fault injection infrastructure itself, avoiding undesired faults accumulation effects during the whole fault injection process, as it occurs on other single-FPGA fault injection platforms [12], [13], [17].

The rest of the paper is organized as follows: Sections II introduces the proposed methodology, while III discusses the fault injection infrastructure. Section IV reports experimental results and, eventually, Section V summarizes the contributions and presents some future works.

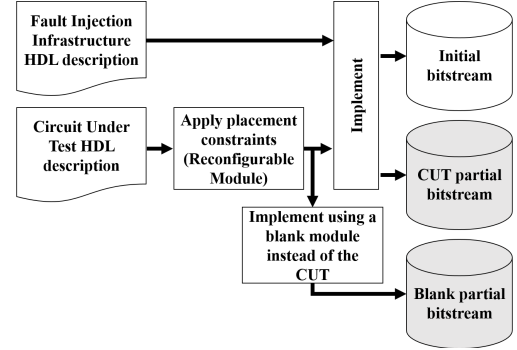
## II. PROPOSED METHODOLOGY

According to the *FARM* model [19], when designing a fault injection environment, one must take into account the adopted *Fault Models*, the *Activation patterns* used to stimulate the system under test, the *Readouts* values collected during the experiment, and the extracted *Measures*.

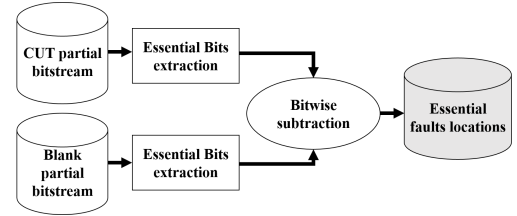
As aforementioned, the fault model adopted by the proposed infrastructure is the Single Event Upset (SEU) in the configuration memory of the FPGA device.

During each fault injection run, the choice of which configuration bit must be flipped is randomly made on-line, choosing from a set of possible locations (*fault list*), generated off-line.

The set of possible locations in which the SEUs will be injected is generated in two steps (Fig. 1).



(a) Step 1: CUT and blank partial bitstreams generation



(b) Step 2: essential faults location generation

Fig. 1: Fault locations generation flow

To ensure the correct operations of the fault injection infrastructure during the whole fault injection process, the basic idea of the proposed approach is to implement the *Circuit Under Test* in a reconfigurable partition of the FPGA, and to extract, from the bitstream, only the locations of the bit associated to the CUT resources. A *reconfigurable partition* is a portion of the FPGA that can be reconfigured at run-time. The size and the position of the reconfigurable partition in the FPGA must be chosen at design-time [16]. Differently from the CUT, the fault injection infrastructure is instead implemented in a non-reconfigurable, or *static*, portion of the device.

During the first phase (Fig. 1a), the HDL description of both the fault injection infrastructure and the CUT are merged and implemented to obtain the *Initial bitstream* configuration file, used to configure at startup the entire FPGA.

In order to implement the CUT in a reconfigurable partition, before the actual implementation, some placement constraints must be provided (i.e., the designer allocates a defined area of the FPGA to the CUT). In this case the CUT is called *Reconfigurable Module*. After that, the implementation process generates also the *CUT partial bitstream* file, that contains the information needed to configure the *Reconfigurable Module*.

The same process is repeated to implement an empty, or *blank*, reconfigurable module instead of the CUT, in the

same previously defined reconfigurable area. The *Blank partial bitstream* file associated with this module contains only the information regarding the input/output interfaces of the CUT (called *partition pins*), and the routing associated to the static part of the design that passes through the reconfigurable partition. As stated in [16], only routing, and not logic, associated with the static part of the design can use hardware resources contained in a reconfigurable partition.

Consequently, since this static routing information cannot be prevented to pass through reconfigurable partitions, an SEU injected in the configuration bits associated to this area can cause errors or failures also in the fault injection infrastructure itself (i.e., the static part of the design) [16]. This can make the fault injection infrastructure unusable, or behaving incorrectly.

To ensure that faults are not injected in a configuration memory cell associated with the fault injection infrastructure, a further step is needed (Fig. 1b).

Starting from the partial bitstreams associated to the CUT and to the blank reconfigurable modules, the Xilinx *Essential Bits* technology is used to restrict the set of potential fault injection locations [18]. This technology offers a functionality, embedded in the Xilinx *BitGen* tool, that allows to identify and to extract, from the bitstream files, the configuration bits that are essential to the design functionality (i.e., the so called *essential bits*). In fact, only a small fraction of the bits are essential to the proper operation of any specific design loaded into the FPGA device [18]. As shown in Fig. 2a, the *Essential Bits* functionality provides a mask in which if a bit is set, the associated bit in the bitstream file is essential, and thus, if flipped, it modifies the circuit functionality (Fig. 2b).

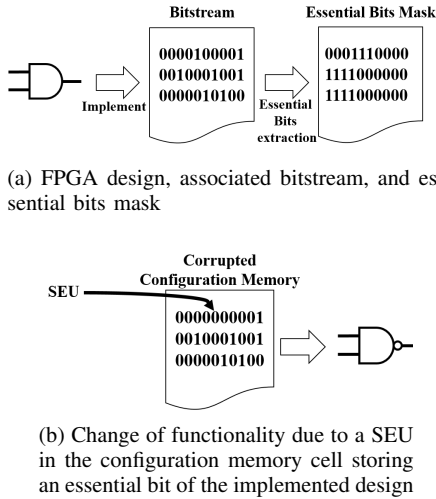


Fig. 2: *Essential bits* meaning

Since the objective of a fault injection infrastructure is to flip those bits associated with the CUT, resorting to the essential bits mask of the blank partial bitstream it is possible to localize the locations of the bits associated with the static routing passing through the reconfigurable module. The mask

containing the position of the essential bits of the CUT (*Essential faults locations* in Fig. 1b) is obtained by bit-wise subtracting the masks associated to both the CUT and the Blank partial bitstreams. It is worth to remember here that, as aforementioned, some bits in the Blank bitstream carry information about the I/O interfaces of the CUT. By applying this method, these bits will be not flipped during the fault injection process, thus their contribution represents an error on the final computed fault injection metrics. However, as will be demonstrated in Section IV, this contribution is very small, thus can be assumed negligible if the fault injection experiments are made in the early design stages, where highly accurate results are not required.

This means that the fault injection infrastructure will always inject a SEU in a position associated with a bit that has no impact on the infrastructure itself, thus ensuring correct operations during the whole injection process.

The second main advantage of using the *Essential Bits* technology for fault injection is that the injection time is dramatically reduced, with respect to inject faults in every possible configuration memory location associated to the CUT partition. In fact, in general, the essential bits of a module implemented in a partition are lower than 20% of the total configuration bits of that partition [18]. Thus, it is useless to inject SEUs in the remaining 80% of the bits composing the bitstream, since it is known a-priori that those faults will not cause any observable functional error. Obviously, these not injected faults are counted as if they have been injected to compute the final output statistics results.

### III. PROPOSED INFRASTRUCTURE

The architecture of the proposed fault injection infrastructure is depicted in Fig. 3.

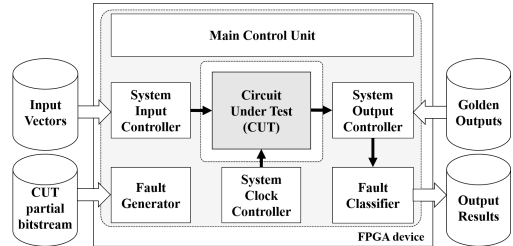


Fig. 3: Proposed fault injection infrastructure architecture

The infrastructure takes in input the bitstream configuration file of the *Circuit Under Test* (CUT), the *Input Vectors* needed to exercise the CUT, and the *Golden Outputs* of the fault-free CUT run. According to the SEU fault model, a single bit-flip is introduced in the configuration memory for each execution run. At the end of the fault injection process, the infrastructure provides in output the results in terms of percentages of faults that caused an observable functional error in the CUT.

The next subsections details the operation of each module composing the infrastructure.

#### A. Fault Generator

The *Essential faults location* mask, extracted as detailed in the previous section, is used at run-time by the *Fault Generator* to choose a possible fault injection location. Using a Linear-Feedback-Shift-Register (LFSR), a pseudo-random injection time and location (among the possible given by the *Essential faults location* mask) are generated. When the execution time reaches the selected injection time, the *Fault Generator* reads the fault-free CUT partial bitstream (Fig. 3) and flips a bit according to the selected position. Afterwards, the execution can be resumed. At the end of the execution, the CUT reconfigurable module is restored with the fault-free partial bitstream. This last reconfiguration overwrites all the configuration *frames* associated with the CUT, avoiding any fault accumulation effect.

It is worth noting that the proposed fault generation method can be easily adapted to emulate also Multiple Bit Upsets (MBUs). This fault model is expected to become a more realistic model, with respect to SEUs, for modern and future technology nodes.

#### B. System Input Controller

Referring to the *FARM* model [19], the Activation patterns, or *Input Vectors*, are provided to the CUT through the *System Input Controller*. The *Input Vectors* can be stored either internally to the FPGA device or in an external memory, depending on their size and on the CUT nature (e.g., processor, datapath, control unit). They can be generated off-line through simulations or other techniques. Thus, the *System Input Controller* consists of a control unit, acting as a memory controller, that reads the vectors from an internal or external memory, and feeds the inputs of the CUT. When the CUT needs pseudo-random input vectors, the *System Input Controller* simply consists of a Linear-Feedback Shift-Register (LFSR) [20].

#### C. System Clock Controller

The *System Clock Controller* is mainly composed of one or more FPGA Digital Clock Managers (DCMs), able to synthesize and manage the clocks needed by the CUT.

It works in conjunction with the *System Input Controller* to synchronize the CUT clock to the *Input Vectors*, and to stop it during the reconfiguration process. In particular, at a random time, during the execution of a run, the CUT clock is stopped and the configuration memory is reconfigured using the faulty CUT partial bitstream, generated as explained in Subsection III-A. Afterwards, the CUT clock is re-activated, resuming the execution of the actual run until the fault is detected, or until the end of the run, if the injected fault does not generate an observable error.

#### D. System Output Collector and Fault Classifier

Referring to the *FARM* model [19], the Readouts and the Measures are performed through the *System Output Collector* and *Fault Classifier* modules.

The *System Output Collector* monitors the outputs of the CUT after each fault injection. Depending on the number

of outputs and number of responses to observe, the designer can choose the output comparison technique that best fit the considered CUT and test case (e.g., outputs compression and signatures comparison, clock-by-clock comparison [20], etc).

If a difference is encountered during the outputs comparison process, the fault in the configuration memory of the FPGA device is targeted as *critical*. In the opposite case, the fault is considered *non-critical* since, even if at the end of the run it can be still present in the configuration memory, it does not generate erroneous results or system failures. If the CUT is equipped with a fault detection mechanism, the error detection signal can be used by the *Fault Classifier* to classify the faults detected by the CUT detection mechanisms, labeling them as *hardware detected*. This functionality can be very useful when, in the early design stages, the designer is interested in evaluating different fault detection techniques.

### IV. EXPERIMENTAL RESULTS

This section presents the experimental results gathered by implementing the proposed fault injection infrastructure on a Xilinx ML605 evaluation board, equipped with a *Virtex-6 VLX240T* FPGA.

Three case studies have been considered:

- *LEON3*-based SoC, including a *LEON3* processor [21] and several peripherals. The processor runs several applications extracted from the *MiBench* benchmark suite [22]. The applications have been selected in order to stimulate different units of the processor;
- two-dimensional convolution datapath, as the one reported in [23], composed of 49 8x8 multipliers and a balanced adder tree including 48 adders. This architecture is often employed when dealing with two-dimensional images filtering;
- triplicated two-dimensional convolution datapath, with a majority voter that sets an *Error detected* signal when it recognizes a mismatch between the three module outputs.

Some modules of the fault injection infrastructure (in particular the *System Input* and *Output Controllers*) must be modified depending on the nature of the actual circuit under test (e.g., CPU or datapath) and on the adopted test procedure. In particular, for the *LEON3*-based SoC case study, the processor runs at 80 MHz, reading the application and the data from an external memory through the *System Input Controller*. The *System Output Controller*, at the end of the execution, reads from the external memory the results produced by the processor, and compares them with the golden ones stored in an internal Block-RAM. The comparison results are sent to the *Fault Classifier* which computes the fault injection metrics.

Instead, in the second test case (i.e., 2D convolution datapath), the *System Input Controller* consists of an LFSR that pseudo-randomly generates 98 8-bit inputs for the 49 multipliers. The inputs generation process is repeated for 307,200 clock cycles in order to emulate the processing of an image composed of 640x480 pixels. The *System Output Controller* compresses the output values using a Multiple-Input Shift-Register (MISR), and compares the output signature with the

golden one stored internally [20]. In the last considered test case, the *Error detected* signal of the TMR Voter is connected to the *Fault Classifier* to notify if a fault is detected by the TMR mechanism.

The *System Clock Controller* and the *Fault Generator* are the same among the three case studies. The *System Clock Controller* is composed of a single Digital Clock Manager (DCM), necessary to synthesize the clock for the CUT. The *Fault Generator* consists of: a memory controller, that reads from the external memory a look-up table storing the positions of each essential bit in the bitstream, two LFSRs, that pseudo-randomly select the injection time and the essential bit position to flip, and a Dynamic Partial Reconfiguration (DPR) controller which writes the CUT bitstreams in the configuration memory through the *Internal Configuration Access Port* (ICAP) of the FPGA device. The ICAP is driven by the *Fault Generator* using the maximum possible clock frequency (i.e., 100 MHz), providing a reconfiguration bandwidth of 3.2 Gbps [16].

Table I summarizes the hardware FPGA resources needed to implement the fault injection infrastructure in the three considered test cases (SIC, SOC and FC represent the *System Input Controller*, the *System Output Controller* and the *Fault Classifier*, respectively). As can be seen from Table I, the resources consumption of the fault injection infrastructure in the three test cases is very limited (around 3.6% of the overall FPGA resources), even if the target device represents a medium-sized FPGA.

Table II shows the partial bitstream sizes ( $BS$ ) associated with the CUTs, the percentage of essential bits  $\%EB$ , the time needed to run the application ( $T_{run}$ ), and the total injection time ( $T_{inj}$ ), performing a number of runs equal to the number of extracted essential bits.

In general, the total injection time can be estimated as:

$$T_{inj} = \#EB \cdot (T_{run} + T_{DPR}) \quad (1)$$

$T_{DPR}$  is the sum of two contributions: the first is the time needed to configure, with the faulty bitstream, the reconfigurable partition in which the CUT is implemented, while the second represents the time needed to restore it, with the golden bitstream, at the end of each run. Since the faulty and the

golden CUT bitstreams have the same size,  $T_{DPR}$  can be computed as:

$$T_{DPR} = 2 \cdot \frac{BS}{f_{ICAP}} \quad (2)$$

where  $BS$  is the bitstream size, and  $f_{ICAP}$  represents the ICAP operating frequency (i.e., 100 MHz in our experiments).

It is worth mentioning that in all the considered test cases, the percentage of essential bits associated with the static part of the design (i.e., the fault injection infrastructure and the I/O interface of the CUT) in the CUT bitstream is less than the 0.7%. As explained in Sec. III, these bits are not included in the fault injection process. This contribution can be treated as an error on the final computed metrics, since the fault injection infrastructure will not be present in the final circuit implementation. However, this error can be considered negligible if this kind of evaluation is made in the early design phases, where a very accurate measure is not required.

Table III shows the *Fault Classifier* results in terms of percentages of faults that caused an observable error (*Critical*), and those that have not caused any error (*Non-Critical*). The number of *Equivalent Injected Faults* (EIF) and the percentages are computed taking into account also the number of non-essential bits. Each non-essential bit composing the bitstream can be flipped without any effect on the circuit functionality, thus a fault in those bits can be considered as *Non-Critical*.

As can be seen from Table III, in the third test case (i.e., *LEON3* running IFFT [22]) the percentage of faults that cause an error is higher than the percentage in the first two test cases, since the application uses the floating point unit, which represents about 40% of the CUT area. In the last test case the number of *Critical* faults is greatly reduced with respect to the fourth test case, since all the faults detected by the TMR mechanism that do not lead to a Voter output error are considered *Non-critical*.

The proposed methodology and infrastructure has been compared, in terms of fault injection execution time, with the integrated methods presented in [17], [12], and [13]. The fault injection platforms presented in [12], [13], [17] require a fixed  $T_{DPR}$  of about  $10\mu s$ , since they reconfigure a single

TABLE I: Fault injection infrastructure hardware resources consumption for the three considered test cases

Module	LUTs	FFs	BRAMs	DCMs
Fault Generator	499	359	-	-
SIC (LEON)	2,292	1,430	-	-
SIC (2D Conv.)	3,332	784	-	-
SIC (TMR)	3,332	784	-	-
Sys. Clock Contr.	-	-	-	1
SOC (LEON)	1,251	315	4	-
SOC (2D Conv.)	166	24	-	-
SOC (TMR)	166	24	-	-
FC (LEON)	403	92	-	-
FC (2D Conv.)	403	92	-	-
FC (TMR)	436	118	-	-
Main Control Unit	1,094	67	-	1
<b>Total (LEON)</b>	<b>5,539 (3.68%)</b>	<b>2,263 (0.75%)</b>	<b>4 (0.96%)</b>	<b>2 (16.7%)</b>
<b>Total (2D Conv.)</b>	<b>5,494 (3.65%)</b>	<b>1,326 (0.44%)</b>	-	<b>2 (16.7%)</b>
<b>Total (TMR)</b>	<b>5,527 (3.67%)</b>	<b>1,352 (0.45%)</b>	-	<b>2 (16.7%)</b>

TABLE II: CUTs Bitstream size, percentage of *Essential Bits*, application execution time, and total injection time

CUT	$BS[KB]$	$\%EB$	$T_{run}[ms]$	$T_{inj}[s]$
L3 Susan	755.6	16.2	37.91	41,415
L3 CRC32	755.6	16.2	20.94	24,552
L3 IFFT	755.6	16.2	395.65	395,514
2D conv.	170.9	13.6	6.14	4,573
2D conv TMR	478.4	13.6	6.14	4,573

TABLE III: Fault Injection classification results

CUT	EIF	% Critical	% Non-Critical
L3 Susan	6.18 M	9.8	90.2
L3 CRC32	6.18 M	7.3	92.7
L3 IFFT	6.18 M	13.6	82.4
2D conv.	1.39 M	12.4	87.6
2D conv TMR	1.39 M	1.3	98.7



