# A Low-Cost SEU Fault Emulation Platform for SRAM-Based FPGAs

P. Kenterlis, N. Kranitis, A. Paschalis
*Department of Informatics & Telecommunications*
*University of Athens, Greece*
{pkent, nkran, paschali}@di.uoa.gr

D. Gizopoulos, M. Psarakis
*Department of Informatics*
*University of Piraeus, Greece*
{dgizop, mpsarak}@unipi.gr

## Abstract

*In this paper, we introduce a fully automated low cost hardware/software platform for efficiently performing fault emulation experiments targeting SEUs in the configuration bits of FPGA devices, without the need for expensive radiation experiments. We propose a method for significantly reducing the fault list by removing the faults on unused LUT bit positions. We also target the design Flip-Flops found in the Configurable Logic Blocks (CLBs) inside the FPGA. Run-time reconfigurability of Virtex devices using JBits is exploited to provide the means not only for fault injection but fault detection as well. First, we consider five possible application scenarios for evaluating different self-test schemes. Then, we apply the least favorite and most time consuming of these scenarios on two 32x32 multiplier designs, demonstrating that transferring the simulation processing workload to FPGA hardware can allow for acceleration of simulation time of more than two orders of magnitude.*

## 1. Introduction

Moving to Very Deep Sub Micron (VDSM) technology, chips become increasingly prone to faulty behavior caused by cosmic or artificial radiation. Such faults are modeled as Single Event Upsets (SEUs) in the form of bit-flips [1]. SRAM-based FPGA devices are steadily becoming the most suitable platform for implementing modern system applications due to their high reconfigurability, low cost and availability, which promote fast time-to-market. SRAM-based FPGAs, however, are much more susceptible to SEUs than ASICs. Recently, SEU effects on SRAM-based FPGA resources has been a field of research [1], [2], [3], [4], [5]. When dealing with SEUs in SRAM-based FPGAs, two types of fault locations need to be considered:

1. Upsets in the Flip-Flops in the form of a bit-flip.
2. Upsets in the configuration bits that control the combinational and sequential logic (Look-Up Tables, MUXes, Flip-Flop initialization bits) and routing.

Ionizing radiation hitting a design Flip-Flop or one of its control signal inputs can invert its stored logic value. These upsets could be harmful if allowed to propagate through states at the rest of the circuit. For example, in the case of FSM state registers where feedback is present, the effect of the upset could be detrimental to the system's operation.

Furthermore, a SEU can "permanently" change the function of the configured circuit by hitting a configuration SRAM cell. For example (Figure 1), ionizing radiation hitting a LUT configuration bit will change the output function to represent another logic gate (or set of gates). No actual permanent damage is inflicted upon the device since after power cycling the FPGA board, a fresh copy of the configuration bitstream will be loaded from the external configuration memory to the FPGA device and eradicate the fault.

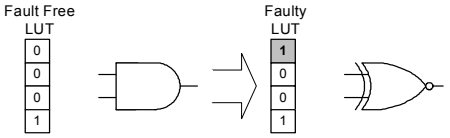


**Figure 1** Effect of a SEU in a LUT

When designing circuits to be implemented in FPGAs, it is unrealistic to consider structural fault models, i.e. stuck-at, transition faults, since FPGAs do not have a gate level representation for which such faults can be modeled. For example, in Figure 1 we show that a SEU changes an AND gate to a XNOR gate. In SRAM-based FPGAs, individual memory cells control the configuration of CLB and routing resources to implement the designed circuit. Attempting to model LUTs as equivalent gate circuits and inject stuck-at faults will not address the effect of SEUs which are the dominant faults in FPGAs. Existing fault simulators employ structural fault models which are not applicable to FPGAs affected by SEUs.

An alternative method to emulate the effects of SEUs is to employ functional simulators of the FPGA devices. However, FPGA fault simulation of large circuits even when performed by multi-GHz workstations requires a vast amount of processing time. When dealing with large system applications the required simulation time becomes prohibitive. Currently, VirtexDS [6] designed by Xilinx is a software device simulator that can be used in order to simulate the functions of FPGA devices, more specifically Xilinx's Virtex series.

By transferring the simulation process of combinational circuits to actual hardware, a speed-up of two orders of magnitude has been indicated over software simulation [7]. For large sequential circuits and long test patterns, due to complexity, we can expect higher performance increases from hardware emulation. Besides, by building hardware emulation capabilities in a prototype board of the developed system with all its peripheral circuitry, it is technically realizable to emulate the effects of SEUs in a prototype as these would appear during in-field operation.

Software simulation introduces outstanding time cost, while radiation experiments designed to introduce SEUs in SRAM-based FPGA devices under operation are expensive in terms of equipment used. In addition, radiation is harmful to the FPGA device itself. A non-destructive, non-intrusive fault emulation platform addressing the effects of SEUs in FPGAs is becoming increasingly challenging.

The same technology feature that makes SRAM-based FPGAs vulnerable to SEUs can also be used to develop techniques that detect and correct the effects of SEUs; Run-Time Reconfiguration (RTR), i.e. reconfiguration of the system during normal operation in order to perform a different function. RTR comes in two variants, full and partial. Partial RTR allows small portions of the FPGA's configuration memory to be altered using small partial

bitstreams. Partial RTR is used to perform minor changes in the configured circuit on the FPGA, accelerating the fault injection process required in our work. VirtexDS supports full and partial RTR operations. We utilize VirtexDS as a fault simulator by changing the configuration memory and using partial RTR to compare with the proposed fault emulation platform.

In this paper we concentrate our interest in the investigation of SEUs in the device configuration bits and user design registers. More specifically, we focus on the faults affecting the used LUT configuration bits of CLB Function Generators and CLB Flip-Flops used by the implemented circuit. As reported in previous works (e.g. [7]), the faults affecting only the LUT configuration bits can be considered as an effective sampling technique. As demonstrated through radiation experiments [3] the LUTs are the most SEU sensitive resources of the device; not only directly by SEUs affecting the LUT configuration bits, but also indirectly, with particles affecting other resources of the device, for example, affecting the LUT write enable line may induce corruptions to all bits in one LUT. We regard the 4-input LUT as a 16-bit vector for which SEUs can occur and alter the implemented function so as to represent an entirely different circuit.

In this paper, a fully automated low cost hardware/software fault emulator platform usable with any Virtex-based board is presented. Selection of board is limited only by the availability of the SelectMAP programming port pins to the user, thus allowing the use of low cost prototyping or evaluation boards. Also, algorithms used for LUT and Flip-Flop fault list extraction and fault injection are presented. A method for significantly reducing the fault list by removing the faults on unused LUT bit positions is proposed. Then, five possible scenarios for fault injection and detection on different types of implemented circuits supported by our fault emulator are described. Finally, experimental results using the implemented fault emulator on two 32x32 multipliers, with and without pipelining, are exhibited, demonstrating that a speed increase of two orders of magnitude is achieved in the case of combinational circuits, while a speed increase of two orders of magnitude is achieved for sequential circuit designs.

## 2. Previous Work

Fault injection by radiation exposure has been investigated in [3] targeting FPGA configuration bits aiming to characterize the sensitivity of the configuration memory to SEUs. A simulation-based fault injection tool is presented for calculating the implemented circuit error rate. An FPGA SEU simulator platform to represent radiation induced configuration bitstream upsets was presented in [4] based on a Virtex FPGA board. The SEU simulator predicted 97% of the output errors observed during radiation testing experiments, thus provided an affordable platform to explore costs and benefits of various SEU mitigation techniques. A fault injection environment was proposed in [2] as an alternative to radiation testing experiments. Publicly available software by Xilinx, JBits, was used for accessing, analyzing and modifying configuration memory resources, while VirtexDS was used to simulate the FPGA device performing fault injection experiments. Experimental results were provided using functional test vectors and fault injection in a random fault list of ITC99 benchmarks. Since VirtexDS is a software device simulator, the fault injection environment is highly CPU-time consuming and not practical for large designs.

In [9], [10], a software/hardware fault injection platform is presented that allows direct manipulation of the configuration bitstream on Xilinx's Spartan2 FPGA devices. An Enhanced Parallel Port (EPP) interface is employed to connect to the fault emulator which is embedded in the target FPGA device. Fault injection process times reported for each fault are in the range of 9s and 13s respectively, for different size of FPGA device being used. Most of these times, as reported, were spent downloading the faulty bitstream, while only a very small fraction actually involved fault injection and emulated circuit execution.

A fault injection tool for SRAM-based FPGAs is presented in [5] based on fault emulation. It provides the capability of fault injection, in particular to the Programmable Interconnection Points (PIPs) by modifying the configuration bitstream, avoiding the use of any standard commercial software and their related limitations. A hardware-software platform for SEU immunity verification targeting Flip-Flops is presented in [11]. The platform is based on proprietary software and specific hardware testing board, using Xilinx's Virtex-II FPGAs, thus making it a somewhat inflexible solution. Due to the fast communications bus used to connect to the testing board (EPP or USB), it is possible to inject and analyze at least 80K faults per hour in a system with 2M test vectors. In our work we make use of existing hardware for minimization of cost.

## 3. The Fault Emulation Platform

The fault emulator platform is a mixture of software and hardware entities. Care has been taken so that these two worlds remain independent as much as possible to allow for changes in each one without harmfully affecting the other.

### 3.1. Software

The software portion of the fault emulator is based on the JBits 2.8 API [12][13] and coded in Java. JBits is a collection of classes that allow user code to manipulate Virtex/Virtex-II device bitstreams. The software portion we developed based on JBits consists of about 4,600 lines of user code in Java.

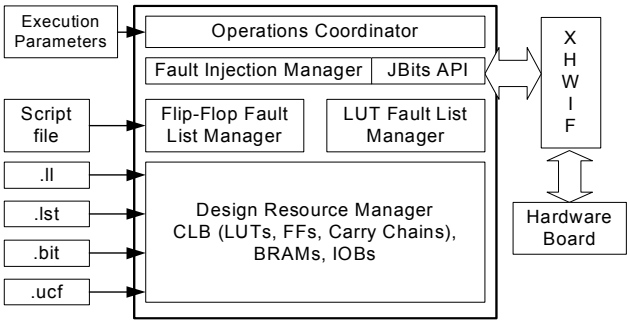


**Figure 2** Software/Hardware partitioning

Exploiting the portability of Java, our software can be executed in any processor hardware platform and operating system. Blocks in Figure 2 describe the basic programming entities implemented.

- The *input logic allocation file (.ll)* contains the human readable report of used resources in the FPGA device such as I/O pins, Flip-Flops, BlockRAMs, and their respective symbol names in the HDL code (or names assigned by the synthesizer). This file is generated by the Xilinx ISE software suite.
- The *input bitstream file (.bit)* contains the configuration information for the target FPGA device.
- The *input constraints file (.ucf)* contains information on hierarchical logic module placement and I/O pin assignments in standard text format.

- The *input symbol name list files (.lst)* contain the signal names of the design Flip-Flops. Such a list is generated by an external tool that we developed, using the logic allocation file. One list is associated with the monitored Flip-Flops and another one is associated with the SEU injection target Flip-Flops.
- The *execution parameters file* contains textual representation of parameters that instruct the fault emulator on how to perform its set of processes. Information contained includes the number of clock cycles to run, input files, board and device data, and fault injection/detection methods to use.
- The *input script file* is used for SEU emulation tests on Flip-Flops and contains the text commands that instruct the fault emulator to inject a SEU in a specific Flip-Flop, at a specific point in emulation time. The developed script language is capable of introducing SEUs and multiple upsets during a *Fault Injection Cycle*. This file is generated by an external utility that we developed, through the use of which the user can introduce a number of SEUs on a list of symbols (as described above) at user-specified, sequential or random times.
- The *Design Resource Manager* is responsible for analyzing the input bitstream file, extracting and creating hierarchical mappings of used FPGA resources.
- The *LUT Fault List Manager* is responsible for extracting information on LUT input usage and producing their fault vectors in a sorted hierarchical list. The fault list is generated exhaustively for all LUTs inside an HDL module defined in the execution parameters file or for the entire design.
- The *Flip-Flop Fault List Manager* is responsible for parsing the *input script file* and generating a list of the internal commands to orchestrate the fault injection in the defined Flip-Flops at the specified point of execution (emulation) time.
- The *Fault Injection Manager* is responsible for sequentially injecting a fault from the LUT/Flip-Flop Fault List (depending on the type of test) and performs the *Fault Injection Cycle*. After each cycle, the Fault Injection Manager processes the test results and determines whether the fault has been detected.
- The *Operations Coordinator* controls the flow of procedures according to the execution parameters file.

### 3.2. Hardware

The hardware portion of the fault emulator is polymorphic. Based on JBits, our software can connect to any FPGA board that has a standard set of the XHWIF interface [14] implemented. For this to be achieved, it is essential that the SelectMAP configuration I/O pins on the FPGA device are not used by the implemented circuit and reserved for the emulation link. The board can be either connected to the local workstation or on a remote computer. For our experiments, a XESS XSV800 board was used with a Virtex XCV800 device, however any other Virtex board could be used as long as the above requirements are met, thus allowing the use of available low-cost prototyping boards. Though XESS provides its own XHWIF port, it proves to be prohibitively slow, with a full device configuration lasting about 2½ minutes. To achieve better performance at a low-cost from the available FPGA board, two new XHWIF ports were implemented. In both implementations, the on-board CPLD connecting the parallel port and the FPGA had to be reconfigured to support the SelectMAP programming port interface. Being our intention to provide a board-independent solution, we have not implemented any special features that would be inapplicable to other boards.

The first XHWIF implementation makes use of a direct connection to the standard parallel port (SPP) of our host computer. Due to the unidirectional nature of the SPP, the readback function is performed over the 4bit status pins of the port, thus nearly doubling the required data transfer time. In an attempt to increase transfer rate, a cost-effective USB approach was adopted. An interface converter device was developed around an Atmel AVR microcontroller. This device provides a USB connection to the host computer and a modified 8bit bidirectional port connecting to the FPGA board. The microcontroller was programmed to implement a simple communications protocol to perform operations such as configuration, readback, reset, and clocking of the FPGA. Both XHWIF implementations provide a clocking facility of fixed frequency, with the USB outperforming the SPP control by an order of magnitude. Comparison of XHWIF implementations is given in Table 1.

| | XESS | SPP | USB |
|---|---|---|---|
| Full Configuration | 2½ min | 7s | 5.8s |
| Full Readback | 2½ min | 13s | 6.5s |
| Partial Reconf. (1 LUT) | 675ms | 32ms | 26ms |
| Download Rate | 3.84kB/s | 82kB/s | 99kB/s |
| Upload Rate | 3.84kB/s | 56kB/s | 88kB/s |
| Clocking (max freq.) | 4kHz | 260kHz | 2MHz |

**Table 1** Comparison between different XHWIF ports

## 4. Fault Injection and Detection

### 4.1. LUT Fault List Extraction and Injection Flow

To inject a fault, first a fault list needs to be available. Since we make use of no external information, we must extract LUT and LUT input usage using only the input bitstream file. In synthesized circuits, it is possible that some of the LUT inputs are not used for the implemented function; we avoid injecting faults in unused LUT bits since their presence does not interfere with the implemented logic function of the LUT.

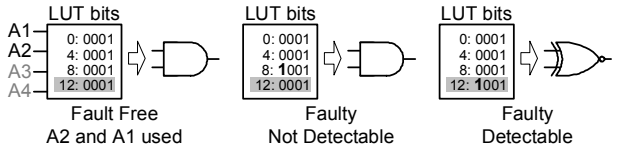


**Figure 3** SEU in used and unused LUT bit positions

If a fault is injected in an unused bit position due to the connected LUT inputs, though the fault is present, it does not affect the output of the LUT (Figure 3). The implemented function will remain the same, though a fault is present. To determine which of the four inputs are being used, we analyze the 16-bit LUT vector, as illustrated in Figure 4. First we transfer the fault free LUT vector in a 4x4 Karnaugh map; by looking at relationships between certain bit positions we can identify which of the four inputs of the LUT are being used. Consider the 16-bit vector as $L_{abcd}$, where *a*, *b*, *c* and *d* are the LUT inputs (A4, A3, A2 and A1 respectively). Assigning logic values to the LUT inputs, a bit value is returned, e.g. for $L_{1011}$ the $12^{th}$ bit of the LUT vector is returned. When the implemented logic function does not make use of all four inputs, only a small set of the 16-bit LUT vector is used (shaded grey in Figure 3). Any unused inputs are tied to $V_{cc}$. The unused section is an identical copy of the used one (see fault free LUT in Figure 3), which we will call *mirror image*. For *u* number of unused inputs, there will be $2^u$-1 mirror images. Symbolically this is represented as $L_{0000} \rightarrow L_{0111}$ being respectively equal to $L_{1000} \rightarrow L_{1111}$ when input *a* (A4) is not used.

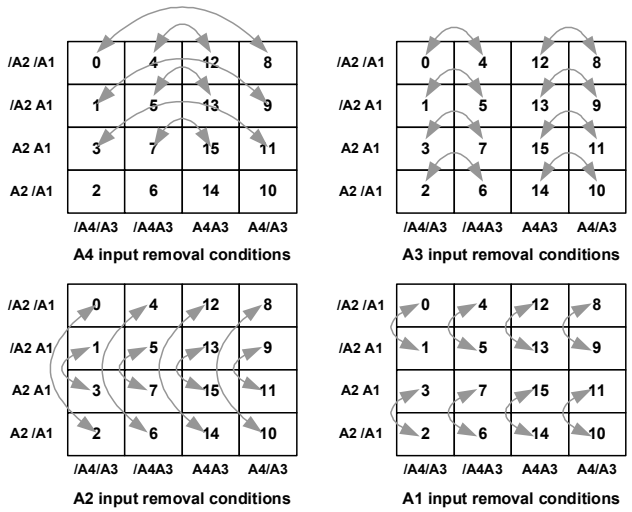


**Figure 4** Identifying unused inputs using Karnaugh maps

The synthesis tools are responsible for these assignments, which we exploit to identify the used LUT inputs. From Figure 4, a*ll* equality relationships marked with arrows between bit-positions must be satisfied to safely deduce that the corresponding input is not used (mirror image identification).

Using the Binary Decision Tree (BDT) of Figure 7 we identify the LUT bits participating in the fault injection process. This BDT is derived from the described mirror image assignments. We have encoded the tree so that the left hand side signifies an unused input bit (1), and the right-hand side signifies a used input bit (X). Fault emulation is a fairly simple sequential process; a *Fault Injection Cycle* consists of the steps depicted in Figure 5. Using the same algorithm, single or multiple upsets can be injected.

```
Configure device with original bitstream;
Pulse Clock N times;
FFR=test responses from fault free device;
A: For (L=0; L<#LUTs; L++) do
   LV ← fault free LUT vector of LUT L;
   B: For (V=0; V<#Faulty_Vectors; V++) do
      Pulse Circuit Reset;
      FV ← Fault_Vector_List[L][V];
      Pulse Clock N times;
      TR ← test responses from device;
      If (TR==FFR) then Fault_Detected ←false;
      Else Fault Detected ←true;
   End loop B;
   Inject LV in LUT L;
End loop A;
```

**Figure 5** LUT Fault injection cycle algorithm

## 4.2. Flip-Flop SEU Fault List Generation and Injection Flow

Fault emulation of SEUs in Flip-Flops unlike in the case of LUTs is a complex and time consuming process since the Virtex architecture is not designed to support direct manipulation of individual Flip-Flops; a simplified logic circuit of a CLB Slice Flip-Flop is given in Figure 6.

As in [8], in order for a Flip-Flop's state to be assigned a desired value, first the Set/Reset switch of *every used* Flip-Flop in the FPGA needs to be changed to represent the current logic state of the Flip-Flop. The S/R switch of the Flip-Flop to be injected with a SEU must carry the inverse value of the Flip-Flop's state (bit flip). Next, the circuit's RESET pin or the Global Set/Reset (GSR) internal signal must be pulsed to force the new reset values.

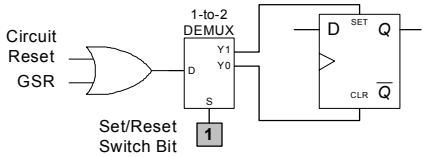


**Figure 6** Set/Reset Simplified Circuit of a CLB Flip-Flop

The process of injecting a SEU in a CLB Flip-Flop requires one readback and one reconfiguration phase. Additionally, in order to recover from the injected SEU a reconfiguration phase is required that will revert the SR switches to their original full-configuration bitstream state.

A *Fault Injection Cycle* consists of the steps depicted in Figure 8. Using the same algorithm, single or multiple upsets can be injected.

```
Configure device with original bitstream;
Readback Flip-Flop Original Start-Up State;
Pulse Clock Until Fault Injection Cycle End;
FFR=test responses from fault free device;
A: For (S=0; S<#Flip-Flop Sites; S++) do
  Pulse Circuit Reset;
  Pulse Clock Until Injection Time;
  Readback State of All Flip-Flops;
  Change S/R Switches of All FFs to match State;
  Invert S/R Switch of Target Flip-Flop;
  Pulse Circuit Reset or GSR;
  Pulse Clock Until Fault Injection Cycle End;
  TR ← test responses from device;
  If (TR==FFR) then Fault_Detected ←false;
  Else Fault Detected ←true;
  Change S/R Switches of All FFs to Original
    Start-Up State;
  End loop A;
```

**Figure 8** Design Flip-Flop Fault injection cycle algorithm

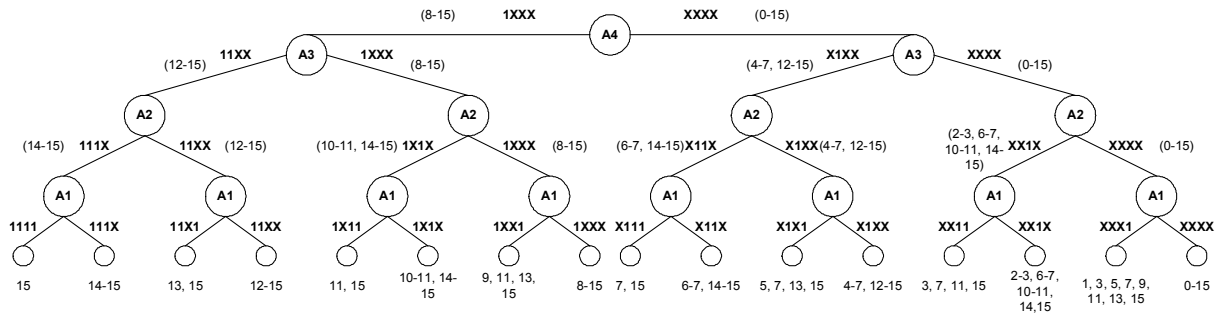


**Figure 7** Binary Decision Tree for selecting LUT used bit positions

The time required to reconfigure the population of Flip-Flops is dependent on their dispersion on the device, due to the organization of the Virtex architecture in frames [15], as well as the number of S/R switches that need to be reconfigured. To allow for faster reconfiguration/readback, the registers in the circuit should be confined within the smallest amount of CLB columns, thus minimizing the number of frames to be read back from the device or reconfigured. A change in future FPGA architectures addressing the described problem would provide the means for faster emulation processes.

### 4.3. Supported Fault Detection Scenarios

In this section we provide five possible fault detection scenarios supported by our fault emulator platform, each applicable to different type of *design under test* (DUT) implementation, as follows. The authors of this paper simply provide examples of use for the proposed fault emulation platform.

1. The *Memory Resident scenario*. This scenario is applicable to a class of DUTs that store their responses in BlockRAM. Thus, in this scenario (Figure 9a), the configuration bitstream contains the DUT, any user-defined TPG and control logic circuit, and sufficient amount of BlockRAM. The TPG produces the test patterns (by means of pseudorandom or deterministic hardware) that are fed to the DUT and the output of the circuit is stored in BlockRAM. An entire test is run for a defined number of patterns with a fault-free DUT. The test responses are read back from the BlockRAM and stored in the host computer's memory. Then, for each fault in the list being injected to the DUT, the test responses are read back and compared to those of the fault-free DUT (Figure 5). The software supports the "no fault dropping" attribute for diagnosis. At the beginning of each fault injection cycle, the BlockRAM needs to be erased to protect from erroneous results already residing in BlockRAM. This scenario is very time consuming in hardware since for each fault injection cycle it requires two partial reconfigurations (one to erase all BlockRAM cells used and one to inject the fault), and a partial readback of the used BlockRAM columns. However, this scenario allows for extensive investigation on the impact of each test pattern applied as input to the DUT. At the end of the emulation process, a list of each fault injected and the list of patterns detecting it is generated.
2. The *Golden Memory Responses scenario*. This is a modified version of the Memory Resident scenario for speed increase (Figure 9b) applicable to the same class of DUTs. The fault-free circuit will store its responses in BlockRAM and any subsequent fault injection cycles will only compare the responses of the DUT to the ones already stored in BlockRAM. An output pin will assert to indicate that a fault has been detected. The host computer can record the clock cycle at which the signal is asserted and stop the test prematurely. In this scenario, no BlockRAM configuration and readback is performed for each fault injection cycle. Since clocking is stopped as soon as the first faulty response is detected (premature fault injection cycle termination), this scenario is extremely fast.
3. The *Hardware Redundancy scenario*. This scenario is applicable to designs with duplication. The Golden DUT (GDUT) provides the fault-free test responses, while the Target DUT (TDUT) is injected with faults. A comparator is connected to the primary outputs of both DUTs to indicate the detection of a fault (Figure 9d). In terms of speed this scenario is equivalent to the *Golden Memory Results scenario*, without the need for the initial fault-free circuit execution of the latter. Premature fault injection cycle termination is supported.
4. The *Built-In Self-Test (BIST) scenario*. This scenario is applicable to a class of DUTs that are integrated with BIST circuits asserting an output signal to indicate the detection of a fault (Figure 9d). There is no requirement for BlockRAM reconfiguration/ readback which transfer large amounts of data between the host computer and the FPGA, as in the Memory Resident scenario; and since injecting a fault requires transferring a very small amount of data, this scenario is considerably faster than the previous two scenarios. Premature fault injection cycle termination is supported.
5. The *Register State scenario*. This scenario is applicable to a class of DUTs that store their responses in distributed Flip-Flops. In this scenario (Figure 9e) we assume that any injected fault will eventually appear in the Flip-Flops of the DUT. After executing a fault injection cycle, the state of the DUT's registers, or a small subset of these, is read back by the fault emulator and compared to that of the fault-free circuit. Besides fault injection (one partial reconfiguration), a series of partial read backs is required (according to the dispersion of Flip-Flops in the device, as mentioned previously). This scenario can be considerably slower than the BIST scenario, yet faster than the Memory Resident scenario depending on the number and topology of Flip-Flops used (restricted within the minimum possible amount of CLB columns). If full readback is required or dispersion is extensive, this scenario is slower than the Memory Resident scenario.

In all scenarios, the user can choose to inject faults in specific areas/ modules of the FPGA or in the entire implemented design, thus including any BIST circuit.

## 5. Experimental Results

### 5.1. LUT SEU Injection Experiments

For the LUT SEU investigation, several experiments were performed. A set of tests using the time consuming *Memory Resident scenario* (Figure 9a) were performed on a behavioral model of a 32x32 bit non-pipelined multiplier which was synthesized using the Xilinx Synthesis Tool (XST). This scenario was chosen to allow for the maximum level of observability in the circuit's test results and allow for direct comparison of our developed XHWIF ports with the VirtexDS simulator. The *Golden Memory Responses*, *Hardware Redundancy* and *Built-In Self-Test*

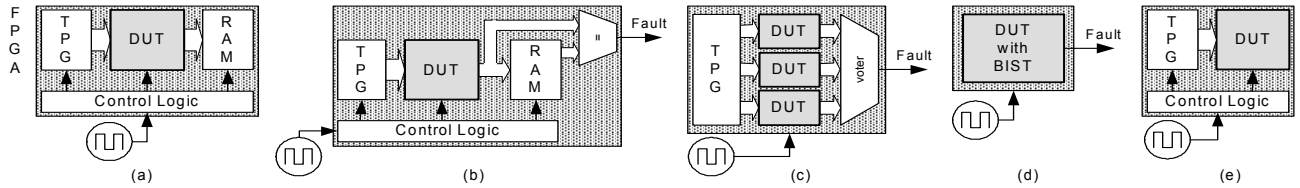


**Figure 9** Supported fault detection scenarios

*scenarios* would have been difficult to use for comparison purposes since VirtexDS would not allow the monitoring of combinational signal outputs. Furthermore, the *Memory Resident scenario* is a good metric for the effect of a slow communications bus in the injection process, which for the other scenarios would not be easily recognizable.

The TPG circuit used to feed the inputs of the multiplier is a 64bit maximal length LFSR. We made use of an FSM test controller requiring two clock cycles per test pattern. The 64bit result from the multiplier is stored in BlockRAM on the second clock cycle. After N number of clock pulses, N/2 patterns have been applied and their results stored. Three different sets of results were captured, one using VirtexDS and two using our XHWIF implementations (SPP and USB). Average times per fault injection cycle were measured (time measurements include BlockRAM erasure, fault injection, circuit clocking, BlockRAM readback and comparison with fault-free circuit responses).

As expected from a software simulator, VirtexDS did not scale well with increasing number of circuit execution clock cycles (Table 2). Contrary to VirtexDS where most time is spent in simulated circuit execution, the use of FPGA hardware emulation spends most of its time in communicating reconfiguration data to the FPGA. As seen through our experiments, hardware fault emulation achieved a speed-up of up to 53.55 times over fault simulation when using SPP. When the faster USB port was used a speed-up of up to 221.14 times was achieved.

| **LUTs=1189, Fault Locations=10297** | | **VirtexDS** | **SPP XHWIF** | **USB XHWIF** |
|---|---|---|---|---|
| Patterns | Fault Coverage | Test Time/Fault (s) | Speed Increase | Speed Increase |
| 64 | 93.95% | 18.3 | 8.71 | 42.56 |
| 100 | 98.68% | 25 | 11.90 | 55.19 |
| 128 | 99.12% | 30 | 14.29 | 62.76 |
| 256 | 99.14% | *59 | 27.96 | 121.40 |
| 512 | 99.14% | *113 | 53.55 | 221.14 |
| * average time estimated for 100 fault injection cycles | | | | |

**Table 2** LUT SEU injection experimental results

Use of any hardware-based fault detection scenario which removes the need for redundant data transfers dramatically decreases the required amount of time per fault injection cycle. In Table 3, times spent per operation in the Memory Resident scenario are analyzed. The above times are complemented with delays introduced by communications bus overhead, software comparison processing and Java code execution. It is apparent, that a faster configuration/readback facility would allow for faster fault injection cycles using the same scenario. Clocking the design does not contribute as much to the overall time as the other factors, while more than 85% of total fault injection cycle time is spent in BlockRAM read and write operations.

| **Hardware Operation** | **Time Spent (%)** |
|---|---|
| Erase BlockRAM | 40,2 |
| LUT Fault Injection | 7,2 |
| Execution | 0,1 |
| Upload BlockRAM | 45,2 |
| LUT Fault Scrubbing | 7,2 |

**Table 3** Times per operation in Memory Resident scenario

Comparing fault injection cycle times for few clock cycles gives a fair impression of the speed-up imposed by the use of hardware. At higher numbers of executed clock cycles, the advantage of fault emulation is far more evident with a speed increase of *two orders of magnitude*. Compared to the work of [8], we have allowed an average speed increase of 39 times in full device configuration using the same FPGA board when using the faster USB XHWIF implementation. This was made possible due to the faster communications bus which allows for faster device reconfiguration and readback, as well as the higher frequency of the system clock used to clock the FPGA.

When hardware based fault detection scenarios were used with the USB XHWIF port, we experienced a maximum fault injection cycle of **50ms**, with two LUT reconfiguration steps and one execution phase with fault injection cycle prematurely ending as soon as the fault is detected. Therefore, in realistic fault emulation experiments the use of hardware-based fault detection (*Golden Memory Responses*, *Hardware Redundancy* and *Built-In Self-Test scenarios*) would be of much more practical value.

## 5.2. Flip-Flop SEU Injection Experiments

For the Flip-Flop SEU investigation, a set of experiments were performed using the *Register State scenario*. The DUT used was a 32x32bit 4-stage pipelined multiplier generated by Xilinx's CORE Generator 6.2. Both inputs of the multiplier were connected to a 64bit maximal length LFSR, also generated using CORE Generator. The multiplier output was connected to a 64bit maximal length MISR. An injection target symbol list file (.lst) was created to include all Flip-Flop symbols used by the multiplier. Another symbol list file was created to include the MISR Flip-Flops, which the fault emulator would monitor for test results. The fault injection point in time for each Flip-Flop was randomly chosen by the script generator utility limited within the range of clock cycles per Fault Injection Cycle. All modules were constrained inside an area of 17 successive CLB columns, with module hierarchy preserved and each module occupying: LFSR (2 columns), multiplier (14 columns) and MISR (1 column). The MISR Flip-Flops were intentionally placed within one column to allow for faster access to the monitored Flip-Flop values. A total of 1,232 Flip-Flops were used by the implemented circuit, out of which, 128 were utilized by the LFSR and MISR modules, and the remaining 1,104 utilized by the multiplier. All experiments resulted in 100% fault coverage.

| **1,104 Flip- Flops** | **VirtexDS** | **SPP XHWIF** | **USB XHWIF** |
|---|---|---|---|
| Patterns | Test Time/Fault (s) | Speed Increase | Speed Increase |
| 64 | 9.48 | 1.75 | 4.10 |
| 128 | 10.65 | 1.96 | 4.59 |
| 256 | 13.25 | 2.44 | 5.71 |
| 512 | 18.26 | 3.35 | 7.87 |
| 1024 | 28.07 | 5.08 | 11.90 |
| 2048 | *44.30 | 7.88 | 18.47 |
| 65536 | *1235.40 | 204.30 | 467.95 |
| * average time estimated for 20 fault injection cycles | | | |

**Table 4** Flip-Flop SEU injection experimental results

As seen from the numerical results in Table 4, VirtexDS scales extremely poorly with increasing number of simulated circuit execution cycles. It is evident that processor intensive simulation of a few thousands or tens of thousands clock cycles for large sequential circuits would be impractical. For 64 test patterns applied to the DUT, the SPP XHWIF port achieves an average speed increase of 1.75 times over VirtexDS, while the USB

XHWIF port is 4.1 times faster than VirtexDS. For 65,536 test patterns, the SPP XHWIF port achieves a speed increase of 204.3 times, while the USB XHWIF port an increase of 467.95 times over VirtexDS. A speed increase of *three orders of magnitude* is practically achieved when transferring from simulation to emulation experiments. With larger circuit designs and higher number of test patterns, an even higher speed increase is achievable.

In each measurement taken using hardware emulation, 50-60% of the test time is spent in Flip-Flop state readback prior to fault injection. This step is extremely time-consuming since the state of *all used Flip-Flops* must be recorded in order to decide which S/R switches need to be reconfigured to an inverted value. However, this decision step allows for shorter reconfiguration times. Higher test speed performance from the USB XHWIF port is attributed to the higher upload transfer rate, compared to that of the SPP port (nearly twice as fast). Clocking the design has little impact on test time per fault. It is evident that much time is spent in communicating reconfiguration and readback data, thus for accelerated emulation experiments it is imperative that a fast communication bus is employed.

## 6. Conclusions

In this paper we have presented a low-cost FPGA fault emulator platform that efficiently performs emulation experiments for evaluation of fault detection and test pattern generation techniques, without the need for expensive and harmful radiation tests or custom hardware. Any Virtex-based FPGA board can be employed without the need for modifications in the software portion. The only requirement is that an XHWIF interface port for the board, using the SelectMAP configuration port, is available or developed.

We have targeted our work in SEU faults affecting the LUT configuration bits as well as the design Flip-Flops. A set of five supported non-intrusive fault detection scenarios was presented for researchers and test engineers to evaluate for use with their applications. We have compared a software simulator and two low-cost hardware implementations with slightly different features based on the same FPGA device and development board. We have also compared fault coverage, fault simulation (using VirtexDS) and fault emulation times for combinational circuits, as well as for sequential circuits. A speed increase of two orders of magnitude has been witnessed for combinational circuits when comparing fault emulation to fault simulation. Moreover, a speed increase of three orders of magnitude was experienced with sequential circuits. A minimum Fault Injection Cycle duration of 50ms per fault in LUT configuration bits was experienced when using our faster supported scenarios. It has become evident that the transition from a fault simulator to a fault emulator environment shifts the problem space from that of processing power to that of communications bandwidth.

In future works we will attempt to include more configuration bits in the fault injection process. Routing is the best candidate since it constitutes the majority of bits in the configuration bitstream.

## References


[1] M. Caffrey, P. Graham, E. Johnson, and M. Wirthlin, “Single-Event Upsets in SRAM FPGAs”, MAPLD International Conference, Laurel MD, USA, 2002.

[2] M. Rebaudengo, M. Sonza Reorda, M. Violante, “Simulation-based analysis of SEU effects of SRAM-based FPGAs”, FPL2002: International Conference on Field Programmable Logic and Application, pp. 607-615, 2002.

[3] M. Ceschia, M. Violante, M. Sonza Reorda, A. Paccagnella, P. Bernardi, M. Rebaudengo, D. Bortolato, M. Bellato, P. Zambolin, and A. Candelori, “Identification and Classification of Single-Event Upsets in the Configuration Memory of SRAM-Based FPGAs”, IEEE Transactions on Nuclear Science, Vol. 50, No. 6, pp. 2088-2094, Dec 2003.

[4] E. Johnson, M. Caffrey, P. Graham, N. Rollins, M. Wirthlin, “Accelerator Validation of an FPGA SEU Simulator”, IEEE Transactions on Nuclear Science, Vol. 50, Issue 6, pp. 2147- 2157, Dec 2003.

[5] M. Alderighi, S. D’Angelo, M. Mancini, G. R. Sechi, “A Fault Injection Tool for SRAM-based FPGAs”, Proceedings of the 9th IEEE International On-Line Testing Symposium, IOLTS, pp. 129-133, 2003.

[6] S. McMillan, B. Blodget, and S. Guccione, “VirtexDS: A Virtex Device Simulator”, SPIE 2000.

[7] A. Parreira, J. P. Teixeira, M. B. Santos, “Built-in self-test preparation in FPGAs”, In Proc. Of the 7th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems, pp. 83-90, Apr 2004.

[8] L. Antoni, R. Leveugle, B. Feher, “Using Run-Time Reconfiguration for Fault Injection Applications”, IEEE Transactions on Instrumentation and Measurement, Vol. 52, pp. 1468-1473, Oct 2003.

[9] P. Bernardi, M. Sonza Reorda, L. Sterpone, M. Violante. “On the Evaluation of SEU Sensitiveness in SRAM-Based FPGAs”, 10th IEEE International On-Line Testing Symposium (IOLTS'04), p. 115, 2004.

[10] F. Lima Kastensmidt, L. Sterpone, L. Carro, M. Sonza Reorda, “On the Optimal Design of Triple Modular Redundancy Logic for SRAM-based FPGAs”, Design, Automation and Test in Europe (DATE'05), Volume 2, pp. 1290-1295, 2005.

[11] M. Aguirre, J.N. Tombs, F. Muñoz, V. Baena-Lecuyer, A. Torralba, L.G. Franquelo, “A Hardware Approach for SEU Immunity Verification using Xilinx FPGA's”, DCIS, XIX Design on Circuits and Integrated Systems Conference, pp. 479-484, Nov 2004.

[12] S. A. Guccione, D. Levi, and P. Sundararajan, "JBits: A Java-based interface for reconfigurable computing," in Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD), Laurel, MD, Sept 1999.

[13] S. Guccione, S. McMillan, and P. Sundararajan, "Testing FPGA Devices using JBits," 4th Military and Aerospace Programmable Logic Devices (MAPLD) International Conference, Laurel, Maryland, Sept 2001.

[14] P.Sundararajan, S.Guccione, D.Levi, “XHWIF: A portable hardware interface for reconfigurable computing”, Proc. of Reconfigurable Technology: FPGAs and Reconfigurable Processors for Computing and Communications, SPIE 4525, pp. 97-102, Aug. 2001.

[15] Xilinx Inc., “Virtex Series Configuration Architecture User Guide”, Xilinx Application Note XAPP151, Version 1.7, Oct 2004.