Microelectronics Journal 123 (2022) 105392

Contents lists available at ScienceDirect

Microelectronics Journal

journal homepage: www.elsevier.com/locate/mejo

The Characterization of Errors in an FPGA-Based RISC-V Processor due to Single Event Transients

Jhalak Sharma *, Nanditha Rao

International Institute of Information Technology, Bangalore, 560100, India

ARTICLE INFO

Keywords:
Fault injection RISC-V Processor MBU
SET
Simulation
FPGA Implementation

ABSTRACT

The reliability of processors is a crucial aspect of processor design. In this study, we design a fault injection framework based on Single Event Transients (SET) to understand its impact on an FPGA-based processor. The key objective of this paper is to integrate the framework with the processor and to study the probability of Multiple Bit Upsets (MBUs) due to the SET. To demonstrate the framework, we choose PicoRV32: a RISC-V based processor implemented on a Xilinx Artix-7 FPGA. We follow a two-step process: RTL Simulation-based fault injection and FPGA-based fault injection. Firstly, we perform a Monte-Carlo based RTL simulation of the processor with the fault injection controller integrated with the processor. We allow the error to propagate and identify the probability of MBUs. As a next step, we use this information to implement the fault injection controller for the FPGA implementation. We observe that the probability of the MBUs (more than 1 bit-flip) is nearly 15.23% on average and 96.8% of the total faults injected propagated to registers. Flip-Flop (FF) is found to be the most vulnerable component in the FPGA (79.1% of resources are mapped to FFs). We also observe that the impact of a fault can last across several clock cycles and is heavily dependent on the instruction being executed by the processor.

1. Introduction

Reliability of processors has become a very essential aspect in providing correct system functionality. SRAM-based FPGAs can provide a vast variety of programmable re-sources such as Look Up Tables (LUTs), FFs, registers, I/o, DSPs and Block RAM (BRAM). FPGAs implementing processors can provide greater flexibility in optimizing the design. On the contrary, ASIC counterparts do not provide the reprogrammability feature that leads to expensive and infeasible design updates. Radiation found in both space and terrestrial environments can prove hazardous to SRAM-based FPGAs and the processors implemented within. If a charged particle impacts at or near a transistor junction, the collected charge can induce an upset to the state of that transistor. If the collected charge is larger than the critical charge (critical charge is the minimum induced charge disturbance needed to change the logic level) of the element, it can flip the stored state [16]. This change in state (or bit-flip) is referred to as a Single Event Upset (SEU). All submicron integrated electronic devices are susceptible to SEUs to some extent. The effects can range from transients causing logical errors, to upsets changing data, to stuck-at faults [15]. Further, with decreasing feature size, the total amount of memory bits per device has increased significantly, and therefore, it is important to study the impact of SETs. (see Table 5)

Fault injection is a useful, low-cost method for understanding the soft error failure models of an FPGA-based system [10]. Several tools and techniques have been introduced for SEU estimation and mitigation on the processors [7,9]. Authors in Ref. [14] present a fault-tolerant version (SHAKTI- F) of the base class processor. Authors in Ref. [18] evaluate the RISC-V Rocket processor embedded in an SRAM-based FPGA under heavy ion induced faults. In Ref. [5], they present a low-cost fault tolerant implementation of the RISC-V architecture. They employ physical and information redundancy to reduce error propagation. In Ref. [3], a scheme that leverages the inherent redundancy created by the FPGA automation tools is presented. Authors in Refs. [2,6,8,10–12] present fault injection in configuration memory of the soft processors. Upsets in the configuration memory bits can affect the routing between nodes and cause functional failures in the user designs. Injecting error in configuration bits involves artificial modification of bitstream and programming the faulty bitstream on the FPGA. This method requires re-programming of the FPGA for every fault injected, that is very time

* Corresponding author.
E-mail address: jhalak.sharma@iiitb.org (J. Sharma).

https://doi.org/10.1016/j.mejo.2022.105392
Received 7 October 2021; Received in revised form 10 February 2022; Accepted 10 February 2022
Available online 15 February 2022

consuming. There is relatively less work on the topic of fault injection in user memory bits or programmable logic on the FPGA. In this paper, we focus on injecting fault in user memory bits.

In a prior work [3], we designed and demonstrated a fault injection framework for an FPGA based design implementing the ISCAS'85 benchmark circuits. We observed that the probability of Multiple Bit Flips (>3 bit upsets) was nearly 29.74% on an average in most circuits and the Flip-Flop was found to be the most vulnerable component in the FPGA. In this paper, we extended the framework to implement a fault injection controller that can be integrated along with the processor's logic or in the user bits of the FPGA. We use a RISC-V based processor as an example to demonstrate our approach using both (a) RTL simulation and (b) FPGA implementation. Our objective is to study the Multiple Bit Upsets (MBU) in the processor when implemented on the FPGA. We assume that the errors that do not get overwritten or masked would have a possibility to propagate to the out-puts. Therefore, we focus on arriving at the probability of such errors propagating to multiple outputs.

Firstly, we perform a Monte-Carlo based RTL simulation of the processor with the fault injection controller integrated with the processor. The analysis selects a random error site (register or memory) and enables the fault (which lasts for just one clock cycle) on one of its bits at a random clock cycle. We allow the error to propagate and identify the probability of MBUs. We identify the nodes which caused a majority of flips in the processor and then use this critical node information to implement the fault injection controller for the FPGA implementation. The fault injection controller is a state machine which is integrated with the processor on the FPGA and runs without any manual intervention. The RTL node screening methodology helps us to narrow down the total number of simulations we need to perform finally on the FPGA and helps reduce the total time required to perform the entire analysis. We observed that running an entire Monte-Carlo simulation on hardware with nearly 10000 simulation decks would have taken a few weeks to complete. Observing the fact that it was nearly impossible to do just a completely hardware based simulation for all the nodes, we arrived at this approach to screen out the non-critical nodes through behavioral simulation. We automate the entire process using python and TCL.

We also note that the critical nodes (causing higher number of multiple flips) screened through the RTL simulation need not always remain critical in the FPGA implementation, since the netlists in the two approaches are different. In our prior work [3], we performed a comparison of the critical nodes from RTL and those from the hardware simulation. We observed that the critical nodes obtained from behav-ioral simulation and the FPGA simulation matched within a 95% confidence interval. Thus, we implemented the same approach in this work on the RISC-V based processor. In the current work, we integrated the Monte-Carlo based fault injection framework on PicoRV32, a Verilog-based processor on the Artix-7 FPGA. We observe that the probability of MBUs (more than 1 bit-flip) is nearly 15.23% on an average and 96.8% of the total faults propagated to registers on the FPGA. Flip-Flop (FF) is found to be the most vulnerable component in the FPGA (79.1% of the FPGA components are mapped to FFs). We characterize the fault propagation across ten clock cycles after the fault was injected. An interesting observation is that the impact across several clock cycles is dependent heavily on the instruction being executed by the processor.

We organize the rest of the paper as follows. In section 2, we provide the proposed fault injection methodology, with three subsections briefly explaining the SET model and experimental setup we used, golden reference generation, and generation of fault injected simulation decks. We present our results in section 3. Subsection 3.1 discusses the statistics of interest from the fault induced on the FPGA, subsec-tion 3.2 describes the bit-flip statistics from simulation, similarly, subsection 3.3 describes the bit-flip statistics of the FPGA implementation of PicoRV32 on Artix-7 FPGA. We present the dependence of fault propagation on the instruc-tions and the impact of a single fault induced across various subsequent clock cycles in subsection 3.4 and 3.5. Lastly, subsection 3.6 describes the dominant fault sites observed on the FPGA fabric. In Section 4, we conclude the paper.

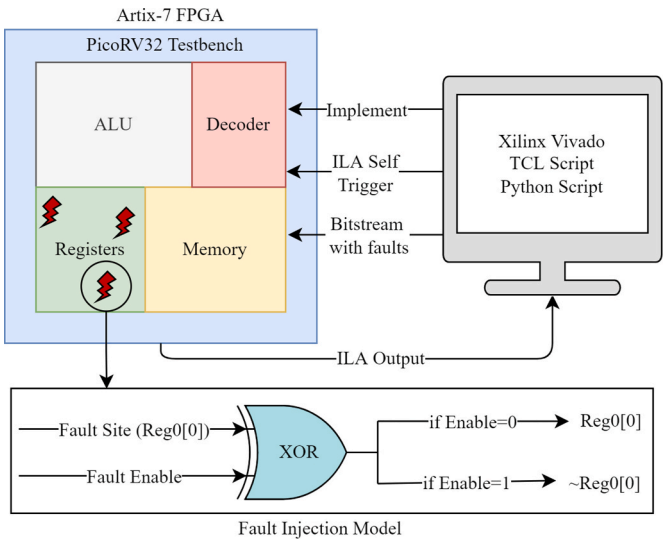


Fig. 1. Experimental setup of fault injection on processors.

2. Proposed fault injection methodology

PicoRV32 is a Verilog-based processor written by Clifford Wolf [17]. It is a CPU core that implements the RISC-V RV32IMC Instruction Set. It uses 9643 LUT, 1596 LUTRAM, 19641 FF, 41 BRAM, a total of 15,88,060 user bits.

We propose a fault injection framework to inject SETs on the FPGA-based processor design. The proposed fault injection methodology is divided into two steps: RTL Simulation based fault injection and FPGA-based fault injection. Firstly, we perform a Monte-Carlo based simulation of the processor with the Fault Injection Controller (FIC) integrated with the processor. The analysis selects a random error site (register/memory) and enables the fault on of its register bits at a random clock cycle. The fault is modeled as a signal that goes high for just one clock cycle, enabling the target node to flip. We allow the error to propagate and then identify the memory locations which flip. We then derive the statistics regarding the probability of MBUs. As a next step, the above information is used for implementing a fault injection controller for the FPGA implementation. Both these steps are automated with python and TCL respectively. We evaluate the probability of an SET propagating and causing MBUs, we study the impact across several consecutive clock cycles, and identify the dominant fault sites.

FPGA-based designs have a unique way of propagating errors due to their inherent design fabric which is based on LUTs, multiplexers, and Flip-Flops. Therefore, we note that the critical nodes (causing higher number of multiple flips) screened through the RTL simulation need not always remain critical in the FPGA implementation, since the netlists in the two approaches are different. In our prior work [3], we performed this comparison and observed that the critical nodes obtained from behavioral simulation and the FPGA simulation matched within a 95% confidence interval. We found that, on an average, 93.77% of the critical nodes from behavioral simulation remained critical in the FPGA implementation, mainly because they were on the critical path and branched out to multiple outputs. With this assumption, we implement the RISC-V processor, namely PicoRV32 on a Xilinx Artix-7 FPGA [1] through the Xilinx Vivado Design Suite 2019.1. The experimental setup is explained in detail in this Section.

Table 1
Test program executed on FPGA.

memory [0] = 32′h 0dc00093;	li	x1,220
memory [1] = 32′h 0000a223;	sw	x0,4 (x1)
memory [2] = 32′h 0040a103;	lw	x2,4 (x1)
memory [3] = 32′h 00410113;	loop: addi	x2,x2,4
memory [4] = 32′h 0020a423;	sw	x2,8 (x1)
memory [5] = 32′h 02010193;	addi	x3,x2,32
memory [6] = 32′h 06312623;	sw	x3,64 (x2)
memory [7] = 32′h 05018213;	addi	x4,x3,80
memory [8] = 32′h 08422023;	sw	x4,128 (x4)
memory [9] = 32′h 402202b3;	sub	x5,x4,x2
memory [10] = 32′h 0052a623;	sw	x5,12 (x5)
memory [11] = 32′h 0a000313;	li	x6,160
memory [12] = 32′h 00632423;	sw	x6,8 (x6)
memory [13] = 32′h 00832383;	lw	x7,8 (x6)
memory [14] = 32′h 00c10113;	addi	x2,x2,12
memory [15] = 32′h 00418193;	addi	x3,x3,4
memory [16] = 32′h 00820213;	addi	x4,x4,8
memory [17] = 32′h 00128293;	addi	x5,x5,1
memory [18] = 32′h 00530313;	addi	x6,x6,5
memory [19] = 32′h 00338393;	addi	x7,x7,3
memory [20] = 32′h fbdff06f;	j< *loop* >	

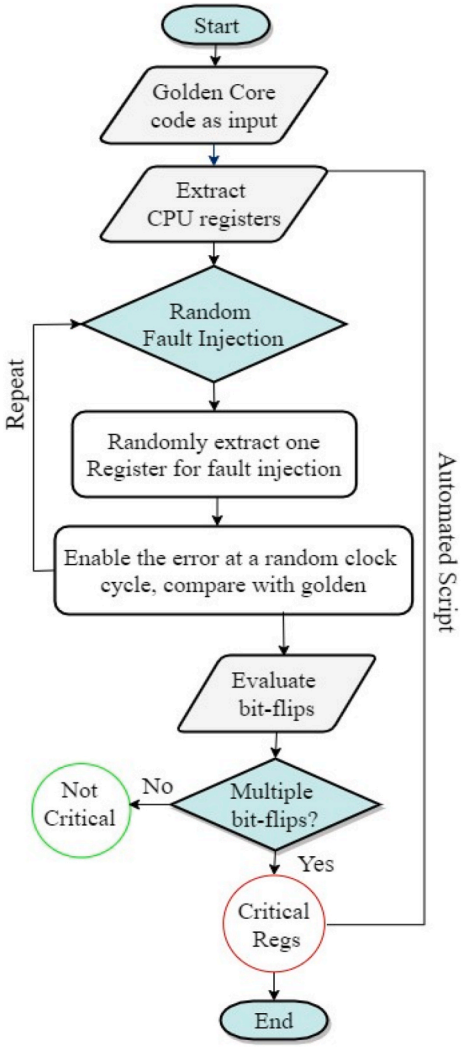


Fig. 2. Fault Injection Flow in Simulation & FPGA Imple-mentation.

2.1. SET model and fault injection setup

Our SET fault model is a simple single bit-flip model (Fig. 1) which is modeled as an XOR operation, with one of its inputs being a fault enable. When the fault enable is high, the corresponding bit in the register flips. These faults can occur in several different components of the processor, such as the registers, memory-locations, program-counter, ALU, decoder, and other critical components. It is important to note that the fault is enabled for a single clock cycle and is then disabled. Our SET or error model, although it spans an entire clock cycle, its impact is different for a strike on a register and strike on any other component such as a LUT, memory or a multiplexer. An SET (spanning an entire clock cycle) on a register can flip its content immediately, but a similar fault on the LUT need not result in an observable error. We have experimented with smaller SET durations such as half clock cycle. But the simplest synthesizable SET model which could latch the error was the one in which the SET was enabled for an entire clock cycle. Hence, we use this SET model throughout this paper.

2.2. Golden reference trace generation

We simulate the PicoRV32 RISC-V processor core which is described in Verilog Hardware Description Lan-gauge with a suitable testbench. The simulation is performed for several thousand clock cycles. The RISC-V core is running a set of instructions through the testbench, which involves several loads, stores from/to memory, and several ALU-related operations, branch instructions, and so on, continuously in a loop. A part of the testbench is shown in Table 1. While this program is running on the core, we note the values of register inputs, register outputs, and memory contents at the rising edge of every clock cycle. These are referred to as the golden reference traces, which will be used as a reference base to compare against the fault-injected simulation outputs.

2.3. Generating fault-injected simulation decks

The processor core in Verilog is used as a template from which we generate several fault-injected simulation decks. The entire process is summarised in the flowchart in Fig. 2. To generate a fault-injected simulation deck or netlist, we choose a random CPU register “r” to inject a fault (in one of the bits of the register) and a random clock cycle “t” to enable the fault in the XOR gate flip-model. Meanwhile, the processor core is running the test program continuously in a loop [Table 1], and therefore, the inputs to the fault-injected simulation deck are automatically provided by the testbench. We use a uniformly distributed random number generator approach to generate the (r, t) for our experiments. Every fault-injected simulation with a unique (r, t) is taken through the process of synthesis, implementation, and bit-stream generation to program it on the FPGA. The output values of the registers and memory locations are captured through the Integrated Logic Analyzer (ILA). In order to automate this entire process, we used python and TCL. The ILA had to be re-initialised and triggered for every fault injection.

To automatically trigger the ILA, we enabled the ILA self trigger feature to pre-arm the trigger immediately after device startup. The ILA trigger feature contained all register settings to “stamp” back onto the already implemented netlist, thereby injecting the fault. A new bit-stream was then automatically created with the self-trigger feature to inject the fault for each Monte-Carlo run. This entire process was time-consuming and took nearly 8–12 min. As part of the efforts to optimize the simulation time, we identified the memory locations which were more susceptible to the fault injected and reduced the number of ILA probes accordingly, thereby reducing the overall resources and the run-time.

Further, to probe all the required registers and memory location, we needed at least 270 (256 memory locations, 7 registers, clock, reset and a few more signals) ILA probes. The ILA probes would consume Block RAM resources on the FPGA. Therefore, not all outputs could be

Table 2
Steps involved in automating a fault injection run on FPGA. Steps 2 through 16 are repeated for every fault injected.

1. Starting the GUI at the beginning
2. Run Synthesis
3. Run Implementation
4. Generate bitstream
5. Program bitstream on the FPGA
6. Programming the FPGA
7. Displaying ILA waveform
8. Setting ILA trigger
9. Write ILA trigger file
10. Open implemented design
11. Apply the trigger at startup file
12. Write trigger at startup bitstream
13. Generate bitstream and program FPGA.
14. Displaying ILA Self Triggered Output Data
15. Programming Trigger at Startup Bitstream
16. Export ILA data to CSV

captured in a single simulation run for a particular (r,t) value. So, we had to also split a single run into multiple simulation runs to capture all results. The automated process involved in performing one fault injection simulation on the FPGA is shown in Table 2. Steps 2 to 16 are repeated for each fault injection, that is, for each (r,t) value.

A single fault-injected simulation with a unique (r, t) is performed for several thousands of clock-cycles. Once the fault is injected at clock cycle "t", the outputs of all regis-ters and memory locations are probed on the ILA and stored. The entire set of Monte-Carlo simulations in the Vivado Design Suite is automated in TCL, to create nearly 2000 unique fault-injected versions of the processor.

The fault-injected output values are compared with the "golden" reference values (when the processor did not have any faults injected). The results are categorized into two categories: (a) the error propagated and is observable at the output or (b) the error got masked. We note that the errors that do not masked would propagate to the outputs. Therefore, we focus on calculating the probability of such errors propagating to multiple outputs. The propagated errors are then mapped to the FPGA resources to identify the output registers or erroneous memory locations (flipped) due to the injected fault.

3. Results

As mentioned earlier, we perform a simulation-based fault injection and an FPGA-based fault injection. For the FPGA version, we implement the PicoRV32 core on a Xilinx Artix-7 FPGA through the setup as shown in Fig. 1. Each FPGA injected simulation as part of the Monte-Carlo setup took nearly 8–12 min to go through the entire design process and generate results. The area overhead of the fault injection circuitry is only 0.19% (0.53% more LUTs and 0.02% more Flip-Flops than the original PicoRV32 core). We present both the simulation results and the FPGA results in this section.

3.1. Statistics of interest from the faults induced on the FPGA

We expect to extract the following results or address the following questions from this set of experiments:

1. What is the probability of MBUs (registers or memory locations) as a result of the SET injected on the FPGA?

3.2. Bit-flip statistics from simulation

We performed the simulations (automated in Python) on behavioral Verilog code of PicoRV32 RISC-V core, using Icarus Verilog and GTKwave. The fault injection circuitry is integrated with the PicoRV32 core. A new set of instruc-tions were written using Jupiter [4], an open-source RISC-V assembler and run-time simulator to be run on the processor. The instructions such as load, store, ALU-based instructions, and branch instructions were included as part of the representative testbench. For each random fault injected on a random register, we compare the fault-injected output with the fault-free (golden) reference value to determine if the output register flipped.

We observe that a majority of the faults propagate to cause observable single bit-flips at the output as shown in Fig. 3 (a) and (b). The entire fault injection controller was automated with python such that it creates a separate sim-ulation deck for each combination of random register and random clock cycle (r, t). We ran nearly 2000–3000 fault-injected simulations. The outputs from the simulation (registers or memory locations) are captured and compared with the golden reference to generate bit-flip statistics. As a result of the SET injected on a certain CPU register, we noticed that the majority of flips are single bit-flips (62% on an average). For instance, the % of bit-flip for a particular case is calculated as equation [1]:

$$\%Zero\ Flips\ in\ Reg1 = \frac{No.of\ Zero\ Flips\ in\ CPUreg1}{Total\ \ Faults\ on\ CPUreg1} \quad (1)$$

3.3. Bit-flip statistics from the FPGA (hardware) implementation

We now present the bit flip statistics from the FPGA hardware implementation of the same processor . In Fig. 3 (b), we present the

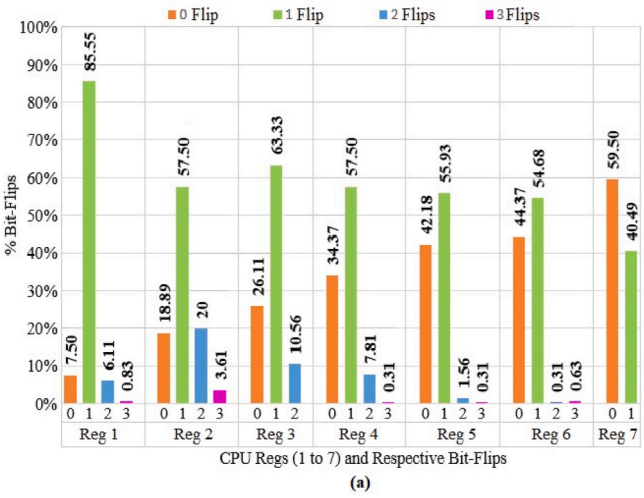


Fig. 3. (a) Simulation (b) FPGA Implementation: Fault injected on CPU registers and corresponding bit-flip statistics.

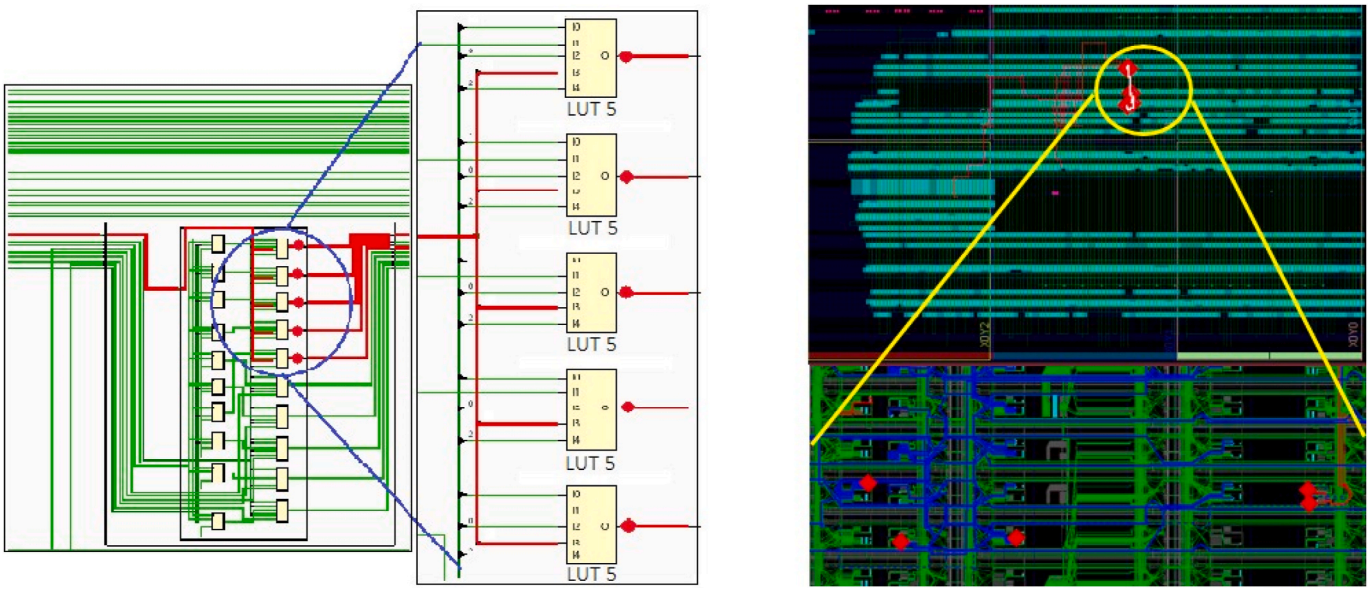


Fig. 4. PicoRV32 showing one error causing 5 Bit-flips (a) Schematic (b) Layout.

2. What are the dominant fault sites or critical regis-ters which resulted in the maximum number of flips? Is there any correlation between their position in the FPGA layout and the faults they cause?
3. Identify the type of FPGA resources mapped to these critical registers to get an understanding of how the FPGA fabric is contributing to the fault propagation. This is a crucial step that differs from a corresponding ASIC design approach.
4. In a processor, up to how many clock cycles can a fault propagate, and what are the reasons for its propaga-tion?
5. Does an error get masked or remain unmasked for sev-eral clock cycles. What are the reasons for this phe-nomenon?

Table 3
Statistics for bit-flip probability in FPGA implementation.

CPU Reg	Mean	Std Dev	95% Confidence Interval
1	1.1281	0.6602	[1.1276, 1.1286]
2	1.1251	0.9070	[1.1231, 1.1269]
3	0.8957	0.5373	[0.8953, 0.8961]
4	0.8535	0.5005	[0.8531, 0.8538]
5	1.1455	0.6869	[1.1449, 1.1460]
6	0.9194	0.5505	[0.9190, 0.9199]
7	0.9478	0.5424	[0.9474, 0.9483]

Table 4
Table indicating the clock cycle number in which the fault was injected on a CPU register & the instruction which was executing in that clock cycle. This table is for the scenario which resulted in **greater** than two output flips.

Clock Cycle	Instruction at memory location	Fault enabled on CPUreg	Register bit which is flipped
370	9	1	16
590	13	1	10
370	9	1	16
710	7	1	7
1010	19	2	3
100	9	3	31
100	9	5	4
370	9	5	27
100	9	5	6
890	7	6	13
700	15	7	12

percentage of bit-flips that propagated to either registers or memory locations, as a result of the fault being injected on one of the CPU reg-isters (CPU Register 1 to 7). Note that, we did not consider a fault in-jection on CPU Register 0, since its value is always zero. The SET is modeled as a single bit-flip in one of these registers. We ob-serve that the majority of flips were single bit-flips, which means that they propagated and flipped a single register or memory element. We show one such ILA

Table 5
Table indicating the clock cycle number in which the fault was injected on a CPU register & the instruction which was executing in that clock cycle. This table is for the scenario which resulted in **less** than two output flips.

Clock Cycle	Instruction at memory location	Fault enabled on CPUreg	Register bit which is flipped
800	7	1	0
790	15	1	11
900	17	1	15
520	15	1	17
570	11	2	1
760	3	3	10
190	9	3	26
510	5	3	13
860	13	4	10
380	19	4	1
300	11	5	10
750	11	5	15
330	5	6	14
110	19	7	13

waveform cap-tured on hardware after programming the FPGA, in Fig. 6. The waveform shows the fault duration, and the incorrect output being captured on CPUreg [7] and memory [2]. On an average, about 8% of the faults propagated and flipped mul-tiple registers or memory elements. Most reliable processor designs implement only single-bit correction schemes, and therefore the multiple bit errors may go un-corrected and re-sult in unexpected outputs in processors.

In Fig. 4 (a) we show the schematic of a part of the PicoRV32 pro-cessor with fault injected at one register. It in-dicates that it is connected to five other registers, which are eventually getting flipped. Similarly, Fig. 4 (b), shows the layout of the implemented PicoRV32 core on the FPGA. The red highlighted spots indicate that a single fault injected on a register is propagating to 5 different FPGA resources. We extract the statistics of interest such that they fall within the 95% confidence in-terval as shown in Table 3 The statistics are generated using JMP 15, a tool for Data Visualization and Exploratory Data Analysis [13]. We

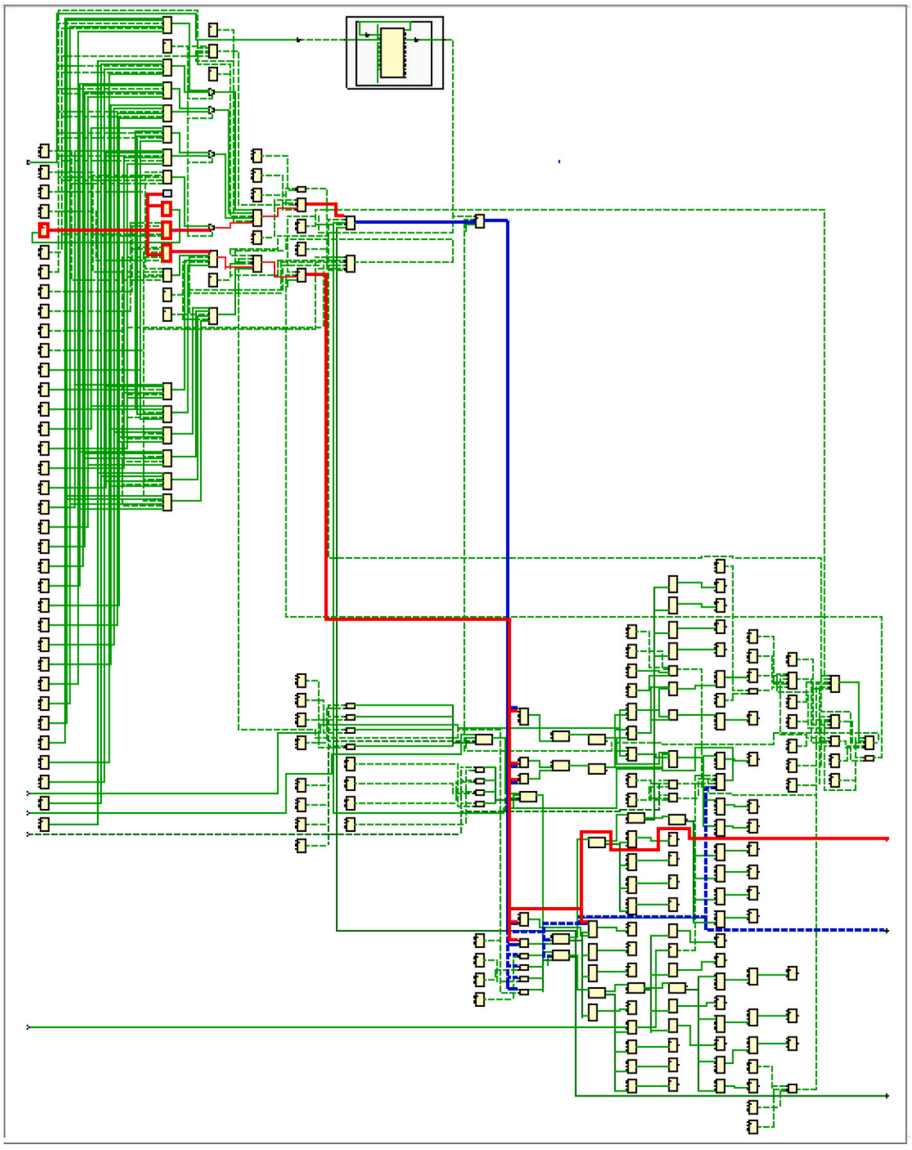


Fig. 5. Schematic indicating fault injected at CPUreg 5 and corrupting 2 outputs.

checked the logic depth from the critical node to outputs, for the scenarios that resulted in multiple output flips. When a fault was injected on a register, it propagated to multiple outputs even if it was 2–3 logic-levels (depth) away from the output. So, the propagation depth was 2–3 logic levels from the critical node, on an average. We show one such schematic in Fig. 5. This schematic shows that the SET injected flip-flop is at least 3 logic levels away from the two output registers which it eventually flipped. A detailed analysis on the logic depth would be sophisticated as it involves inspecting the post-implementation netlist or the schematics for each sim-ulation run.

3.4. Error propagation across multiple clock cycles and unmasked errors

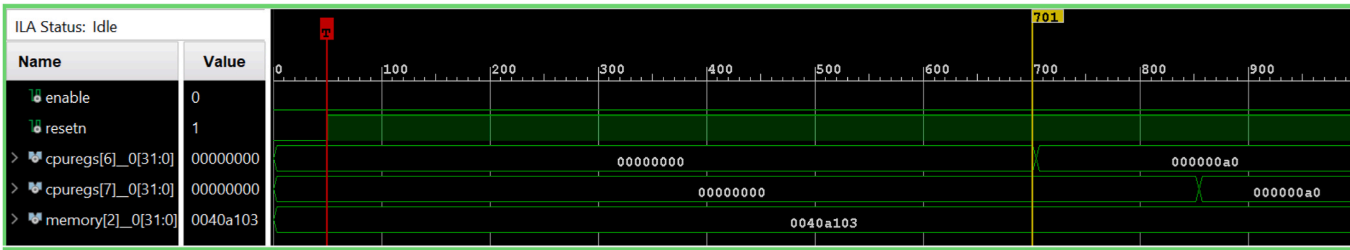
In the FPGA implementation, we observed the outputs of registers and memory locations for the subsequent 10 clock cycles after the fault was enabled. We intended to observe the number of clock cycles across which the fault would propagate to the registers (or memory) to get an observable error.

We also intended to note if any errors were remaining unmasked (remained as an error) even after the fault was dis-abled. We observed the following different patterns of fault propagation across clock cycles:

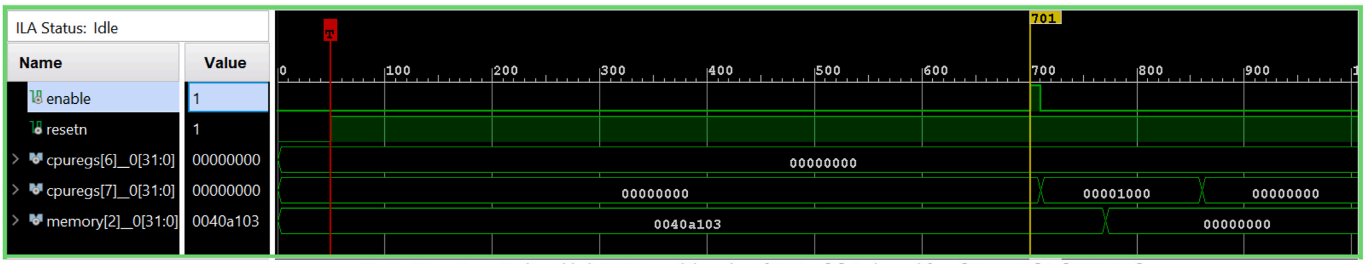
1. Scenario 1: The flipped value returned to its original value after a few clock cycles (after the fault was dis-abled).
2. Scenario 2: The fault propagated to more registers or memory location(s) in subsequent clock cycles.

We observe the percentage of flips observed in the reg-ister and the memory elements, as a result of the fault being injected on the CPU register (1–7) respectively. On an av-erage, 96.8% of the total faults injected propagated to CPU registers, while only 3.2% propagated to memory locations. This could be a direct result of the type of instructions we execute on the processor.

Fig. 8 shows the impact of the fault across 10 clock cycles. We observe that majority of the faults injected were “observable” at a register or a memory location in the very next clock cycle. We can explain our observation as follows. Suppose that the processor is running an “ADDI x2, x2, 4” instruction at the time of the fault injection on register x2. The erroneous value in register x2 is used for the compu-tation of ADD, which gets reflected as an erroneous value in the output register x2 in the very next clock cycle (the processor generates the results in one clock cycle). This ex-plains the reason for observing the faults in the subsequent cycle. The erroneous x2 continues to remain erroneous un-



Golden Waveform: Fault is disabled and the original value of CPUreg[7]_0 is [00000000] at 700 ns



Faulty Waveform: Fault enabled at 700 ns and the value of CPUreg[7]_0 changed from [00000000] to [00001000]

Fig. 6. Ila output waveform illustrating golden and faulty register values due to fault injection.

Case 1		Case 2		Case 3		Case 4		Case 5		Case 6		Case 7		Case 8		Case 9	
CPUreg_Bit	Flips	CPUreg_Bit	Flips	CPUreg_Bit	Flips	CPUreg_Bit	Flips	CPUreg_Bit	Flips	CPUreg_Bit	Flips	CPUreg_Bit	Flips	CPUreg_Bit	Flips	CPUreg_Bit	Flips
1_0	0	1_0	1	1_10	1	1_12	1	4_7	1	1_16	0	1_7	1	4_0	0	1_18	0
1_0	0	1_0	1	1_10	1	1_12	1	4_7	1	1_16	2	1_7	2	4_0	1	1_18	1
1_0	0	1_0	1	1_10	1	1_12	1	4_7	1	1_16	2	1_7	2	4_0	1	1_18	1
1_0	0	1_0	1	1_10	1	1_12	1	4_7	1	1_16	2	1_7	2	4_0	1	1_18	1
1_0	0	1_0	1	1_10	1	1_12	1	4_7	1	1_16	2	1_7	2	4_0	1	1_18	1
1_0	0	1_0	1	1_10	1	1_12	1	4_7	1	1_16	2	1_7	2	4_0	1	1_18	2
1_0	0	1_0	1	1_10	2	1_12	1	4_7	3	1_16	2	1_7	2	4_0	1	1_18	1
1_0	0	1_0	1	1_10	2	1_12	1	4_7	3	1_16	3	1_7	3	4_0	0	1_18	1
1_0	0	1_0	1	1_10	2	1_12	1	4_7	3	1_16	3	1_7	3	4_0	0	1_18	1
1_0	0	1_0	1	1_10	2	1_12	2	4_7	3	1_16	3	1_7	3	4_0	0	1_18	1

Fig. 7. Instruction dependent error propagation Trends observed from FPGA implementation.

3. Scenario 3: The flipped content of the register re-mained flipped across several clock cycles.

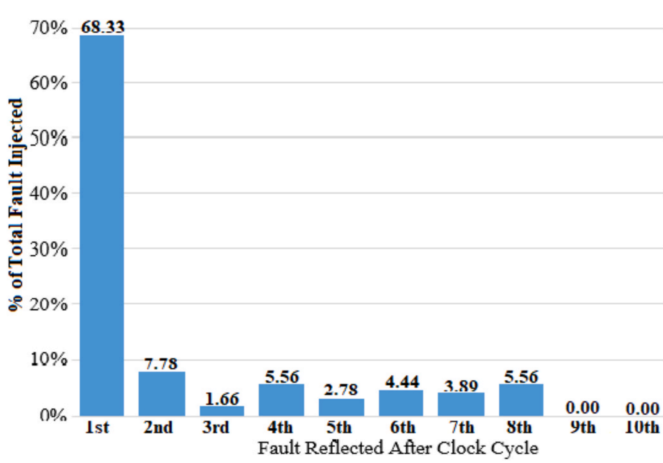


Fig. 8. Fault Impact across ten subsequent clock cycles.

less re-written by another instruction (Scenario 3). Suppose that this ADDI instruction is followed by a store instruction to the memory such as SW x2, 9 (x1). The erroneous value of x2 is propagated to a memory location in subsequent clock cycles (depending on when the store is executed) resulting in an error in the memory value (Scenario 2). Thus, we see a varied pattern of fault propagation across several clock cy-cles. Subsequently, the fault is disabled and new instructions are executed, which could overwrite the erroneous registers (to correct values: Sce-nario 1). Thus, we see several pat-terns in Fig. 7. We present a more detailed instruction-dependent fault analysis in the next subsection.

3.5. *Instruction dependent fault propagation*

Fig. 7 illustrates different cases of fault propagation tendency. Case 1 shows the zero-flip scenario. It indicates that the error is injected at CPU reg1_bit [0] (1_0) at a ran-dom clock cycle, but it neither propagates to flip a register nor a memory location. Similarly, Case 2 indicates that a single bit-flip remained flipped across 10 subsequent clock cycles. While, case 3, 4, 5, 6, and 7 explain Scenario 2 pre-sented in subsection 3.4. The number of flips increases af-ter a certain number of clock cycles, because the erroneous value propagates and affects more registers or memory loca-tions in subsequent clock cycles. On the contrary, in case 8 and 9, the number of flips decrease after a few clock cycles. This is because the erroneous registers are overwritten by the correct values after the fault is disabled. Hence, we can infer that the fault propagation is dependent on the type of instructions being executed next after fault injection on the processor, or in other words, it is application-dependent.

We picked a few sample clock cycles and subset of in-structions to determine their role in fault propagation. Ta- bles 4 and 5 show a subset of instructions, the clock cycle in which they were being executed and the register (and its bit) which had the error injected. In addition to the clock cycle number and the instruction being executed in that particular clock cycle, it is important to note that the register on which the error is

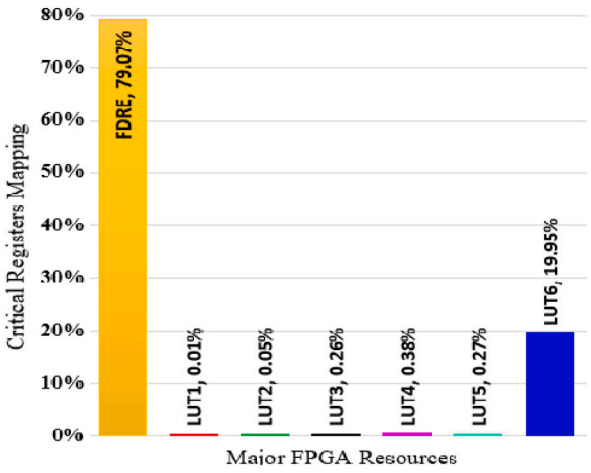


Fig. 9. Dominant Sites Mapped to FPGA resources.

enabled plays a crucial role. If the affected reg-ister is not part of the instruction being executed, it does not affect the instruction and future instructions. Therefore, in addition to the instructions, the sequence of specific types of instructions plays a significant role in the propagation of flips. For example, a "SUB x5, x5, x2" (instruction 9) fol-lowed by an instruction such as "SW x5, 12 (x5)" (instruction 10) would result in corrupting of the memory location due to a fault on x5. Thus, we arrived at this sequence of in-structions shown in Table 1 such that the error propagation is highly influenced by the instruction sequence. This entire sequence in Table 1 can be considered as one of the worst case sequences and is run in a loop continuously till a ran-dom error is injected and its influence is captured.

3.6. Dominant fault sites on the FPGA

The registers on which we injected the faults are mapped certain resources on the FPGA fabric. These registers which resulted in a majority of bit-flips are referred to as "dominant" sites. We plot these dominant sites in Fig. 9. We observe that a majority of the registers are getting mapped to Flip-Flops (FDRE: D Flip-Flop with Clock En-able and Synchronous Reset) in the FPGA after synthesis. Few other components are mapped to the LUT. Therefore, we observe that the Flip-Flop component on the FPGA is highly vulnerable and will need to be protected. In an ear-lier work [20], the author has performed an in-depth analysis and quantified the contribution of SET on flip-flops, towards multiple bit upsets. We found that an SET on a logic gate or a non-register element has a higher chance of getting logi-cally masked before arriving at the register (in which it gets captured). On the other hand, an SET on a register or a flip-flop has a higher chance of flipping the content of the same register, and therefore has a higher chance of propagating to outputs (or multiple outputs).(see Table 4)

4. Conclusion

We have designed a fault injection framework for an FPGA-based RISC-V processor to study the probability of Multiple Bit Upsets and to identify the most vulnerable FPGA resources. We proposed a two-step approach in which we perform a Monte-Carlo based simulation of the proces-sor with the controller integrated with the processor. We experimented on the PicoRV32 RISC-V processor and im-plemented it on a Xilinx Artix-7 FPGA. The entire simula-tion flow is automated in python, similarly, the FPGA flow is automated via TCL commands. We evaluate the probability of Multiple Bit Upsets as a result of SET injected. We also found that majority of flips were single bit-flips (62% on an average) and nearly 8% of the faults propagated and caused multiple bit-flips. We note that a majority of faults propa-gate to registers in the next clock cycle (68.33%). The fault propagation is dependent on the instruction being executed on the processor or in other words it is application depen-dent. Further, we found that Flip-Flop is the most dominant fault site on the FPGA fabric (79.1% utilization of Flip-Flops on FPGA). Therefore, there is a need to protect Flip-Flops, to reduce the number of bit-flips. This paper demonstrated the framework on a simple processor model running a small program. The designed framework can be easily migrated to, and integrated into a multi-core processor running large applications. However, this will need more effort to probe all the outputs on such multiple processors. It may also in-volve saving the register states at multiple and frequent time instants to keep track of the errors across cores.

Credit author statement

Jhalak Sharma: Literature review, Script writing, Carry out simulation and hardware implementation, - Formal analysis, Writing draft. Nanditha Rao: Problem conceptualization, Defining methodology, Project administration, - Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] 7 Series FPGAs Data Sheet: Overview, Xilinx, 2020. Available, https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf [Online]).
[2] J. A. M. A. R. a P. Reviriego, Characterizing a RISC-V SRAM-Based FPGA Implementation against Single Event Upset Using Fault Injec- Tions, Elsevier: Microelectronics Reliability, 2017.
[3] J. Sharma, N. Rao, O.A. Mohamed, fault injection controller based framework to characterize multiple bit upsets for FPGA de- signs, in: 2020 IEEE International Symposium on the Physical and Failure Analysis of Integrated Circuits (IPFA), 2020, pp. 1–5, https://doi.org/10.1109/IPFA49335.2020.9260845.
[4] A. U. a. J. A M. Pedro Reviriego Alexis Ramos, Efficient protec- tion of the register file in soft-processors implemented on Xilinx FPGAs, in: IEEE International Symposium on Defect and Fault Toler- ance in VLSI and Nanotechnology Systems (DFT) 67, 2018, pp. 299–304, 2.
[5] A. Castellanos, Jupiter Open-Source RISC-V Assembler and Runtime Simulator, 2018. Available: https://github.com/andrescv/Jupiter ([Online]).
[6] L. M. L. e. D A. Santos, A low-cost fault-tolerant RISC-V pro- cessor for space systems, in: IEEE 15th Design & Technology of Integrated Systems in Nanoscale Era (DTIS), 2020.
[7] A. S.-M. a. J A.M. Alexis Ramos Imran Wali, Analyzing the im- pact of the operating system on the reliability of a RISC-V FPGA implementation, in: 27th IEEE International Conference on Electron- Ics, Circuits and Systems (ICECS), 2020.
[8] V.B. Christophe Deleuze, , et al.Johan Laurent, On the importance of analysing microarchitecture for accurate software fault models, in: IEEE 21st Euromicro (DSD), 2018.
[9] T.W.e. a. Lingkan Gong, A programmable configuration con- troller for fault-tolerant applications, in: International Conference on Field-Programmable Technology (FPT), 2016.
[10] M. B. F V.e. a. Luigi Blasi, Fault resilience analysis of a RISC- V microprocessor design through a dedicated UVM environment, in: IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2020.
[11] M. R. G. a M.J.W. LukeW, Hsiao Nathan A. Harward, Estimat- ing soft processor soft error sensitivity through fault injection, in: IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, 2015.
[12] N. H. R. a M.J. Wirthlin, Software fault-tolerant techniques for softcore processors in commercial SRAM-based FPGAs, in: 24th International Conference on Architecture of Computing Systems, 2011.
[13] N. H. R. a M.J. Wirthlin, Analysis of the critical bits of a RISC-V processor implemented in an SRAM-based FPGA for space appli- cations, Electronics 9 (1) (2020).
[14] JMP 15, A Tool for Data Visualization and Exploratory Data Analy- Sis, SAS, 1989 ([Online]).
[15] N. G. a V.K.G. Madhusudan Sukrat Gupta, SHAKTI-F: a Fault tolerant microprocessor architecture, in: IEEE 24th Asian Test Symposium, ATS), 2015.
[16] D. White, Considerations Surrounding Single Event Effects in FPGAs, ASICs and Processor, 2012 [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp402_S-EE_Considerations.pdf.
[17] Single Event Upset, Wikipedia, 2009 [Online]. Available: https://en.wikipedia.org/wiki/Single-event_upset.

[18] C. Wolf, PicoRV32 - A Size-Optimized RISC-V CPU, 2019 [On- line]. Available: https://github.com/cliffordwolf/picorv32.
[19] Lucas A. Tambara, Ádria B. de Oliveira, et al., Evaluating soft core RISC-V processor in SRAM-based FPGA under radiation effects, IEEE Trans. Nucl. Sci. 67 (7) (2020) 1503–1510.
[20] N.P. Rao, M.P. Desai, A Detailed Characterization of Errors in Logic Circuits Due to Single-Event Transients, Euromicro DSD, 2015, pp. 714–721, https://doi.org/10.1109/DSD.2015.58, 2015.