UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering,
Mathematics & Computer Science

Accelerating AVF analysis using statistical fault injection

K.M. Schrama

MSc. Thesis
April 2025

Contents

Summary

This thesis introduces a statistically based fault injection campaign generator to speed up fault injection-based architectural vulnerability factor (AVF) analysis.

Microelectronics are increasingly more used in radiation-harsh environments. This, in combination with microelectronics becoming smaller and denser, the risk of a SEU becomes significantly higher. That is why it is important to test the vulnerability of microelectronics against radiation and soft errors. However, testing microelectronics against radiation can be costly and destructive to the device. An alternative to this can be emulation-based fault injections. This is where the hardware is emulated on an FPGA and injected with the errors to simulate the soft errors caused by radiation.

Through a fault injection campaign, the vulnerability of hardware to soft errors can be measured. However, a fault injection campaign can become time-consuming when the hardware to be tested gets bigger and the program on the hardware gets longer.

A solution to the long measurement time is Statistical fault injection (SFI). Statistical fault injection can be used in fault injection campaigns to reduce the number of fault injections needed to determine the system's vulnerability. SFI implies that only a part of the injection space has to be tested to get an accurate measurement of the AVF. The resulting AVF should be within a certain margin of error of the actual AVF.

Statistical fault injection is implemented in a fault injection campaign generator. The campaign is generated based on the device under test (DUT) and the benchmark. The part of the total campaign is selected, the size of which is based on a SFI formula.

The campaign generator is tested with two experiments. A smaller DUT is used in the first experiment, to see how a SFI campaign compares to a full campaign. This resulted in AVF measurements that were scattered around the AVF of a full campaign, but with a larger margin than what was given as a parameter. However, the mean of the 10 SFI campaign does result in an accurate measurement within 1%.

The second experiment is performed with a large DUT, the NEORV32 processor. A full campaign on this processor could not be performed in a feasible amount of time. This experiment is to show the behaviour of a fault injection campaign on a large DUT using differently sized campaigns. This experiment showed that the smaller the campaigns are, the more scattered the measurements are. With increasing sizes of the campaigns, the scatter converges to a point which can be assumed to be the AVF of the system. When looking at the mean of 10 measurements of the same campaign size, the AVF becomes stable after 20 times, and higher, the calculated SFI campaign size. This shows that it is possible to get an AVF measurement of a large DUT.

This research shows that employing Statistical fault injection campaigns substantially accelerates the measurement process of a system's AVF, while maintaining accuracy. This method enables the assessment of AVF in systems that were formerly too large to be measured.

Acronyms

ACE architecturally correct execution.

AVF architectural vulnerability factor.

CPU central processing unit.

DUE detected unrecoverable errors.

DUT device under test.

FIT failures in time.

FPGA field-programmable gate array.

IC integrated circuit.

ISA instruction set architecture.

RTL register-transfer level.

SCFIT Shadow Components-based Fault Injection Technique.

SDC silent data corruption.

SEE single event effect.

SEFI single event functional interrupts.

SET single event transient.

SEU single event upset.

SFI Statistical fault injection.

SOC system on chip.

Glossary

Neorv32 a customizable microcontroller-like system on chip (SoC) built around the NEORV32 RISC-V CPU, see section 2.8.

RISC-V an open standard Instruction Set Architecture.

Chapter 1

Introduction

This thesis covers a method of accelerating fault injection campaigns for radiation hardness assurance. Radiation has become a big problem for microelectronics, and it has become essential to measure the vulnerability of hardware to soft errors due to radiation. In this chapter, the effects of radiation on hardware and the necessity to measure the vulnerability of hardware are introduced. After that, an introduction to the metric to measure the vulnerability is given, and how to measure it. Further, the goal of this thesis is explained. Finally, the structure of this thesis is explained.

1.1 Radiation on hardware

The use of electronics in radiation-harsh environments is increasing, for example, in space [1], [2]. The chance of a single event upset (SEU) is higher, because of the increased radiation in space [3]. This, in combination with microelectronics becoming smaller and denser, the risk of an SEU becomes significantly higher. SEU on a system can cause data corruption or crashes. That is why it is essential to determine the vulnerability of these electronics before they are released into these environments. The vulnerability shows the effect of errors on the system. A metric for the vulnerability will give insight into the structure of the processor and give developers a way to improve the design and counteract the errors. The vulnerability of circuits to SEUs can be measured with the architectural vulnerability factor (AVF) [4].

1.2 Metric for vulnerability

The AVF is the chance in percentage that an error in the processor will result in a visible change in the output of a program. This means an AVF of 0% will never result in a change on the output when an error is introduced. An AVF of 100% will always result in a change on the output. A visible change on the output can be a corruption of the data on the output, or the system crashes and hangs. An error in the system is not guaranteed to propagate to the output or crash the system. This can happen when the corruption is overwritten on the following clock cycles, or a part of the system that is not used at that moment gets an error.

1.3 Measuring the vulnerability

The AVF can be found using architecturally correct execution (ACE) analysis or with fault injection campaigns. ACE analysis is a theoretical analysis where each bit is analysed by its effect on the output when a fault occurs on that bit [4]. A fault injection campaign is a practical analysis where bits in a system are injected with faults and the effect on the output is measured [5]. The AVF is then based on the number of times that the output is different from normal behaviour after a fault injection, divided by the total number of injected faults.

1.4 Fault injection

The principle of fault injections is that a fault is injected into a system during operation to simulate the hitting of a particle and change the state of the bit that was hit. There are multiple ways to inject faults. One method is a hardware-based approach. This is where an IC runs its program as usual, and a fault injection environment either interfaces with the IC through connected pins or energy particles are sent into the IC. This is a fast approach, but at the cost of accuracy and an expensive external hardware environment. Another method is through simulation of the IC on a simulator, and the fault is injected during compile time or during run time. This method is, however, the slowest of the methods, because the hardware is simulated as software and can not run at its intended speed. This does come with the advantage of a great amount of insight into the way the error propagates through a system, and the error can be injected at a specific point and time in the program. Lastly, fault injection can be done with an emulation-based method. This method uses FPGAs to run the hardware on. By using an FPGA, there is no need for expensive external hardware, and the hardware can run at almost the intended speed. With emulation of the hardware, an external environment can pause the hardware at a specific moment to provide an accurate fault injection. This does, however, come with the overhead of the time it takes to inject while the hardware is paused.

A fault injection campaign is a sequence of single fault injections. A fault injection campaign usually injects into every possible point at every possible moment of the system. This is done to measure the AVF of a system by dividing the number of fault injections that resulted in a visible error on the output by the total number of fault injections performed. So by running a campaign, the AVF can be measured.

However, the time to run a campaign can increase significantly the larger the system becomes. The number of fault injections needed in a campaign is based on the device under test (DUT) and the program running on it. The DUT can have a large amount of memory in which a fault can occur, and this then has to be tested for every moment program. So when the vulnerability of, for example, a central processing unit (CPU), like the Neorv32, needs to be tested, the number of points in the system is already in the thousands. The program running on the CPU can also go into the thousands of moments that can be injected. This will result in an injection space in the millions. Every injection, the campaign needs to let the program fully run to see the output. So based on the length of the program, the campaign can take weeks if not months. To be able to measure the AVF of a large system, the campaign needs to be accelerated.

1.5 Accelerating fault injection campaigns

This thesis will research a way to accelerate fault injection campaigns. With faster fault injection campaigns, bigger DUTs with longer benchmarks can be analysed for architectural vulnerability. To achieve this, the major reason for a too long campaign can be improved. That is the large injection space. The injection space is the total number of fault injection points in space and time, and the type of fault. The size of the DUT and the length of the benchmark determine the injection space. For bigger DUTs and longer benchmark programs, it is almost impossible to do a complete AVF analysis, because not all possible faults can be injected in a reasonable amount of time. Statistical fault injection (SFI) can be used to solve this problem [6]. SFI implies that not every point in the injection space needs to be injected with a fault to determine a representative AVF. By needing fewer injections randomly distributed over the injection space for a representative AVF analysis, the number of injection points can be decreased.

1.6 Thesis outline

In Chapter 2 an overview is given of the background of fault injections, AVF analysis and tools used in this thesis. Chapter 3 gives an overview of existing methods and alternatives to accelerate fault injection campaigns. Chapter 4 gives an overview of the developed campaign generator. An overview of the experiments performed, and the results is given in Chapter 5. Finally, a conclusion and discussion are formulated in Chapter 6.

Chapter 2

Background

This chapter gives background on the concepts used in this thesis. The first section, the cause and effects of soft errors. Followed by the metric to measure the vulnerability to soft errors and methods to measure the vulnerability. Then, some more detail is given about fault injection and how to use that to determine the vulnerability. Finally, some background is provided about the system that is used as the DUT in one of the experiments.

2.1 Soft errors

Soft errors are data corruption events that can be induced by radiation. The errors are not harmful to a device, but can corrupt data or crash a system. Soft errors due to radiation have become a major problem for the reliability of systems [7], [8]. Soft errors are also called single event upsets (SEU), but SEU's can also apply to hard errors where a device is permanently damaged.

2.1.1 Soft errors due to radiation

Radiation occurs naturally in space. When a system is deployed into space, it will be hit by this radiation. The radiation particles can hit the microelectronics in that system and cause a single event effect (SEE). An SEE can affect a system in multiple ways as a soft error [3]:

- single event transients (SET): causes a change in the output voltage of a gate
- single event upsets (SEU): causes an invert in memory value
- single event functional interrupts (SEFI): causes a system to crash until reset

These events are caused when a transistor collects a charge from energetic particles, like alpha particles or neutron particles. When enough charge is collected, the state of the transistor can change and cause a bit in the logic to be inverted. This change can corrupt the memory or make the system crash.

2.1.2 Soft error mitigation

With microelectronics becoming smaller and denser, soft errors become a larger problem even at sea level [8]. To combat these types of errors, mitigations are set in place to detect or prevent these errors.

A method to detect these errors is parity. With parity, a data block has a bit added to it that provides information on whether the total bits are odd or even. A change in the data block means a change from odd to even or even to odd. This can be checked with the parity bit, and a bit flip can be detected. When an error gets detected, it can be prevented from propagating through the system or changed back.

Another method to prevent soft errors is to reduce charge collection by the transistor. This can be done by device-level hardening. This is where the design of the semiconductors in silicon are changed by adding capacitance to sensitive areas.

2.1.3 Reliability metrics

An SEU can cause an error, when this error remains undetected, it is called a silent data corruption (SDC) [7]. When the error does get detected, but cannot be rectified, it is called detected unrecoverable errors (DUE). Generally, SDC and DUE are expressed in FIT (failures in time). One FIT means that there is one error in one billion hours of operation. These metrics imply the chance of an error occurring, but do not give insight into the effects of an error on the system.

2.2 Architectural Vulnerability Factor

Not every error in a system would cause a visible error on the output. For this, the architectural vulnerability factor (AVF) is used. It assesses the system's vulnerability by determining the extent to which an error impacts the system. The AVF of a system is the probability that a single-bit fault will be visible in the output of the system [9] [4]. This means that an AVF of 0% indicates that a fault will not be seen in the output, and an AVF of 100% indicates that a fault will be seen in the output. AVF can be determined with different methods. AVF can be determined using Architecturally Correct Execution (ACE) analysis [9]. This is a theoretical analysis. AVF can also be determined using fault injections [6] [5]. This is a more practical analysis.

2.3 ACE analysis

One method of estimating the AVF of a system is by using ACE bits [9]. A bit in a system can be labelled as an ACE bit or an un-ACE bit. When a bit is labelled as an ACE bit, then a fault in that bit will result in a visible change in the output, for example, a change in the output data or the system crashing. When a bit is an un-ACE bit, then the fault in the bit will have no effect on the output. Un-ACE bits are not required for architecturally correct execution.

Whether a bit is an ACE bit can change depending on the state of the system and the time in the program. For example, a fault in a register only affects the output if the data in the register is used after the fault. If the data is overwritten, then the fault also disappears and will not be seen in the output. If the register is to be read, the bit is an ACE bit; if the register is written to, the bit is an un-ACE bit. The AVF of a system can be calculated using Formula (2.1) [9]:

$$AVF = \frac{\sum \text{residency of all ACE bits in a structure}}{\text{total number of bits} \cdot \text{total execution cycles}} \tag{2.1}$$

2.4 Fault injection

To estimate the AVF of a system, fault injections can be used. AVF describes the vulnerability of a system to single-bit faults. By using fault injections, among others, single-bit faults can be simulated. There are multiple fault injection techniques, hardware-based, simulation-based and emulation-based fault injections [5].

2.4.1 Hardware-based fault injection

Hardware-based fault injection is performed by either contact-based approaches or without contact approaches [5]. Contact-based approaches are where the integrated circuit is connected via pins onto an external interface [10]. Without contact approaches are performed by, for example, sending energy particles onto the IC with a laser beam to simulate a faulty environment [11] or by exposing the IC with ionizing radiation [12]. These approaches require an expensive external hardware environment to inject the faults, with a risk of damaging the IC during testing. The hardware-based fault injection campaigns are, however, the fastest of all the approaches, because each test is run at the standard operating speed of the IC without high overhead.

2.4.2 Simulation-based fault injection

Simulation-based fault injection makes use of a simulation of a system to perform the fault injections on. A simulator like Verilator [13] can be used to inject the faults into. The machine code is changed during compile time or during run time to simulate the errors that would have happened on the hardware. The downside of a simulation-based approach is that the fault injections are significantly slower compared to the emulation or the hardware-based approach.

2.4.3 Emulation-based fault injection

Emulation-based fault injections make use of FPGAs to run the system on. The faults can be injected during runtime by setting the bits in the FPGA [5]. The advantage of using FPGAs to emulate the system is that there is no additional expensive external hardware needed to inject the faults into the system. Also, because the circuit is emulated on hardware, it will run as fast as the hardware-based approach. The only downside is that for a fault to be injected at a specific point during run time, the program needs to be stopped for the tool to be able to inject the fault [14], [15]. The added time will make it slower than a hardware-based approach.

2.5 Fault injection campaign

A fault injection campaign is how the AVF of a system can be measured when using fault injections. A campaign is a sequence of fault injections, where after each injection, the output is checked, and the system is reset for the next injection. Every injection is at a different point in the system at a different time in the program. By doing this, every possible point and moment at which an error can occur is tested. The total of injection points that result in a failure are counted and divided by the total number of injection points to calculate the AVF of the system. With Formula 2.2, the AVF can be calculated. f is the output of a single fault injection at point X_i. This is 1 if the output is incorrect and 0 if the output is correct. The sum of the outputs is divided by n the total number of fault injections, and results in the AVF.

$$AVF = \frac{1}{n}\sum_{i=1}^{n} f(X_i) \tag{2.2}$$

2.6 Fault injection tool

A fault injection tool used for injecting faults into an FPGA is the injection tool from [15]. This fault injection tool uses the Fretz fault injector from [14] to inject the faults into the Device Under Test (DUT). The fault injection tool has the components, as can be seen in Figure 2.1. The campaign generator creates a list of injection points with the corresponding time of injection. The stimuli generator sends a stop point to the extra hardware on the target board that controls the clock of the DUT to stop the DUT at the moment of injection. The grey parts are existing components from the Fretz framework that inject the fault into the DUT. The fault injector sends the location for injection to the external board, and the external board injects the fault. The golden standard is the correct output of the DUT without any faults and is based on the DUT. The result analyser compares the output of the DUT with the golden standard. The result then gets stored in a database.

2.7 Statistical fault injection

To decrease the length of a fault injection campaign, statistical fault injection (SFI) can be used. SFI is based on Formula 2.3 from [6].

$$n = \frac{N}{1 + e^2 \frac{N-1}{t^2 p(1-p)}} \tag{2.3}$$

n is the number of fault injections needed for a representative analysis. N is the initial injection space, so the total number of points that can be injected into. p is the estimated probability of faults resulting in a failure. This is a priori unknown; a conservative approach is to set this value to maximize n, which is at 0.5. e is the margin of error. t is the cut-off point corresponding to the confidence level, this is the probability that the value is within the error margin. In Figure 2.2 can the influence

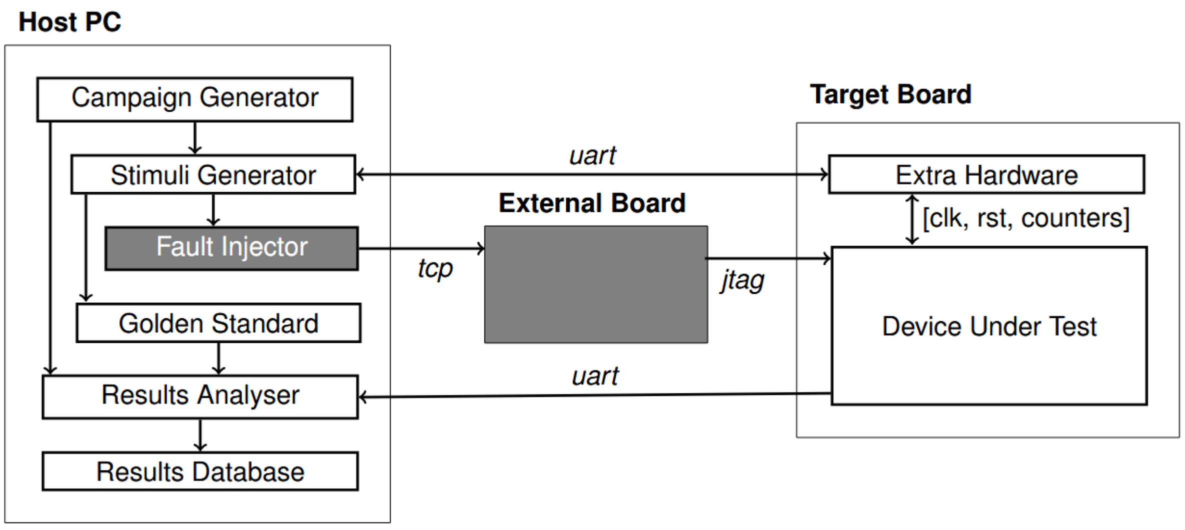


Figure 2.1: Components of the fault injection tool from [15]

of the estimated probability can be seen on n. At 0.5 n is maximized and falls as p gets bigger or smaller. Figure 2.3 shows the influence of the margin of error on n. n grows exponentially, the smaller e is chosen. It has the biggest influence on n of the parameters. Figure 2.4 shows the influence of the cut-off point on n. The cut-off point is computed with respect to the normal distribution, below 50%, t becomes negative. This will be squared the same as when it is set as positive, so only 0.5 to 1.0 is shown.

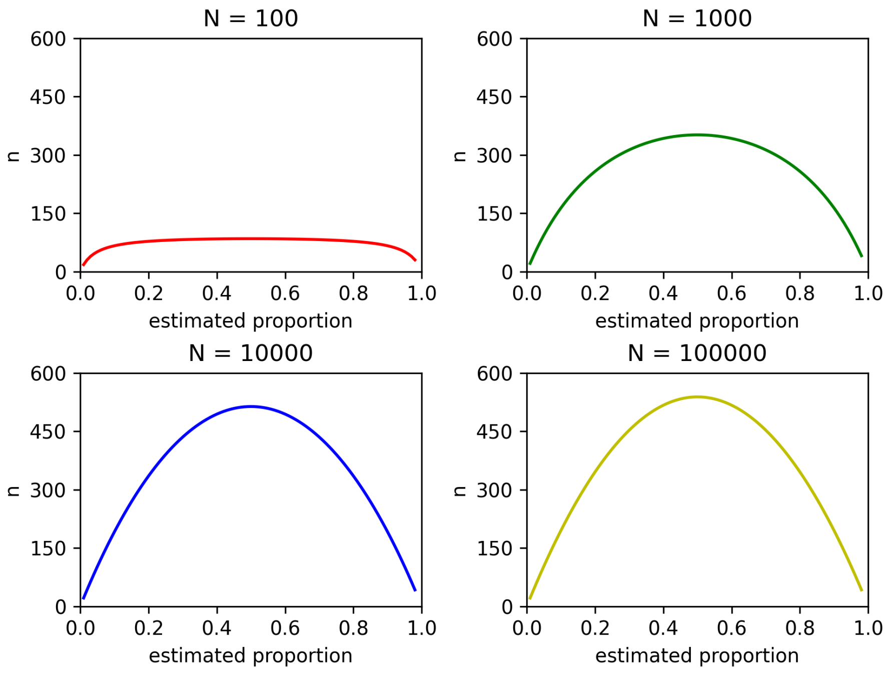


Figure 2.2: The effect of the estimated proportion on the number of injection points needed for SFI.

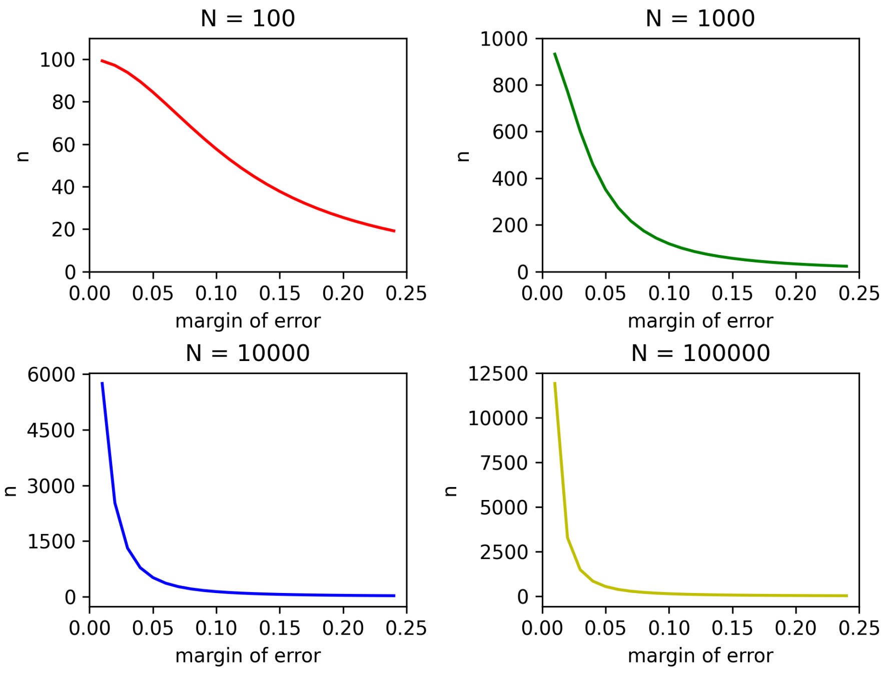


Figure 2.3: The effect of the margin of error on the number of injection points needed for SFI.

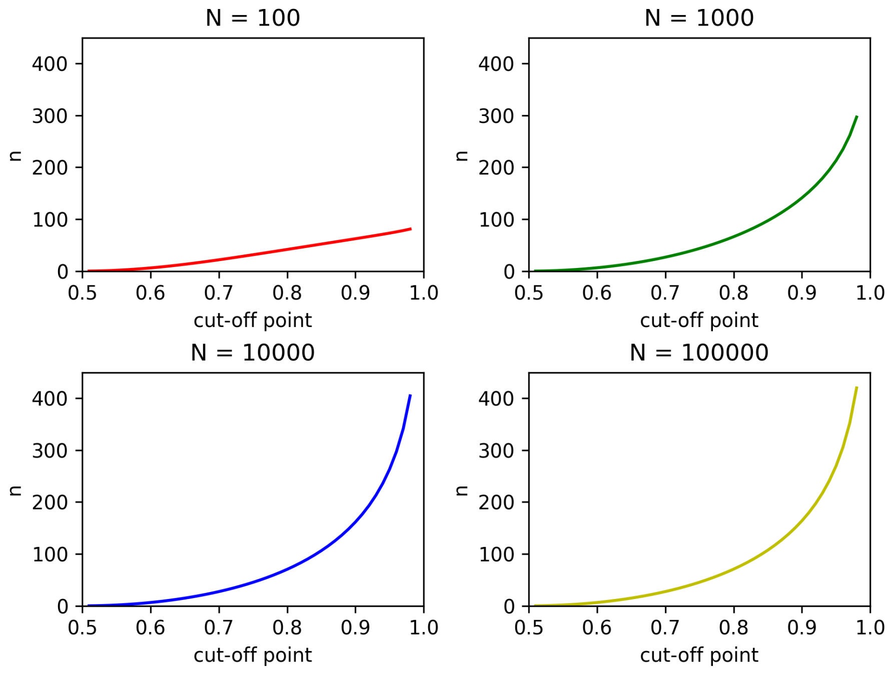


Figure 2.4: The effect of the cut-off point on the number of injection points needed for SFI.

2.8 Neorv32

The Neorv32 is an open source processor that is a system on chip (SOC) [16]. It is built around a RISC-V CPU that is written in VHDL. This makes it platform independent and able to be run on almost all FPGA's.

The RISC-V CPU architecture can be seen in Figure 2.5. It is a 32-bit little-endian pipelined architecture. It is a modified Harvard architecture. The CPU can be configured to run as a single or dual-core. The CPU is highly configurable with instruction sets and extensions.

The extensions also include a custom functions unit (CFU). This unit can be used to implement custom RISC-V instructions. The CFU is intended to run functions that are not efficient when implemented in software.

©

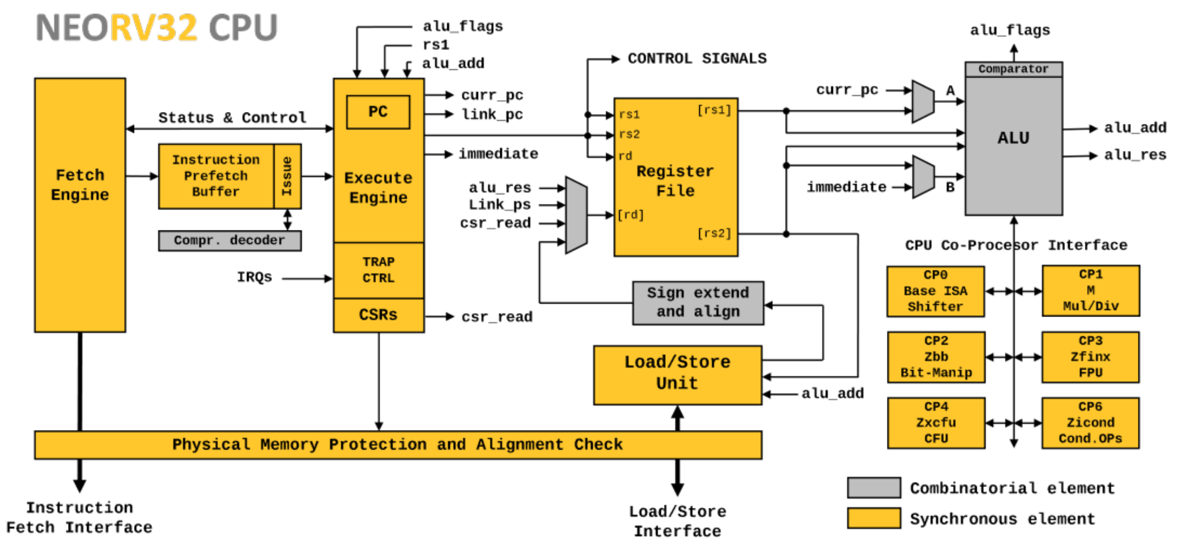


Figure 2.5: Neorv32 CPU architecture [16]

Chapter 3

Related work

Several methods for determining the AVF of a system were already mentioned in Chapter 2. This chapter will extend on that and also explore other methods of determining the AVF, including methods to speed up the AVF analysis.

3.1 AVF analysis

ACE analysis is a fast method of analysing the AVF of a system. While it is fast, it does come at the cost of accuracy, because of it being a conservative method where every bit is labelled as an ACE bit unless proven otherwise [9]. This means an ACE analysis can only give an upper bound of the AVF. The AVF estimated using ACE is an overestimation of about 3.5x [17] compared to an AVF analysis using fault injection. While a refinement of the ACE analysis does bring the overestimation to 2.6x [17], it is still only an upper bound and does not provide an accurate analysis.

Another very accurate method is register-transfer level (RTL) injection [18]. RTL injection is a simulation-based fault injection method. The downside of this method is the long simulation time. This creates a problem for architectures with a significant number of points to inject and a large workload. A couple of simulators exist that allow for fault injection, for example, vRTLmod [19] or another fault injector [20] based on the Gem5 simulator [21].

A method that combines speed and accuracy is an emulation based fault injection, as in [15], where Fretz [14] is utilized to run injection campaigns. Another emulation based fault injection technique is proposed in [22], where the debugging facilities of the Altera software are used for fault injections, called SCFIT. While emulation based fault injection tools do increase the speed of fault injection campaigns compared to simulation-based fault injections, very large DUTs still form a problem for analysing the AVF in a feasible amount of time.

3.2 Accelerating AVF analysis

A way to accelerate fault injection campaigns is proposed in [6]. The method of using statistical fault injection to decrease the amount of injections needed was already explored in [23] with emulation-based fault injections, but without a basis for a minimum number of injections as described in [6].

A way to improve statistical fault injections has been proposed in [24] and [25]. They propose two similar fault injection models based on SFI with the addition of ACE-analysis. Both have performed the analysis on a simulation-based fault injection setup.

The proposed method works by doing an ACE analysis combined with statistical fault injection. The ACE-analysis will classify intervals at points where an injection could cause a failure (ACE) and intervals where an injection definitely doesn't cause a failure (un-ACE). The approach from [24] and [25] will only inject into the intervals that are classified as ACE and skip the parts that will never cause a failure. By doing the ACE-analysis before the injection campaign, the number of fault injections can be reduced, thus reducing the amount of time needed to run the campaign. The average speed up that was obtained using this approach was 13.5 times. Depending on the DUT, the speed-up differs. A DUT with a high percentage of ACE intervals will have a lower speed up than a DUT with fewer.

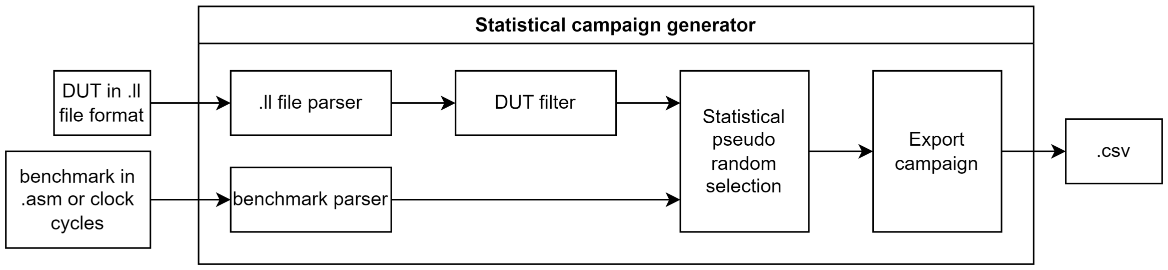
Chapter 4

Statistical fault injection campaign generator

This chapter will describe the statistical fault injection campaign generator. First, an overview is given of the whole campaign generator. After that, each part is explained in more detail.

4.1 The campaign generator

The goal of a fault injection campaign is to find the AVF of the device under test (DUT) by injecting into the DUT. This is achieved by injecting at every memory point in the DUT at every clock cycle of the program in the system. The campaign is based on the DUT and the benchmark, which consists of the points in space combined with the points in time. The points in space are the memory cells, and the points in time are the clock cycles. Thus, every point in the system can be injected at every clock cycle.



Statistical campaign generator
DUT in .ll file format
benchmark in .asm or clock cycles
.ll file parser
DUT filter
benchmark parser
Statistical pseudo random selection
Export campaign
.csv

Figure 4.1: Statistical fault injection campaign generator

4.2 LL file parser

The .ll file is a descriptive file of every memory cell used in the DUT. The .ll file is a file generated by Vivado when generating the bitstream of the DUT. This file contains the information about every memory cell in the implemented design. The information described is the location of the memory cell, the block, the latch and the user net associated with the memory cell. These are the points of the device under test (DUT) that a fault can be injected into. The file parser reads the file and stores the contents in a pandas dataframe to later be able to extract the injection points from it.

4.3 Filter for DUT

As can be seen in Section 2.1 and described in Section 2.6, the target board contains the DUT and extra hardware. Both of these parts are part of the bitstream that the FPGA is programmed with. The data from the .ll file needs to be filtered so that only the memory cells of the DUT remain to be injected. What needs to be filtered depends on the DUT, but the extra hardware of the controller that is used by the fault injection tool always needs to be excluded for injection. Injecting into a part of the controller can break the fault injection tool. The data will also be filtered from BRAM points. Due to limitations of the fault injection tool, it is unable to inject into BRAM. During the experiments, the BRAM has been left out of scope, because the goal of this thesis is to accelerate the measurement of AVF in general and not to determine the AVF of a specific DUT, which would require injection into the BRAM.

4.4 Benchmark parser

When injecting into a soft-core CPU like the , the assembly file of the program can be used to select when a fault injection needs to take place based on the program counter. This can be used to inject into a specific part of the benchmark program. The benchmark parser will parse the assembly file for the program counters to later use them when selecting fault injection points.

Program counters can span over multiple clock cycles. To get a more specific moment for the injection, a clock counter can also be used to indicate when in the program to inject a fault. This can be set via arguments, the start, the end and the clock counts in between that need to be skipped. This can later also be used when selecting fault injection points. Only one of the two methods will be able to be used at a time.

4.5 Statistical pseudo-random selection

For statistical fault injection, Formula 2.3, as shown in Section 2.7, is used to calculate how many injection points are needed for a within the parameters accurate AVF measurement. This is calculated using the function, as can be seen in Listing 4.1. Based on the value from the equation, the injection points are chosen from the total list of memory points, combined with each clock cycle count or program count at random.

The selection is performed using a pseudo-random function. The random function is the random library in Python. The use of the function can be seen in Listing 4.2. The function is not truly random because it is an algorithm based on a seed, which determines the output. The algorithm used is the Mersenne Twister [26]. which is also the recommended algorithm by [6]. The algorithm is deterministic, which means a seed can be set to receive a known output. If the seed is not set, the system time will be used as the seed.

It does not matter if the random selection is not truly random, because the goal of the selection is to have an equal spread of injection points. This can also be achieved by using a pseudo-random selection.

```
FUNCTION number_of_FI (population_size, estimated_proportion = 0.5,
  ↪ margin_of_error = 0.05, cut_off_point = 0.99, cop_percentage = True)
    IF cut_off_point is a percentage:
        cut_off_point = get normal quantile equivalent of cut_off_point

    RETURN population_size/(((margin_of_error**2) * ((population_size - 1)
  ↪ /((cut_off_point**2) * estimated_proportion * (1 -
  ↪ estimated_proportion)))) + 1)
```

Listing 4.1: Algorithm to calculate number of injection points for SFI.

```
FUNCTION random_sampled_combine (DUT_DATA, BENCHMARK_DATA):
    TOTAL_FI_SPACE = length of DUT_DATA * length of BENCHMARK_DATA
    SAMPLED_SPACE = number_of_FI(TOTAL_FI_SPACE)

    FRAMEADDRESS_LIST = get column "frameaddress" from DUT_DATA and create
  ↪ a list from it
    FRAMEOFFSET_LIST = get column "frameoffset" from DUT_DATA and create a
  ↪ list from it

    INJECTIONPOINTS = empty list
    FOR each 0 to SAMPLED_SPACE:
        DUT_SAMPLE = get index of random selection from DUT_DATA
        BENCHMARK_SAMPLE = get index of random selection from
  ↪ BENCHMARK_DATA
        INJECTIONPOINTS append a list of FRAMEADDRESS_LIST(DUT_SAMPLE),
  ↪ FRAMEOFFSET_LIST(DUT_SAMPLE) and BENCHMARK_DATA(BENCHMARK_SAMPLE)

    RETURN INJECTIONPOINTS
```

Listing 4.2: Algorithm for the selection of injection points based on the calculation of the number of injection points needed for SFI.

4.6 Exporting campaign

The selected injection points are exported as a CSV file. The CSV file contains the frame address, the frame offset and the moment to inject, which can be the clock counter or the program counter.

4.7 Fault injection tool

The fault injection tool described in Section 2.6 is changed to work together with the statistical fault injection campaign generator. The campaign generator exports the campaign as a CSV file. The fault injection tool was changed to run a campaign based on the CSV file.

A feature to run multiple campaigns one after the other was also added. This feature makes it easy to run multiple AVF measurements.

Chapter 5

Experiments and results

In this chapter, the experiments that were performed to evaluate the statistical fault injection campaign generator are described, and the results are analysed. There were two experiments done, one where a small DUT was used to run a full fault injection campaign on and a statistical campaign, and an experiment where a large DUT was used, where a full campaign would not be feasible.

5.1 Experimental setup

The experimental setup is mostly the same as the fault injection tool described in Section 2.6. The difference is that the campaign generator was changed to the statistical campaign generator.

5.2 Experiment: Full campaign compared to statistical campaign

This experiment is to show the use of SFI campaign on a DUT compared to a full campaign. This was done on a smaller DUT where the full campaign can be done in a feasible amount of time.

5.2.1 DUT

The DUT used in this experiment is a shift and add multiplier. The multiplier algorithm can be seen in Figure 5.1. The DUT is implemented with A and B as 16-bit unsigned values.

An example of the workings of the multiplier can be seen in Figure 5.2. The multiplier works by adding A to the result if the LSB of B is 1; if it is 0, it does nothing. After that, A is shifted left and B is shifted right, and the addition is done again if the LSB is 1 again. This repeats for the number of bits of A and B. For an n-bit A and B, the multiplier takes n clock cycles to compute the product.

The multiplier was chosen as DUT, because it provides injection points that will definitely make it to the output, for example, the product register, which maps directly to the output and is not overwritten, because it writes to itself with A added to it. The DUT also has injection points that will definitely not be shown on the output, for example, the bits of B that are shifted in each cycle that are not read anymore. This should

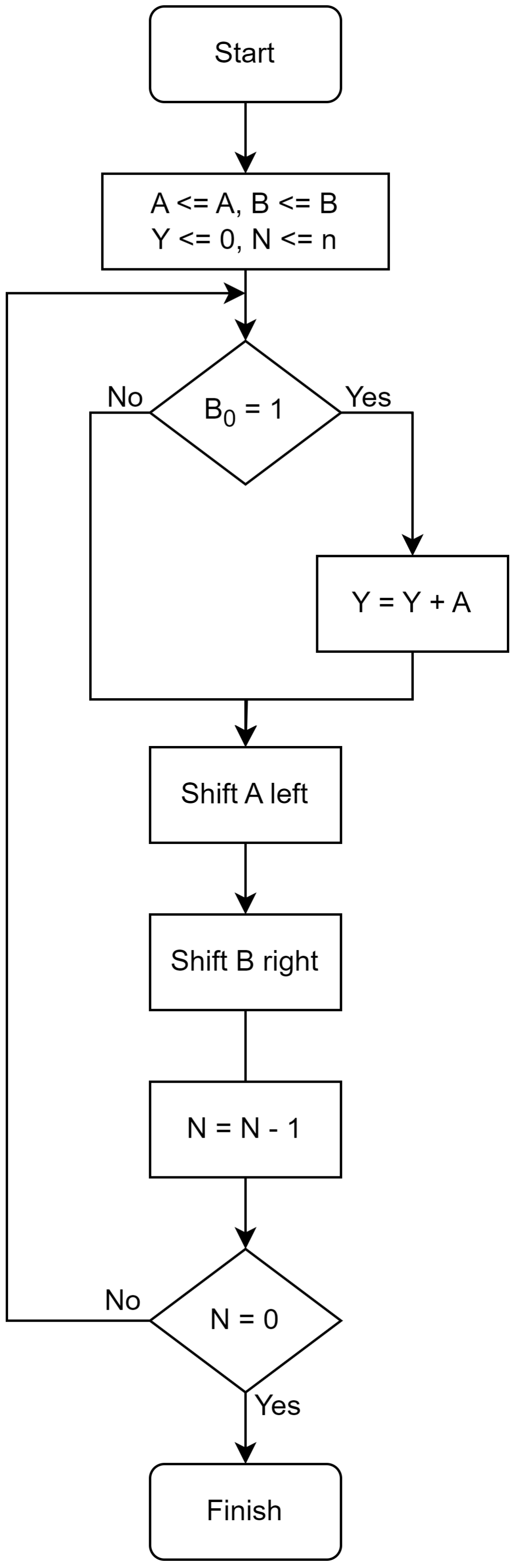


Figure 5.1: Implementation of a shift and add multiplier. A and B are the inputs of the multiplier, and Y is the output. n is the number of bits of A and B.

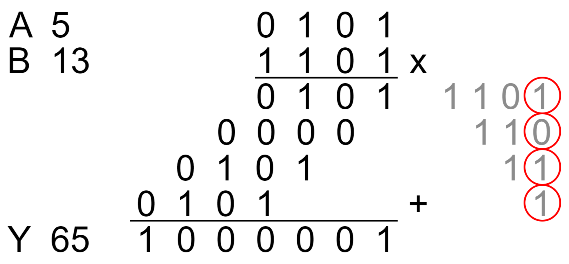


Figure 5.2: Example of the multiplier

provide an AVF value between 0 and 1 and not 0 or 1. With this, the difference can be shown between the full campaign and the SFI campaign.

5.2.2 Benchmark

The DUT does not run a benchmark. That is why the injections are based on the clock cycle count. The multiplier will need 16 clock cycles to do the multiplication; that is why the injections are chosen to be able to happen at the clock cycles between 0 and 16.

5.2.3 Parameters

For this experiment, the following parameters were used in the SFI campaign generator:

- $p = 0.5$
- $e = 5\%$
- $t = 99\%$

p was chosen as 0.5. p is a priori unknown and is chosen as 0.5 to maximize n. e is set to 5% and t is set to 99%. These are chosen relatively strictly, due to this n will result in a higher number, which is chosen to see if the results are able to fall within these margins. If that is the case, the margins can be loosened to see if the accuracy is still acceptable.

Several measurements were performed to see the variance of the measured AVF and to see if the mean of the measurements is around the AVF of the full campaign. Each campaign was generated with a different seed.

The full injection campaign was also performed multiple times to verify that the full campaign AVF is constant as it is expected due to the deterministic behaviour of the DUT.

5.2.4 Results

The experiment was performed 10 times for the SFI and the full campaigns. The results of the experiment of SFI campaigns compared to a full campaign can be seen in Figure 5.3. The full campaign had 2057 injection points. In the SFI campaign, there were 428 injection points. The AVF of the SFI are spread. This should follow a normal distribution around the AVF of the full campaign.

It can be seen that the actual AVF falls outside the error margin of some of the measurements, this can be seen with measurements S3, S4, S8 and S10. The confidence level was chosen as 99%, which would mean that for only 1% of the measurements the AVF should fall outside the error margin.

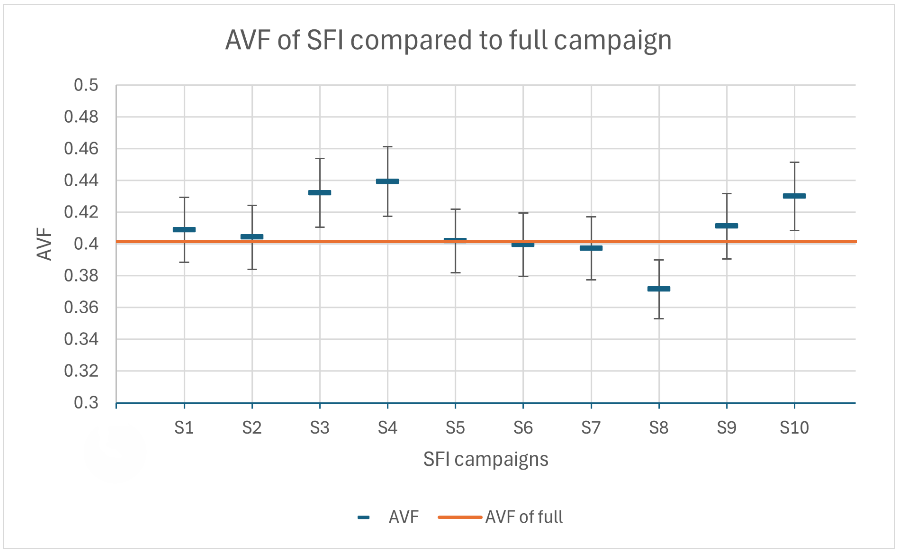


Figure 5.3: SFI campaigns compared to a full campaign

When looking at Figure 5.4, it can be seen how the spread of the measured AVF of the SFI campaigns compare to the full campaign. The full campaigns provided the same AVF every measurement. This means that the experimental setup does not influence the measured AVF.

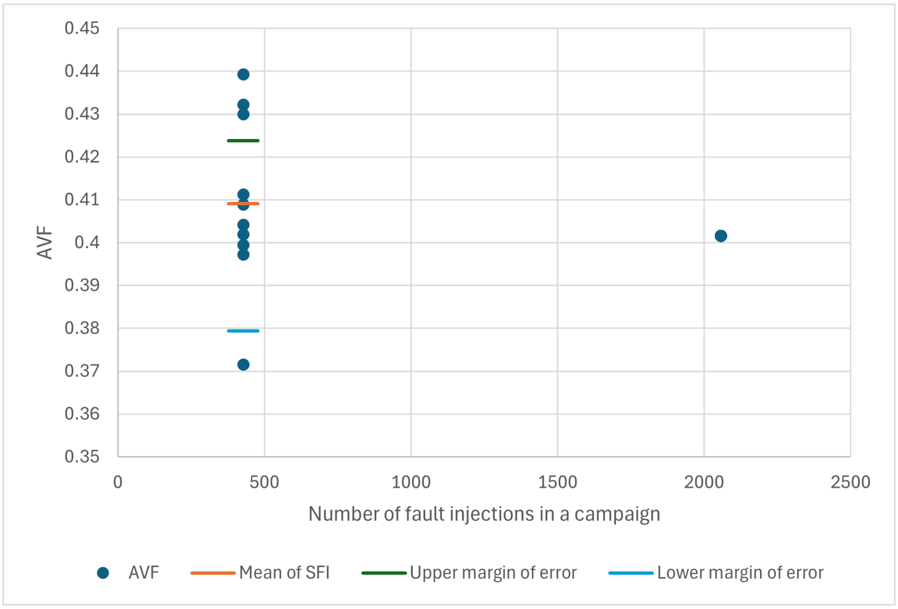


Figure 5.4: SFI campaigns spread compared to full campaigns, upper and lower margin based on the AVF of the full campaign

When looking at the mean of all the measurements, we get the results in Table 5.1. The mean of the SFI campaigns is 0.409 compared to the 0.402 of the full campaigns. This is a difference of 1.7%. This is a better result than a single SFI AVF measurement, which could result in a higher difference, for example, measurement S4 from Figure 5.3. S4 has measured an AVF of 0.439. This is a difference of 6.8%. This could be due to the normal distribution that the result should follow around the actual AVF.

	SFI	full
Campaign length	428	2057
Mean AVF	0.409	0.402

Table 5.1: Mean AVF of SFI campaigns compared to AVF of a full campaign

5.2.5 Increase number of campaigns to run

To see if a multiple number of campaigns influences the average AVF, the experiment was also done with 20 to 100 campaigns with increments of 10.

Figure 5.5a shows the spread of each experiment, with the respective number of campaigns. The mean of each experiment can also be seen. Figure 5.5b shows the difference of each mean AVF compared to the AVF of the full campaign found in Table 5.1.

5.2.6 Analysis

Looking at these results, it provides evidence that a single SFI injection campaign does not provide an accurate AVF. However, with multiple campaigns, a more accurate AVF can be derived from the mean of the campaigns to an accuracy of within 1%. However, this would not be beneficial for smaller DUT, because a normal full campaign takes fewer fault injections than 10 SFI campaigns.

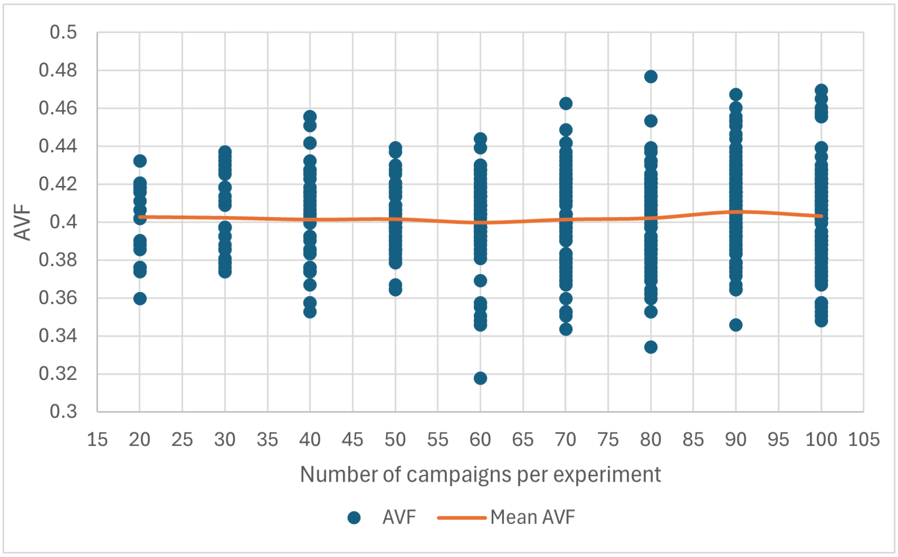
5.3 Experiment: Statistical campaign on a large DUT

5.3.1 DUT

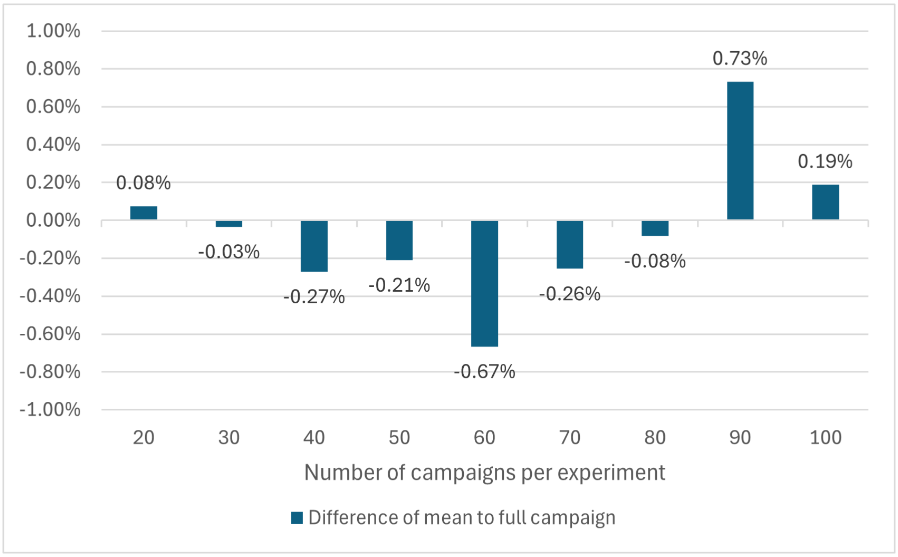
In this experiment, the Neorv32 with the RISC-V ISA was used as the DUT. The goal of this experiment is to show the use of SFI on a DUT of which a full campaign would be too large to run in a feasible amount of time. The architecture of the Neorv32 is shown in Section 2.8.

5.3.2 Benchmark

The benchmark running on the DUT is a quick sort algorithm, Listing 5.1. The algorithm is from [27], where it was used to test microcontrollers and FPGA's against radiation. The quick sort algorithm is a simple version of the algorithm where the pivot is chosen to be in the middle of the array. This is not an optimal implementation of the



(a) Multiple number of campaigns per experiment



(b) Difference between the average of the mean AVF compared to the full campaign AVF

Figure 5.5: The spread of the campaign outputs can be seen in 5.5a. Each row is a different experiment with the number of campaigns as in the graph. The mean is based on the campaigns from each respective experiment. The difference of the mean of each experiment and the actual AVF of the full campaign from Table 5.1 can be seen in 5.5b.

algorithm, but the algorithm is purely to run as a benchmark for fault injections and doesn't have to be as quick as possible. An example of how the quick sort works can be seen in Figure 5.6.

The algorithm is provided with an array of 180 numbers between -100 and 100. After the benchmark has run, the sorted array will be checked with a checker function against a golden standard. After which, an output will be given based on whether the sorted array was correct or not.

```c
void quick_sort(int *a, int n){
    if (n < 2)
        return;

    int p = a[n / 2];
    int *l = a;
    int *r = a + n - 1;

    while (l <= r) {
        if (*l < p) {
            l++;
        } else if (*r > p) {
            r--;
        } else {
            int t = *l;
            *l = *r;
            *r = t;
            l++;
            r--;
        }
    }
    quick_sort(a, r - a + 1);
    quick_sort(l, a + n - l);
}
```

Listing 5.1: A quick sort algorithm implemented in C to run on the Neorv32.

5.3.3 Parameters

For the SFI Formula 2.3, the same parameters were used as in Section 5.2.3. The same parameters were used to be able to compare the results of this experiment with the smaller DUT. With these parameters and an injection space of 341107970, the SFI formula results in an injection campaign of 541 injection points.

The campaign will only be run as an SFI campaign based on Formula 2.3. A full campaign has an injection space of 341107970. If a single injection takes, for example, 50ms, the total time of one campaign would take 197 days. Previous findings from [15] yielded a time of 720ms per injection, which would mean a significantly longer duration of a full campaign.

By not being able to run a full campaign, the AVF of a full campaign will be unknown. To be able to know the accuracy of the system, the experiment is performed with an increasing number of injection points per campaign, starting from the SFI campaign and multiplying by 5 to 50, with increments of 5.

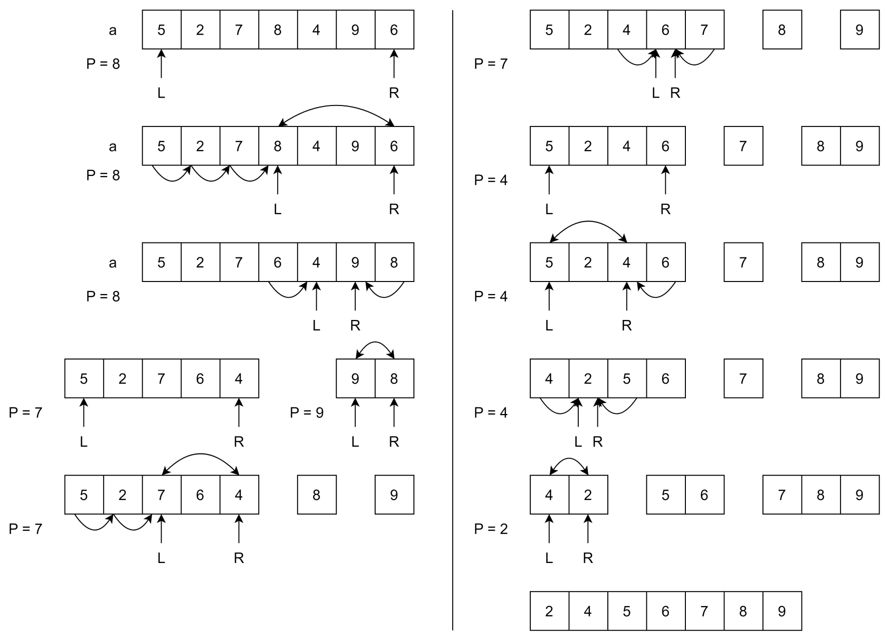


Figure 5.6: Example of the quick sort algorithm.

5.3.4 Results

The SFI campaigns were run as described in Section 5.3.3. Each campaign was run 10 times. The results of the experiment can be seen in Figure 5.7. The AVF of each campaign can be seen. The mean is based on the campaigns with the same number of injection points.

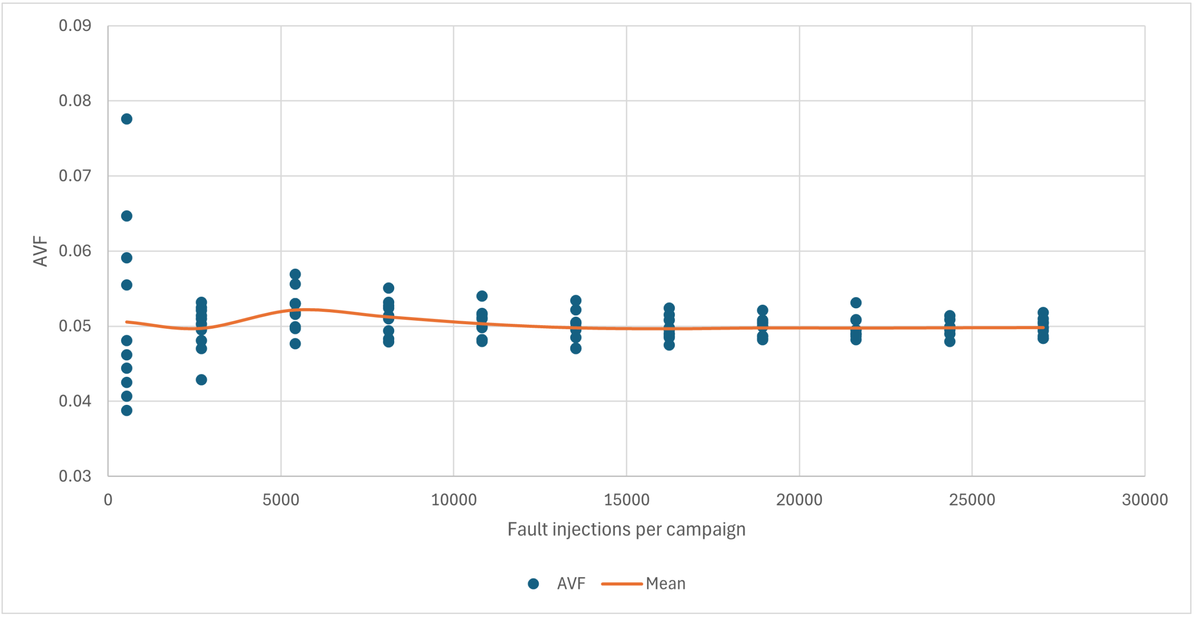


Figure 5.7: AVF of the campaigns with differing number of fault injections per campaign. The mean is the average of the campaigns with the same number of fault injections.

The AVF of the campaigns are more spread the lower the number of injections per

Campaign length	541	2705	5410	8115	10820	13525
Mean AVF	0.0504	0.0519	0.0522	0.0512	0.0503	0.0498
Campaign length	16230	18935	21640	24345	27050	
Mean AVF	0.0497	0.0498	0.0497	0.0498	0.0498	

Table 5.2: Mean AVF of 10 campaigns with the respective campaign lengths.

campaign. This is to be expected, because a smaller area of the DUT is injected, so a smaller chance to have an accurate measurement compared to the real AVF. The more injection points, the more the AVF converges. This can be seen in the AVF, but also in the mean of the AVF. The mean AVF varies with the number of injection points, but the mean AVF converges to a constant. It can be assumed that the point the data converges to is the actual AVF of the system, because the way the SFI Formula 2.3 behaves, the error margin gets closer to 0, the larger the number of injection points. This means the AVF of the system should be 0.05. In Table 5.2, the mean AVF of each campaign length can be seen.

The mean time of an injection in this experiment was 77ms. This is dependent on the DUT and the benchmark, because the time to run the benchmark is also in the 77ms. During the running of the benchmark, the benchmark is paused at the moment of injecting and continues after injecting. The time that the benchmark is paused for is 2ms. This is included in the 77ms of the total time per injection.

When you look at the results in Figure 5.7, it can be seen that there are gaps at the lowest 2 amount of injection points. To be able to analyse the error from the expected AVF of 0.05 at every number of injection points, it is necessary to fill the gaps. The campaigns of 541 and 2705 injection campaigns were run again 60 times each. While these measurements were done at another time than the previous measurements, the environment, parameters and injection tool were the same. The additional measurements can be seen in Figure 5.8. Here can be seen that the spread of the points follows a curve. This curve is the error margin of the data compared to the actual AVF of the system. An approximation of the error margin can be seen in Figure 5.9. This is based on the measured data points. The actual error margin based on the parameters can be seen in Figure 5.10. The approximated margin of error is an error margin 10 times larger than the expected margin of error used as a parameter in Formula 2.3.

5.3.5 Analysis

When looking at these results, the behaviour that is shown is expected, where the mean of multiple campaigns is close to the probable AVF. Especially when the number of fault injections in the campaign is increased.

What is unexpected is the resulting error margin. The approximate error margin is significantly larger than the error margin that was used as a parameter. This behaviour could be caused by the use of the SFI Formula 2.3 in this experimental setup. To prove this, the experiment should be repeated with another experimental setup to see if this behaviour changes.

However, the results do show an accurate AVF measurement when taking the mean of 10 campaigns with more than 10820 injection points. This would mean a

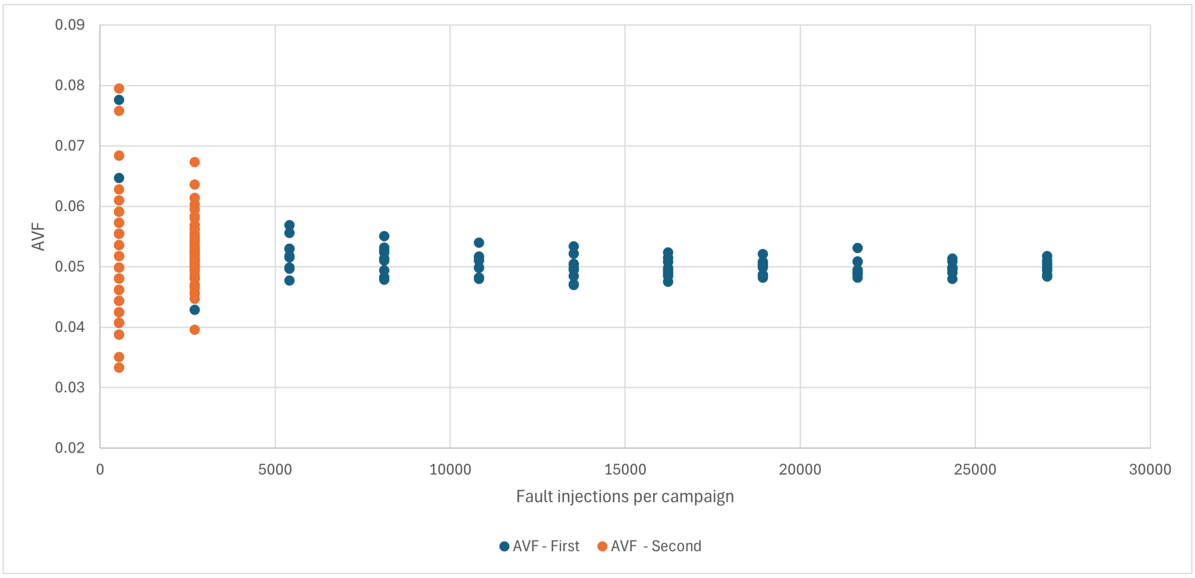


Figure 5.8: Additional AVF measurements of 541 and 2705 injection point campaigns in orange.

total of 108200 injections. Compared to the 341107970 injection points of the full injection space, this is an improvement of 3100 times. With an average injection time of 77ms, an accurate AVF measurement would only take 2.3 hours instead of 304 days for a campaign of the full injection space.

Another noticeable result is the distribution of the points of the lowest number of injections per campaign in Figure 5.7. The points are equally spaced and not more dense around the middle than would be expected. This can be explained by the way the AVF is calculated for SFI campaigns. The number of faults detected, divided by the total number of injections in the campaign. The AVF can only be as accurate as $\frac{1}{n}$ where n is the result of Formula 2.3. In this experiment $n = 541$, so the precision of the measurement is $\frac{1}{541} = 0.0018$. This means a difference of 1 fault to an AVF of 0.05 is already a difference of 3.7%.

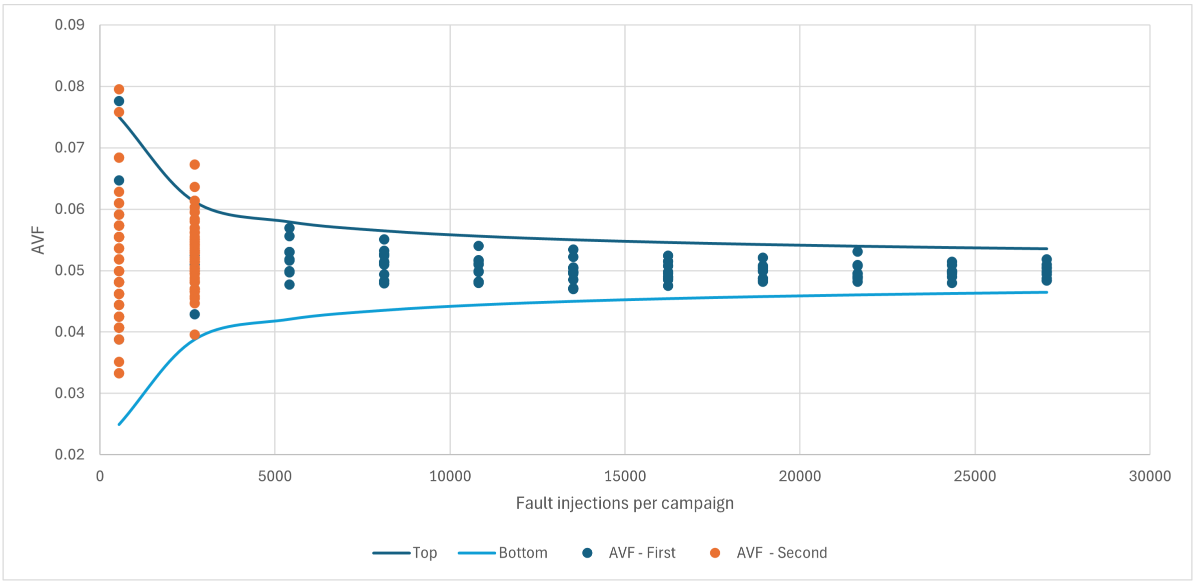


Figure 5.9: The approximate margin of error based on the data points.

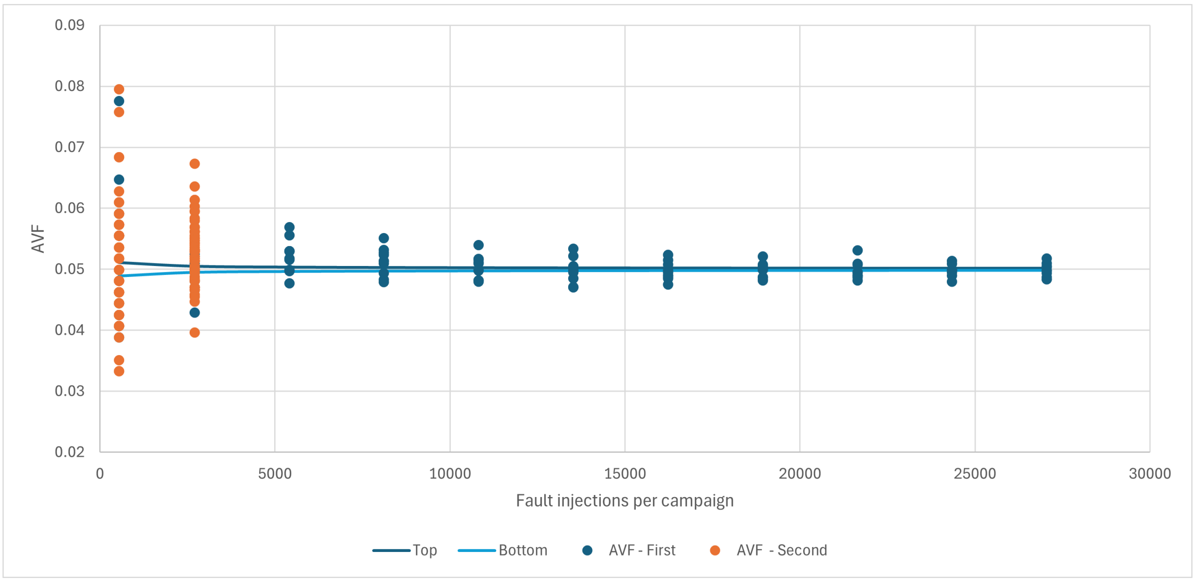


Figure 5.10: The margin of error based on the parameters set.

Chapter 6

Conclusion and discussion

Microelectronics are increasingly more used in radiation-harsh environments. This, in combination with microelectronics becoming smaller and denser, the risk of an SEU becomes significantly higher. That is why it is crucial to test the vulnerability of microelectronics against radiation and soft errors. However, testing microelectronics against radiation can be costly and destructive to the device. An alternative to this can be emulation-based fault injections. This is where the hardware is emulated on an FPGA and injected with the errors to simulate the soft errors caused by radiation. Through a fault injection campaign, the vulnerability of hardware to soft errors can be measured. However, a fault injection campaign can become time-consuming when the hardware to be tested gets bigger and the program on the hardware gets longer.

This thesis has presented a way to accelerate fault injection campaigns. This was done by utilising statistical fault injections in a campaign generator, where only a limited number of points are injected with an error. By using this, a campaign can be significantly shortened, to be able to test hardware that was before too large to test in a feasible amount of time. The speed-up that was achieved was that an SFI campaign is 3100x less than a full campaign. This brings an injection space that would take 304 days to only 2.3 hours, while still keeping an accurate measurement within 1%.

Further, this thesis has made the contribution of creating a practical implementation of SFI and has provided research into the use of SFI in an emulation-based fault injection tool. This thesis has also shown the use of SFI in combination with the Fretz framework to create a fast fault injection setup.

With this setup, an AVF analysis was performed on a large processor, the Neorv32 with a RISC-V architecture. By using the SFI campaign generator, vulnerable elements in a system can be found, and mitigation techniques in processors can be evaluated faster.

The experiments presented in this paper have shown the acceleration of using SFI campaigns compared to a full campaign, with a speed-up of 3100x. Significantly decreasing the number of injections from 341107970 to 108200 for a Neorv32 with a quick sort as a benchmark. The experiments have also shown that the mean of multiple AVF measurements using SFI campaigns are accurate to within 1%.

However, the accuracy of a single SFI campaign was not as expected and showed an increase of error margin up to 10 times. Future research should investigate the cause of this inaccuracy and possibly reduce the amount of fault injections needed for an accurate AVF measurement. Further research should also look into the use of the campaign generator with other fault injection tools to see if these results are reproduced or possibly improved.

Another limitation of SFI is that the AVF can only be as precise as $\frac{1}{n}$ where n is the result of Formula 2.3. The precision is especially impactful on DUTs with a small AVF like the Neorv32. This could also be the reason for the high error margin on the experiment of the Neorv32. This was not further explored in this thesis and should be investigated in further research. Systems keep increasing in size, and thus, an increased analysis duration. A lower error margin would decrease the analysis duration and increase the number of systems that can be analysed.

In conclusion, this thesis has shown that employing Statistical fault injection campaigns substantially accelerates the measurement process of a system's AVF, while maintaining accuracy. This method enables the assessment of AVF in systems that were formerly too large to be measured.

Bibliography

[1] G. Trinh, O. Formoso, C. Gregg, E. Taylor, K. Cheung, D. Catanoso, and T. Olatunde, “Hardware Autonomy for Space Infrastructure,” in *2023 IEEE Aerospace Conference*, Mar. 2023, pp. 1–6, iSSN: 1095-323X. [Online]. Available: https://ieeexplore.ieee.org/document/10115601

[2] Q. Huang and J. Jiang, “An overview of radiation effects on electronic devices under severe accident conditions in NPPs, rad-hardened design techniques and simulation tools,” *Progress in Nuclear Energy*, vol. 114, pp. 105–120, Jul. 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0149197019300563

[3] H. M. Quinn, D. A. Black, W. H. Robinson, and S. P. Buchner, “Fault Simulation and Emulation Tools to Augment Radiation-Hardness Assurance Testing,” *IEEE Transactions on Nuclear Science*, vol. 60, no. 3, pp. 2119–2142, Jun. 2013, conference Name: IEEE Transactions on Nuclear Science. [Online]. Available: https://ieeexplore.ieee.org/document/6519339

[4] G. Papadimitriou and D. Gizopoulos, “Demystifying the System Vulnerability Stack: Transient Fault Effects Across the Layers,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2021, pp. 902–915, iSSN: 2575-713X. [Online]. Available: https://ieeexplore.ieee.org/document/9499847

[5] M. Eslami, B. Ghavami, M. Raji, and A. Mahani, “A survey on fault injection methods of digital integrated circuits,” *Integration*, vol. 71, pp. 154–163, Mar. 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016792601930402X

[6] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, “Statistical fault injection: Quantified error and confidence,” in *Automation & Test in Europe Conference & Exhibition 2009 Design*, Apr. 2009, pp. 502–506, iSSN: 1558-1101. [Online]. Available: https://ieeexplore.ieee.org/document/5090716/?arnumber=5090716

[7] S. Mukherjee, J. Emer, and S. Reinhardt, “The soft error problem: an architectural perspective,” in *11th International Symposium on High-Performance Computer Architecture*, Feb. 2005, pp. 243–247, iSSN: 2378-203X. [Online]. Available: https://ieeexplore.ieee.org/document/1385945

[8] T. Heijmen, “Soft Errors from Space to Ground: Historical Overview, Empirical Evidence, and Future Trends,” in *Soft Errors in Modern Electronic Systems*, M. Nicolaidis, Ed. Boston, MA: Springer US, 2011, pp. 1–25. [Online]. Available: https://doi.org/10.1007/978-1-4419-6993-4_1

[9] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, “A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor,” in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, Dec. 2003, pp. 29–40. [Online]. Available: https://ieeexplore.ieee.org/document/1253181/?arnumber=1253181

[10] H. Madeira, M. Rela, F. Moreira, and J. G. Silva, “RIFLE: A general purpose pin-level fault injector,” in *Dependable Computing — EDCC-1*, K. Echtle, D. Hammer, and D. Powell, Eds. Berlin, Heidelberg: Springer, 1994, pp. 197–216.

[11] S. P. Buchner, F. Miller, V. Pouget, and D. P. McMorrow, “Pulsed-Laser Testing for Single-Event Effects Investigations,” *IEEE Transactions on Nuclear Science*, vol. 60, no. 3, pp. 1852–1875, Jun. 2013, conference Name: IEEE Transactions on Nuclear Science. [Online]. Available: https://ieeexplore.ieee.org/document/6510515/?arnumber=6510515

[12] T. Vanát, J. Pospíil, F. Kríek, J. Ferencei, and H. Kubátová, “A System for Radiation Testing and Physical Fault Injection into the FPGAs and Other Electronics,” in *2015 Euromicro Conference on Digital System Design*, Aug. 2015, pp. 205–210. [Online]. Available: https://ieeexplore.ieee.org/document/7302271/?arnumber=7302271

[13] “Veripool.” [Online]. Available: https://www.veripool.org/verilator/

[14] A. Sari, V. Vlagkoulis, and M. Psarakis, “An open-source framework for Xilinx FPGA reliability evaluation,” in *Proc. Workshop Open Source Design Autom.(OSDA)*, 2019, pp. 1–6.

[15] T. T. Smit, “Investigating RISC-V hardware for autonomy in Space,” Apr. 2024.

[16] S. Nolting and A. T. A. Contributors, “The NEORV32 RISC-V Processor,” Feb. 2025. [Online]. Available: https://github.com/stnolting/neorv32

[17] N. J. Wang, A. Mahesri, and S. J. Patel, “Examining ACE Analysis Reliability Estimates Using Fault-Injection.”

[18] M. Maniatakos, N. Karimi, C. Tirumurti, A. Jas, and Y. Makris, “Instruction-Level Impact Analysis of Low-Level Faults in a Modern Microprocessor Controller,” *IEEE Transactions on Computers*, vol. 60, no. 9, pp. 1260–1273, Sep. 2011, conference Name: IEEE Transactions on Computers. [Online]. Available: https://ieeexplore.ieee.org/document/5432157

[19] J. Geier and D. Mueller-Gritschneder, “vRTLmod: An LLVM based Open-source Tool to Enable Fault Injection in Verilator RTL Simulations,” in *Proceedings of the 20th ACM International Conference on Computing Frontiers*, ser. CF ’23.

New York, NY, USA: Association for Computing Machinery, Aug. 2023, pp. 387–388. [Online]. Available: https://doi.org/10.1145/3587135.3591435

[20] A. Chatzidimitriou and D. Gizopoulos, “Anatomy of microarchitecture-level reliability assessment: Throughput and accuracy,” in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2016, pp. 69–78. [Online]. Available: https://ieeexplore.ieee.org/document/7482075

[21] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[22] M. Ebrahimi, A. Mohammadi, A. Ejlali, and S. G. Miremadi, “A fast, flexible, and easy-to-develop FPGA-based fault injection technique,” *Microelectronics Reliability*, vol. 54, no. 5, pp. 1000–1008, May 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0026271414000067

[23] P. Ramachandran, P. Kudva, J. Kellington, J. Schumann, and P. Sanda, “Statistical Fault Injection,” in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, Jun. 2008, pp. 122–127, iSSN: 2158-3927. [Online]. Available: https://ieeexplore.ieee.org/document/4630080

[24] M. Ebrahimi, N. Sayed, M. Rashvand, and M. B. Tahoori, “Fault injection acceleration by architectural importance sampling,” in *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct. 2015, pp. 212–219. [Online]. Available: https://ieeexplore.ieee.org/document/7331384

[25] M. Kaliorakis, D. Gizopoulos, R. Canal, and A. Gonzalez, “MeRLiN: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment,” in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2017, pp. 241–254. [Online]. Available: https://ieeexplore.ieee.org/document/8192475/?arnumber=8192475

[26] M. Matsumoto and T. Nishimura, “Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator,” *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, Jan. 1998. [Online]. Available: https://dl.acm.org/doi/10.1145/272991.272995

[27] “lanl/benchmark_codes,” Sep. 2024, original-date: 2015-12-21T22:30:04Z. [Online]. Available: https://github.com/lanl/benchmark_codes

Appendix A

SFI campaign generator code listing

```python
import pandas as pd
import argparse
import random

from scipy.stats import norm
import re
import time
import os

class StatisticalInjection:
    def __init__(self, seed = 0):
        if seed != 0:
            self._set_seed(seed)
        self._verbose = False

    def _set_seed(self, seed = 0):
        random.seed(seed)

    def _random_seed(self):
        random.seed()

    def _normal_quantile(self, percentage, mean=0, std_dev=1):
        # if not (0 <= percentage <= 1):
        #     raise ValueError("Percentage must be between 0 and 1.")
        return norm.ppf(percentage, loc=mean, scale=std_dev)

    def _number_of_FI(self, population_size, estimated_proportion = 0.5, margin_of_error = 0.05, cut_off_point = 0.99, cop_percentage = True):
        if cop_percentage:
            cut_off_point = self._normal_quantile(cut_off_point)
        return population_size/(((margin_of_error**2) * ((population_size - 1)/((cut_off_point**2) * estimated_proportion * (1 - estimated_proportion)))) + 1)

    def _ll_parser(self, file=""):
        if file == "":
            return None
        data = pd.read_csv(file, sep="\s+", comment=";",
                           names=["type", "offset", "frameaddress", "frameoffset", "block", "info1", "info2"],
                           skiprows=[0, 32], low_memory=False)
        data["location"] = data["block"].str.extract(r'X(?P<location_x>\d+)Y(?P<location_y>\d+)').astype(int).apply(
            tuple, axis=1)
        data["block_type"] = data["block"].str.extract(r'(?<==)(?P<type>\w+)(?=_)')
        data["ram_id"] = data["info1"].str.extract(r'(?<=Ram=)(?P<ram_id>\w+)(?=:)')
        bit_info = data["info1"].str.extract(r'(?<=:)(?P<bit_info>\w+)', expand=False)
        data["bit_type"] = bit_info.str.extract(r'(?P<bit_type>\D+)')
        data["bit_number"] = bit_info.str.extract(r'(?P<bit_type>\d+)')
        data["latch_info"] = data["info1"].str.extract(r'(?<=Latch=)(?P<latch_info>\S+)')
        data["net_info"] = data["info2"].str.extract(r'(?<=Net=)(?P<net_info>\S+)')

        return data

    def _filter_bram(self, data:pd.DataFrame):
        dropIndex = []
        for index, row in data.iterrows():
            if "RAMB" in row["block_type"]:
                dropIndex.append(index)
        if self._verbose: print("Dropping", len(dropIndex), "of BRAM")
        data.drop(dropIndex, axis=0, inplace=True)
        return data

    def _filter_muldiv(self, data:pd.DataFrame):
        dropIndex = []
        for index, row in data.iterrows():
            if "muldiv" in row["net_info"]:
                dropIndex.append(index)
        if self._verbose: print("Dropping", len(dropIndex), "of muldiv")
        data.drop(dropIndex, axis=0, inplace=True)
        return data

    def _filter_controller(self, data:pd.DataFrame):
        dropIndex = []
        for index, row in data.iterrows():
            if "controller" in row["net_info"]:
                dropIndex.append(index)
        if self._verbose: print("Dropping", len(dropIndex), "of controller")
        data.drop(dropIndex, axis=0, inplace=True)
        return data

    def _filter_cycle_counter(self, data:pd.DataFrame):
        dropIndex = []
        for index, row in data.iterrows():
            if "cycle_counter" in row["net_info"]:
                dropIndex.append(index)
        if self._verbose: print("Dropping", len(dropIndex), "of cycle_counter")
        data.drop(dropIndex, axis=0, inplace=True)
        return data

    def _read_asm(self, file=""):
        if not file:
            file = "neorv32/sw/benchmark/qsort/main.asm"
        programCounters = []
        with open(file, "r") as f:
            for x in f:
                if ":" in x[:5]:
                    # print(x)
                    m = re.search('(.+?):', x)
                    if m:
                        programCounters.append(m.group(1).strip())
        # print(programCounters)
        programCounters = [int(num, 16) for num in programCounters]
        return [hex(num) for num in programCounters]

    def _clock_count_fip(self, end:int, skip:[[int, int]] = None):
        clock_counters = []
        for i in range(end+1):
            clock_counters.append(i)

        for j in skip:
            begin, end = 0, 0
            for index, i in enumerate(clock_counters):
                if i == j[0]:
                    begin = index
                if i == j[1]:
                    end = index
            if begin != end and begin < end:
                clock_counters = clock_counters[:begin] + clock_counters[
↪ end:]

        # print(clock_counters)
        return list(map(str, clock_counters))


    def _list_to_csv(self, outlist, file="out.csv"):
        with open(file, "w") as f:
            f.write("frameaddress,frameoffset,program_counter\n")
            for point in outlist:
                f.write(point["frameaddress"] + "," + point["frameoffset"]
↪  + "," + point["program_counter"] + "\n")

    def _filter_important_data(self, data:pd.DataFrame):
        data.drop(["type", "offset", "block", "info1", "info2",
                   "location", "block_type", "ram_id", "bit_type",
                   "bit_number", "latch_info", "net_info"], axis=1,
↪ inplace=True)
        data.reset_index(drop=True, inplace=True)
        return data

    def _random_sampled_combine(self, lldata:pd.DataFrame, bmdata:[str],
↪ mult = 1):
        if mult == 0: mult = 1

        total_fi_space = lldata.shape[0] * len(bmdata)
        sampled_space = int(self._number_of_FI(total_fi_space)) * mult

        if self._verbose: print("Sampled space is", sampled_space)

        frameaddress_list = lldata["frameaddress"].values.tolist()
        frameoffset_list = lldata["frameoffset"].values.tolist()

        injectionPoints = []
        for i in range(sampled_space):
            llsample = random.choice(range(len(frameaddress_list)))
            bmsample = random.choice(range(len(bmdata)))
            injectionPoints.append({"frameaddress": frameaddress_list[
↪ llsample], "frameoffset": str(frameoffset_list[llsample]), "
↪ program_counter": bmdata[bmsample]})

        return injectionPoints

    def _full_injection_combine(self, lldata:pd.DataFrame, bmdata:[str]):
        total_fi_space = lldata.shape[0] * len(bmdata)
        if self._verbose: print("Full space is", total_fi_space)
        frameaddress_list = lldata["frameaddress"].values.tolist()
        frameoffset_list = lldata["frameoffset"].values.tolist()

        injectionPoints = []
        for fa in range(len(frameaddress_list)):
            for bm in range(len(bmdata)):
                injectionPoints.append({"frameaddress": frameaddress_list[
↪ fa], "frameoffset": str(frameoffset_list[fa]), "program_counter":
↪ bmdata[bm]})

        return injectionPoints

    def run(self, args=None):
        parser = argparse.ArgumentParser(
            prog="Statistical Injection",
            description="Hardware and benchmark parser for statistical
↪ fault injection")
        benchmark = parser.add_mutually_exclusive_group(required=True)
        benchmark.add_argument("-a", "--asm-file", help="path to .asm file
↪ ", type=str)
        benchmark.add_argument("-c", "--clock-cycle-count", action="
↪ store_true", help="fault injection based on clock cycle count")

        group = parser.add_argument_group("Campaign generation settings")
        group.add_argument("-l", "--logic-location-file", help="path to .
↪ ll file", type=str)
        group.add_argument("-s", "--cc-start", help="Specify the start of
↪ the clock cycle count", type=int)
        group.add_argument("-e", "--cc-end", help="Specify the start of
↪ the clock cycle count", type=int)
        group.add_argument("-o", "--output-file", help="path to output
↪ file", type=str)
        group.add_argument("-m", "--multiple", help="Generate multiple
↪ campaign files, specify the number of campaigns to generate", type=
↪ int)
        group.add_argument("-b", "--filter-bram", action="store_false",
↪ help="Specify to filter bram from injection points, default is true"
↪ )
        group.add_argument("--filter-muldiv", action="store_true", help="
↪ Specify to filter muldiv from injection points, default is false")
        group.add_argument("--skip", nargs="+", help="Skip these values of
↪  clock counters, specify even amount of numbers, last odd will be
```

```
    ↪ ignored. Only works with -c flag”, type=int)
        group.add_argument(”--seed”, help=”Specify the seed value for the
    ↪ random selection of the injetion points”, type=int)
        group.add_argument(”-d”, ”--output-dir”, help=”output directory
    ↪ relative to the base of the project, default is campaigns”, type=str
    ↪ )
        group.add_argument(”-f”, ”--full-campaign”, help=”Generate a full
    ↪ campaign, default is false, ignores increasing and multiplied”,
    ↪ action=”store_true”)

        multiply_group = parser.add_mutually_exclusive_group(required=
    ↪ False)
        multiply_group.add_argument(”--increasing”, help=”When generating
    ↪ multiple campaigns, increase the size of each campaign multiplied by
    ↪  the index of the campaign multiplied by the number given, eg --
    ↪ increasing 5 -> multiplier of 1 times 5, 2 times 5 etc., ”
                                      , type=int)
        multiply_group.add_argument(”--multiplied”, help=”When generating
    ↪ campaigns multiply the sample space by this amount”, type=int)

        settings_group = parser.add_argument_group(”Settings”)
        settings_group.add_argument(”-v”, ”--verbose”, action=”store_true”
    ↪ , help=”Turn on verbose mode, default is false”)

        args = parser.parse_args()

        llFile = ”zedboard-vivado/bitstream.ll”
        asmFile = ”neorv32/sw/benchmark/qsort/main.asm”

        self._verbose = args.verbose

        if args.logic_location_file:
            llFile = args.logic_location_file

        if args.asm_file:
            asmFile = args.asm_file
        elif None in [args.cc_start, args.cc_end]:
            raise parser.error(”Specify the start and end of the clock
    ↪ cycle count”)

        skip = []
        if args.skip:
            for i, value in enumerate(args.skip):
                if i%2 == 0 and not i == len(args.skip)-1: skip.append([
    ↪ args.skip[i], args.skip[i+1]])

        outfile = ”campaign.csv”
        if args.output_file:
            outfile = args.output_file

        outDir = ”campaigns/”
        if args.output_dir:
            outDir = args.output_dir
            if outDir[-1] != ”/”:
                outDir += ”/”

        if self._verbose: print(”Reading logic location file: ” + llFile)
```

```python
        llData = self._ll_parser(llFile)

        # filter for all RAMB, because injection there is not possible yet
        if args.filter_bram:
            llData = self._filter_bram(llData)

        if args.filter_muldiv:
            llData = self._filter_muldiv(llData)

        # filter for all registers belonging to cycle counter, because FI here will kill the program
        llData = self._filter_cycle_counter(llData)

        # filter for all registers belonging to controller, is the uart communication, FI here will kill communication
        llData = self._filter_controller(llData)

        # delete unnecessary info for csv file
        llData = self._filter_important_data(llData)

        if args.asm_file:
            if self._verbose: print("Reading benchmark program: " + asmFile)
            bmData = self._read_asm(asmFile)
        else:
            skip = [[0, args.cc_start]] + skip
            if self._verbose: print("Skipping the following clock cycle intervals:", skip)
            bmData = self._clock_count_fip(args.cc_end, skip)
            # bmData = self._clock_count_fip(153571, [[90587, 96180]])

        if self._verbose: print("Total injection space = ", str(llData.shape[0]), "*", str(len(bmData)), " = " + str(llData.shape[0] * len(bmData)))

        time_per_injection = 0.060 #s

        # I am truly sorry for this next part, but I was not going to refactor the whole thing
        # (if it works don't touch it)
        if not args.multiple:
            index = outfile.find(".csv")
            if index != -1 and index != 0:
                file_extension = outfile[index:]
                outfile = outfile[:index]
            else:
                file_extension = ".csv"
            if args.full_campaign:
                selection = self._full_injection_combine(llData, bmData)
                outfile = outfile + "-full" + str(len(selection))
            else:
                if args.seed:
                    self._set_seed(args.seed)
                    outfile = outfile + "-seed" + str(args.seed)
                if args.multiplied:
                    selection = self._random_sampled_combine(llData, bmData, args.increasing)
                    outfile = outfile + "-multiplied" + str(args.multiplied)
                else:
                    selection = self._random_sampled_combine(llData, bmData)
            file = outDir + outfile + time.strftime("-%Mm_%Ss", time.gmtime(int(len(selection)*time_per_injection))) + file_extension
            os.makedirs(os.path.dirname(file), exist_ok=True)
            self._list_to_csv(selection, file)
            print("Campaign generated and stored at:", file)
        else:
            self._random_seed()
            index = outfile.find(".csv")
            if index != -1 and index != 0:
                file_extension = outfile[index:]
                outfile = outfile[:index]
            else:
                file_extension = ".csv"
            for i in range(args.multiple):
                if args.seed and not args.full_campaign:
                    self._set_seed(args.seed + i)
                    file = outDir + outfile + "-" + str(i+1) + "-seed" + str(args.seed + i)
                else:
                    file = outDir + outfile + "-" + str(i+1)
                if args.full_campaign:
                    selection = self._full_injection_combine(llData, bmData)
                    file = file + "-full" + str(len(selection))
                else:
                    if args.increasing:
                        if args.increasing == 1:
                            selection = self._random_sampled_combine(llData, bmData, i+1)
                            file = file + "-injectionpoints" + str(len(selection)) #str(i*args.increasing)
                        else:
                            selection = self._random_sampled_combine(llData, bmData, i*args.increasing)
                            file = file + "-injectionpoints" + str(len(selection)) #str(i*args.increasing)
                    elif args.multiplied:
                        selection = self._random_sampled_combine(llData, bmData, args.multiplied)
                        file = file + "-injectionpoints" + str(len(selection)) #str(args.multiplied)
                    else:
                        selection = self._random_sampled_combine(llData, bmData)
                file = file + time.strftime("-%Mm_%Ss", time.gmtime(int(len(selection)*time_per_injection))) + file_extension
                os.makedirs(os.path.dirname(file), exist_ok=True)
                self._list_to_csv(selection, file)
                print("Campaign", i+1, "generated and stored at:", file)
```

Appendix B

Fault injection user application code listing

```python
from Communication.CommandManager import CommandManager
import Controller
import serial
import threading
import time
import pandas as pd
import os
import re

from tqdm import tqdm

class UserApplication:
    def __init__(self, project):
        self._command_manager = CommandManager(project.FpgaDevice,
                                               project.IpAddress,
                                               project.TcpPort)

        if os.name == 'nt':
            ser = serial.Serial("COM9", 115200) # windows
            self._dut_uart = serial.Serial("COM8", 19200)
        elif os.name == 'posix':
            ser = serial.Serial("/dev/ttyUSB1", 115200) # linux
            self._dut_uart = serial.Serial("/dev/ttyUSB0", 19200)
        else:
            ser = serial.Serial("COM9", 115200) # windows
            self._dut_uart = serial.Serial("COM8", 19200)
            # ser = serial.serial_for_url("socket://192.168.2.17:4196",
    ↪ baudrate=115200)
            # self._dut_uart = serial.serial_for_url("socket
    ↪ ://192.168.2.16:4196", baudrate=19200)
        self._clock_controller = Controller.UartController(ser)

        self._dut_output = b''
        self._deamon = threading.Thread(target=self._recorder, daemon=True
    ↪ )

    def _recorder(self):
        while self._dut_uart:
            size = self._dut_uart.in_waiting
            if size > 0:
                # print(size)
                data = self._dut_uart.read(size)
                self._dut_output += data

    def _clear_dut_output(self):
        self._dut_output = b''

    def _import_injection_file(self, file = ""):
        data = pd.read_csv(file)
        injectionCampaign = {"frameaddress": data["frameaddress"].values.tolist(), "frameoffset": data["frameoffset"].values.tolist(), "program_counter": data["program_counter"].values.tolist()}
        # print(injectionCampaign)
        return injectionCampaign

    def neo_injection(self):
        print("Injecting into neo design")
        self._deamon.start()
        cc = self._clock_controller
        with self._command_manager as cm:
            board_info = cm.ReadId()
            print(f"Board Info: {vars(board_info)}")
            cc.reset()

            # cc.set_prgm_stop(0x308) # clock cycle 90587(0x161db), start checker
            cc.set_prgm_stop(0x1dc) # clock cycle 5256(0x1488), nop before quicksort
            cc.enable(glbl=True, prgm=True)
            # cc.set_cycle_stop(90587)
            # cc.enable(glbl=True, cycle=True)
            while not cc.ready():
                pass

            print("Cycle count: " + hex(cc.cycle_counter()))
            # cm.InjectFault(0x0042141f, 1794, True)

            cc.set_prgm_stop(0x338) # clock cycle 96180(0x177b4), end of checker
            cc.enable(glbl=True, prgm=True)
            # cc.set_cycle_stop(96180)
            # cc.enable(glbl=True, cycle=True)
            while not cc.ready():
                pass

            print("Cycle count: " + hex(cc.cycle_counter()))

            # cc.set_prgm_stop(0x220)  # clock cycle 153571(0x257e3), end of benchmark
            # cc.enable(glbl=True, prgm=True)
            cc.set_cycle_stop(0x257e3)
            cc.enable(glbl=True, cycle=True)
            while not cc.ready():
                pass

            print("Cycle count: " + hex(cc.prgm_counter()))

            print(self._dut_output.hex(), self._dut_output)
            # print(self._dut_output.decode('utf-8', "ignore"))

    def mcm_injection(self):
        self._deamon.start()
        cc = self._clock_controller
        with self._command_manager as cm:
            board_info = cm.ReadId()
            print(f"Board Info: {vars(board_info)}")
            cc.reset()

            cc.set_cycle_stop(5)
            cc.enable(glbl=True, cycle=True)
            while not cc.ready():
                pass

            cm.InjectFault(0x00420d9f, 3172, True)

            cc.set_cycle_stop(17)
            cc.enable(glbl=True, cycle=True)
            while not cc.ready():
                pass

            print("Result:", hex(cc.prgm_counter()))

    def mcm_injection_campaign(self, campaign_dir = "src/Campaign/"):
        campaign_files = os.listdir(campaign_dir)
        campaign_files.sort()
        self._deamon.start()
        cc = self._clock_controller
        with self._command_manager as cm:
            for file in campaign_files:
                if os.path.isdir(campaign_dir + file):
                    sub_campaign_files = os.listdir(campaign_dir + file)
                    sub_campaign_files.sort()
                    for sub_file in sub_campaign_files:
                        if "campaign" in sub_file[:8]:
                            self._mcm_injection(cc, cm, campaign_dir +
    ↪ file + "/" + sub_file,
                                                            f"{campaign_dir}
    ↪ results-{file}.txt")
                else:
                    if "campaign" in file[:8]:
                        self._mcm_injection(cc, cm, campaign_dir + file, f
    ↪ "{campaign_dir}results.txt")

    def _mcm_injection(self, cc, cm, file="", result_file="results-mcm.txt
    ↪ "):
        injectionCampaign = self._import_injection_file(file)
        print("Start injecting campaign with:", file)
        board_info = cm.ReadId()
        # print(f"Board Info: {vars(board_info)}")

        total_correct = 0
        total_timeout = 0
        total_incorrect = 0
        campaign_length = len(injectionCampaign["frameaddress"])
        duration = []
        injection_time = []
        timeout_point = []
        incorrect_point = []
```

```python

    for index in tqdm(range(campaign_length), desc="Running campaign ..."):
        begin_time = time.time_ns()
        frameaddress = int(injectionCampaign["frameaddress"][index], 16)
        frameoffset = injectionCampaign["frameoffset"][index]
        cycle_counter = injectionCampaign["program_counter"][index]

        correct_output = 0x4444

        cc.reset()

        cc.set_cycle_stop(cycle_counter)
        cc.enable(glbl=True, cycle=True)
        while not cc.ready():
            pass

        pause_time = time.time_ns()
        cm.InjectFault(frameaddress, frameoffset, True)
        pause_time = time.time_ns() - pause_time

        cc.set_cycle_stop(17)
        cc.enable(glbl=True, cycle=True)
        while not cc.ready():
            if (time.time_ns() - begin_time) >= 5_000_000_000:
                total_timeout += 1
                timeout_point.append([str(frameaddress), str(frameoffset), str(cycle_counter)])
                break
            pass

        if cc.prgm_counter() == correct_output:
            total_correct += 1
        else:
            total_incorrect += 1
            incorrect_point.append([str(frameaddress), str(frameoffset), str(cycle_counter)])
        end_time = time.time_ns()
        duration.append(end_time - begin_time)
        injection_time.append(pause_time)

    with open(result_file, "a") as f:
        f.write(time.strftime("%a %d %b %Y - %H:%M:%S UTC +0000", time.gmtime()) + "\n")
        f.write("Campaign: " + file + "\n")
        f.write("AVF: " + str((campaign_length - total_correct) / campaign_length) + "\n")
        f.write("Campaign length: " + str(campaign_length) + "\n")
        f.write("Total correct: " + str(total_correct) + "\n")
        f.write("Total timeout: " + str(total_timeout) + "\n")
        mean_duration = int(sum(duration) / len(duration))
        f.write("Mean duration per injection: " + str(mean_duration / 1_000_000) + "ms" + "\n")
        f.write("Total duration of campaign: " + str(sum(duration) / 1_000_000) + "ms" + "\n")
        total_injectionTime = sum(injection_time) / 1_000_000
```

```python
            f.write("Mean time of injecting: " + str(total_injectionTime / len(injection_time)) + "ms" + "\n")
            f.write("\n")

        with open("src/Campaign/timeout_points-mcm.txt", "a") as f:
            f.write(time.strftime("%a %d %b %Y - %H:%M:%S UTC +0000", time.gmtime()) + "\n")
            f.write("Result file: " + result_file + "\n")
            f.write("Campaign: " + file + "\n")
            f.write("Timeout points:\n")
            f.write("Frameaddress - Frameoffset - Cycle counter\n")
            for point in timeout_point:
                f.write("{: <12}".format(point[0]) + " - " + "{: <11}".format(point[1]) + " - " + "{: <13}".format(
                    point[2]) + "\n")
            f.write("\n")

        with open("src/Campaign/incorrect_points-mcm.txt", "a") as f:
            f.write(time.strftime("%a %d %b %Y - %H:%M:%S UTC +0000", time.gmtime()) + "\n")
            f.write("Result file: " + result_file + "\n")
            f.write("Campaign: " + file + "\n")
            f.write("Incorrect points:\n")
            f.write("Frameaddress - Frameoffset - Cycle counter\n")
            for point in incorrect_point:
                f.write("{: <12}".format(point[0]) + " - " + "{: <11}".format(point[1]) + " - " + "{: <13}".format(
                    point[2]) + "\n")
            f.write("\n")

        print("AVF =", (campaign_length - total_correct) / campaign_length)
        print("Mean duration per injection:", mean_duration / 1_000_000, "ms")

    def fir_single_injection(self):
        self._deamon.start()
        cc = self._clock_controller
        with self._command_manager as cm:
            board_info = cm.ReadId()
            print(f"Board Info: {vars(board_info)}")
            cc.reset()

            correct_output = [0, 0, 3, 7, 12, 18, 25, 33, 42, 52, 52]
            correct = True

            # print("Inject led")
            # cm.InjectFault(0x0042221f, 2915, True)

            for i in range(11):
                cc.set_cycle_stop(i)
                cc.enable(glbl=True, cycle=True)
                while not cc.ready():
                    pass

                # if i == 1: cm.InjectFault(0x0040111f, 132, True)
                if not cc.prgm_counter() == correct_output[i]:
                    correct = False
                print(i, ":", cc.cycle_counter(), cc.prgm_counter(), cc.prgm_counter() == correct_output[i])

                # cc.set_cycle_stop(i+1)
                # cc.enable(glbl=True, cycle=True)
                # while not cc.ready():
                #     pass

                # print(cc.cycle_counter(), cc.prgm_counter())
            print("Successful run:", correct)


    def fir_injection(self, campaign_dir = "src/Campaign/"):
        campaign_files = os.listdir(campaign_dir)
        campaign_files.sort()
        self._deamon.start()
        cc = self._clock_controller
        with self._command_manager as cm:
            for file in campaign_files:
                if os.path.isdir(campaign_dir + file):
                    sub_campaign_files = os.listdir(campaign_dir + file)
                    sub_campaign_files.sort()
                    for sub_file in sub_campaign_files:
                        if "campaign" in sub_file[:8]:
                            self._fir_injection_campaign(cc, cm, campaign_dir + file + "/" + sub_file, f"{campaign_dir}results-{file}.txt")
                else:
                    if "campaign" in file[:8]:
                        self._fir_injection_campaign(cc, cm, campaign_dir + file, f"{campaign_dir}results.txt")

    def _fir_injection_campaign(self, cc, cm, file = "", result_file = "results-fir.txt"):
        injectionCampaign = self._import_injection_file(file)
        print("Start injecting campaign with:", file)
        board_info = cm.ReadId()
        # print(f"Board Info: {vars(board_info)}")

        total_correct = 0
        total_timeout = 0
        total_incorrect = 0
        campaign_length = len(injectionCampaign["frameaddress"])
        duration = []
        injection_time = []
        timeout_point = []
        incorrect_point = []

        for index in tqdm(range(campaign_length), desc="Running campaign..."):
            begin_time = time.time_ns()
            frameaddress = int(injectionCampaign["frameaddress"][index], 16)
            frameoffset = injectionCampaign["frameoffset"][index]
            cycle_counter = injectionCampaign["program_counter"][index]

            correct_output = [0, 0, 3, 7, 12, 18, 25, 33, 42, 52, 52]
            correct = True
            end = False

            cc.reset()

            pause_time = 0

            for i in range(11):
                cc.set_cycle_stop(i)
                cc.enable(glbl=True, cycle=True)
                while not cc.ready():
                    if (time.time_ns() - begin_time) >= 5_000_000_000:
                        total_timeout += 1
                        timeout_point.append([str(frameaddress), str(frameoffset), str(cycle_counter)])
                        end = True
                        break
                    pass

                if end:
                    break

                if i == 1:
                    pause_time = time.time_ns()
                    cm.InjectFault(0x0040111f, 132, True)
                    pause_time = time.time_ns() - pause_time

                if not cc.prgm_counter() == correct_output[i]:
                    correct = False
                    break
                # print(i, ":", cc.cycle_counter(), cc.prgm_counter(), cc.prgm_counter() == correct_output[i])


            if correct:
                total_correct += 1
            else:
                total_incorrect += 1
                incorrect_point.append([str(frameaddress), str(frameoffset), str(cycle_counter)])
            end_time = time.time_ns()
            duration.append(end_time - begin_time)
            injection_time.append(pause_time)

        with open(result_file, "a") as f:
            f.write(time.strftime("%a %d %b %Y - %H:%M:%S UTC +0000", time.gmtime()) + "\n")
            f.write("Campaign: " + file + "\n")
            f.write("AVF: " + str((campaign_length-total_correct)/campaign_length) + "\n")
            f.write("Campaign length: " + str(campaign_length) + "\n")
            f.write("Total correct: " + str(total_correct) + "\n")
            f.write("Total timeout: " + str(total_timeout) + "\n")
            mean_duration = int(sum(duration) / len(duration))
            f.write("Mean duration per injection: " + str(mean_duration/1_000_000) + "ms" + "\n")
```

```python
            f.write("Total duration of campaign: " + str(sum(duration)/1_000_000) + "ms" + "\n")
            total_injectionTime = sum(injection_time)/1_000_000
            f.write("Mean time of injecting: " + str(total_injectionTime/len(injection_time)) + "ms" + "\n")
            f.write("\n")

        with open("src/Campaign/timeout_points-fir.txt", "a") as f:
            f.write(time.strftime("%a %d %b %Y - %H:%M:%S UTC +0000", time.gmtime()) + "\n")
            f.write("Result file: " + result_file + "\n")
            f.write("Campaign: " + file + "\n")
            f.write("Timeout points:\n")
            f.write("Frameaddress - Frameoffset - Cycle counter\n")
            for point in timeout_point:
                f.write("{: <12}".format(point[0]) + " - " + "{: <11}".format(point[1]) + " - " + "{: <13}".format(point[2]) + "\n")
            f.write("\n")

        with open("src/Campaign/incorrect_points-fir.txt", "a") as f:
            f.write(time.strftime("%a %d %b %Y - %H:%M:%S UTC +0000", time.gmtime()) + "\n")
            f.write("Result file: " + result_file + "\n")
            f.write("Campaign: " + file + "\n")
            f.write("Incorrect points:\n")
            f.write("Frameaddress - Frameoffset - Cycle counter\n")
            for point in incorrect_point:
                f.write("{: <12}".format(point[0]) + " - " + "{: <11}".format(point[1]) + " - " + "{: <13}".format(point[2]) + "\n")
            f.write("\n")

        print("AVF =", (campaign_length-total_correct)/campaign_length)
        print("Mean duration per injection:", mean_duration/1_000_000, "ms")

    def example_injection(self):
        print("Injecting into example design")
        with self._command_manager as cm:
            board_info = cm.ReadId()
            print(f"Board Info: {vars(board_info)}")
            # Inject into latch - LD0
            cm.InjectFault(0x0042239f, 3044, True)
            # Inject into bram - LD3
            cm.InjectFault(0x00c20280, 2912, False)

    def neo_inject_pc(self):
        print("Injecting into neo design")
        self._deamon.start()
        cc = self._clock_controller
        with self._command_manager as cm:
            board_info = cm.ReadId()
            print(f"Board Info: {vars(board_info)}")
            cc.reset()

            cc.set_prgm_stop(0x1dc)
            cc.enable(glbl=True, prgm=True)
            while not cc.ready():
                pass

            print("1: " + hex(cc.prgm_counter()))
            cc.set_cycle_relative_stop(6)
            cc.enable(glbl=True, cycle=True)
            while not cc.ready():
                pass

            print("Inject")
            print("2: " + hex(cc.prgm_counter()))
            # cm.InjectFault(0x0042151f, 541, True) #bit 3 next pc
            # cm.InjectFault(0x0042141f, 579, True) #bit 4 next pc
            # cm.InjectFault(0x0042149f, 643, True) #bit 5 next pc
            # cm.InjectFault(0x0042141f, 546, True) #bit 6 next pc
            # cm.InjectFault(0x0042139f, 669, True) #bit 7 next pc
            # cm.InjectFault(0x0042139f, 708, True) #bit 8 next pc

            cc.set_cycle_relative_stop(8)
            cc.enable(glbl=True, cycle=True)
            # cc.set_prgm_stop(0x220)
            # cc.enable(glbl=True, prgm=True)
            # cc.set_cycle_stop(0xf00000)
            # cc.enable(glbl=True, cycle=True)
            while not cc.ready():
                pass
            print("3: " + hex(cc.prgm_counter()))

            cc.set_prgm_stop(0x220)
            cc.enable(glbl=True, prgm=True)
            while not cc.ready():
                pass

            print(self._dut_output.hex(), self._dut_output)
            # print(self._dut_output.decode('utf-8', "ignore"))

    def injection_campaign(self, file = ""):
        if not file:
            print("Campaign file required for injection campaign")
            return
        injectionCampaign = self._import_injection_file(file)
        print("Start injecting campaign with:", file)
        self._deamon.start()
        cc = self._clock_controller
        with self._command_manager as cm:
            board_info = cm.ReadId()
            # print(f"Board Info: {vars(board_info)}")

            total_correct = 0
            total_timeout = 0
            campaign_length = len(injectionCampaign["frameaddress"])
            duration = []

            # for index in range(campaign_length):
            for index in tqdm(range(campaign_length), desc="Running
    ↪ campaign..."):
                begin_time = time.time_ns()
                frameaddress = int(injectionCampaign["frameaddress"][index
   ↪ ], 16)
                # frameoffset = int(injectionCampaign["frameoffset"][index
   ↪ ], 10)
                frameoffset = injectionCampaign["frameoffset"][index]
                # program_counter = int(injectionCampaign["program_counter
   ↪ "][index], 10)
                program_counter = injectionCampaign["program_counter"][
   ↪ index]

                # print("Injection", index, "at:")
                # print("    Position:", hex(frameaddress), str(frameoffset
   ↪ ))
                # print("    Moment:   ", str(program_counter))

                cc.reset()

                # set stop at point in benchmark program
                cc.set_cycle_stop(program_counter)
                cc.enable(glbl=True, cycle=True)
                while not cc.ready():
                    pass

                # inject at frameaddress and frameoffset
                cm.InjectFault(frameaddress, frameoffset, True)

                cc.set_prgm_stop(0x220)
                cc.enable(glbl=True, prgm=True)
                while not cc.ready():
                    if time.time_ns() - begin_time >= 500_000_000:
                        total_timeout += 1
                        break
                    pass

                # print(self._dut_output)
                if self._dut_output == b'\x00\xff':
                    total_correct += 1
                self._clear_dut_output()
                end_time = time.time_ns()
                duration.append(end_time - begin_time)
                # print("Time of injection is:", end_time - begin_time, "
   ↪ ns")

            print("Total correct:", total_correct, "out of",
   ↪ campaign_length)
            print("AVF =", (campaign_length-total_correct)/campaign_length
   ↪ )
            print("Total timeout:", total_timeout)
            mean_duration = int(sum(duration) / len(duration))
            print("Mean duration per injection:", mean_duration/1_000_000,
   ↪  "ms")
            print("Total duration of campaign:", sum(duration)/1_000_000,
   ↪ "ms")

    def _injection_campaign(self, cc, cm, file = "", result_file = "
   ↪ results.txt"):
        injectionCampaign = self._import_injection_file(file)
        print("Start injecting campaign with:", file)
```

```python
        board_info = cm.ReadId()
        # print(f"Board Info: {vars(board_info)}")

        total_correct = 0
        total_timeout = 0
        total_incorrect = 0
        campaign_length = len(injectionCampaign["frameaddress"])
        duration = []
        injection_time = []
        timeout_point = []
        incorrect_point = []

        for index in tqdm(range(campaign_length), desc="Running campaign ..."):
            begin_time = time.time_ns()
            frameaddress = int(injectionCampaign["frameaddress"][index], 16)
            # frameoffset = int(injectionCampaign["frameoffset"][index], 10)
            frameoffset = injectionCampaign["frameoffset"][index]
            # program_counter = int(injectionCampaign["program_counter"][index], 10)
            program_counter = injectionCampaign["program_counter"][index]

            # print("Injection", index, "at:")
            # print("    Position:", hex(frameaddress), str(frameoffset))
            # print("    Moment:   ", str(program_counter))

            cc.reset()

            # set stop at point in benchmark program
            cc.set_cycle_stop(program_counter)
            cc.enable(glbl=True, cycle=True)
            while not cc.ready():
                pass

            pause_time = time.time_ns()

            # inject at frameaddress and frameoffset
            cm.InjectFault(frameaddress, frameoffset, True)

            pause_time = time.time_ns() - pause_time

            cc.set_prgm_stop(0x220)
            cc.enable(glbl=True, prgm=True)
            while not cc.ready():
                if (time.time_ns() - begin_time) - pause_time >= 1_000_000_000:
                    total_timeout += 1
                    timeout_point.append([str(frameaddress), str(frameoffset), str(program_counter)])
                    break
                pass

            # print(self._dut_output)
            if self._dut_output == b'\x00\xff':
                total_correct += 1
            else:
                total_incorrect += 1
                incorrect_point.append([str(frameaddress), str(frameoffset), str(program_counter)])
            self._clear_dut_output()
            end_time = time.time_ns()
            duration.append(end_time - begin_time)
            injection_time.append(pause_time)

        with open(result_file, "a") as f:
            f.write(time.strftime("%a %d %b %Y - %H:%M:%S UTC +0000", time.gmtime()) + "\n")
            f.write("Campaign: " + file + "\n")
            f.write("AVF: " + str((campaign_length-total_correct)/campaign_length) + "\n")
            f.write("Campaign length: " + str(campaign_length) + "\n")
            f.write("Total correct: " + str(total_correct) + "\n")
            f.write("Total timeout: " + str(total_timeout) + "\n")
            mean_duration = int(sum(duration) / len(duration))
            f.write("Mean duration per injection: " + str(mean_duration/1_000_000) + "ms" + "\n")
            f.write("Total duration of campaign: " + str(sum(duration)/1_000_000) + "ms" + "\n")
            total_injectionTime = sum(injection_time)/1_000_000
            f.write("Mean time of injecting: " + str(total_injectionTime/len(injection_time)) + "ms" + "\n")
            f.write("\n")

        with open("src/Campaign/timeout_points.txt", "a") as f:
            f.write(time.strftime("%a %d %b %Y - %H:%M:%S UTC +0000", time.gmtime()) + "\n")
            f.write("Result file: " + result_file + "\n")
            f.write("Campaign: " + file + "\n")
            f.write("Timeout points:\n")
            f.write("Frameaddress - Frameoffset - Cycle counter\n")
            for point in timeout_point:
                f.write("{: <12}".format(point[0]) + " - " + "{: <11}".format(point[1]) + " - " + "{: <13}".format(point[2]) + "\n")
            f.write("\n")

        with open("src/Campaign/incorrect_points.txt", "a") as f:
            f.write(time.strftime("%a %d %b %Y - %H:%M:%S UTC +0000", time.gmtime()) + "\n")
            f.write("Result file: " + result_file + "\n")
            f.write("Campaign: " + file + "\n")
            f.write("Incorrect points:\n")
            f.write("Frameaddress - Frameoffset - Cycle counter\n")
            for point in incorrect_point:
                f.write("{: <12}".format(point[0]) + " - " + "{: <11}".format(point[1]) + " - " + "{: <13}".format(point[2]) + "\n")
            f.write("\n")

        print("AVF =", (campaign_length-total_correct)/campaign_length)
        print("Mean duration per injection:", mean_duration/1_000_000, "ms")

    def multiple_campaign(self, campaign_dir = "src/Campaign/"):
        campaign_files = os.listdir(campaign_dir)
        campaign_files.sort()
        self._deamon.start()
        cc = self._clock_controller
        with self._command_manager as cm:
            for file in campaign_files:
                if os.path.isdir(campaign_dir + file):
                    sub_campaign_files = os.listdir(campaign_dir + file)
                    sub_campaign_files.sort()
                    for sub_file in sub_campaign_files:
                        if "campaign" in sub_file[:8]:
                            self._injection_campaign(cc, cm, campaign_dir + file + "/" + sub_file, f"{campaign_dir}results-{file}.txt")
                else:
                    if "campaign" in file[:8]:
                        self._injection_campaign(cc, cm, campaign_dir + file, f"{campaign_dir}results.txt")

    def run(self, campaign_file = ""):
        print("Current working directory:", os.getcwd())
        # self.example_injection()
        # self.neo_injection()
        if campaign_file:
            print("Starting campaign from file: " + campaign_file)
            self.injection_campaign(campaign_file)
        else:
            # print("No campaign file, running injection on pc")
            # self.neo_inject_pc()
            # self.neo_injection()
            # self.multiple_campaign()
            # self.fir_injection()
            # self.fir_single_injection()
            self.mcm_injection_campaign()
```

Appendix C

Multicycle multiplier DUT code listing

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library work;

entity multicycle_mult is
    Port (
        clk     : in  STD_LOGIC;
        reset   : in  STD_LOGIC;
        start   : in  STD_LOGIC;
        a       : in  STD_ULOGIC_VECTOR(15 downto 0);
        b       : in  STD_ULOGIC_VECTOR(15 downto 0);
        result  : out STD_ULOGIC_VECTOR(31 downto 0);
        done    : out STD_LOGIC;
      cycle_counter_o : out STD_ULOGIC_VECTOR(31 downto 0)
    );
attribute dont_touch : boolean;
attribute dont_touch of multicycle_mult : entity is true;
end multicycle_mult;

architecture Behavioral of multicycle_mult is
    signal a_reg, b_reg : STD_ULOGIC_VECTOR(31 downto 0);
    signal product      : unsigned(31 downto 0);
    signal count        : INTEGER range 0 to 16;
    signal busy         : STD_LOGIC;

    signal cycle_counter : unsigned(31 downto 0) := (others => '0');
begin
    process(clk, reset)
    begin
        if reset = '0' then
            a_reg   <= (others => '0');
            b_reg   <= (others => '0');
            product <= (others => '0');
            count   <= 0;
            busy    <= '0';
            done    <= '0';
          result  <= (others => '0');
          cycle_counter <= (others => '0');
        elsif rising_edge(clk) then
            if start = '1' and busy = '0' then
                a_reg   <= (31 downto 16 => '0') & a;
                b_reg   <= (31 downto 16 => '0') & b;
                product <= (others => '0');
                count   <= 0;
                busy    <= '1';
                done    <= '0';
            elsif busy = '1' then
                if count < 16 then
                    if b_reg(0) = '1' then
                        product(31 downto 0) <= product(31 downto 0) + unsigned(a_reg);
                    end if;
                    a_reg <= a_reg(30 downto 0) & '0'; -- shift left
                    b_reg <= '0' & b_reg(31 downto 1); -- shift right
                    count <= count + 1;
                else
                    busy <= '0';
                    done <= '1';
                result <= std_ulogic_vector(product);
                end if;
            end if;
          cycle_counter <= cycle_counter + 1;
          cycle_counter_o <= std_ulogic_vector(cycle_counter);
        end if;
    end process;

    --result <= product;
end Behavioral;
```

Appendix D

Benchmark quick sort code listing

```c
/**********************************************************************/
    ↪ /**
* @file benchmark/qsort/main.c
* @author Kevin Schrama
* @brief Qsort benchmark program.
**********************************************************************/

#include <neorv32.h>

/**********************************************************************/
    ↪ /**
* @name User configuration
**********************************************************************/
/**@{*/
/** UART BAUD rate */
#define BAUD_RATE 19200
/**@}*/

#define array_elements 180

void quick_sort(int *a, int n);
int checker(int golden_array[], int dut_array[]);

/**********************************************************************/
    ↪ /**
* Main function;
*
* @return 0 if execution was successful
**********************************************************************/
int main()
{

    int pattern[array_elements] = {
        23, -7, 42, 18, 0, -13, 56, 89, -22, 4,
        67, -99, 12, 33, 45, 78, -11, -8, 60, 14,
        -35, 50, 7, 24, -46, 92, -71, 8, 31, -6,
        100, -44, 9, 29, -53, 81, -25, 17, -19, 36,
        5, 72, -80, -1, 49, 3, 27, -64, 88, -90,
        19, 34, -72, 11, 44, -18, 68, 73, -84, -33,
        95, 26, 48, 13, -50, 6, 55, -15, 41, 70,
        -2, 20, 59, -28, 12, 87, 21, -61, 76, 39,
        28, -36, 74, 9, -47, 82, -4, 31, 62, -10,
        -99, 25, 57, 40, -85, 63, 35, 53, -26, 96,
        -54, 46, 77, -67, 15, 22, 38, -34, 64, -81,
        89, 14, 58, -5, 30, -9, 66, 47, -63, 91,
        -29, 42, 80, 3, -48, 18, 75, -27, 50, 98,
        -74, 37, 4, -41, 65, 11, 16, -32, 84, 44,
        -88, 12, 52, 79, -13, 26, 71, -19, 68, 6,
        -56, 45, 99, -8, 31, -22, 92, 38, -70, 25,
        -39, 85, 54, 7, -17, 40, 61, -77, 36, 43,
        -20, 69, 2, -49, 81, 17, -66, 74, 28, 19};

    int correct_pattern[array_elements] = {
        -99, -99, -90, -88, -85, -84, -81, -80, -77, -74,
        -72, -71, -70, -67, -66, -64, -63, -61, -56, -54,
        -53, -50, -49, -48, -47, -46, -44, -41, -39, -36,
        -35, -34, -33, -32, -29, -28, -27, -26, -25, -22,
        -22, -20, -19, -19, -18, -17, -15, -13, -13, -11,
        -10, -9, -8, -8, -7, -6, -5, -4, -2, -1,
        0, 2, 3, 3, 4, 4, 5, 6, 6, 7,
        7, 8, 9, 9, 11, 11, 12, 12, 12, 13,
        14, 14, 15, 16, 17, 17, 18, 18, 19, 19,
        20, 21, 22, 23, 24, 25, 25, 26, 26, 27,
        28, 28, 29, 30, 31, 31, 31, 33, 34, 35,
        36, 36, 37, 38, 38, 39, 40, 40, 41, 42,
        42, 43, 44, 44, 45, 45, 46, 47, 48,
        49, 50, 50, 52, 53, 54, 55, 56, 57, 58,
        59, 60, 61, 62, 63, 64, 65, 66, 67, 68,
        68, 69, 70, 71, 72, 73, 74, 74, 75, 76,
        77, 78, 79, 80, 81, 81, 82, 84, 85, 87,
        88, 89, 89, 91, 92, 92, 95, 96, 98, 99, 100};

    int n = sizeof pattern / sizeof pattern[0];

    // capture all exceptions and give debug info via UART
    // this is not required, but keeps us safe
    // neorv32_rte_setup();

    // setup UART at default baud rate, no interrupts
    neorv32_uart0_setup(BAUD_RATE, 0);

    // check available hardware extensions and compare with compiler flags
    //neorv32_rte_check_isa(0); // silent = 0 -> show message if isa
    ↪ mismatch

    asm("nop");
    quick_sort(pattern, n);
    asm("nop");
    int errors = checker(correct_pattern, pattern);

    neorv32_uart0_putc(errors);
    neorv32_uart0_putc(0xff);
    // for (int i = 0; i < array_elements; i++){
    //     neorv32_uart0_putc(pattern[i]);
    // }

    while(neorv32_uart0_tx_busy()){}

    asm("nop");

    return 0;
```

```
}

void quick_sort(int *a, int n)
{
    if (n < 2)
        return;
    int p = a[n / 2];
    int *l = a;
    int *r = a + n - 1;
    while (l <= r)
    {
        if (*l < p)
        {
            l++;
        }
        else if (*r > p)
        {
            r--;
        }
        else
        {
            int t = *l;
            *l = *r;
            *r = t;
            l++;
            r--;
        }
    }
    quick_sort(a, r - a + 1);
    quick_sort(l, a + n - l);
}

int checker(int golden_array[], int dut_array[])
{
    int num_of_errors = 0;

    for (int i = 0; i < array_elements; i++)
    {
        if (golden_array[i] != dut_array[i])
        {
            // printf("Element %d was wrong: %d -> %d\n", i, dut_array[i],
    ↪ golden_array[i]);
            num_of_errors++;
        }
    }

    return num_of_errors;
}
```