UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering,
Mathematics & Computer Science

Investigating RISC-V hardware for autonomy in space

Exact emulation-based fault injection
on a hardware accelerator

T. T. Smit

MSc. Thesis
April 2024

Acknowledgement

I would like to express my gratitude to Dr. ir. M. Ottavi for his support, and the opportunities he facilitated throughout the duration of this thesis. His extensive network provided me with unique insights and access to resources that allowed me to work with the ESA and enrich my research experience.

I am also thankful to Bruno Endres Forlin, for his dedicated guidance, constructive feedback, and continuous encouragement. I admire his expertise, scientific skills and positive attitude. This was invaluable in shaping the direction of my research and refining my ideas. The same goes for Dr. ing. K.H. Chen, who always took the time to check in on me and be interested in my latest developments.

Special thanks are due to Peter, Stijn and Ruben who generously devoted their time and effort to proofread and provide feedback on this thesis. Your insightful comments and suggestions have significantly improved the quality of this work.

Last but certainly not least, I am deeply grateful to Wike, my parents and other family, for their unwavering love, encouragement, and support throughout this journey. Their patience, understanding, and encouragement sustained me through the challenges and triumphs of completing this thesis.

To all those mentioned above, and to many others who have contributed in various ways, I extend my heartfelt thanks. This thesis would not have been possible without your guidance, support, and encouragement.

Tijmen

Summary

In this thesis, a novel emulation-based fault injection (FI) tool is introduced to assess the reliability of new space hardware.

Higher autonomy in outer space hardware, with the help of artificial intelligence (AI) accelerators, is a significant change expected in the coming century. This is necessitated by an increased number of satellites, more deep space missions, and bandwidth scarcity. Simultaneously, RISC-V based processors will become more prominent in the same industry, which presents the opportunity for new RISC-V based AI accelerators. However, electronic components are vulnerable to radiation, which can induce soft errors. Therefore, new space hardware requires mitigation techniques to reduce vulnerability.

It is crucial to verify the effectiveness of these techniques, which can be accomplished with the help of FI. Emulation-based FI offers a cost-effective and direct insight into hardening strategies with minimal campaign time, while ensuring translatable results to physical implementations. The presented FI tool eliminates time-area tradeoffs contained in traditional emulation-based FI and minimizes customization efforts. It consists of essential elements, including an automatic fault list generator, injector, and results analyser. The tool enables precise fault injection at specific instructions and specific hardware locations.

The effectiveness of the FI tool is demonstrated with a series of experiments. An AI accelerator named SPARROW, implemented in a RISC-V processor, served as a target in these experiments. During the experiments, some benchmark programs are executed by utilizing SPARROW to assess its behaviour under FI. This way, insights are gained regarding SPARROW's architectural vulnerability factor (AVF). There is minimal deviation observed between calculated and experimentally determined AVF. Additionally, the influence of the executed benchmark program on the system's vulnerability is highlighted, emphasizing the importance of considering software characteristics in AVF evaluations.

It is found that while the located FI tool offers precise fault injection capabilities, it may present an optimistic view of vulnerability due to program influence. Further research is needed to distinguish vulnerabilities stemming from either system architecture or software, directing hardening efforts effectively. Exploring the tool's capabilities for other fault models and memory cells, alongside potential optimizations, is crucial. Verification through irradiation experiments is essential to justify its abstraction from physical sources. Overall, the thesis contributes to gaining insight into hardware vulnerability through FI.

Contents

Glossary and Acronyms

ACE	architecturally correct execution
AI	artificial intelligence
ALU	arithmetic logic unit
API	application programming interface
ASIC	application-specific integrated circuit
AVF	architectural vulnerability factor
CFU	custom functions unit available in NEORV32
CISC	complex instruction set computer
CPU	central processing unit
CRC	cyclic redundancy code
CSR	control and status register
DUE	detected unrecoverable error
DUT	device under test
ECC	error-correction code
FI	fault injection
FIT	failure in time, inversely related to MTBF
FPGA	field-programmable gate array
FREtZ	FPGA Reliability Evaluation through JTAG
IC	integrated circuit
ISA	instruction set architecture
JTAG	named after the Joint Test Action Group (see Section 2.6)
LLVM	a set of compiler and tool chain technologies
LUT	look up table
MAC	multiply–accumulate operation
MBU	multipel-bit upset, the upset of two or more bits in the same word

MCU	multiple-cell upset, upset in two or more memory cells or latches
ML	machine learning
MTBF	mean time between failures
NEORV32	32-bit RISC-V soft-core CPU and microcontroller-like SoC (see Section 2.2)
PL	programmable logic
PS	processing system
PVF	program vulnerability vector
RHA	radiation hardness assurance
RISC-V	open standard instruction set architecture five (see Section 2.1)
RISC	reduced instruction set computer
SBU	single-bit upset, a bit-flip in a memory cell or latch as a result of a particle strike
SDC	silent data corruption
SEE	single-event effects
SEFI	single-event functional interrupt, event causes functionality loss due to disruption of a control register, clock signal, reset signal, etc
SEL	single-event latchup, a particle strike requires a power reset or causes permanent damage to the device
SER	soft error rate
SET	singel-event transient, a voltage glitch caused by a particle strike is captured by a storage element
SEU	single-event upset
SIMD	single instruction, multiple data (see Section 2.3)
SoC	system on a chip
SPARROW	low-cost SIMD accelerator for AI operations (see Section 2.4)
SWAR	SIMD within a register
TCP	transmission control protocol
Tcl	tool command language
TMR	triple modular redundancy
UART	universal asynchronous receiver/transmitter

Chapter 1

Introduction

This thesis presents a new fault injection environment for hardness assurance. Hardness assurance is a requirement for new space hardware. This chapter introduced the need for new space hardware and the concept of radiation hardness assurance. Furthermore, a brief introduction into fault injection will be provided. At last, an outline for the rest of the thesis is given.

1.1 New hardware in space

Higher autonomy in outer space hardware, in the form of artificial intelligence (AI), is a big change foreseen in the coming century [1]–[3]. This will be driven by multiple developments in the space industry. First, with the introduction of more complex satellites and larger constellations, ground operations will face challenges in maintaining spacecraft control and processing telemetries. On-board detection of anomalies will be a requirement for future satellites, such that failures can be predicted, and ground operations can be warned before the error can propagate through the satellite. Furthermore, challenges are expected for satellite telecommunication [1], [4], with an increasing amount of satellites in orbit, bandwidth becomes scarce. And, with deep space missions, the distance between the transmitter and the receiver becomes extremely large. This is not only challenging because of the relative visibility geometry and the additional attenuation of the signal power, but also because long delays prevent real-time operations. Requirements for satellite and deep-space communication can be lowered with increased autonomy. However, this autonomy can only be achieved with the introduction of new and more complex hardware in the space industry.

Another prospect is the introduction of the RISC-V ISA in the space industry [5]. Due to the open and modular design of RISC-V, the architecture can be extended to best suit the new space hardware needs, ranging from low-power microcontrollers to high-performance CPUs, and dependable processors capable of managing numerous tasks simultaneously. This includes the ability to extend processors with AI accelerators, of which already multiple exist [6]. With these opportunities to utilize RISC-V hardware-based accelerators for space applications, questions are raised about how well this new hardware will behave in the harshness of the space environment.

1.2 Space environment and hardware resilience

New hardware, as introduced in the previous section, can not reliably be used in outer space as is. Electronic components are susceptible to radiation from both space and Earth's atmosphere, which can induce failures in unprotected systems [7], [8]. As further explored in Section 2.7, radiation can trigger soft errors, leading to incorrect computations or system failure. Therefore, strategies are employed to mitigate the impact of soft errors on the system. Yet, it is crucial to verify the effectiveness of mitigation techniques and the hardening of systems to suppress radiation-induced faults. It is best to utilize diverse tools for designing and assessing the effectiveness of various radiation-hardening techniques already during the system design stage. Among these, radiation hardness assurance (RHA) stands out as the most trustworthy measure. RHA entails conducting physical tests using radiation sources to determine whether a system can function properly in a radiation-harsh environment. This environment is simulated through the use of external sources such as natural or accelerated particle radiation tests, laser beams, or pin forcing [9].

However, RHA impose significant development costs [10]. Besides, it may be helpful to obtain feedback on selected techniques early in the design cycle for guidance. To evaluate the effectiveness of a chosen radiation protection, developers can resort to other hardness verification techniques, including fault injection (FI) [9], [10]. This includes emulation-based FI, where a hardware design is emulated and injected on a field-programmable gate array (FPGA). There are a couple of advantages to using this technique. Foremost, studying faults at architecture levels gives direct insight into ways to target hardening and selective node hardening approaches, and, the probability relation between a soft error and a software error is maintained [11]. Added to that, studying the actual behaviour of a circuit in an application environment allows for considering real-time interactions [12]. Lastly, no special facility is required for emulation based FI, making them more cost-effective besides making it feasible to validate the circuit early with no restrictions on selecting fault locations [11].

1.3 Exact fault injection

This work presents a new emulation-based fault injection environment. As outlined in Chapter 3, two variations on emulation-based tools exist [13]. In the first, static fault injection, each fault is separately injected into a net list and programmed to a FPGA, or similarly, a bitstream gets modified. The second approach is the replacing of flip-flops in a design with a saboteur circuit, allowing to alter its content. There is a tradeoff between these approaches. The reconfiguration-based approaches suffer from reconfiguration overheads and the latter is fast but has a huge area overhead [14].

This work does not make this trade-off between time and area, as it operates directly on the target FPGA for fault injection. This has an additional advantage as campaigns take considerably less time, with a recorded injection rate of about 700 ms. Moreover, the software running on the host PC is developed in Python. This facilitates users to customize and expand the provided setup according to their requirements effortlessly. Finally, no logical changes to the device under test (DUT) are required, ensuring that the results obtained can be extrapolated to setups beyond the fault injection environment.

1.4 Target hardware

To provide insight into the effectiveness of the proposed tool, an example campaign will be provided. As the target, and in line with the observations made in the earlier sections, SPARROW has been selected. SPARROW is intended for AI acceleration, and has been presented by Bolnet and Kosmidis [15]. It was originally designed for the space-qualified LEON3 processor but has been ported to a RISC-V core for this project. SPARROW makes use of a single instruction, multiple data (SIMD) architecture. In this architecture, the integer pipeline is extended with additional short vector operations, by placing a SIMD unit parallel to the arithmetic logic unit (ALU). As outlined in Section 2.3, SIMD does not add performance cost to the operations of the base processor.

1.5 Thesis outline

The rest of this thesis is organized as follows. Chapter 2 will introduce the basic concepts as used throughout the thesis. Chapter 3 provides an overview of RHA and alternatives including FI, together with a list of existing emulation based FI tools.

Then, Chapter 4 introduces the developed toolset, which gets tested in Chapter 5. Finally, in Chapter 6 a conclusion and discussion is formulated.

Chapter 2
Background

This chapter gives an overview of concepts used throughout the rest of this thesis. The first sections cover the RISC-V instruction set architecture and one of its implementations, followed by additional hardware for CPUs researched in this thesis. Then the JTAG standard is discussed. Finally, the focus is shifted towards hardware reliability and soft errors.

2.1 RISC-V

RISC-V is an open ISA [16]. It is a RISC-style load-store instruction set architecture (ISA) [17]. A reduced instruction set computer (RISC) is designed to minimize and simplify the individual instructions. As opposed to complex instruction set computer (CISC), more instructions are required to fulfil a particular task. However, the speed of each instruction can be optimized because of their simplicity. Development of RISC-V started in 2010 at the UC Berkely [18], but is now being guided by the RISC-V International Association.

A key feature of RISC-V is its open standard, which allows anyone to use, modify and contribute to the ISA besides the freedom to develop their own hardware. This sparked the rise of a great community of researchers, developers, and companies contributing to its development. Numerous compatible processors have been developed in the last few years.

RISC-V exhibits a modular architectural design, comprising alternative foundational components complemented by optional extensions. The ISA base prescribes the structure of instructions, encodings, control flow, register specifications, integer manipulation logic, and auxiliary components. Popular extensions include Integer, Multiplication and Division, Floating-Point and Vector Operations.

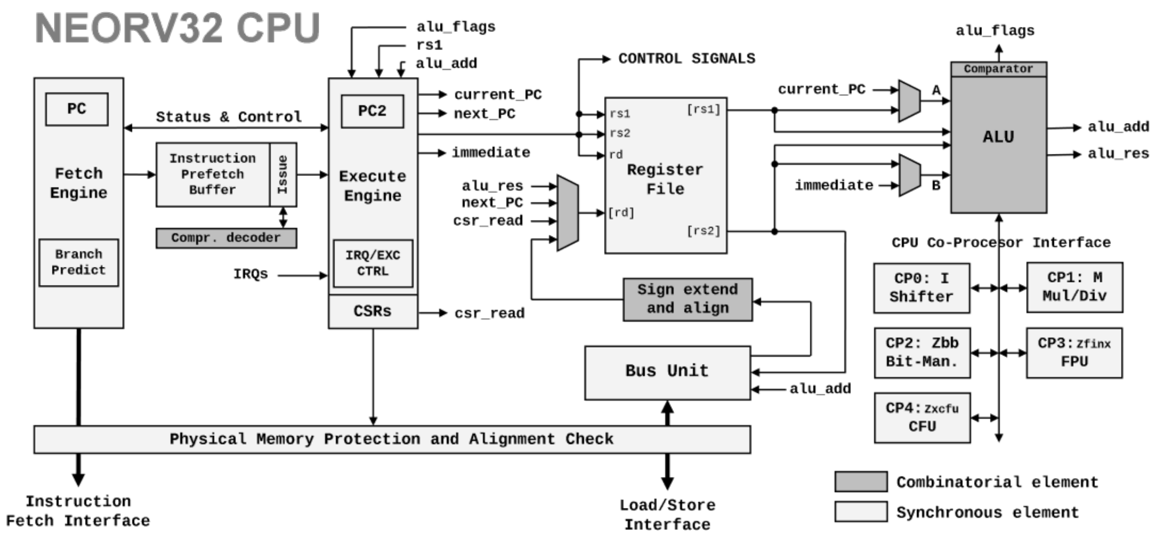

Figure 2.1: NEORV32 CPU overview [19].

2.2 NEORV32

The NEORV32 processor is an open-source RISC-V compatible processor system, which includes a central processing unit (CPU) and an system on a chip (SoC) implementation [19]. The NEORV32 project aims to provide a simple-to-understand, easy-to-use yet powerful and flexible RISC-V implementation. Besides, special care is taken to ensure execution safety using full virtualization. The NEORV32 CPU uses a 2-stage pipelined multi-cycle architecture, an instruction fetch (front-end) and instruction execution (back-end) are de-coupled to operate independently of each other. An overview of the CPU is shown in Figure 2.1 The system's full customizability includes optional common peripherals like embedded memories, timers, serial interfaces, general purpose IO ports and an external bus interface to connect custom IP like memories, network on a-chip sets and other peripherals. By the RISC-V standard, the CPU can be extended with standard and custom ISA extensions.

The latter includes a custom functions unit (CFU), which allows implementing custom RISC-V instructions. Since the CFU has direct access to the core's register file, it allows for the implementation of small logic accelerators. These operations should be able to be completed in a few clock cycles since the pipeline is stalled till completion. The CFU is implemented as a coprocessor to the arithmetic logic unit (ALU), see Figure 2.1.

Böhmer et al. [20] has performed a first-time characterization of the NEORV32 core.

2.3 SIMD

Flynn [21] classifies high-speed computers into four categories: single instruction stream, single data stream; single instruction, multiple data (SIMD) streams; multiple instruction streams, single data stream; multiple instruction streams, multiple data streams. A computing process is in essence performing a sequence of instructions on a set of data, where each instruction performs a combinational manipulation on one or two elements of data. A program is an ordered set of instructions, which is executed by the computer. This execution sequence is the instruction stream. Similarly, the data stream is the sequence of data called for by the instruction stream. Parallelism can be achieved by multiplying one or both of these streams.

SIMD does not add any performance cost to the operation of the base processor. As shown by Lai et al. [22] it can contribute to a performance improvement besides a memory footprint reduction. In their work, SIMD 16-bit multiply–accumulate (MAC) instructions, are used to optimize matrix multiplication and convolution kernels. The latter archives 4.6 times faster throughput while being 4.9 times more energy efficient. Similarly, ReLU activation layers are optimized, providing a 4 times speed-up.

As outlined by Lee [23], in subword parallelism a word is partitioned into smaller units. The same operation can be performed on each subword in parallel, creating a form of SIMD. As pointed out by Lee, subword parallelism is a low-cost solution for SIMD parallelism within a word-oriented processor. The implementation requires no additional register file and very little hardware overhead. For example, the same data path can be used for word and subword operations. In other texts, subword parallelism is referred to as packed parallelism or SIMD within a register (SWAR). Furthermore, the subwords are called lanes.

2.4 SPARROW

SPARROW is a low-cost SIMD accelerator for artificial intelligence (AI) operations, as introduced by Bonet et al. [15]. The *low-cost SIMD accelerator for AI operations* (SPARROW) reuses the integer register file, and is fixed with four lanes of 8-bit integers. The process involves two stages, which are specialized for AI applications. The first stage performs parallel computations and the second reduction operations. SPARROW extends the integer pipeline, without any performance cost in the rest of the operations of the base processor. SPARROW has been written in VHDL, and support for it has been added to a set of compiler and tool chain technologies (LLVM) [24].

The design has been guided by its intended machine learning (ML) applications. Primarily, as Bonet [25] observed, matrix multiplications serve as the foundation for

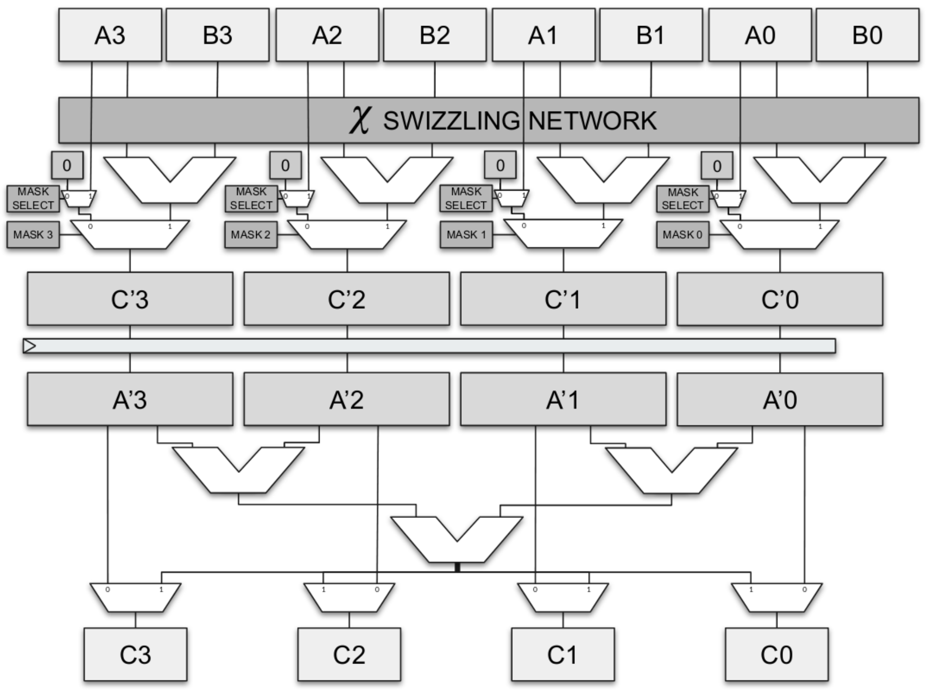

Figure 2.2: Outline of the SPARROW module. Data travels from the top to the bottom [15].

the majority of ML operations, whose computation is based on the dot product. Furthermore, it has been pointed out that the precision of arithmetic operations involved in ML inference operations has been reduced to 8-bit integers. This latter observation accommodates the reuse of the integer register file, by dividing one 32-bit register into four lanes.

The full two-stage design is shown in Figure 2.2. The first stage executes four operations in parallel, adding swizzling and masking vector modifiers. The resulting four lanes of the initial stage can be forwarded to the module output without any supplementary modifications. Or, the four can be combined in reduction operations to yield a 32-bit outcome or subsequently transmitted. A saturation option is included in both stages to mitigate possible overflow of 8-bit values.

Data can be signed or unsigned and available first-stage operations include *add, sub, mul, max, min, shift, move b, and, or, xor, nand, nor* and *xnor*. The second stage supports *sum, max, min* and *xor* operands.

Table 2.1 shows the signals, registers, and their interactions contained in SPARROW. Signal names are a clock-latched copy of the signal in the column before. Furthermore, Figure 2.3 shows how a wave illustrates this signal propagation through the different stages. Irrelevant values are greyed out. Also, the setting of the control register is shown upfront.

2.5 Implementation of SPARROW in NEORV32

For this thesis, SPARROW has been implemented in the NEORV32 as a CFU coprocessor. Because no instruction operated in SPARROW requires memory access, and the SPARROW does not generate any exception, the pipeline could be kept as is. Figure 2.4 shows the interconnections between of the NEORV32 to the CFU and SPARROW. Besides the rewiring of the shown signals as outlined below, an additional control and status register (CSR) has been added to NEORV32. This CSR holds the contents of the control register as outlined in Table 2.1, and was put at address $0x800$ as suggested by the RISC-V standard [26]. This CSR can be set using the default RISC-V CSR instructions. Also, this CSR was added to NEORV32's software environment, together with a library for setting different control signals with ease. In Appendix D the applied changes in software and hardware are shown.

As can be observed from Figure 2.4, there is a mismatch between the NEORV32's bus to the CFU and SPARROW. Therefore, some translation needs to be made between them. The signals contained in the *sdi* and *sdo* (SPARROW data in/out) are listed in Table 2.1. Starting with the former, *sdi*, which is a combination of the operands and partially the control signal. From the control signal, both stage oper-

Table 2.1: List of SPARROWs internal register. Subsequential column cells show the copying of register values from one stage to the next.

		In	Registers			Out
Description	Width (bits)	sdi	s1	s2	s3	sdo
Operand 1 data	32	ra	ra			
Operand 2 data	32	rb	rb	rb		
Stage 1 operation	5	op1	op1			
Stage 2 operation	3	op2	op2	op2		
Stage 2 skipping	1	rc_we	en	en		
Control register:	22[a]	ctrl	ctrl	ctrl	ctrl	ctrl
Masking register	4	mk				
Masking enable	1	ms				
Swizzling operand 1	8	sa				
Swizzling operand 2	8	sb				
Output type[b]	(2)	ol				
Output duplication[b]	(4)	od				
Use rhd	1	hp				
Stage 1 operand overwrite	32	bpv	bpv			
bpv overwrite setting	2	bp	bp			
Shift register intermediate result	32			(ra)[c]	rdh	
Stage 1 result	64			ra	—	s1bp[d]
Setting for saturation of result	1		(op1)[c]	sat		
Stage 2 result	32				rc	result
Stage 2 result	32				(rc)[c]	s2bp
Total bits (per cycle)	338[e]		129	123	86	

[a] total of registers listed below excluding unimplemented registers, [b] not implemented in used design, [c] next stage register is (partial) copy of enclosed value, [d] s1bp is a 32 reduced version of ra and only valid during the stage 2 cycle, [e] total of register bits present in the SPARROW architecture (e.g. *rb* is counted twice since being present in both stage 1 and 2)

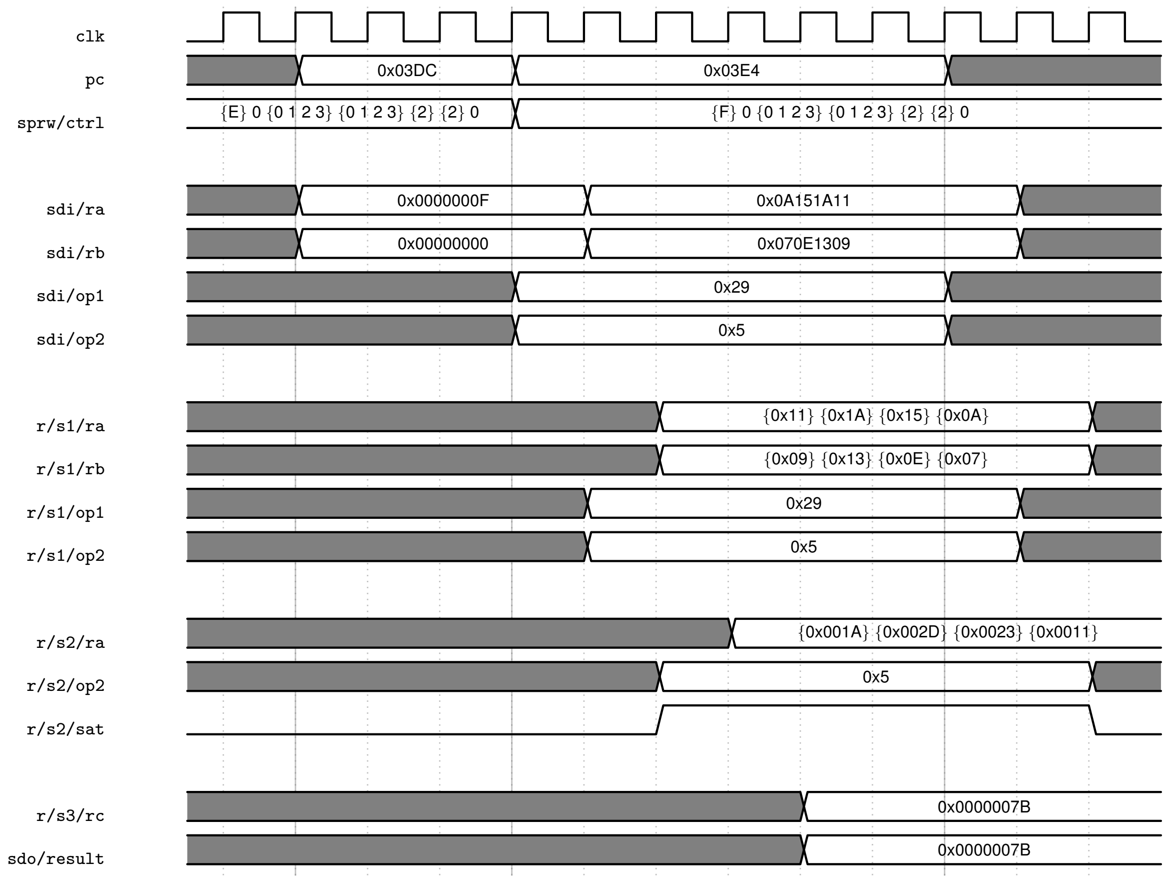

Figure 2.3: A timing diagram showing data propagating through the stages of SPARROW. The instruction issued at program counter `0x03DC` shows the setting of the SPARROW CSR. The next instruction is the the summing of all input lanes. Notice the date travelling through the two stages.

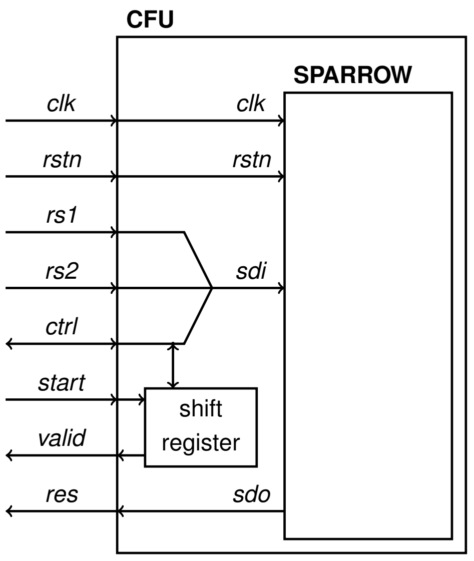

Figure 2.4: CFU and SPARROW interfaces within the NEORV32.

ations are taken out, and the aforementioned CSR is extracted. From *sdo*, only the final result is selected and connected to the result signal.

The start signal indicates the other input signals are valid and the CFU can start a new operation. As soon as the computation is completed, the valid signal is set to indicate completion. This will complete CFU instruction operation and will also write the processing result back to the register file. Since SPARROW is guaranteed to be finished in three cycles, the start signal is fed into a shift register, setting the valid signal after these cycles. No specific exceptions can be raised by the SPARROW CFU.

Instructions are issued using the RISC-V *custom-0* opcode (see [26, Table 24.1]). An R-type instruction is used, where the stage one and two opcodes are encoded in the *func3* and *func7* fields.

2.6 JTAG

JTAG (named after the Joint Test Action Group) is a debug and test access port, defined in IEEE standard 1149. The original standard defines instructions that can be used to perform functional and interconnect tests, as well as built-in self-test procedures. Later expansions of the standard allow for a variety of new applications, including the use of an FPGA programming and debugging interface.

The original IEEE 1149.1 standard [27] describes JTAG as a boundary-scan device, containing a test access port (TAP), instruction register and data registers. On the periphery of an integrated circuit (IC) device, boundary scan registers are added and connected to each external device pin to drive stimuli and capture responses. The registers are concatenated in a scan chain, which can be accessed via the test data input (TDI) and output (TDO) pins on the TAP. A single boundary-scan allows performing a full interconnect test to check its integrity. As this test is extremely thorough, it can provide an extremely high percentage of structural fault coverage [28].

2.7 Soft Errors

Radiation-induced soft errors have become a key challenge in modern computer system designs. Soft errors, also referred to as single-event upsets (SEUs), are data corruption events, but where the device itself is not permanently damaged. This is contrary to cumulative effects, where a fault can become permanent. The reliability of ICs is threatened by radiation-induced soft errors [29], [30]. Soft errors can result in data (detectable and undetectable) corruption on the system level, circuit malfunction or system crash.

2.7.1 Effects of radiation on hardware

Soft errors in microelectronics are caused by highly energetic particles present in the natural space environment striking sensitive regions of a microelectronic circuit [31]. Heavy ions include protons, neutrons, and alpha particles. When a heavy ion strikes, some of its charge is released in a semiconductor device. Either direct ionization by the incident particle itself or ionization via secondary particles created by nuclear reactions between the incident particle and the struck device.

Direct ionization occurs via two principles. A particle passing through a semiconductor material loses energy through freeing electron-hole pairs along its path. When all energy is lost, the particle comes to rest in the semiconductor. Direct ionization is the main mechanism for memory circuit upsets, and is primarily induced by heavy ions with atomic numbers greater than two. Lighter particles like protons and neutrons typically cause insufficient charge deposition. However, these particles can both produce significant upset rates due to indirect mechanisms. As a high-energy proton or neutron enters the semiconductor lattice, it can collide with a target nucleus in several ways. Firstly, due to an elastic collision, a silicon atom can recoil, causing the nucleus to be displaced from its normal position in the crystal lattice. Or a target nucleus emits alpha or gamma particles and decays into a daughter particle, which in turn recoils. Or lastly, the particle induces nuclear fission, splitting a nucleus into two fragments, each of which can recoil. Each of these recoils has the potential to release energy along their trajectories through direct ionization. Because these particles are much heavier than the original proton or neutron, they deposit higher charge densities as they travel.

Ionization can generate currents through charge collection. The reverse-biased p-n junctions are the most vulnerable area for particle strikes. The strong electric field within the depletion region of a reverse-biased junction can effectively collect charge induced by the particle via drift processes, causing a transient current at the junction contact. Strikes close to a depletion region can lead to notable transient currents as diffusion toward the depletion region. Similarly, for direct impacts, carriers generated beyond the depletion region can diffuse back toward the junction. When this, for example, happens at a source-drain junction of a transistor, the electron potential immediately following the strike leads to a significant source-drain conductance mimicking the transistor's open state. In the same line of reasoning, storage elements such as DRAM and SRAM can be particularly vulnerable. This text will, however, not go into the specifics.

A particle strike resulting in a soft error in a logic circuit is not guaranteed. The presence of active pathways from the struck node to latches, the arrival time of the erroneous signal at the latches, and the erroneous pulse time profile at the latch input determine if an erroneous data signal resulting from a strike is captured by a latch or other storage element. Even then, the erroneous information may be blocked by other logic during the following clock cycles.

2.7.2 Effects of radiation

A particle strike may result in unnoticeable effects, a brief interruption of circuit function, a shift in the logic state, or possibly irreversible harm to the device or IC [31], [32]. Single-event interactions are localized effects that can result in a seemingly spontaneous transient within a circuit. This is contrary to total dose radiation that causes a gradual global degradation of device parameters and dose-rate radiation that causes photocurrents in every circuit junction. A single event that affects a node that is storing information can lead to an upset, which is the corruption of the data to an unrecognized, unreadable, or unstable state. If the valid information stored in or travelling through the circuit is altered by this damaged state, it might cause an upset that ultimately results in a circuit error. That is, an upset turns into an error when it latches or when another circuitry misinterprets it as legitimate data. Because of their destructive, uncorrectable origins, permanent faults also get referred to as hard errors. Contrary to hard errors, SEU and multipel-bit upsets (MBUs) are static, but can be corrected. The circuit's stored information is overwritten by these soft errors, but a rewrite or power cycle restores or resets the component to normal operation without causing irreversible harm.

Once a soft error is identified or its probability is calculated (as shown in Section 2.8), insight is gained into a circuit's vulnerability to single events and critical paths that could weaken its single-event tolerance. However, knowing about single events does not provide actual upset metrics that correspond to how a circuit operates in orbit or during beam experiments. Internal single events might not be visible at a circuit's interface pins or output of a subcircuit (see Section 2.8). Nevertheless, if the soft error eventually reaches an output, an externally observable error occurs, which is defined as an error event. One soft error could lead to incorrect information across multiple output ports, over several clock cycles.

SEUs can cause localized information mistakes that are either temporary, persistent, or static. Singel-event transients (SETs) are spurious impulses that can travel through the circuit routes in a single clock cycle. These asynchronous signals may either overpower the circuit's valid synchronous signals or travel to a latch and become static. One important factor influencing the likelihood of mistakes is the timing of the radiation-induced signals in relation to the synchronous signals. The most significant applications for these kinds of faults are analog subsystems and combinational circuits.

Soft errors are a result of single-bit upset (SBU) or MBU, which are types of single-event effects (SEE). Other SEE categories include multiple-cell upset (MCU), SET, single-event functional interrupt (SEFI) and single-event latchup (SEL) [29] (see Glossary).

2.7.3 Mitigation techniques

Soft error mitigation techniques can be applied at different levels in a hardware design, at the system architecture level, circuit level and by technology- or device-level hardening [31]. The latter aims to reduce the charging collection at sensitive nodes by changing the design of semiconductor devices in the silicon. This can be challenging since fundamental changes in the manufacturing process are required. Appling hardening at the circuit-level, overcomes this, by removing the radiation tolerance requirements from the technology level and moving them to the design level. Then any silicon technology, including state-of-the-art low die shrinks, can be used. Generally, hardening is accomplished by the addition of some capacitance to sensitive nodes, which is very effective against ionization events triggered by protons and neutrons [33].

System-level hardening is accomplished by detecting and correcting errors using redundancy. A processor without redundancy cannot detect errors [34]. Redundancy can be created in three dimensions: spatial, temporal and information. With spatial, or physical, redundancy, hardware is replicated and results from each replica are compared. To illustrate, with triple modular redundancy (TMR), the output of the majority of three replicas is chosen by a voter to be the output of the system. Physical redundancy can be implemented at various levels of granularity, ranging from replicating entire processors or cores to finer replication of ALUs or registers. Being a tradeoff between finer diagnosis and the relative overhead of the voter. Additionally, redundancy does not necessitate identical hardware, allowing for design diversity. However, the primary costs of redundancy, including hardware, power, and energy consumption, can be considerable.

Temporal redundancy involves performing an operation twice and comparing the results, doubling the total execution time and halving performance. Unlike physical redundancy, no extra hardware or power cost is incurred, but active energy consumption doubles. Pipelining can mitigate latency, reducing the penalty, but throughput still suffers. This method does not address the energy penalty, maintaining double the active energy usage.

Lastly, information redundancy involves adding extra bits to detect errors in data. Error-correction code (ECC) can detect and sometimes correct errors. Parity, the simplest ECC, adds a bit to ensure even or odd total bits in the codeword, detecting single-bit errors. Parity is favoured for its simplicity, affordability, and decent error-detection capabilities.

2.7.4 Metrics

As outlined, a SBU can, on a functional level, result in an error. Undiscovered errors are referred to as silent data corruption (SDC), and discovered errors are known as detected unrecoverable error (DUE) [30]. Given that a bit is influencing the ultimate outcome of the system, a defective bit that remains unread is not deemed an error. In cases where a defective bit is read and error detection and correction mechanisms are accessible, the bit can be rectified, and the defect is not considered an error. If the defective bit is only detectable, an assessment is required to ascertain whether it impacts the program's outcome. It will be designated as a genuine DUE if it does, and as a false DUE when not. When there is no error detection and the defective bit is not affecting the program's output, it is categorized as error-free since the fault remains undiscovered. The classification of SDC is assigned when the defective bit alters the program's outcome without detection.

Generally, SDC and DUE rates are quantified in terms of FIT. One failure in time (FIT) signifies a single failure occurrence every 10^9 hours, or one billion hours. Since FIT rates accumulate, determining a system's FIT rate involves adding the rates of its components. The collective term often used to describe this is the soft error rate (SER). While not additive, mean time between failures (MTBF) is often a more intuitive measure. MTBF represents the average time until a system fails and is inversely proportional to FIT.

2.8 Architectural Vulnerability Factor

As outlined in previous section, introducing soft error mitigation techniques comes with significant costs in performance, power or size. Furthermore, not all soft errors will affect the final outcome of a program. Therefore, having a metric to give insight into the vulnerability of the micro-architectural structure can be insightful. The probability that a fault in a processor structure will result in a visible error in the final output of a program is called the structure's architectural vulnerability factor (AVF) [35]. For example, a branch predictor's AVF is 0 % and the program counter's is 100 %. Furthermore, the total error rate of micro-architectural components is the product of its raw fault rate and its AVF. By summing up these rates across all system components, it can be evaluated whether the design meets its error rate objectives. This analysis helps in identifying cost-effective strategies for fault protection.

Estimating AVF is done by identifying bits required for architecturally correct execution (ACE). Soft errors in any of the ACE bits will cause a visible error in the final output of a program. The remaining bits are un-ACE bits. The AVF of a single-bit storage cell is the fraction of time it holds ACE bits. Assuming the same raw error rate for each bit, the AVF of a system is the average AVF of its storage cells, or the average fraction of its cells that hold ACE bits at any point in time.

For computing AVFs, identifying ACE and un-ACE bits is key. Makherjee et al. [35] defines the final output as the program's generated value that is sent to an I/O device. Given a specific programme execution, only the correctness of this output matters. Thus, the ACE-ness of a bit does not necessarily correspond to the precise semantics of the architecture. Furthermore, Makherjee et al. provides some pointers for estimating a conservative AVF, which starts by assuming all bits are ACE and then removing any identifiable un-ACE bit. This includes processor states which cannot influence the instruction path, such as data or status bits which are idle or don't contain valid data, predictor structures and dead bits. Bits become dead after their last use, for example, a value in a register is dead after it is stored in memory. Likewise, some processor states do not influence the system output. Such as, *NOP*, performance-enhancing and dynamically dead instructions. The latter refers to instructions whose result is not used. This can be either because the result is not read by any other instruction, or because the result is read-only by another dynamically dead instruction. Lastly, logical masking via bitwise operations can create un-ACE bits.

As discussed, the AVF of a storage cell is the fraction of time an upset in that cell will cause a visible error in the final output of a program. For a hardware structure, the AVF is the average AVF of all bits in the structure, assuming the same raw FIT rate.

$$\begin{aligned}\text{AVF} &= \frac{\bar{r}}{B} \\ &= \frac{\sum_{i \in B} r_i}{BC}\end{aligned} \tag{2.1}$$

The AVF can be calculated with Equation 2.1, where B are the bits in the hardware structure, r a bit's ACE cycles count, $\bar{r}$ the average number of ACE bits per cycle, and C the total execution cycles.

It is important to note, as Biswas et al. [36] points out, that a performance model's ACE analysis is only as good as the model. It may take more information for AVF computation to project performance than it might for a performance model. Thus, an ACE analysis might require much effort, or end up with a pessimistic AVF estimation.

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

With fault injections, as outlined in the next chapter, the AVF of a system can be approximated [37]. With Equation 2.2, the approximated AVF is calculated. An evaluation function $f(X)$, returns 0 when a campaign output is correct, else 1. Furthermore, n faults are evaluated using a fault injection technique.

Chapter 3

Related Work

The concept of radiation hardness assurance and fault injection was introduced in Chapter 1, and will be broadened in this chapter. Furthermore, existing tools and environments for emulation-based fault injection will be listed.

3.1 Radiation hardness assurance

Physical irradiation campaigns (radiation hardness assurance) are considered to resemble closely what one would expect the system to perform in a radiation-harsh environment [9]. However, this procedure is complicated and expensive. Besides, radiation hardness assurance (RHA) always targets a device under test (DUT) in its totality and does not give insight into the vulnerability of sub-structures of the DUT and therefore no feedback on the chosen techniques in the design cycle or evaluation of the effectiveness of a chosen radiation protection. This can be provided by using modelling and computer simulations. RHA possibilities and alternatives are shown in Figure 3.1

As summed up by Huang and Jinang [9], many modelling and simulation tools exist. Such as ones using the Monto Carlo method, closely mimicking RHA, and ones predicting cumulative effects. Besides, modelling and simulation techniques for simulating single event effects exist.

3.2 Simulating single event effects

The simulation and modelling of SEE focuses on a variety of levels in the analysis of interactions between ionizing particles and the matter, from semiconductor up to system level target [9], [31]. These include physical-based device models, multidimensional device simulations, circuit simulations, and mixed devices. Physical device simulations primarily aim to forecast how devices respond to incoming radiation. In contrast, circuit simulations try to model how circuits react to SEE, while

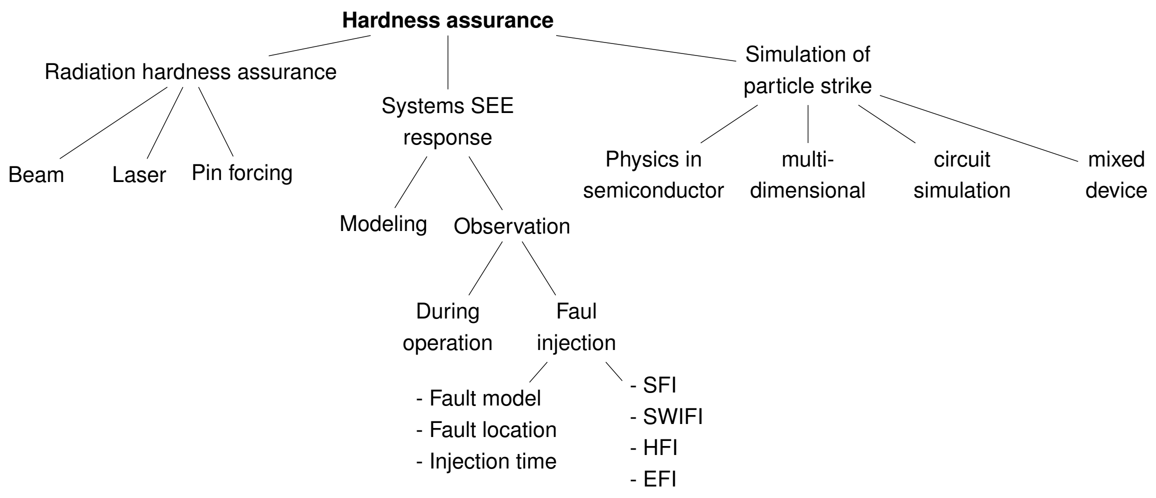

Figure 3.1: Options for performing hardness assurance.

analysis codes are utilized for predicting error rates.

Stepping up in a hierarchical view, effects of SEE can be investigated from a system's perspective, in particular analytical modelling or experimental measuring the effect of SEEs on the system's functioning [38]. The former includes traditional dependability analysis techniques, which try to associate various faults and their causes. The latter, experimental measurement techniques, can be done by recording errors and failures in a large set of systems during operation. However, to speed up this process, faults can be injected into a running system.

3.3 Fault injection

Fault injection (FI) is a method used to quantify the reliability and resilience of a system against soft errors, via assessing the system's ability to detect, locate, and/or mitigate fault occurrences [11], [38], [39]. A FI campaign can be characterized by the used fault model and fault injection locations, and categorized by the technique used for injection.

Fault models describe the type of real-world error being simulated, either transient or persistent faults. Furthermore, the fault model describes the temporal and spatial characteristics of the campaign, which allows the modelling of SBU and MBU. A common model is the bit-flip model, where SEU are simulated. Alternatively, in the set and reset fault models, bits are set to a fixed value regardless of their initial value. Finally, a stuck-at model fixes a bit's value permanently, simulating persistent faults. Furthermore, as faults occur in components that make up systems, a campaign can target specific parts of the system. This includes data registers, address registers, data-fetching units, control registers, and ALUs in a CPU or a memory's controller and stored bits. In the latter particularly, a particle strike in a memory cell can result in both SBUs and MBUs, especially with the ever-decreasing silicon die sizes [8], [38].

FI techniques can be categorized into hardware FI, emulation-based FI, software FI and simulation-based FI [11], [38], [40]. The first, hardware FI, can be performed by either disrupting an IC with faults via input pins or irradiation campaigns. Alternatively, advantage can be taken of built-in hardware debugging facilities [41]. FI at a software level reproduces the errors that would have been produced when faults occur in the hardware, many tools for this exist [42]. Machine code is changed during compile time or run time such that the contents of registers and memory elements are changed, emulating the effect of real-world hardware faults. Contrary, simulation-based FI the DUT hardware is simulated using its hardware description, wherein fault gets injected. Different tools, including Quick EMUlator (QEMU) based, exist [11], [43]. Lastly, with emulation-based FI, faults are injected in high-level models [38]. These models can run on reconfigurable hardware or via an instrumentation-based approach. When using the former, using for example an FPGA, faults are injected by partially reconfiguring the hardware with the DUT. In the latter, the DUT is altered such that errors can be introduced during program execution. Emulation at this architecture level, introducing soft errors at specific times and locations, shows the response of a running application to unwanted changes. Kooli and Di Natale [40] list a variety of fault injection tools and environments.

Aponte-Moreno et al. [43] did a comparison between two simulation-based, an emulation-based FI and irradiation campaigns. Results show that simulation tools give about 10 % optimistic estimates of the reliability when compared to results obtained by emulation. They note that “ISA-level simulation models should be used for a preliminary assessment of the reliability of the system during development” [43, p. 8]. However, fault coverage is very similar in both simulation and emulation and thus can both be used to pre-evaluate different versions of a given hardening technique. Furthermore, it was found that reasonable consistency was preserved between simulation and irradiation experiments with neutrons when evaluating the effectiveness of the mitigation techniques.

3.4 Emulation-based fault injection tools

Various tools and environments for emulation-based fault injection tools exist. Most are based on any of the following techniques: logic modification or reprogramming after netlist modification or bitstream modification [12], [13], [44]. The techniques all have a different campaign time and area tradeoff [44].

The first is static fault injection by reconfiguration. This starts with a list of fault injection locations, and a fault-free netlist is compiled and programmed into an field-programmable gate array (FPGA). Then, selecting one fault after the other from the FI locations list, it gets injected into the fault-free implementation, recompiled and reprogrammed. Once completed, a test sequence can be applied. Campaigns are sped up by partial reconfiguration of FPGAs [12]. This method only allows emulating stuck-at faults, and a full campaign is a lengthy process [13].

A technique that omits recompilation relies on bitstream modification [12], [45]. For this, the bitstream is changed such that the contents of look up tables (LUTs) related to the FI targets are altered, so the output is flipped or fixed. An extension to the bitstream alteration technique, using read-modify-write operations, provides temporal freedom [46]. In general, the proposed techniques follow these steps. First, a FPGA is programmed with a bitstream and execution is started. At an injection cycle, execution is stopped. Then, the current state of all flip-flops is captured, faults get injected, and the result is programmed back to the FPGA. Execution can then be continued.

Secondly, the instrumented circuit technique allows for more dynamic FI, allowing the injections of faults in a single FPGA configuration by replacing flip-flops in the design with saboteur circuitry. In this technique, faults are injected using specific instrumentation hardware located in the flip-flops of the circuit. This allows transient and stuck-at faults to be injected via an external signal. Fault injection is coordinated by a controller. Different, but similar implementations of this setup exist [47]. Time synchronization between the DUT and the controller can be introduced by utilizing shift registers connected to a scan-path chain, named fault-mask, through all instrumentation hardware.

Different tools exist which are based on the aforementioned techniques.

FIDYCO [48] is a hardware/software fault injection environment where both the fault injector and DUT are running on a FPGA. The tool adds extra supporting hardware to the DUT design to allow FI. A separate host interface running in software on a computer instructs the fault injector. Furthermore, it allows implementing the DUT hardware a second time on the FPGA, which output is used for comparison to that of the injected DUT. Temporal FI flexibility is provided by having the DUT send a trigger to the fault injector.

Autonomous emulation [44] replicates all flip-flops of the DUT circuit, to be able to store the correct and faulty state. This is besides additional saboteur flip-flops added for fault injection through instrumented circuit technique. This way, the golden standard can be compared to the injected DUT during execution. Emulation could then be stopped for silent faults, whose effects disappear after a couple of clock cycles. This technique minimizes campaign execution time. This technique can also be extended to monitor memory [49].

FT-UNSHADES [50] acts on the basis of the read-modify-write with bitstream alteration and partial reconfiguration. The tool allows limiting a fault campaign to only a part of the DUT, by utilizing back-annotation information generated by the Xilinx design flow. Besides, injection of MBU is possible. Temporal synchronization is accomplished by counting clock cycles. The tool has been simpler to operate in a new version [51].

FITVS [52] extends the concept of instrumented circuit techniques by inserting the saboteurs in the library modules and modifying the DUT automatically. On a FPGA, besides the DUT, an emulation controller is added, which connects to all the saboteur circuitry. **FITO** [53] follows the same approach as FITVS, but for hardware designs described in Verilog.

Automated FI [54] reuses the concept of autonomous emulation, but optimizes the saboteur circuitry. This included replacing formerly added flip-flops with LUT, and introducing an additional flip-flop for storing the DUT's pre-injection state. This latter improvement reduces campaign time and guarantees coherency between successive experiments. Furthermore, part of the FI environment is implemented in the processing system (PS) of a SoC, besides the DUT in the programmable logic (PL).

FIFA [55] is a platform-independent implementation of the instrumented circuit technique, with the extension of autonomous emulation. The tool is fully parametrizable, which allows designers to establish trade-offs between the complexity and completeness of the analysis. It handles SBUs and MBUs, both transient and permanent FI.

AMUSE [56] adds the implementation of arbitrary circuit delays to the instrumented circuit technique. This is done by modelling a circuit's gate-level characteristics in a register-transfer level design. In the design, gates are replaced by a saboteur, which also includes a shift register to delay signals similar to delays in an application-specific integrated circuit (ASIC) implementation. This way, SET effects can be studied with circuit emulation on a register transfer level, speeding up SET assessment campaigns.

SCFIT [57] is an instrumentation-based fault injection technique which completely relies on commercial tools for the placement of saboteurs. It combines two techniques for FI, the In-System Memory Content Editor available in Altera FPGAs for observing and altering memory elements, through JTAG from Quartus. And, for targeting individual registers, a variant of the instrumented circuit technique with a scan chain is used. This scan chain is controlled from Quartus as well, utilizing In-System Sources and Probes available on Aletra FPGAs. A campaign is executed with the help of tool command language (Tcl) scripts. The tool was extended to be able to inject multiport memories as well [58].

DFI [59] applies the FI technique of using a saboteur to both single registers and full register files. In the latter, a multiplexer is added to the write input for intercepting a write operation or introducing one to inject a fault in a memory slot. Only a little hardware is added, thus having a low area overhead. **NETFI** from the same authors [60] expands on this concept, by changing the built-in library of Xilinx.

Fault-aware LUT mapper [14] alters the contents of LUTs to inject stuck at faults which get activated using Xilinx ICAP interface.

EFIC-ME [61] is a fault injection environment which parses a netlist and inserts saboteur circuitry in the DUT. It allows for stuck-at-fault and bit-flips. Faults can be injected at specific clock cycles. However, only simple DUTs are presented.

Chapter 4

Fault Injection Tool

As briefly introduced in Chapter 1, this thesis presents a new FI environment. This chapter will outline the tool in detail.

4.1 A fault injection system

An FI environment is generally composed of the same elements [47], [55], [62], see Figure 4.1. A fault list or campaign generator defines a fault model based on user parameters. This list is provided to a fault injector and stimuli generator. The latter controls the execution of a DUT, while the former injects a specific type of fault at the location and moment according to the fault model. A result analyser checks the execution and output of the DUT, and discriminates between a crash, timeout or SDC. For this, a golden standard is required, which can be either a clean copy of the DUT being instructed by the stimuli generator as well, or a recorded execution run. Lastly, a results database saves for all campaigns the corresponding result, for later or direct analysis.

The following sections will discuss these elements as used in the proposed tool in more detail. A hardware overview is presented in Figure 4.1. The campaign generator, results analyser and creation of the results database is done on the host PC, all implemented in the Python programming language. This implementation can be found in Appendix A. The fault injector is implemented on an external FPGA board, and the DUT and stimuli generator on the target FPGA.

4.2 Device Under Test

The device under test (DUT) is programmed on the target FPGA. This DUT can be any hardware. However, the discussed tool setup is aimed to inject faults into CPUs. Contrary to tools presented in the previous chapter, no drastic changes are made to the DUT hardware, which guarantees operation is representative to that outside the

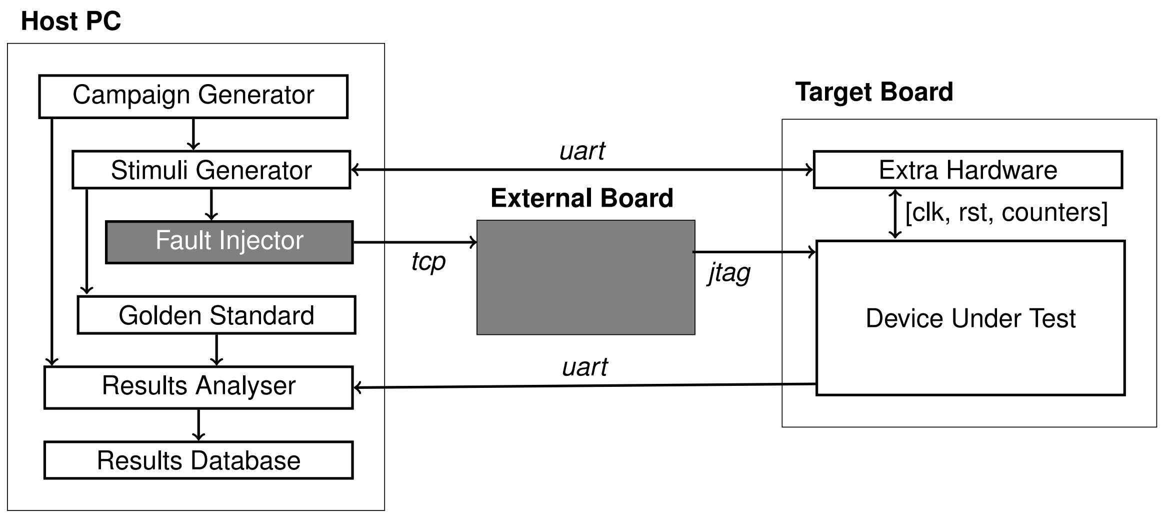

Figure 4.1: Tools components, hardware and their relations, the grey parts are existing components from the FREtZ environment.

FI environment. Furthermore, the target board needs to be programmed only once, which reduces campaign run-time compared to reprogramming-based FI tools.

Nonetheless, one change to the DUT needs to be made. To allow for temporal synchronisation between the stimuli generator and DUT, various machine counters from the CPU need to be wired to the outside. Since this does not add any new functionality, this should be trivial. Any CPU as DUT should do, but for extracting the counters the source needs to be open. Having to make these changes might be considered a disadvantage. Though, temporal synchronisation is required for exact FI, and the alternative is to have the DUT CPU manage this synchronisation by setting a pin, for example. However, this will inevitably disrupt the state of the CPU, deviating from its execution. The stimuli generator as described in the next section is adaptable to allow for such an implementation as well.

4.3 Stimuli generator

To allow fault injection at specific instructions, execution of the DUT needs to be paused. This is accomplished by interrupting the clock signal going to the target. Therefore, to be in close connection with each other, parts of the stimuli generator are implemented on the same FPGA as the DUT. This extra hardware is shown in Figure 4.2. The stimulation generator sets a handful of registers via universal asynchronous receiver/transmitter (UART) on the target FPGA. From the DUT processor, the program counter, cycle counter, and instructions-retired counter are wired to the hardware wrapper, which is compared against the said register. When a value is matched, the clock is disabled. Which counter to compare against can be set via

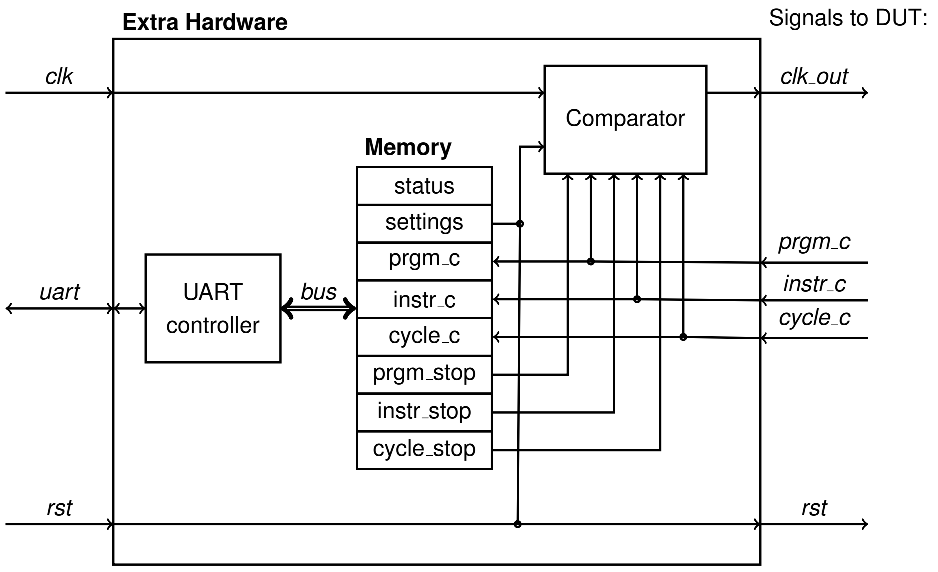

Figure 4.2: Simplified overview of additional hardware included on the target FPGA interacting with the DUT.

Table 4.1: Hardware utilization of extra introduced hardware for FI tool, as obtained from Vivado after implementation.

	Utilization
LUT	208
FF	88
IO	16
BUFG	2

an additional register. Multiple counters can be compared simultaneously, killing the clock whenever one matches first. Furthermore, via the same UART connection, the current value of each counter and board status, including the clock enable, can be retrieved. Using a combination of the counter allows targeting specific clock cycles after an instruction fetch. Lastly, the DUT can be reset via the UART interface. The introduced area overhead as a result of this added hardware is shown in Table 4.1.

4.4 Campaign generator

The presented tool provides an easily extendable and user-friendly campaign generator, implemented using the Python programming language. For this, a logic location file associated with an DUT implementation needs to be provided. The logic location file is an ASCII file that contains information on each of the nodes in the design that can be captured for reading back. It can be exported from Xilinx Vivado using its BitGen utility or by executing a *write_bitstream* Tcl command. The file contains the absolute bit position in the read back stream, frame address, frame offset, logic resource used, and name of the component in the design.

The campaign generator parses this file and creates a data set of all injectable memory elements and latches. To the latter, the associated nets and their hierarchy from the DUT design are associated as well. The user can provide a net or hierarchical component to which the campaign generator can filter and include in the final fault model. For filtering memory locations, no specific implementation is provided, and also not included in experiments. However, the campaign generator can be extended with ease. Allowing researchers to define injection locations programmatically is a big advantage of this tool because it gives great adaptability to different DUTs.

The campaign generator also allows for fixing temporal injection points in the fault model. Because of the versatile stimuli generator, an endless combination of options for this is available. For example, after examining the compiled assembly of a benchmark program, the user can provide a list of target instructions. With the help of the stimuli generator, the campaign generator records all instances when these instructions are executed, utilizing the DUT's instruction counter. These specific instructions can then be included in the fault model. Alternatively, a list of clock cycles to which injections need to be performed can be provided. Again, injection points can be defined programmatically, providing users freedom.

4.5 Fault injector

For injecting faults into the DUT, the FREtZ framework has been utilized. The FPGA Reliability Evaluation through JTAG (FREtZ) framework is developed by Sari et al. [63]. FREtZ provides access to the FPGA configuration memory and circuit logic via the JTAG protocol, intending to provide FPGA design engineers with a tool to improve fault detection isolation and repair strategies. As the authors point out, JTAG is a good option for FPGA configuration tools due to its universality, low overhead and small radiation-sensitive cross-section. Furthermore, besides injecting faults, JTAG can be used for memory scrubbing and debugging purposes. The FREtZ framework provides a Python application programming interface (API) to a Tcl interface handler communicating with a JTAG communication engine to read and write configuration memory and configuration registers. This JTAG communication engine is running on the external FPGA board, communicating to the host PC via a transmission control protocol (TCP) connection.

Injecting faults such as described above, omits the need to either resynthesize and reprogram the DUT for every campaign or having to introduce new hardware in the DUT. These two alternatives, as thoroughly described in the previous chapter, are a time-area tradeoff. The presented tool introduces only a little extra hardware. Campaign durations will be examined in the next chapter.

4.6 Results analyser and database

The job of the results analyser is to categorize the outcome of each campaign as correct, silent data corruption (SDC) or a timeout. The output of the DUT is captured by the results analyser. Output recording is currently limited to UART output. The output is compared against a golden standard, which is a saved output after a campaign run without any fault injected.

Besides comparing the DUT output to the golden standard and identifying SDC, the analyser also has a timer. The execution time of each campaign run is recorded, excluding the time required for injecting a fault. Furthermore, a limit is defined as a user parameter. When execution takes longer than this limit, the campaign is marked as a timeout whereafter the next can be run.

During the execution of the fault model, the number of campaigns and the amount of SDCs and timeouts are recorded and presented to the user in real-time. Besides, results, as judged by the analyser, are stored in the results database, combined with the campaigns from the fault model as generated by the campaign generator. Each entry includes the injection location, injection moment, campaign run time, recorded output and classification. Results are gathered and saved in batches to reduce memory overhead. Afterwards, with the help of a simple utility, all results can be combined into one complete dataset for further analysis.

4.7 User parameters

As mentioned above, the creation of a fault model is guided by a couple of user parameters. Furthermore, the results analyser requires a couple of parameters to be set. All parameters, as taken from the program running on the host PC, are shown in Listing 4.1.

4.8 Tool novelties

As already mentioned throughout this chapter, the presented tool has some benefits over those discussed in the previous chapter. This is added to the advantages regarding emulation-based FI as mentioned in the same chapter. To reiterate, this

```
ADDR_LIST = [0x2bc, 0x30c, 0x334, 0x350,
             0x4c0, 0x4c8, 0x538, 0x540]
CLK_OFFSETS = [2, 3, 4]
INSTRUCTION_END = 0x7fc
LATCH_NAME = "/sprw/"
TIMEOUT_TIME = .8e9 #ns (1e9 ns = 1 second)
GOLDEN_STANDARD = [b'#0$']
LOGIC_FILENAME = "fault_injection.ll"
```

Listing 4.1: User parameters, as part of the application shown in Appendix A.

tool omits time-area tradeoff between logic modification and reprogramming based FI tools, because it works with fault injection on the target FPGA. Furthermore, the software implemented on the host PC is written in Python, a language already established in academia, and with a wide variety of libraries available [64]. This allows users to modify and extend the presented setup to their own needs with ease. Lastly, no logical hardware needs to be added to the DUT, which ensures results are translatable to a setup outside the FI environment.

4.9 Hardware

For the setup, two Digilent Zedboards, containing a Zynq-7000 SoC are used. One as the target FPGA running the DUT and additional UART and comparator hardware, and one as the external FPGA for the fault injector. Two Digilent PmodUSBUART modules are utilized for connecting the target to the host PC. The external board is connected to the target via JTAG and to the host PC via an Ethernet connection. The host PC runs Windows 11, and creates a local network via network sharing. The software runs with Python 3.10.

Chapter 5
Experiments and Results

The tool presented in the previous chapter is evaluated in experiments, which are presented and analysed in this chapter. Two experiments are presented. Firstly, the AVF of the SPARROW unit is calculated and thereafter verified with an injection campaign. Secondly, registers in the same hardware are identified with high and low ACE.

5.1 Experimental setup

For all experiments presented in this chapter, the same setup is used, revolving around the environment as presented in the previous chapter. However, as discussed, the FI environment can handle other types of DUT as well. The hardware used is as described in Section 4.9.

5.1.1 Target CPU and DUT

In the experiments, SPARROW is used as DUT, implemented in the NEORV32 CPU (see Section 2.4). From this CPU, the machine counters are wired to the outside for temporal synchronization. Besides, the NEORV32 has a flexible memory configuration, providing the processor with several boot scenarios. This allows the instruction memory to be pre-initialized with a benchmark program and have the NEORV32 directly boot this on startup.

5.1.2 Benchmark

When choosing a benchmark, it is important to make heavy use of the specific DUT. Suggested algorithms include a bubble sort and matrix multiplication [60]. Furthermore, when one wants to have temporal synchronization using the clock counter, the program's execution must be deterministic. This also requires that program inputs are always the same.

```
     210: 37 25 15 0a    lui a0, 41298
     214: 13 05 15 a1    addi  a0, a0, -1519
     218: b7 15 0e 07    lui a1, 28897
     21c: 93 85 95 30    addi  a1, a1, 777
;    asm volatile ("add_usum %0, %1, %2":"=r"(c):"r"(a), "r"(b));
     220: 0b 54 b5 04    add_usum  s0, a0, a1
```

Listing 5.1: Machine code for summing eight values using SPARROW.

In the experiments described later in this chapter, two different programs are run on the DUT. In the first, the values of the eight SIMD registers from SPARROW are summed, see Equation 5.1. The values in register A are $10, 21, 26, 17$ (0x0A151A11) and for B $7, 14, 19, 9$ (0x070E1309), yielding a sum of 123. The assembly for calculating this sum is shown in Listing 5.1, and the second half of Figure 2.3 shows SPARROW's internal signals for this calculation.

$$c = (a_0 + b_0) + (a_1 + b_1) + (a_2 + b_2) + (a_3 + b_3) \tag{5.1}$$

$$\begin{aligned} C &= AB \\ D &= C * \omega \end{aligned} \qquad \text{with } \omega = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{5.2}$$

A second program aims to utilize SPARROW with its intended ML capabilities. Therefore, it contains a matrix operation and convolutional kernel operation, see Equation 5.2. The function implementations are shown in Appendix B. The matrix multiplication is visualized in Figure 5.1 and its implementation follows the colloquial shown in Equation 5.3. Within one instruction, the dot product of two 4D vectors gets calculated. This is repeated a couple of times (the matrix A column size, or matrix B row size, divided by four), where the results of these dot products are summed to find one element of the result matrix. When the column or row size is not dividable by four, SPARROW's masking capabilities get utilized. When this is done, the last four values of a row get loaded into the A and B registers, to ensure memory safety. A final remark, in the presented implantation, both matrices are indexed by rows, effectively doing a matrix product with one transposed matrix, i.e. $C = A^T B$.

$$\begin{aligned} C &= AB \\ c_{ij} &= \sum_{k=1}^{n} a_{ik} b_{kj} \\ &= \sum_{k \in \{0,4,8,\ldots\}} \sum_{o=0}^{4} a_{(i+o)k} b_{(k+o)j} \end{aligned} \tag{5.3}$$

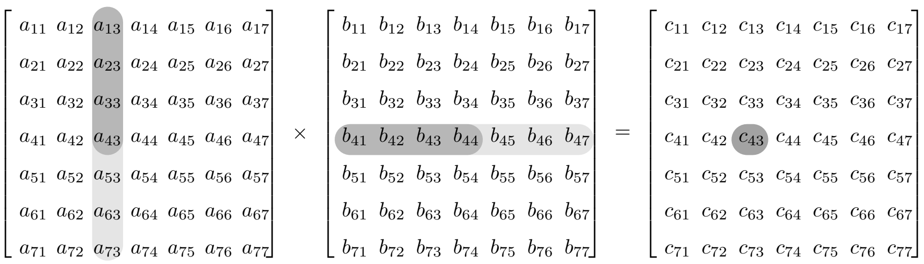

Figure 5.1: Matrix multiplication as implemented in SPARROW.

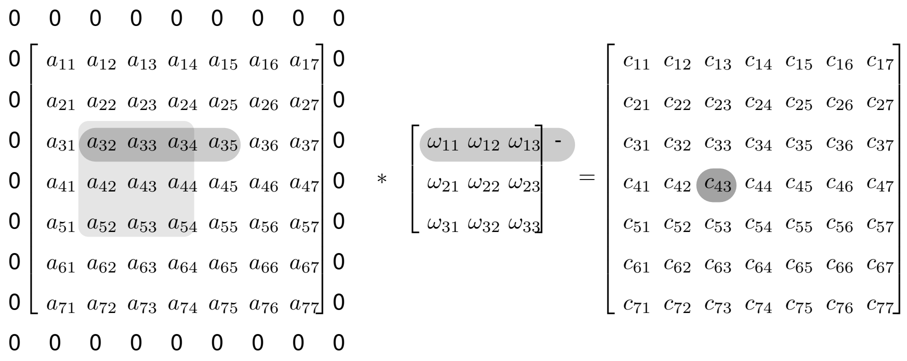

Figure 5.2: A 3x3 kernel convolution calculation as implemented in SPARROW.

The SPARROW implementation for doing a convolution is illustrated in Figure 5.2, with the underlying colloquial shown in Equation 5.4. Before calculating the convolution, the input matrix is padded with zeros. Then, each convolution is calculated in three steps, one row of the kernel separately by doing 3D dot products. Again, SPARROW's masking capabilities are utilized, while taking care of memory safety.

$$
\begin{aligned}
C &= A * \omega \\
c_{ij} &= \sum_{u=-k}^{k} \sum_{v=-k}^{k} a_{(i+u)(j+v)} \omega_{uv}
\end{aligned}
\tag{5.4}
$$

Matrices A and B, as presented in Equation 5.2, are initiated with pseudorandom values between one and five. As kernel, the one shown in Equation 5.2 is used. Small values are used to prevent the whole final matrix from being fully saturated. From the values of matrix D (see Equation 5.2) a 32-bit cyclic redundancy code (CRC) is calculated, with polynomial `0xF8C9140A` as selected using the work from Koopman [65]. This CRC value is printed to a UART output using a simple printing function. To limit computational overhead, the use of printf functionality is avoided. The implementation of this printing function can be found in Appendix B.

5.2 Experiment results

This section will firstly discuss the theoretical AVF of SPARROW, and will then verify this with an experiment. A last experiment will look into the vulnerability of SPARROW when executing the benchmark of the previous section.

5.2.1 Theoretical AVF

As described in Section 2.8, the AVF of a hardware structure can be calculated using Equation 2.1. Table 2.1 provides a list with all registers in SPARROW and its sizes. From this, the number of registers per cycle can be counted, from which the theoretical AVF can be calculated as shown in Equation 5.5. However, only the operands, operations, and saturation signals are targeted. These account for 109 bits in total, see Equation 5.6. For the expected AVF, this difference does not matter. Because of the multi-stage design, with no registers shared between these stages, SPARROWs partial AVF will always be $1/3$.

$$\begin{aligned}\text{AVF}_{\text{sparrow}} &= \frac{\sum_{i \in B} r_i}{BC} \\ &= \frac{129 + 123 + 86}{338 \cdot 3} = \frac{1}{3}\end{aligned} \tag{5.5}$$

$$\text{AVF'}_{\text{sparrow}} = \frac{109}{109 \cdot 3} = \frac{1}{3} \tag{5.6}$$

5.2.2 AVF from fault injection

A fault injection campaign has been performed on SPARROW as DUT while running a program performing the calculation from Equation 5.1. Injections were done at clock offsets 2, 3 and 4 from the start of instruction `0x220` (see Listing 5.1). Latches to inject were automatically selected using the method outlined in Chapter 4.

This campaign resulted in 516 injections. However, this also included the injections on the input and output signals of SPARROW. Filtering these out leaves 324 injections, of which 105 suffered SDC. This yields, using Equation 2.2, an AVF of $\frac{105}{321} = 0.327$, about 1.9 % off from $1/3$. Analysing the results further shows that the last bit of the second stage operation is not vulnerable. A bit flip in this bit changes this stage operation from an unsigned sum to a signed sum. With the small values chosen as inputs, this does not make a difference for the output. Hence, the found AVF differs slightly from the calculated AVF.

5.2.3 Experiment statistics

The experiment, as described above, was extended to inject with clock offsets $0-7$. It was then run twice, such that outputs could be compared. In these two campaigns, in total 2752 injections were performed. Between the two runs, the results of 5 injections did not match, i.e. results were in one campaign labelled as correct and in the second one as SDC.

Running the campaign with these 2752 took in total 33 minutes, yielding 720 ms per injection. Of this, about 35 ms is spent on the execution of the benchmark on the DUT CPU. This big overhead is partially because some experiments needed to be rerun sometimes. Results communicated from the DUT to the host PC via UART sometimes suffer errors, where some characters are not being received, (e.q. `#13$` instead of `#123$`). In these cases, the received output of the benchmark program is classified as incorrect. To prevent false negatives, a campaign is redone up to four times when a result is received incorrectly. This happens for about 16 % of the campaigns.

5.2.4 AVF per register

A second FI campaign has run with the second benchmark program, as outlined earlier in this chapter, being executed on the DUT. Faults were injected on program counters where SPARROW instructions are executed, with clock offsets 2 - 4. The addresses can be found in Appendix C, and are `0x2BC`, `0x30C`, `0x334` and `0x350` from the convolution part, and `0x4C0`, `0x4C8`, `0x538`, and `0x540` from the matrix product part. The same registers as in the previous experiment were selected for injection. This resulted in 106596 injections, of which 21179 were marked as incorrect and 113 as a timeout. This gives an AVF of $(21179 + 113)/106596 = 0.20$. This is off by about 0.13 from the above calculated AVF, and it is fair to say this is all due to masking effects. Interestingly, however, is the appearance of time-out errors. As outlined in Section 2.4, with the CFU implementation with a shift register, timeouts as a result of an injection in SPARROW can not stall the pipeline or processor in any way. This could be an artefact of the 16 % retries of campaigns, but further research should unveil what causes this effect.

In Figure 5.3, the vulnerabilities of individual registers are shown, and only the four least and most vulnerable registers are displayed. These are all centred around the 20 % AVF as found as average AVF. There does not appear to be a clear pattern within these results. On the other hand, the vulnerability for injections at different instructions, as shown in Figure 5.4, does show clear differences between them. This provides more evidence that the benchmark running on a DUT has a high influence on its AVF.

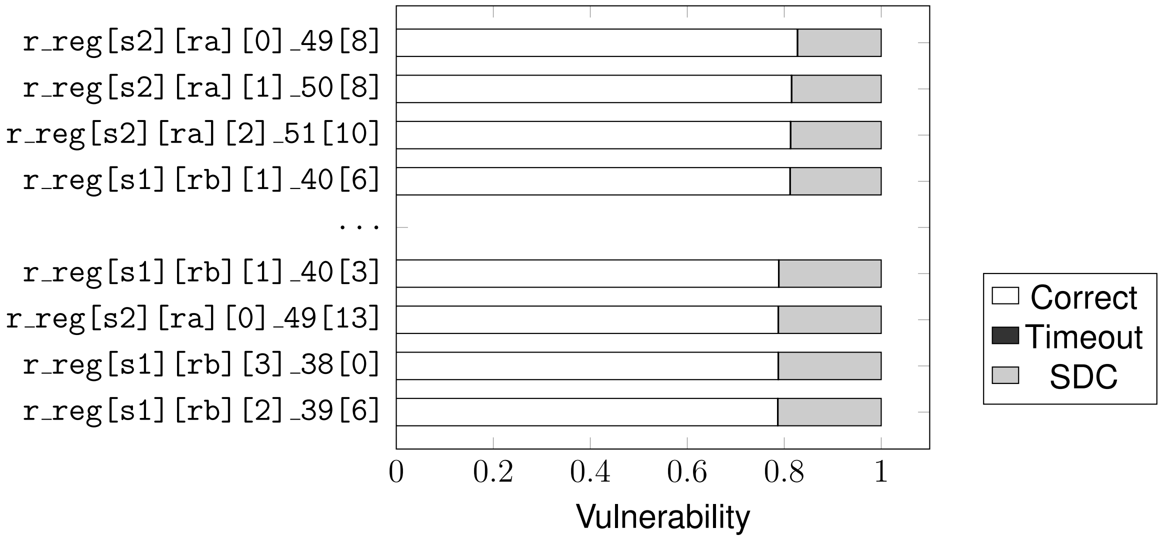

Figure 5.3: AVF of least (top) and most vulnerable registers.

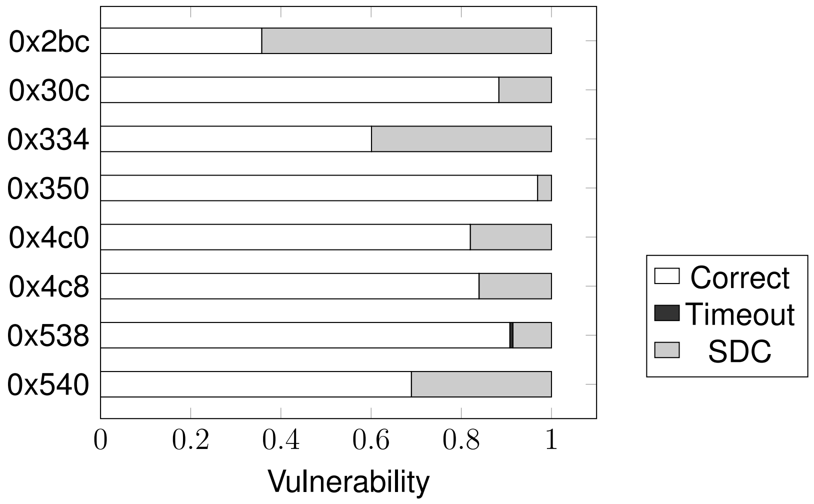

Figure 5.4: Vulnerability when of the SPARROW system when injecting at different instructions.

5.3 Analysis

Both experiments illustrate that FI can be utilized to give insight into the AVF of a system. Moreover, the presented tool provides a way to do this for a subsystem. It also illustrates that a calculated AVF, as presented in Section 2.8, yields a pessimistic vulnerability. This is also pointed out by Mukherjee et al. [35].

However, the last experiment illustrates that the software running on the DUT highly influences the found AVF through FI. Masking is the reason a different AVF is found in this experiment. A metric which can capture this information is program vulnerability vector (PVF). The PVF is a systematic method to efficiently evaluate the error resilience of software under hardware faults, and is a subset of a systems AVF [66], [67].

Chapter 6

Conclusion and Discussion

Artificial Intelligence is expected to be integrated into space hardware development for enhanced autonomy. However, robust resilience measures are necessary because radiation-induced soft errors can cause problems for electronic components. RHA involves rigorous tests to assess system functionality in harsh environments, albeit incurring significant costs. Alternative verification techniques such as FI offer cost-effective ways to evaluate radiation protection. Emulation-based FI provides insights into targeted hardening and real-time interactions without specialized facilities, enabling early circuit validation. This thesis has presented a new located FI tool. It is novel in allowing the specification of fault injection locations in a design. As well as injecting at specified moments, for example, synchronized with the execution of a program on a CPU. Besides creating a FI tool by extending the FREtZ framework, a contribution is made by implementing SPARROW in the NEORV32 CPU.

Experiments have illustrated these capabilities by showing that the tool can be used to find the AVF of a (sub)system, up to individual latches and registers. It also showed the influence a benchmark program has on the found AVF. Therefore, further research should examine the distinction between vulnerabilities stemming from the system architecture or hardware and those arising from software. This investigation will provide deeper insight into where hardening efforts should be directed.

Similarly, further work should examine the tools' ability to inject with other fault models, including stuck-at faults and MBUs, as the thesis did not show this. Besides, FREtZ allows for injecting memory cells besides latches. Likewise, the presented tool is useful for comparing vulnerability before and after applying hardening techniques. This capability is not explored in this thesis, but would be valuable to the field of radiation hardening engineering.

Another limitation of the tool is the required additional hardware on the target FPGA, and the DUT needs to be altered for temporal synchronization. It might be possible for CPUs as targets to utilize (JTAG) on-chip debuggers, when present, for this. This will reduce, besides the complexity of the design, the development time and effort for setting up campaigns. Besides, the required hardware, as listed in

Section 4.9, can pose limitations. That is, being limited to specific FPGA boards and requiring multiple of these. Perhaps the use of all programmable SoCs such as Xilinx's Zynq boards, gives opportunities for simplifying the setup.

Besides the hardware overhead, the presented tool suffers from some inefficiencies. For example, to reduce false negatives as a result of erroneous UART communication, campaigns are rerun multiple times, introducing additional time per injection campaign. Other optimizations include storing the parsed result of logic location files for later reuse; programming the FPGA with a DUT CPU in which the benchmark is already advanced to the first injection point; using the FREtZ framework without its graphical user interface; or extracting the FI capabilities from FREtZ; alternatively, utilize FREtZ's memory scraping functionality to inspect effects of injections before finalizing a benchmark, shortening campaign times; having the DUT CPU's instruction memory in an external memory, such that no new bitstream needs to be generated for new benchmarks.

Lastly, the effectiveness of the presented tool should be verified with irradiation experiments. Since FI abstracts effects in hardware from its physical sources. A comparison between the results of a FI campaign and irradiation test can justify this abstraction.

In conclusion, this thesis has shown that with the help of FI, insight into the vulnerability of hardware designs can be gained. By emulating the hardware, as opposed to simulating, this is achieved with a minimal setup, and with little setup time.

Bibliography

[1] M. Ghiglione and V. Serra, “Opportunities and challenges of AI on satellite processing units”, en, in *Proceedings of the 19th ACM International Conference on Computing Frontiers*, Turin Italy: ACM, May 2022, pp. 221–224, ISBN: 978-1-4503-9338-6. DOI: 10.1145/3528416.3530985.

[2] Z. Pengcheng, W. Wenbo, L. Qiang, and C. Chenghua, “Software-Hardware Cooperative Lightweight Research of Remote Sensing Target Detection Algorithms for Space-Borne Edge Computing”, en, in *Proceedings of the 7th International Symposium of Space Optical Instruments and Applications*, H. P. Urbach and H. Jiang, Eds., vol. 295, Series Title: Springer Proceedings in Physics, Singapore: Springer Nature Singapore, 2023, pp. 668–679, ISBN: 978-981-9940-97-4. DOI: 10.1007/978-981-99-4098-1_55.

[3] G. Trinh, O. Formoso, C. Gregg, *et al.*, “Hardware Autonomy for Space Infrastructure”, in *2023 IEEE Aerospace Conference*, Big Sky, MT, USA: IEEE, Mar. 2023, pp. 1–6, ISBN: 978-1-66549-032-0. DOI: 10.1109/AERO55745.2023.10115601.

[4] P. Tortora, D. Modenini, M. Zannoni, *et al.*, “Ground and Space Hardware for Interplanetary Communication Networks”, en, in *A Roadmap to Future Space Connectivity*, C. Sacchi, F. Granelli, R. Bassoli, F. H. P. Fitzek, and M. Ruggieri, Eds., Series Title: Signals and Communication Technology, Cham: Springer International Publishing, 2023, pp. 107–138, ISBN: 978-3-031-30761-4. DOI: 10.1007/978-3-031-30762-1_5.

[5] S. Di Mascio, A. Menicucci, G. Furano, C. Monteleone, and M. Ottavi, “The Case for RISC-V in Space”, en, in *Applications in Electronics Pervading Industry, Environment and Society*, S. Saponara and A. De Gloria, Eds., vol. 573, Series Title: Lecture Notes in Electrical Engineering, Cham: Springer International Publishing, 2019, pp. 319–325, ISBN: 978-3-030-11972-0. DOI: 10.1007/978-3-030-11973-7_37.

[6] S. Kalapothas, M. Galetakis, G. Flamis, F. Plessas, and P. Kitsos, “A Survey on RISC-V-Based Machine Learning Ecosystem”, en, *Information*, vol. 14, no. 2, p. 64, Jan. 2023, ISSN: 2078-2489. DOI: 10.3390/info14020064.

[7] M. Nicolaidis, Ed., *Soft Errors in Modern Electronic Systems* (Frontiers in Electronic Testing), en. Boston, MA: Springer US, 2011, vol. 41, ISBN: 978-1-4419-6992-7. DOI: 10.1007/978-1-4419-6993-4.

[8] 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, ISSN: 0018-9499, 1558-1578. DOI: 10.1109/TNS.2013.2259503.

[9] 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", en, *Progress in Nuclear Energy*, vol. 114, pp. 105–120, Jul. 2019, ISSN: 01491970. DOI: 10.1016/j.pnucene.2019.02.008.

[10] M. Eslami, B. Ghavami, M. Raji, and A. Mahani, "A survey on fault injection methods of digital integrated circuits", en, *Integration*, vol. 71, pp. 154–163, Mar. 2020, ISSN: 01679260. DOI: 10.1016/j.vlsi.2019.11.006.

[11] Y. B. Bekele, D. B. Limbrick, and J. C. Kelly, "A Survey of QEMU-Based Fault Injection Tools & Techniques for Emulating Physical Faults", *IEEE Access*, vol. 11, pp. 62 662–62 673, 2023, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3287503.

[12] L. Antoni, R. Leveugle, and B. Feher, "Using run-time reconfiguration for fault injection applications", in *IMTC 2001. Proceedings of the 18th IEEE Instrumentation and Measurement Technology Conference. Rediscovering Measurement in the Age of Informatics (Cat. No.01CH 37188)*, vol. 3, Budapest, Hungary: IEEE, 2001, pp. 1773–1777, ISBN: 978-0-7803-6646-6. DOI: 10.1109/IMTC.2001.929505.

[13] Kwang-Ting Cheng, Shi-Yu Huang, and Wei-Jin Dai, "Fault emulation: A new methodology for fault grading", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 10, pp. 1487–1495, Oct. 1999, ISSN: 02780070. DOI: 10.1109/43.790625.

[14] A. Ullah, E. Sanchez, L. Sterpone, L. Cardona, and C. Ferrer, "An FPGA-based dynamically reconfigurable platform for emulation of permanent faults in ASICs", en, *Microelectronics Reliability*, vol. 75, pp. 110–120, Aug. 2017, ISSN: 00262714. DOI: 10.1016/j.microrel.2017.06.032.

[15] M. S. Bonet and L. Kosmidis, "SPARROW: A Low-Cost Hardware/Software Co-designed SIMD Microarchitecture for AI Operations in Space Processors", in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, ISSN: 1558-1101, Mar. 2022, pp. 1139–1142. DOI: 10.23919/DATE54114.2022.9774730.

[16] "About RISC-V – RISC-V International". en-US. (), [Online]. Available: https://riscv.org/about/ (visited on 10/10/2023).

[17] "RISC-V: An Open Standard for SoCs". (Aug. 2014), [Online]. Available: https://www.eetimes.com/risc-v-an-open-standard-for-socs/ (visited on 10/10/2023).

[18] "History – RISC-V International". en-US. (), [Online]. Available: https://riscv.org/about/history/ (visited on 10/10/2023).

[19] S. Nolting and A. t. A. Contributors. "The NEORV32 RISC-V Processor". (Aug. 2023), [Online]. Available: https://github.com/stnolting/neorv32.

[20] K. Böhmer, B. Forlin, C. Cazzaniga, *et al.*, "Neutron Radiation Tests of the NEORV32 RISC-V SoC on Flash-Based FPGAs", in *2023 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Juan-Les-Pins, France: IEEE, Oct. 2023, pp. 1–6. DOI: 10.1109/DFT59622.2023.10313556.

[21] M. Flynn, "Very high-speed computing systems", *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966, ISSN: 0018-9219. DOI: 10.1109/PROC.1966.5273.

[22] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs", 2018, Publisher: arXiv Version Number: 1. DOI: 10.48550/ARXIV.1801.06601.

[23] R. Lee, "Multimedia extensions for general-purpose processors", *1997 IEEE Workshop on Signal Processing Systems. SiPS 97 Design and Implementation formerly VLSI Signal Processing*, 1997. DOI: 10.1109/SIPS.1997.625683.

[24] M. S. Bonet and L. Kosmidis, "Compiler Support for an AI-oriented SIMD Extension of a Space Processor", en, *ACM SIGAda Ada Letters*, vol. 42, no. 1, pp. 95–99, Dec. 2022, ISSN: 1094-3641. DOI: 10.1145/3577949.3577968.

[25] M. S. Bonet, "Hardware-software co-design for low-cost AI processing in space processors", eng, Accepted: 2022-02-02T08:19:46Z, M.S. thesis, Universitat Politècnica de Catalunya, Oct. 2021.

[26] A. Waterman and K. Asanovi, "The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 20191213", RISC-V Foundation, Tech. Rep., Dec. 2019.

[27] "IEEE Standard for Reduced-Pin and Enhanced-Functionality Test Access Port and Boundary-Scan Architecture", IEEE, Tech. Rep. DOI: 10.1109/IEEESTD.2010.5412866.

[28] Be Van Ngo, P. Law, and A. Sparks, “Use of JTAG boundary-scan for testing electronic circuit boards and systems”, in *2008 IEEE AUTOTESTCON*, Salt Lake City, UT, USA: IEEE, Sep. 2008, pp. 17–22, ISBN: 978-1-4244-2225-8. DOI: 10.1109/AUTEST.2008.4662576.

[29] 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, ISBN: 978-1-4419-6993-4. DOI: 10.1007/978-1-4419-6993-4_1.

[30] S. Mukherjee, J. Emer, and S. Reinhardt, “The Soft Error Problem: An Architectural Perspective”, in *11th International Symposium on High-Performance Computer Architecture*, San Francisco, CA, USA: IEEE, 2005, pp. 243–247, ISBN: 978-0-7695-2275-3. DOI: 10.1109/HPCA.2005.37.

[31] P. Dodd and L. Massengill, “Basic mechanisms and modeling of single-event upset in digital microelectronics”, *IEEE Transactions on Nuclear Science*, vol. 50, no. 3, pp. 583–602, Jun. 2003, ISSN: 0018-9499, 1558-1578. DOI: 10.1109/TNS.2003.813129.

[32] E. Petersen, *Single Event Effects in Aerospace*, en, 1st ed. Wiley, Sep. 2011, ISBN: 978-0-470-76749-8. DOI: 10.1002/9781118084328.

[33] F. Faccio, “Radiation effects and hardening by design in cmos technologies”, in *Analog Circuit Design: Robust Design, Sigma Delta Converters, RFID*, Springer, 2011, pp. 69–87.

[34] D. J. Sorin, *Fault Tolerant Computer Architecture* (Synthesis Lectures on Computer Architecture), en. Cham: Springer International Publishing, 2009, ISBN: 978-3-031-00595-4. DOI: 10.1007/978-3-031-01723-0.

[35] 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 *22nd Digital Avionics Systems Conference. Proceedings (Cat. No.03CH37449)*, San Diego, CA, USA: IEEE Comput. Soc, 2003, pp. 29–40, ISBN: 978-0-7695-2043-8. DOI: 10.1109/MICRO.2003.1253181.

[36] A. Biswas, P. Racunas, J. Emer, and S. Mukherjee, “Computing Accurate AVFs using ACE Analysis on Performance Models: A Rebuttal”, *IEEE Computer Architecture Letters*, vol. 7, no. 1, pp. 21–24, Jan. 2008, ISSN: 1556-6056. DOI: 10.1109/L-CA.2007.19.

[37] 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)*, Amsterdam: IEEE, Oct. 2015, pp. 212–219, ISBN: 978-1-4673-8321-9. DOI: 10.1109/CODESISSS.2015.7331384.

[38] S. A. Alazawi and M. N. Al-Salam, "Review of Dependability Assessment of Computing System with Software Fault-Injection Tools", en, *Journal of Southwest Jiaotong University*, vol. 54, no. 4, p. 27, 2019, ISSN: 0258-2724. DOI: 10.35741/issn.0258-2724.54.4.27.

[39] H. Ziade, R. A. Ayoubi, R. Velazco, *et al.*, "A survey on fault injection techniques", *Int. Arab J. Inf. Technol.*, vol. 1, no. 2, pp. 171–186, 2004.

[40] M. Kooli and G. Di Natale, "A survey on simulation-based fault injection tools for complex systems", in *2014 9th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, Santorini, Greece: IEEE, May 2014, pp. 1–6, ISBN: 978-1-4799-4972-4. DOI: 10.1109/DTIS.2014.6850649.

[41] A. Aponte-Moreno, J. Isaza-Gonzalez, A. Serrano-Cases, A. Martinez-Alvarez, S. Cuenca-Asensi, and F. Restrepo-Calle, "An Experimental Comparison of Fault Injection Tools for Microprocessor-based Systems", in *2020 IEEE Latin-American Test Symposium (LATS)*, Maceio, Brazil: IEEE, Mar. 2020, pp. 1–6, ISBN: 978-1-72818-731-0. DOI: 10.1109/LATS49555.2020.9093668.

[42] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing Dependability with Software Fault Injection: A Survey", en, *ACM Computing Surveys*, vol. 48, no. 3, pp. 1–55, Feb. 2016, ISSN: 0360-0300, 1557-7341. DOI: 10.1145/2841425.

[43] A. Aponte-Moreno, J. Isaza-González, A. Serrano-Cases, A. Martínez-Álvarez, S. Cuenca-Asensi, and F. Restrepo-Calle, "Evaluation of fault injection tools for reliability estimation of microprocessor-based embedded systems", en, *Microprocessors and Microsystems*, vol. 96, p. 104 723, Feb. 2023, ISSN: 01419331. DOI: 10.1016/j.micpro.2022.104723.

[44] C. Lopez-Ongil, M. Garcia-Valderas, M. Portela-Garcfa, and L. Entrena-Arrontes, "Autonomous transient fault emulation on FPGAs for accelerating fault grading", in *11th IEEE International On-Line Testing Symposium*, French Riviera, France: IEEE, 2005, pp. 43–48, ISBN: 978-0-7695-2406-1. DOI: 10.1109/IOLTS.2005.18.

[45] T. Slaughter and C. Stroud, "Fault injection emulation for field programmable analog arrays", in *Southwest Symposium on Mixed-Signal Design, 2003.*, Las Vegas, NV, USA: IEEE, 2003, pp. 212–216, ISBN: 978-0-7803-7778-3. DOI: 10.1109/SSMSD.2003.1190429.

[46] L. Antoni, R. Leveugle, and M. Feher, "Using run-time reconfiguration for fault injection in hardware prototypes", in *17th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2002. DFT 2002. Proceedings.*, Vancouver, BC, Canada: IEEE Comput. Soc, 2002, pp. 245–253, ISBN: 978-0-7695-1831-2. DOI: 10.1109/DFTVS.2002.1173521.

[47] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Reorda, and M. Violante, "Exploiting circuit emulation for fast hardness evaluation", *IEEE Transactions on Nuclear Science*, vol. 48, no. 6, pp. 2210–2216, Dec. 2001, ISSN: 0018-9499, 1558-1578. DOI: 10.1109/23.983197.

[48] B. Rahbaran, A. Steininger, and T. Handl, "Built-in Fault Injection in Hardware - The FIDYCO Example", in *Second IEEE International Workshop on Electronic Design, Test and Applications*, Perth, Australia: IEEE, 2004, pp. 327–327, ISBN: 978-0-7695-2081-0. DOI: 10.1109/DELTA.2004.10070.

[49] M. Garcia-Valderas, M. Portela-Garcia, C. Lopez-Ongil, and L. Entrena, "Emulation-based Fault Injection in Circuits with Embedded Memories", in *12th IEEE International On-Line Testing Symposium (IOLTS'06)*, Como, Italy: IEEE, 2006, pp. 183–184, ISBN: 978-0-7695-2620-1. DOI: 10.1109/IOLTS.2006.29.

[50] M. Aguirre, D. Merodio, J. Tombs, *et al.*, "Selective Protection Analysis Using a SEU Emulator: Testing Protocol and Case Study Over the Leon2 Processor", *IEEE Transactions on Nuclear Science*, vol. 54, no. 4, pp. 951–956, Aug. 2007, ISSN: 0018-9499, 1558-1578. DOI: 10.1109/TNS.2007.895550.

[51] J. Mogollon, H. Guzman-Miranda, J. Napoles, J. Barrientos, and M. Aguirre, "FTUNSHADES2: A novel platform for early evaluation of robustness against SEE", in *2011 12th European Conference on Radiation and Its Effects on Components and Systems*, Sevilla, Spain: IEEE, Sep. 2011, pp. 169–174, ISBN: 978-1-4577-0586-1. DOI: 10.1109/RADECS.2011.6131392.

[52] H. Zheng, L. Fan, and S. Yue, "FITVS: A FPGA-Based Emulation Tool For High-Efficiency Hardness Evaluation", in *2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*, Sydney, Australia: IEEE, Dec. 2008, pp. 525–531, ISBN: 978-0-7695-3471-8. DOI: 10.1109/ISPA.2008.46.

[53] M. Shokrolah-Shirazi and S. G. Miremadi, “FPGA-Based Fault Injection into Synthesizable Verilog HDL Models”, in *2008 Second International Conference on Secure System Integration and Reliability Improvement*, Yokohama, Japan: IEEE, Jul. 2008, pp. 143–149. DOI: 10.1109/SSIRI.2008.47.

[54] R. Leveugle and A. Prost-Boucle, “A new automated instrumentation for emulation-based fault injection”, in *2010 First IEEE Latin American Symposium on Circuits and Systems (LASCAS)*, Foz do Iguacu: IEEE, Feb. 2010, pp. 200–203, ISBN: 978-1-5090-2076-8. DOI: 10.1109/LASCAS.2010.7410245.

[55] L. Naviner, J.-F. Naviner, G. Dos Santos, E. Marques, and N. Paiva, “FIFA: A fault-injection–fault-analysis-based tool for reliability assessment at RTL level”, en, *Microelectronics Reliability*, vol. 51, no. 9-11, pp. 1459–1463, Sep. 2011, ISSN: 00262714. DOI: 10.1016/j.microrel.2011.06.017.

[56] L. Entrena, M. Garcia-Valderas, R. Fernandez-Cardenal, A. Lindoso, M. Portela, and C. Lopez-Ongil, “Soft Error Sensitivity Evaluation of Microprocessors by Multilevel Emulation-Based Fault Injection”, *IEEE Transactions on Computers*, vol. 61, no. 3, pp. 313–322, Mar. 2012, ISSN: 0018-9340, 1557-9956, 2326-3814. DOI: 10.1109/TC.2010.262.

[57] A. Mohammadi, M. Ebrahimi, A. Ejlali, and S. G. Miremadi, “SCFIT: A FPGA-based fault injection technique for SEU fault model”, in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Dresden: IEEE, Mar. 2012, pp. 586–589, ISBN: 978-1-4577-2145-8. DOI: 10.1109/DATE.2012.6176538.

[58] M. Ebrahimi, A. Mohammadi, A. Ejlali, and S. G. Miremadi, “A fast, flexible, and easy-to-develop FPGA-based fault injection technique”, en, *Microelectronics Reliability*, vol. 54, no. 5, pp. 1000–1008, May 2014, ISSN: 00262714. DOI: 10.1016/j.microrel.2014.01.002.

[59] W. Mansour and R. Velazco, “SEU Fault-Injection in VHDL-Based Processors: A Case Study”, en, *Journal of Electronic Testing*, vol. 29, no. 1, pp. 87–94, Feb. 2013, ISSN: 0923-8174, 1573-0727. DOI: 10.1007/s10836-013-5351-6.

[60] W. Mansour and R. Velazco, “An Automated SEU Fault-Injection Method and Tool for HDL-Based Designs”, *IEEE Transactions on Nuclear Science*, vol. 60, no. 4, pp. 2728–2733, Aug. 2013, ISSN: 0018-9499, 1558-1578. DOI: 10.1109/TNS.2013.2267097.

[61] Z. U. Abideen and M. Rashid, “EFIC-ME: A Fast Emulation Based Fault Injection Control and Monitoring Enhancement”, *IEEE Access*, vol. 8, pp. 207 705–207 716, 2020, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3038198.

[62] G. Cieslewski, A. D. George, and A. Jacobs, “Acceleration of FPGA Fault Injection Through Multi-Bit Testing”, in *Proceedings of the 2010 International Conference on Engineering of Reconfigurable Systems & Algorithms, ERSA 2010, July 12-15, 2010, Las Vegas Nevada, USA*, T. P. Plaks, D. Andrews, R. F. DeMara, *et al.*, Eds., CSREA Press, 2010, pp. 218–224.

[63] A. Sari, M. Psarakis, and V. Vlagkoulis, “An Open-source Framework for Xilinx FPGA Reliability Evaluation”, in *Open Source Design Automation*, Florence, Mar. 2019.

[64] V. M. Ayer, S. Miguez, and B. H. Toby, “Why scientists should learn to program in Python”, en, *Powder Diffraction*, vol. 29, no. S2, S48–S64, Dec. 2014, ISSN: 0885-7156, 1945-7413. DOI: `10.1017/S0885715614000931`.

[65] P. Koopman and T. Chakravarty, “Cyclic redundancy code (CRC) polynomial selection for embedded networks”, in *International Conference on Dependable Systems and Networks, 2004*, Jun. 2004, pp. 145–154. DOI: `10.1109/DSN.2004.1311885`.

[66] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, “ePVF: An Enhanced Program Vulnerability Factor Methodology for Cross-Layer Resilience Analysis”, in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Toulouse, France: IEEE, Jun. 2016, pp. 168–179, ISBN: 978-1-4673-8891-7. DOI: `10.1109/DSN.2016.24`.

[67] V. Sridharan and D. R. Kaeli, “Using hardware vulnerability factors to enhance AVF analysis”, en, in *Proceedings of the 37th annual international symposium on Computer architecture*, Saint-Malo France: ACM, Jun. 2010, pp. 461–472, ISBN: 978-1-4503-0053-7. DOI: `10.1145/1815961.1816023`.

[68] M. S. Bonet. “SPARROW”. en. (), [Online]. Available: `https://gitlab.bsc.es/msolebon/sparrow` (visited on 03/14/2023).

Appendix A
User Application code listing

UserApplication.py

```python
from __future__ import annotations
from typing import List
import sys, time, pydevd
from PySide2.QtCore import Qt, QObject, QThread, Signal, Slot
from Project.ProjectSettings import ProjectSettings
from Frames.FrameParser import FrameParser
from Frames.Frame import Frame
from Frames.EbdFrame import EbdFrame
from Frames.FrameAddress import FrameAddress
from Utilities.Constants import Constants
from Utilities.Log import Log
from UI.UISignals import UISignals
from Communication.CommandManager import CommandManager
# from Communication.SerialPort import SerialPort
from Data.ExecutionStatus import ExecutionStatus

from Controller.controller import Controller
from Log.logicdata import LogicData
from Log.ficampaign import FiCampaign
import serial
import pandas as pd

class UserApplicationSignals(QObject):
    """This class holds the signals user application
    """
    #: Signal to be called to stop the user application
    ApplicationStopped = Signal(bool)

class UserApplication(QThread):
    """This class can be used as user application placeholder which ↩
        could be able to be loaded and executed dynamically
    """
    def __init__(self, project : ProjectSettings, frameParser : FrameParser, remoteDeviceSwIdReceived : QObject, parent=None):
        """Class constructor

        :param project: The project settings
        :type project: ProjectSettings
        :param frameParser: The :class:FrameParser as created by the caller
        :type frameParser: FrameParser
        :param remoteDeviceSwIdReceived: Callbacl function to be called when a SW version and ID are received
        :type remoteDeviceSwIdReceived: QObject
        :param parent: The parent object, defaults to None
        :type parent: [type], optional
        """
        super().__init__(parent)
        self._execute = False
        self._project = project
        self._frameParser = frameParser
        self._commandManager = None
        self._uiSignals = UISignals()
        self._uiSignals.RemoteDeviceSwIdReceived.connect(remoteDeviceSwIdReceived)

    def Exit(self):
        """Exits the user application
        This method terminates the thread and the closes the :class:CommandManager
        """
        self.StopClicked()
        try:
            if self._commandManager:
                self._commandManager.Close()
        except Exception as e:
            Log.PrintException(f'UserApplication.Exit: {str(e)}')

    @Slot(bool)
    def StopClicked(self, value : bool = True):
        """Stops the thread

        :param value: User provided value (from the parent object), defaults to True
        :type value: bool, optional
        """
        self._execute = False
        if self.isRunning():
            self.terminate()

    def waitAndRecored(self, timeout_time, contr, uart_dut = None):
        timer_start = time.time_ns()
        outp = b''
        while not contr.getReady():
            # Gather ouput text
            if uart_dut != None:
                size = uart_dut.in_waiting
                outp += uart_dut.read(size)
            # Check for timeout
            if (time.time_ns() - timer_start) > timeout_time:
                return (-1, outp)
        timer_end = time.time_ns()
        if uart_dut != None:
                size = uart_dut.in_waiting
                outp += uart_dut.read(size)
        return (timer_end - timer_start, outp)

    def gatherInjectionPoints(self, contr, instructions, INSTRUCTION_END):
        ip = pd.DataFrame(columns=['pc', 'instr'])
        for pc in instructions:
            contr.setCycleStop(0)
            contr.setEnable(glbl = True, cycle = True)
            contr.reset()

            end = False
            while not end:
                contr.setPrgmStop(pc)
                contr.setEnable(glbl = True, prgm = True)
                (time, _ ) = self.waitAndRecored(2e9, contr, None)
                if time == -1:
                    end = True
                    continue
                instr = contr.read(contr.INSTR_COUNTER)
                ip = pd.concat(
                        [pd.DataFrame([[pc, instr]], columns=ip.columns), ip],
                        ignore_index=True)
                contr.setCycleStopRelative(8)
                contr.setEnable(glbl=True, cycle=True)
        return ip


    def run(self):
        global_timer = time.time()
        """This method holds the user implementation executed by ↩
            this thread
        """

        ADDR_LIST = [0x2bc, 0x30c, 0x334, 0x350,
                     0x4c0, 0x4c8, 0x538, 0x540]
        CLK_OFFSETS = [2, 3, 4]
        INSTRUCTION_END = 0x7fc
        LATCH_NAME = "/sprw/"
        TIMEOUT_TIME = .8e9 #ns (1e9 ns = 1 second)
        GOLDEN_STANDARD = [b'#0$']
        LOGIC_FILENAME = "fault_injection.ll"

        UART_CONTROLLER = "COM7"
        UART_DUT = "COM6"
        RETRIES = 5

        # Counters
        c_run = 0
        c_tries = 0
        c_true = 0
        c_fales = 0
        c_timeout = 0

        try:
            self._commandManager = CommandManager(self._project. ↩
                FpgaDevice, self._project.IpAddress, self._project. ↩
                TcpPort)
            self._execute = True

            # Read S/N ans SW version
            Log.Print("Reading ID")
            boardInfo = self._commandManager.ReadId()
            serialNumberHex = '{0:0{1}X}'.format(boardInfo. ↩
                SerialNumber, 8)
            self._uiSignals.RemoteDeviceSwIdReceived.emit(f'{ ↩
                serialNumberHex} : {boardInfo.Version}')

            # Connect via UART to controller and DUT
            Log.Print("Connecting to controller and DUT")
            contr = Controller(UART_CONTROLLER)
            uart_dut = serial.Serial(UART_DUT, 19200, timeout=2)
            # uart_dut = None

            if False:
                # Load and filter data
                Log.Print("Importing and filtering Data")
                data = LogicData(LOGIC_FILENAME)
                dut = data.getLatchName(LATCH_NAME)
                dut['net_short'] = dut['Net'].replace(r'^.*/sprw/', "", regex=True)
                dut.to_csv("logs/injection_points.csv")

                Log.Print("Selecting injection insructionts")
                it = self.gatherInjectionPoints(contr, ADDR_LIST, INSTRUCTION_END)
                it.to_csv("logs/injction_instrucions.csv")

                Log.Print("Creating full injection datasheet")
                clkoffset = pd.DataFrame({'clk_offset': CLK_OFFSETS})
                INJECTION_CAMPAIGN = dut.join(it, how='cross').join(clkoffset, how='cross')
                INJECTION_CAMPAIGN.to_csv("logs/injection_campaign.csv")
            else:
                INJECTION_CAMPAIGN = pd.read_csv("logs/injection_campaign.csv")

            result = INJECTION_CAMPAIGN
            result['output'] = b''
            result['correct'] = None
            result['runtime'] = 0
            result['timeout'] = False
            result['retries'] = 0
            result['time_ns'] = 0


            contr.setCycleStop(0)
            contr.setEnable(glbl=True, cycle=True)
            contr.reset()

            TOTAL_INJECTIONS = len(INJECTION_CAMPAIGN.index)

            prev_offset = 0
            for index, row in result.iterrows():
                c_run += 1

                pc = row['pc']
                instr = row['instr']
                clk_offset = row['clk_offset']
                Log.Print(f"[{index/TOTAL_INJECTIONS:2.2%}] Running campaign at offset {row['offset']}, pc 0x{pc:x}, instr 0x{instr:x}, skip {clk_offset}")

                result.at[index, 'time_ns'] = time.time_ns()

                tries = RETRIES
                while tries > 0:
                    c_tries += 1
                    result.at[index, 'output'] = b''
                    result.at[index, 'timeout'] = False

                    if uart_dut != None:
                        uart_dut.reset_input_buffer()

                    # Set stopping position to instruction
                    contr.setInstrStop(instr)
                    contr.setPrgmStop(pc)
                    # contr.setPrgmStop(INSTRUCTION_END)
                    contr.reset(); time.sleep(0.1)
                    contr.setEnable(glbl = True, instr = True)
                    # contr.setEnable(glbl = True, prgm = True)

                    # Wait for DUT to be ready and time execution
                    (timer_first, outp) = self.waitAndRecored( ↩
                       TIMEOUT_TIME, contr, uart_dut)

                    # Save output
                    result.at[index, 'output'] += outp
                    # Continue if timeout
                    if timer_first == -1:
                        Log.Print("Timeout 1")
                        result.at[index, 'timeout'] = True
                        c_timeout+=1
                        tries-=1
                        contr.reset(); time.sleep(0.1)
                        continue

                    contr.setCycleStopRelative(clk_offset)
                    contr.setEnable(glbl = True, cycle = True)

                    time.sleep(0.1)

                    # Preform Fault injection
                    frameaddress = int(row['frameaddress'], 16)
                    frameoffset = int(row['frameoffset'])
                    # status = self._commandManager.InjectFault( ↩
                       frameaddress, frameoffset, True)
                    # Log.Print(status)

```

```python
                # Resume execution
                contr.setPrgmStop(INSTRUCTION_END)
                contr.setEnable(glbl = True, prgm=True)
                (timer_second, outp) = self.waitAndRecored(TIMEOUT_TIME - timer_first,
                                                        contr, uart_dut)

                # Save output
                result.at[index, 'output'] += outp
                # Continue if timeout
                if timer_second == -1:
                    Log.Print(f"Timeout 2 at PC 0x{contr.read(contr.PRGM_COUNTER):x}\n{outp=}")
                    result.at[index, 'timeout'] = True
                    c_timeout+=1
                    tries-=1
                    contr.reset(); time.sleep(0.1)
                    continue

                # Save timing info
                result.at[index, 'runtime'] = timer_first + timer_second
                result.at[index, 'retries'] = RETRIES-tries

                if fi.get_output(id).find(b'$') == -1:
                    Log.Print(f"Output not complete\n{outp=}")
                    c_fales+=1
                    tries-=1
                elif fi.get_output(id).find(b'#0') == -1:
                    Log.Print(f"Output not corect\n{outp=}")
                    c_fales+=1
                    tries = -1
                else:
                    c_true+=1
                    result.at[index, 'correct'] = True
                    Log.Print(f"{correct:}, Done in {timer_first + timer_second} ns in {RETRIES-tries} retries (r{c_run}/f{c_fales}/t{c_timeout})")
                    tries = -1

        result.to_csv("logs/injection_results.csv")
        # Closing UART devices
        Log.Print("End of campaign, closing UART devices")
        contr.close()
```

```
            if uart_dut != None:
                uart_dut.close()
            Log.Print(f"Total campaign took {time.time() - global_timer}s\nRan {c_run} campaigns with {c_tries} tries, {c_true} correct, {c_fales} fails and {c_timeout} timeouts.")

        except Exception as e:
            contr.close()
            if uart_dut != None:
                uart_dut.close()
            Log.PrintException(f'UserApplication.run: {str(e)}')
```

Appendix B
Benchmark code listing

main.c

```c
int main() {
    // Setup
    neorv32_uart0_setup(BAUD_RATE, 0);

    if (neorv32_crc_available() == 0) {
        neorv32_uart0_putc('X');
        return 1;
    }

    if (!get_implementation()) {
        neorv32_uart0_putc('Y');
    }

    // Variable Setup
    unsigned char A[N][N], B[N][N], C[N][N], D[N][N];
    unsigned int previous = 134775813U;
    previous = init((unsigned char*) A, sizeof(A[0]), previous);
    previous = init((unsigned char*) B, sizeof(B[0]), previous);
    char f_1[3][3] = { {1, 0, 1},
                       {0, 2, 0},
                       {0, 0, 0}};

    // Benchmark
    product((unsigned char*) A, (unsigned char*) B, sizeof(A[0]), (
        unsigned char*) C);
    conv_filter((unsigned char*) C, (unsigned char*) D, sizeof(C
        [0]), (unsigned char*) f_1);

    // Print Result
    neorv32_crc_setup(CRC_MODE32, 0xf8c9140a, 0);
    int result = neorv32_crc_block((uint8_t*)D, sizeof(D));
    uint8_t crc[4] = {0xb4, 0x9f, 0x6a, 0x70};
    result = neorv32_crc_block((uint8_t*)crc, sizeof(crc));
    neorv32_uart0_putc('#');
    print_uint(result);
    neorv32_uart0_putc('$');

    while ((NEORV32_UART0->CTRL & (1<<UART_CTRL_TX_BUSY)));

    return 0;
}
```

```
unsigned int init(unsigned char* matrix, size_t edge_size, unsigned ↩
    int previous){
    for(int i =0; i < edge_size*edge_size; i++){
        previous = previous*48271U;
        *matrix++ = previous%5;
    }
    return previous;
}

void print_uint(uint32_t num){
    if (num > 9) print_uint(num/10);
    neorv32_uart0_putc('0' + (num%10));
}
```

functions.c (Convolution filter)

```
void conv_filter(unsigned char* src, unsigned char* dst, const ↩
   size_t edge_size, const unsigned char* filter) {
    memset(dst, 0, edge_size*edge_size*sizeof(char));
    union mask_t mk;
    sprw_ctrl_ms_clear();
    // Limits change for every row of kernel
    const int limit[3][3] = {{0,-1,1}, {0,0,0}, {1,0,0}};

    int kern, value, dot;
    for(int k = 0; k < 3; k++){ // Loop through every row of kernel
        // kern = *(int *) (filter + k*3);
        memcpy(&kern, filter + k*3, sizeof(int));
        const int max = edge_size * (edge_size + limit[k][1]);
        unsigned char *dst_p = dst + limit[k][2]*edge_size;
        for(int j = limit[k][0]*edge_size; j < max; j+= edge_size){ ↩
            // Row index of src
            // first
            mk.elem = (struct mask_lane_t) {0, 1, 1, 0}; ↩
               sprw_ctrl_mk_set(mk);
            // value = *(int *) (src+j);
            memcpy(&value, src + j, sizeof(int));
            value = value << 8;
            asm volatile ("usdot %0, %1, %2" : "=r"(dot) : "r"(value), "r"(kern));
            *dst_p++ += dot;
            // neorv32_uart0_printf("%d, k: %x, v: %x, d: %x %x (first)\n",dst_p - dst, kern, value, dot, *(dst_p-1));


            mk.elem = (struct mask_lane_t) {1, 1, 1, 0}; sprw_ctrl_mk_set(mk);
            for(int i = 1; i < edge_size - 2; i++) { // Column index of src (3 at the time)
                // value = *(int *) (src+j+i-1);
                memcpy(&value, src + j + i-1, sizeof(int));
                asm volatile ("usdot %0, %1, %2" : "=r"(dot) : "r"(value), "r"(kern));
                *dst_p++ += dot;
                // neorv32_uart0_printf("%d, k: %x, v: %x, d: %x %x\n",dst_p - dst, kern, value, dot, *(dst_p-1));
            }


            //second laste
            value = value >> 8;
            asm volatile ("usdot %0, %1, %2" : "=r"(dot) : "r"(value), "r"(kern));
            *dst_p++ += dot;
            // neorv32_uart0_printf("%d, k: %x, v: %x, d: %x %x (last 1)\n",dst_p - dst, kern, value, dot, *(dst_p-1));
            //second laste
            mk.elem = (struct mask_lane_t) {1, 1, 0, 0}; sprw_ctrl_mk_set(mk);
            value = value >> 8;
            asm volatile ("usdot %0, %1, %2" : "=r"(dot) : "r"(value), "r"(kern));
            *dst_p++ += dot;
            // neorv32_uart0_printf("%d, k: %x, v: %x, d: %x %x (last 2)\n",dst_p - dst, kern, value, dot, *(dst_p-1));
        }

    }
    sprw_ctrl_reset();

}
```

functions.c (Matrix product)

```c
void product(const unsigned char* A, const unsigned char* B, const size_t edge_size, unsigned char* C) {
    memset(C, 0, edge_size*edge_size*sizeof(char));

    // Some administration to handle resting part of calculation not fitting
    // in width 4 sparrow ALU
    int mod = edge_size%4;
    const int max = edge_size - mod;
    union mask_t mk;
    mk.vector = 0xf0 >> mod & 0x0f;
    sprw_ctrl_ms_clear(); // Set ms to 0 such that masking vector masks to 0

    int* matA = malloc(sizeof(int));
    int* matB = malloc(sizeof(int));

    for (size_t i = 0; i < edge_size; i++) {
        for (size_t j = 0; j < edge_size; j++) {
            int dot;
            int sum = 0;
            for (size_t k = 0; k < max; k += 4) {
                memcpy(matA, A+i*edge_size+k, sizeof(int));
                memcpy(matB, B+j*edge_size+k, sizeof(int));
                asm("usdot %0, %1, %2" : "=r"(dot) : "r"(*matA), "r"(*matB));
                // asm("nop");
                asm("usadd_ %0, %1, %2" : "=r"(sum) : "r"(sum), "r"(dot));
            }
            if (mod != 0) {
                sprw_ctrl_mk_set(mk);
                memcpy(matA, A+i*edge_size+edge_size-4, sizeof(int));
                memcpy(matB, B+j*edge_size+edge_size-4, sizeof(int));
                asm volatile ("usdot %0, %1, %2" : "=r"(dot) : "r"(*matA), "r"(*matB));
                sprw_ctrl_mk_clear();
                asm volatile ("usadd_ %0, %1, %2" : "=r"(sum) : "r"(sum), "r"(dot));
            }
            C[i * edge_size + j] = sum;
        }
    }
    sprw_ctrl_reset();
}
```

Appendix C
Compiled benchmark

main.asm (convolutional filter)

```
00000194 <conv_filter>:
; void conv_filter(unsigned char* src, unsigned char* dst, const size_t edge_size, const unsigned char* filter) {
     194: 13 01 01 fb  ␣addi␣␣sp, sp, -80
     198: 23 26 11 04  ␣sw␣␣ra, 76(sp)
     19c: 23 24 81 04  ␣sw␣␣s0, 72(sp)
     1a0: 23 22 91 04  ␣sw␣␣s1, 68(sp)
     1a4: 23 20 21 05  ␣sw␣␣s2, 64(sp)
     1a8: 23 2e 31 03  ␣sw␣␣s3, 60(sp)
     1ac: 23 2c 41 03  ␣sw␣␣s4, 56(sp)
     1b0: 23 2a 51 03  ␣sw␣␣s5, 52(sp)
     1b4: 23 28 61 03  ␣sw␣␣s6, 48(sp)
     1b8: 23 26 71 03  ␣sw␣␣s7, 44(sp)
     1bc: 23 24 81 03  ␣sw␣␣s8, 40(sp)
     1c0: 13 89 06 00  ␣mv␣␣s2, a3
     1c4: 13 0b 06 00  ␣mv␣␣s6, a2
     1c8: 93 89 05 00  ␣mv␣␣s3, a1
     1cc: 93 0a 05 00  ␣mv␣␣s5, a0
;       memset(dst, 0, edge_size*edge_size*sizeof(char));
     1d0: 33 06 c6 02  ␣mul␣a2, a2, a2
     1d4: 13 85 05 00  ␣mv␣␣a0, a1
     1d8: 93 05 00 00  ␣mv␣␣a1, zero
     1dc: ef 00 40 6f  ␣jal␣0x8d0 <memset>
     1e0: 13 05 00 01  ␣addi␣␣a0, zero, 16
;     asm volatile ("csrc %[reg], %[val]":: [reg] "i" (CSR_SPRW), [val] "r" (SPRW_MS_MSK) );
     1e4: 73 30 05 80  ␣csrc␣␣2048, a0
;       const int limit[3][3] = {{0,-1,1}, {0,0,0}, {1,0,0}};
     1e8: 13 05 41 00  ␣addi␣␣a0, sp, 4
     1ec: 13 06 40 02  ␣addi␣␣a2, zero, 36
     1f0: 13 0a 41 00  ␣addi␣␣s4, sp, 4
     1f4: 93 05 00 00  ␣mv␣␣a1, zero
     1f8: ef 00 80 6d  ␣jal␣0x8d0 <memset>
     1fc: 93 03 00 00  ␣mv␣␣t2, zero
     200: 13 05 f0 ff  ␣addi␣␣a0, zero, -1
;      const int limit[3][3] = {{0,-1,1}, {0,0,0}, {1,0,0}};
     204: 23 24 a1 00  ␣sw␣␣a0, 8(sp)
     208: 93 05 10 00  ␣addi␣␣a1, zero, 1
     20c: 23 26 b1 00  ␣sw␣␣a1, 12(sp)
     210: 23 2e b1 00  ␣sw␣␣a1, 28(sp)
     214: 13 06 eb ff  ␣addi␣␣a2, s6, -2
     218: 33 be c5 00  ␣sltu␣␣t3, a1, a2
;      for(int k = 0; k < 3; k++){ // Loop through every row of ↩
    kernel
     21c: 93 0e db ff  ␣addi␣␣t4, s6, -3
     220: 13 08 c0 00  ␣addi␣␣a6, zero, 12
     224: 93 0f f0 00  ␣addi␣␣t6, zero, 15
     228: 93 08 60 00  ␣addi␣␣a7, zero, 6
     22c: 93 02 70 00  ␣addi␣␣t0, zero, 7
     230: 13 03 30 00  ␣addi␣␣t1, zero, 3
;          for(int j = limit[k][0]*edge_size; j < max; j+= edge_size ↩
    ){ // Row index of src
     234: 33 86 03 03  ␣mul␣a2, t2, a6
     238: 33 06 ca 00  ␣add␣a2, s4, a2
     23c: 03 26 06 00  ␣lw␣␣a2, 0(a2)
;          const int max = edge_size * (edge_size + limit[k][1]);
     240: 33 05 65 01  ␣add␣a0, a0, s6
     244: 33 0f 65 03  ␣mul␣t5, a0, s6
;          for(int j = limit[k][0]*edge_size; j < max; j+= edge_size ↩
    ){ // Row index of src
     248: b3 0b 66 03  ␣mul␣s7, a2, s6
     24c: 63 d2 eb 13  ␣bge␣s7, t5, 0x370 <conv_filter+0x1dc>
     250: 13 95 13 00  ␣slli␣␣a0, t2, 1
     254: 33 05 75 00  ␣add␣a0, a0, t2
     258: 33 05 a9 00  ␣add␣a0, s2, a0
     25c: 03 46 15 00  ␣lbu␣a2, 1(a0)
     260: 83 46 05 00  ␣lbu␣a3, 0(a0)
     264: 03 47 35 00  ␣lbu␣a4, 3(a0)
     268: 03 45 25 00  ␣lbu␣a0, 2(a0)
     26c: 13 16 86 00  ␣slli␣␣a2, a2, 8
     270: 33 66 d6 00  ␣or␣␣a2, a2, a3
     274: 93 16 87 00  ␣slli␣␣a3, a4, 8
     278: 33 e5 a6 00  ␣or␣␣a0, a3, a0
     27c: 13 15 05 01  ␣slli␣␣a0, a0, 16
     280: 33 65 c5 00  ␣or␣␣a0, a0, a2
;          unsigned char *dst_p = dst + limit[k][2]*edge_size;
     284: b3 85 65 03  ␣mul␣a1, a1, s6
     288: b3 85 b9 00  ␣add␣a1, s3, a1
;          for(int j = limit[k][0]*edge_size; j < max; j+= edge_size ↩
    ){ // Row index of src
     28c: 33 8c 7a 01 ␣add␣s8, s5, s7
;    asm volatile ("csrc %[reg], %[val]":: [reg] "i" (CSR_SPRW), [ ↩
    val] "r" (SPRW_MK_MSK) );
     290: 73 b0 0f 80 ␣csrc␣␣2048, t6
;    asm volatile ("csrs %[reg], %[val]":: [reg] "i" (CSR_SPRW), [ ↩
    val] "r" (mask) );
     294: 73 a0 08 80 ␣csrs␣␣2048, a7
;               memcpy(&value, src + j, sizeof(int));
     298: 33 86 7a 01 ␣add␣a2, s5, s7
     29c: 83 46 16 00 ␣lbu␣a3, 1(a2)
     2a0: 03 44 06 00 ␣lbu␣s0, 0(a2)
     2a4: 03 46 26 00 ␣lbu␣a2, 2(a2)
     2a8: 93 96 86 00 ␣slli␣␣a3, a3, 8
     2ac: b3 e6 86 00 ␣or␣␣a3, a3, s0
     2b0: 13 16 06 01 ␣slli␣␣a2, a2, 16
     2b4: 33 66 d6 00 ␣or␣␣a2, a2, a3
;               value = value << 8;
     2b8: 93 16 86 00 ␣slli␣␣a3, a2, 8
;               asm volatile ("usdot %0, %1, %2" : "=r"(dot) : "r"( ↩
    value), "r"(kern));
     2bc: 0b d6 a6 7c ␣usmul_usum␣␣a2, a3, a0
;               *dst_p++ += dot;
     2c0: 03 84 05 00 ␣lb␣␣s0, 0(a1)
     2c4: 33 06 c4 00 ␣add␣a2, s0, a2
     2c8: 23 80 c5 00 ␣sb␣␣a2, 0(a1)
;    asm volatile ("csrc %[reg], %[val]":: [reg] "i" (CSR_SPRW), [ ↩
    val] "r" (SPRW_MK_MSK) );
     2cc: 73 b0 0f 80 ␣csrc␣␣2048, t6
;    asm volatile ("csrs %[reg], %[val]":: [reg] "i" (CSR_SPRW), [ ↩
    val] "r" (mask) );
     2d0: 73 a0 02 80 ␣csrs␣␣2048, t0
     2d4: 13 86 15 00 ␣addi␣␣a2, a1, 1
     2d8: 13 04 0c 00 ␣mv␣␣s0, s8
     2dc: 93 84 0e 00 ␣mv␣␣s1, t4
;               for(int i = 1; i < edge_size - 2; i++) { // Column ↩
    index of src (3 at the time)
     2e0: 63 08 0e 04 ␣beqz␣␣t3, 0x330 <conv_filter+0x19c>
;                   memcpy(&value, src + j + i-1, sizeof(int));
     2e4: 83 45 14 00 ␣lbu␣a1, 1(s0)
     2e8: 83 46 04 00 ␣lbu␣a3, 0(s0)
     2ec: 83 47 34 00 ␣lbu␣a5, 3(s0)
     2f0: 03 47 24 00 ␣lbu␣a4, 2(s0)
     2f4: 93 95 85 00 ␣slli␣␣a1, a1, 8
     2f8: b3 e5 d5 00 ␣or␣␣a1, a1, a3
     2fc: 93 96 87 00 ␣slli␣␣a3, a5, 8
     300: b3 e6 e6 00 ␣or␣␣a3, a3, a4
    304: 93 96 06 01  ␣slli␣a3, a3, 16
    308: b3 e6 b6 00  ␣or␣a3, a3, a1
;                    asm volatile ("usdot %0, %1, %2" : "=r"(dot) : "r"(value), "r"(kern));
    30c: 8b d5 a6 7c  ␣usmul_usum␣a1, a3, a0
;                    *dst_p++ += dot;
    310: 03 07 06 00  ␣lb␣a4, 0(a2)
    314: b3 05 b7 00  ␣add␣a1, a4, a1
    318: 23 00 b6 00  ␣sb␣a1, 0(a2)
    31c: 13 06 16 00  ␣addi␣a2, a2, 1
;                for(int i = 1; i < edge_size - 2; i++) { // Column index of src (3 at the time)
    320: 93 84 f4 ff  ␣addi␣s1, s1, -1
;                    memcpy(&value, src + j + i-1, sizeof(int));
    324: 13 04 14 00  ␣addi␣s0, s0, 1
;                for(int i = 1; i < edge_size - 2; i++) { // Column index of src (3 at the time)
    328: e3 9e 04 fa  ␣bnez␣s1, 0x2e4 <conv_filter+0x150>
;                value = value >> 8;
    32c: 93 05 f6 ff  ␣addi␣a1, a2, -1
    330: 13 d4 86 40  ␣srai␣s0, a3, 8
;                asm volatile ("usdot %0, %1, %2" : "=r"(dot) : "r"(value), "r"(kern));
    334: 0b 54 a4 7c  ␣usmul_usum␣s0, s0, a0
;                *dst_p++ += dot;
    338: 83 04 06 00  ␣lb␣s1, 0(a2)
    33c: 33 84 84 00  ␣add␣s0, s1, s0
    340: 23 00 86 00  ␣sb␣s0, 0(a2)
;   asm volatile ("csrc %[reg], %[val]":: [reg] "i" (CSR_SPRW), [val] "r" (SPRW_MK_MSK) );
    344: 73 b0 0f 80  ␣csrc␣2048, t6
;   asm volatile ("csrs %[reg], %[val]":: [reg] "i" (CSR_SPRW), [val] "r" (mask) );
    348: 73 20 03 80  ␣csrs␣2048, t1
;                value = value >> 8;
    34c: 13 d6 06 41  ␣srai␣a2, a3, 16
;                asm volatile ("usdot %0, %1, %2" : "=r"(dot) : "r"(value), "r"(kern));
    350: 0b 56 a6 7c  ␣usmul_usum␣a2, a2, a0
;                *dst_p++ += dot;
    354: 83 86 25 00  ␣lb␣a3, 2(a1)
    358: 33 86 c6 00  ␣add␣a2, a3, a2
    35c: 23 81 c5 00  ␣sb␣a2, 2(a1)
    360: 93 85 35 00  ␣addi␣a1, a1, 3
;            for(int j = limit[k][0]*edge_size; j < max; j+= edge_size){ // Row index of src
    364: b3 8b 6b 01  ␣add␣s7, s7, s6
```

```
     368: 33 0c 6c 01  ␣add␣s8, s8, s6
     36c: e3 c2 eb f3  ␣blt␣s7, t5, 0x290 <conv_filter+0xfc>
;      for(int k = 0; k < 3; k++){ // Loop through every row of ↩
    kernel
     370: 93 83 13 00  ␣addi␣␣t2, t2, 1
     374: 63 8c 63 00  ␣beq␣t2, t1, 0x38c <conv_filter+0x1f8>
     378: 33 85 03 03  ␣mul␣a0, t2, a6
     37c: b3 05 aa 00  ␣add␣a1, s4, a0
;          const int max = edge_size * (edge_size + limit[k][1]);
     380: 03 a5 45 00  ␣lw␣␣a0, 4(a1)
;          unsigned char *dst_p = dst + limit[k][2]*edge_size;
     384: 83 a5 85 00  ␣lw␣␣a1, 8(a1)
     388: 6f f0 df ea  ␣j␣0x234 <conv_filter+0xa0>
     38c: 37 a5 5c 00  ␣lui␣a0, 1482
     390: 13 05 f5 c9  ␣addi␣␣a0, a0, -865
;    asm volatile ("csrw %[reg], %[val]":: [reg] "i" (CSR_SPRW), [ ↩
    val] "r" (SPRW_DEFAULT) );
     394: 73 10 05 80  ␣csrw␣␣2048, a0
; }
     398: 03 2c 81 02  ␣lw␣␣s8, 40(sp)
     39c: 83 2b c1 02  ␣lw␣␣s7, 44(sp)
     3a0: 03 2b 01 03  ␣lw␣␣s6, 48(sp)
     3a4: 83 2a 41 03  ␣lw␣␣s5, 52(sp)
     3a8: 03 2a 81 03  ␣lw␣␣s4, 56(sp)
     3ac: 83 29 c1 03  ␣lw␣␣s3, 60(sp)
     3b0: 03 29 01 04  ␣lw␣␣s2, 64(sp)
     3b4: 83 24 41 04  ␣lw␣␣s1, 68(sp)
     3b8: 03 24 81 04  ␣lw␣␣s0, 72(sp)
     3bc: 83 20 c1 04  ␣lw␣␣ra, 76(sp)
     3c0: 13 01 01 05  ␣addi␣␣sp, sp, 80
     3c4: 67 80 00 00  ␣ret
```

main.asm (Matrix product)

```
000003c8 <product>:
; void product(const unsigned char* A, const unsigned char* B, ↩
    const size_t edge_size, unsigned char* C) {
     3c8: 13 01 01 fd  ␣addi␣␣sp, sp, -48
     3cc: 23 26 11 02  ␣sw␣␣ra, 44(sp)
     3d0: 23 24 81 02  ␣sw␣␣s0, 40(sp)
     3d4: 23 22 91 02  ␣sw␣␣s1, 36(sp)
     3d8: 23 20 21 03  ␣sw␣␣s2, 32(sp)
     3dc: 23 2e 31 01  ␣sw␣␣s3, 28(sp)
     3e0: 23 2c 41 01  ␣sw␣␣s4, 24(sp)
     3e4: 23 2a 51 01  ␣sw␣␣s5, 20(sp)
     3e8: 23 28 61 01  ␣sw␣␣s6, 16(sp)
     3ec: 23 26 71 01  ␣sw␣␣s7, 12(sp)
     3f0: 13 8a 06 00 ␣mv␣␣s4, a3
     3f4: 93 0a 06 00 ␣mv␣␣s5, a2
     3f8: 13 89 05 00 ␣mv␣␣s2, a1
     3fc: 93 09 05 00 ␣mv␣␣s3, a0
;     memset(C, 0, edge_size*edge_size*sizeof(char));
     400: 33 06 c6 02 ␣mul␣a2, a2, a2
     404: 13 85 06 00 ␣mv␣␣a0, a3
     408: 93 05 00 00 ␣mv␣␣a1, zero
     40c: ef 00 40 4c ␣jal␣0x8d0 <memset>
     410: 13 05 00 01 ␣addi␣␣a0, zero, 16
;   asm volatile ("csrc %[reg], %[val]":: [reg] "i" (CSR_SPRW), [val] "r" (SPRW_MS_MSK) );
     414: 73 30 05 80 ␣csrc␣␣2048, a0
;     for (size_t i = 0; i < edge_size; i++) {
     418: 63 80 0a 16 ␣beqz␣␣s5, 0x578 <product+0x1b0>
     41c: 13 08 00 00 ␣mv␣␣a6, zero
     420: 13 f5 3a 00 ␣andi␣␣a0, s5, 3
     424: 13 fb ca ff ␣andi␣␣s6, s5, -4
     428: 13 06 00 0f ␣addi␣␣a2, zero, 240
     42c: 33 56 a6 00 ␣srl␣a2, a2, a0
     430: 93 78 e6 00 ␣andi␣␣a7, a2, 14
     434: 93 33 1b 00 ␣seqz␣␣t2, s6
     438: 13 3e 15 00 ␣seqz␣␣t3, a0
     43c: 13 83 ca ff ␣addi␣␣t1, s5, -4
     440: 93 02 f0 00 ␣addi␣␣t0, zero, 15
     444: 93 8b 09 00 ␣mv␣␣s7, s3
     448: 93 0f 00 00 ␣mv␣␣t6, zero
     44c: b3 0e 58 03 ␣mul␣t4, a6, s5
     450: b3 06 d3 01 ␣add␣a3, t1, t4
     454: 33 8f d9 00 ␣add␣t5, s3, a3
     458: 93 06 09 00 ␣mv␣␣a3, s2
;             for (size_t k = 0; k < max; k += 4) {
     45c: 63 92 03 10 ␣bnez␣␣t2, 0x560 <product+0x198>
     460: 93 07 00 00 ␣mv␣␣a5, zero
     464: 13 07 00 00 ␣mv␣␣a4, zero
;                 memcpy(matA, A+i*edge_size+k, sizeof(int));
     468: b3 84 fb 00 ␣add␣s1, s7, a5
     46c: 03 c4 14 00 ␣lbu␣s0, 1(s1)
     470: 03 c5 04 00 ␣lbu␣a0, 0(s1)
     474: 83 c5 34 00 ␣lbu␣a1, 3(s1)
     478: 83 c4 24 00 ␣lbu␣s1, 2(s1)
     47c: 13 14 84 00 ␣slli␣␣s0, s0, 8
     480: 33 65 a4 00 ␣or␣␣a0, s0, a0
     484: 93 95 85 00 ␣slli␣␣a1, a1, 8
     488: b3 e5 95 00 ␣or␣␣a1, a1, s1
     48c: 93 95 05 01 ␣slli␣␣a1, a1, 16
     490: 33 e5 a5 00 ␣or␣␣a0, a1, a0
```

```
;                    memcpy(matB, B+j*edge_size+k, sizeof(int));
     494: b3 85 f6 00  ␣add␣a1, a3, a5
     498: 03 c4 15 00  ␣lbu␣s0, 1(a1)
     49c: 83 c4 05 00  ␣lbu␣s1, 0(a1)
     4a0: 03 c6 35 00  ␣lbu␣a2, 3(a1)
     4a4: 83 c5 25 00  ␣lbu␣a1, 2(a1)
     4a8: 13 14 84 00  ␣slli␣␣s0, s0, 8
     4ac: 33 64 94 00  ␣or␣␣s0, s0, s1
     4b0: 13 16 86 00  ␣slli␣␣a2, a2, 8
     4b4: b3 65 b6 00  ␣or␣␣a1, a2, a1
     4b8: 93 95 05 01  ␣slli␣␣a1, a1, 16
     4bc: b3 e5 85 00  ␣or␣␣a1, a1, s0
;                    asm("usdot %0, %1, %2" : "=r"(dot) : "r"(*matA), ↩
    "r"(*matB));
     4c0: 0b 55 b5 7c  ␣usmul_usum␣␣a0, a0, a1
;                for (size_t k = 0; k < max; k += 4) {
     4c4: 93 87 47 00  ␣addi␣␣a5, a5, 4
;                    asm("usadd_ %0, %1, %2" : "=r"(sum) : "r"(sum), " ↩
    r"(dot));
     4c8: 0b 07 a7 74  ␣usadd_␣␣a4, a4, a0
;                for (size_t k = 0; k < max; k += 4) {
     4cc: e3 ee 67 f9  ␣bltu␣␣a5, s6, 0x468 <product+0xa0>
     4d0: 63 1a 0e 06  ␣bnez␣␣t3, 0x544 <product+0x17c>
     4d4: 33 85 5f 03  ␣mul␣a0, t6, s5
;    asm volatile ("csrc %[reg], %[val]":: [reg] "i" (CSR_SPRW), [ ↩
    val] "r" (SPRW_MK_MSK) );
     4d8: 73 b0 02 80  ␣csrc␣␣2048, t0
;    asm volatile ("csrs %[reg], %[val]":: [reg] "i" (CSR_SPRW), [ ↩
    val] "r" (mask) );
     4dc: 73 a0 08 80  ␣csrs␣␣2048, a7
;                    memcpy(matA, A+i*edge_size+edge_size-4, sizeof( ↩
    int));
     4e0: 83 45 1f 00  ␣lbu␣a1, 1(t5)
     4e4: 03 46 0f 00  ␣lbu␣a2, 0(t5)
     4e8: 83 47 3f 00  ␣lbu␣a5, 3(t5)
     4ec: 03 44 2f 00  ␣lbu␣s0, 2(t5)
     4f0: 93 95 85 00  ␣slli␣␣a1, a1, 8
     4f4: b3 e5 c5 00  ␣or␣␣a1, a1, a2
     4f8: 13 96 87 00  ␣slli␣␣a2, a5, 8
     4fc: 33 66 86 00  ␣or␣␣a2, a2, s0
     500: 13 16 06 01  ␣slli␣␣a2, a2, 16
     504: b3 65 b6 00  ␣or␣␣a1, a2, a1
;                    memcpy(matB, B+j*edge_size+edge_size-4, sizeof( ↩
    int));
     508: 33 05 a3 00  ␣add␣a0, t1, a0
     50c: 33 05 a9 00  ␣add␣a0, s2, a0
     510: 03 46 15 00  ␣lbu␣a2, 1(a0)
     514: 83 47 05 00 ␣lbu␣a5, 0(a0)
     518: 03 44 35 00 ␣lbu␣s0, 3(a0)
     51c: 03 45 25 00 ␣lbu␣a0, 2(a0)
     520: 13 16 86 00 ␣slli␣a2, a2, 8
     524: 33 66 f6 00 ␣or␣a2, a2, a5
     528: 93 17 84 00 ␣slli␣a5, s0, 8
     52c: 33 e5 a7 00 ␣or␣a0, a5, a0
     530: 13 15 05 01 ␣slli␣a0, a0, 16
     534: 33 65 c5 00 ␣or␣a0, a0, a2
;                 asm volatile ("usdot %0, %1, %2" : "=r"(dot) : "r ↩
    "(*matA), "r"(*matB));
     538: 0b d5 a5 7c ␣usmul_usum␣a0, a1, a0
;   asm volatile ("csrs %[reg], %[val]":: [reg] "i" (CSR_SPRW), [ ↩
    val] "r" (SPRW_MK_MSK) );
     53c: 73 a0 02 80 ␣csrs␣2048, t0
;                 asm volatile ("usadd_ %0, %1, %2" : "=r"(sum) : " ↩
    r"(sum), "r"(dot));
     540: 0b 07 a7 74 ␣usadd_␣a4, a4, a0
;             C[i * edge_size + j] = sum;
     544: 33 85 df 01 ␣add␣a0, t6, t4
     548: 33 05 aa 00 ␣add␣a0, s4, a0
     54c: 23 00 e5 00 ␣sb␣a4, 0(a0)
;         for (size_t j = 0; j < edge_size; j++) {
     550: 93 8f 1f 00 ␣addi␣t6, t6, 1
     554: b3 86 56 01 ␣add␣a3, a3, s5
     558: e3 92 5f f1 ␣bne␣t6, s5, 0x45c <product+0x94>
     55c: 6f 00 00 01 ␣j␣0x56c <product+0x1a4>
     560: 13 07 00 00 ␣mv␣a4, zero
     564: e3 08 0e f6 ␣beqz␣t3, 0x4d4 <product+0x10c>
     568: 6f f0 df fd ␣j␣0x544 <product+0x17c>
;     for (size_t i = 0; i < edge_size; i++) {
     56c: 13 08 18 00 ␣addi␣a6, a6, 1
     570: b3 8b 5b 01 ␣add␣s7, s7, s5
     574: e3 1a 58 ed ␣bne␣a6, s5, 0x448 <product+0x80>
     578: 37 a5 5c 00 ␣lui␣a0, 1482
     57c: 13 05 f5 c9 ␣addi␣a0, a0, -865
;   asm volatile ("csrw %[reg], %[val]":: [reg] "i" (CSR_SPRW), [ ↩
    val] "r" (SPRW_DEFAULT) );
     580: 73 10 05 80 ␣csrw␣2048, a0
; }
     584: 83 2b c1 00 ␣lw␣s7, 12(sp)
     588: 03 2b 01 01 ␣lw␣s6, 16(sp)
     58c: 83 2a 41 01 ␣lw␣s5, 20(sp)
     590: 03 2a 81 01 ␣lw␣s4, 24(sp)
     594: 83 29 c1 01 ␣lw␣s3, 28(sp)
     598: 03 29 01 02 ␣lw␣s2, 32(sp)
     59c: 83 24 41 02 ␣lw␣s1, 36(sp)
    5a0: 03 24 81 02  ␣lw␣␣s0, 40(sp)
    5a4: 83 20 c1 02  ␣lw␣␣ra, 44(sp)
    5a8: 13 01 01 03  ␣addi␣␣sp, sp, 48
    5ac: 67 80 00 00  ␣ret
```

Appendix D

SPARROW implementation in NEORV32

This chapter outlines the added hardware and software to the NEORV32 project [19] for implementing SPARROW [15], [68] as a CFU. First hardware additions are shown, starting with the CFU to SPARROW interface. Also, changes to NEORV32s control bus are included. Then, a small addition to the CSR library file is listed, whereafter a new library for interfacing with the SPARROW CSR is provided.

neorv32_cpu_cp_cfu_sparrow.vhd

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library neorv32;
use neorv32.neorv32_package.all;

library sparrow;
use sparrow.sparrow.all;

entity neorv32_cpu_cp_cfu is
  port (
    -- global control --
    clk_i   : in  std_ulogic; -- global clock, rising edge
    rstn_i  : in  std_ulogic; -- global reset, low-active, async
    ctrl_i  : in  ctrl_bus_t; -- main control bus
    start_i : in  std_ulogic; -- trigger operation
    -- data input --
    rs1_i   : in  std_ulogic_vector(XLEN-1 downto 0); -- rf source ↩
        1
    rs2_i   : in  std_ulogic_vector(XLEN-1 downto 0); -- rf source ↩
        2
    rs3_i   : in  std_ulogic_vector(XLEN-1 downto 0); -- rf source 3
    rs4_i   : in  std_ulogic_vector(XLEN-1 downto 0); -- rf source 4
    -- result and status --
    res_o   : out std_ulogic_vector(XLEN-1 downto 0); -- operation result
    valid_o : out std_ulogic -- data output valid
  );
end neorv32_cpu_cp_cfu;

architecture neorv32_cpu_cp_cfu_rtl of neorv32_cpu_cp_cfu is

  -- CFU Control - do not modify! ----------------------------
  -- ------------------------------------------------------------

  type control_t is record
    busy   : std_ulogic; -- CFU is busy
    done   : std_ulogic; -- set to '1' when processing is done
    result : std_ulogic_vector(XLEN-1 downto 0); -- user's processing result (for write-back to register file)
    rtype  : std_ulogic_vector(1 downto 0); -- instruction type, see constants below
    funct3 : std_ulogic_vector(2 downto 0); -- "funct3" bit-field from custom instruction
    funct7 : std_ulogic_vector(6 downto 0); -- "funct7" bit-field from custom instruction
  end record;
  signal control : control_t;

  -- instruction format types --
  constant r3type_c  : std_ulogic_vector(1 downto 0) := "00"; -- R3-type instructions (custom-0 opcode)
  constant r4type_c  : std_ulogic_vector(1 downto 0) := "01"; -- R4-type instructions (custom-1 opcode)
  constant r5typeA_c : std_ulogic_vector(1 downto 0) := "10"; -- R5-type instruction A (custom-2 opcode)
  constant r5typeB_c : std_ulogic_vector(1 downto 0) := "11"; -- R5-type instruction B (custom-3 opcode)

  -- User Logic --------------------------------------------
  -- ------------------------------------------------------------

  -- multiply-add unit (r4-type instruction example) --
  type sprwctl_t is record
    sreg : std_ulogic_vector(2 downto 0); -- 3 cycles latency in arbitration shift register
    done : std_ulogic;
  end record;
  signal sprwctl : sprwctl_t;


  signal sdi: sprw_in_type;
  signal sdo: sprw_out_type;
  signal holdn: std_ulogic := '1';
  signal sprw_reg: sprw_ctrl_reg_type;

  for all : sprw_module use entity sparrow.sprw_module(rtl);

  function swizzling_set(sz_i : std_ulogic_vector(VSIZE*LOGSZ-1 downto 0)) return swizzling_reg_type is
      variable res_val : swizzling_reg_type;
  begin
      for i in 0 to (XLEN/VLEN)-1 loop
          res_val(i) := to_integer(unsigned(sz_i(i*LOGSZ+LOGSZ-1 downto i*LOGSZ)));
      end loop;
      return res_val;
  end function swizzling_set;

  function to_scr(data : std_ulogic_vector) return sprw_ctrl_reg_type is
      variable reg : sprw_ctrl_reg_type;
  begin
      reg.mk := to_stdlogicvector(data(3 downto 0));
      reg.ms := data(4);
      reg.sa := swizzling_set(data(12 downto 5));
      reg.sb := swizzling_set(data(20 downto 13));
      reg.ol := to_stdlogicvector(data(22 downto 21));
      reg.od := to_stdlogicvector(data(26 downto 23));
      reg.hp := data(27);
      return reg;
  end to_scr;

begin

-- ******************************************************************************
-- This controller is required to handle the CPU/pipeline interface. Do not modify!
-- ******************************************************************************

  -- CFU Controller ↩
      ------------------------------------------------------------------------ ↩

  -- ↩
      ----------------------------------------------------------------------------------

  cfu_control: process(rstn_i, clk_i)
  begin
    if (rstn_i = '0') then
      res_o        <= (others => '0');
      control.busy <= '0';
    elsif rising_edge(clk_i) then
      res_o <= (others => '0'); -- default; all CPU co-processor ↩
          outputs are logically OR-ed
      if (control.busy = '0') then -- idle
        if (start_i = '1') then
          control.busy <= '1';
        end if;
      else -- busy
        if (control.done = '1') or (ctrl_i.cpu_trap = '1') then -- ↩
            processing done? abort if trap (exception)
          res_o        <= control.result; -- output result for just ↩
              one cycle, CFU output has to be all-zero otherwise
          control.busy <= '0';
        end if;
      end if;
    end if;
  end process cfu_control;

  -- CPU feedback --
  valid_o <= control.busy and control.done; -- set one cycle before ↩
      result data

  -- pack user-defined instruction type/function bits --
  control.rtype  <= ctrl_i.ir_opcode(6 downto 5);
  control.funct3 <= ctrl_i.ir_funct3;
  control.funct7 <= ctrl_i.ir_funct12(11 downto 5);

  sprw: sprw_module port map(clk => clk_i,
                             rstn => rstn_i,
                             holdn => holdn,
                             sdi => sdi,
                             sdo => sdo);


  sprw_reg <= ( mk => (others => '1'),
                ms => '0',
                sa => (0, 1, 2, 3),
                sb => (0, 1, 2, 3),
                ol => "10",
                od => (others => '0'),
                hp => '0');

  sdi.ra <= to_stdlogicvector(rs1_i);
  sdi.rb <= to_stdlogicvector(rs2_i);
  sdi.op1 <= to_stdlogicvector(control.funct7(5 downto 1));
  sdi.op2 <= to_stdlogicvector(control.funct3);
  sdi.rc_we <= '1';
  sdi.ctrl <= to_scr(ctrl_i.alu_sprw);
  -- sdi.ctrl <= sprw_reg;
  sdi.bpv <= (others => '0');
  sdi.bp <= "00";

  sprw_control: process(rstn_i, clk_i)
  begin
    if (rstn_i = '0') then
        sprwctl.sreg <= (others => '0');
    elsif rising_edge(clk_i) then
        if (control.busy = '0') and
            (start_i = '1') and
            (control.rtype = r3type_c) then
          sprwctl.sreg(0) <= '1';
        else
          sprwctl.sreg(0) <= '0';
        end if;
        sprwctl.sreg(sprwctl.sreg'left downto 1) <= sprwctl.sreg(sprwctl.sreg'left - 1 downto 0);
    end if;
  end process sprw_control;

  sprwctl.done <= sprwctl.sreg(sprwctl.sreg'left);


  -- Output select ---------------------------------------------------------------------------
      -- ----------------------------------------------------------------------------------------

  out_select: process(control, sprwctl, sdo)
  begin
    case control.rtype is

```

```
      -- --------------------------------------------------------
      when r3type_c => -- R3-type instructions
      -- --------------------------------------------------------
        control.result <= to_stdulogicvector(sdo.result);
        control.done   <= sprwctl.done; -- iterative, wait for unit ↩
            to finish

      -- --------------------------------------------------------
      when others => -- undefined
      -- --------------------------------------------------------

        control.result <= (others => '0');
        control.done   <= '0';

    end case;
  end process out_select;


end neorv32_cpu_cp_cfu_rtl;
```

neorv32_package.vhd

```
  -- sparrow CSR --
  constant csr_sprw_c           : std_ulogic_vector(11 downto 0) := ↩
      x"800";

  -- Main CPU Control Bus ↩
     ------------------------------------------------------------------- ↩

  -- ↩
     ------------------------------------------------------------------------------

  type ctrl_bus_t is record
    -- register file --
    rf_wb_en      : std_ulogic; -- write back enable
    rf_rs1        : std_ulogic_vector(04 downto 0); -- source ↩
       register 1 address
    rf_rs2        : std_ulogic_vector(04 downto 0); -- source ↩
       register 2 address
    rf_rs3        : std_ulogic_vector(04 downto 0); -- source ↩
       register 3 address
    rf_rd         : std_ulogic_vector(04 downto 0); -- destination ↩
       register address
    rf_mux        : std_ulogic_vector(01 downto 0); -- input source ↩
        select
    rf_zero_we    : std_ulogic;                     -- allow/force ↩
       write access to x0
    -- alu --
    alu_op        : std_ulogic_vector(02 downto 0); -- ALU  ↩
       operation select
    alu_opa_mux   : std_ulogic;                     -- operand A  ↩
       select (0=rs1, 1=PC)
    alu_opb_mux   : std_ulogic;                     -- operand B  ↩
       select (0=rs2, 1=IMM)
    alu_unsigned  : std_ulogic;                     -- is unsigned  ↩
       ALU operation
    alu_frm       : std_ulogic_vector(02 downto 0); -- FPU rounding ↩
        mode
    alu_cp_trig   : std_ulogic_vector(05 downto 0); -- co-processor ↩
        trigger (one-hot)
    alu_sprw      : std_ulogic_vector(XLEN-1 downto 0);
    -- bus interface --
    bus_req_rd    : std_ulogic;                     -- trigger  ↩
       memory read request
    bus_req_wr    : std_ulogic;                     -- trigger  ↩
       memory write request
    bus_mo_we     : std_ulogic;                     -- memory  ↩
       address and data output register write enable
    bus_fence     : std_ulogic;                     -- fence  ↩
       operation
    bus_fencei    : std_ulogic;                     -- fence.i  ↩
       operation
    bus_priv      : std_ulogic;                     -- effective  ↩
       privilege level for load/store
    -- instruction word --
    ir_funct3     : std_ulogic_vector(02 downto 0); -- funct3 bit  ↩
       field
    ir_funct12    : std_ulogic_vector(11 downto 0); -- funct12 bit  ↩
       field
    ir_opcode     : std_ulogic_vector(06 downto 0); -- opcode bit  ↩
       field
    -- cpu status --
    cpu_priv      : std_ulogic;                     -- effective  ↩
       privilege mode
    cpu_sleep     : std_ulogic;                     -- set when CPU ↩
        is in sleep mode
    cpu_trap      : std_ulogic;                     -- set when CPU ↩
        is entering trap exec
    cpu_debug     : std_ulogic;                     -- set when CPU ↩
        is in debug mode
  end record;
```

neorv32_cpu_csr.h

```
  /* sparrow unit control */
  CSR_SPRW          = 0x800, /**< 0x800 - sprw:   Sparrow control ↩
     register */
```

neorv32_sparrow.h

```
#ifndef neorv32_sprw_h
#define neorv32_sprw_h

#ifdef __cplusplus
extern "C" {
#endif

#define SPRW_DEFAULT (0x005C9C9F)
#define SPRW_HP_MSK (0x08000000)
#define SPRW_OD_MSK (0x07800000)
#define SPRW_OL_MSK (0x00600000)
#define SPRW_SB_MSK (0x001FE000)
#define SPRW_SA_MSK (0x00001FE0)
#define SPRW_MS_MSK (0x00000010)
#define SPRW_MK_MSK (0x0000000F)

struct swizz_lane_t {
  uint8_t  a : 2;
  uint8_t  b : 2;
  uint8_t  c : 2;
  uint8_t  d : 2;
};

union swizz_t {
  struct swizz_lane_t elem;
  uint8_t vector;
};

struct mask_lane_t {
  uint8_t a : 1;
  uint8_t b : 1;
  uint8_t c : 1;
  uint8_t d : 1;
};

union mask_t {
  struct mask_lane_t elem;
  uint8_t vector : 4;
};

/**********************************************************************//* ↩
```

```
 * Reset the SPARROW control register to default settings.
 *
 * Default: no swilling, mask with masking vector 1111, ol 10 and
 * no other settings enabled.
 *
 **********************************************************************/ ↩

inline void __attribute__ ((always_inline)) sprw_ctrl_reset() {
  asm volatile ("csrw %[reg], %[val]":: [reg] "i" (CSR_SPRW), [val] ↩
      "r" (SPRW_DEFAULT) );
}

/*********************************************************************//* ↩

 * Set ms to 1.
 *
 **********************************************************************/ ↩

inline void __attribute__ ((always_inline)) sprw_ctrl_ms_set() {
  asm volatile ("csrs %[reg], %[val]":: [reg] "i" (CSR_SPRW), [val] ↩
      "r" (SPRW_MS_MSK) );
}

/*********************************************************************//* ↩

 * Clear ms to 0.
 *
 **********************************************************************/ ↩

inline void __attribute__ ((always_inline)) sprw_ctrl_ms_clear() {
  asm volatile ("csrc %[reg], %[val]":: [reg] "i" (CSR_SPRW), [val] ↩
      "r" (SPRW_MS_MSK) );
}

/*********************************************************************//* ↩

 * Set mk.
 *
 * @param[in] mk Masking vector for mk (union mask_t).
 **********************************************************************/ ↩

inline void __attribute__ ((always_inline)) sprw_ctrl_mk_set(union  ↩
   mask_t mk) {
  uint32_t mask = mk.vector;
  asm volatile ("csrc %[reg], %[val]":: [reg] "i" (CSR_SPRW), [val] ↩
      "r" (SPRW_MK_MSK) );
  asm volatile ("csrs %[reg], %[val]":: [reg] "i" (CSR_SPRW), [val] "r" (mask) );
}

/*******************************************************************//*
 * Set mk.
 *
 * @param[in] mask Masking vecotor for mk (uint32_t, 4 LSB).
 ********************************************************************/
/*
inline void __attribute__ ((always_inline)) sprw_ctrl_mk_set(uint32_t mask) {
  asm volatile ("csrc %[reg], %[val]":: [reg] "i" (CSR_SPRW), [val] "r" (SPRW_MK_MSK) );
  asm volatile ("csrs %[reg], %[val]":: [reg] "i" (CSR_SPRW), [val] "r" (mask) );
}
*/
/*******************************************************************//*
 * Clear mk to 1111.
 *
 ********************************************************************/
inline void __attribute__ ((always_inline)) sprw_ctrl_mk_clear() {
  asm volatile ("csrs %[reg], %[val]":: [reg] "i" (CSR_SPRW), [val] "r" (SPRW_MK_MSK) );
}


/*******************************************************************//*
 * Set sa.
 *
 * @param[in] sa Masking vector for sa (union swizz_t).
 ********************************************************************/
inline void __attribute__ ((always_inline)) sprw_ctrl_sa_set(union swizz_t sa) {
  uint32_t mask = sa.vector << 5;
  asm volatile ("csrc %[reg], %[val]":: [reg] "i" (CSR_SPRW), [val] "r" (SPRW_SA_MSK) );
  asm volatile ("csrs %[reg], %[val]":: [reg] "i" (CSR_SPRW), [val] "r" (mask) );
}
```

```

/*******************************************************************//*
 * Set sa.
 *
 * @param[in] mask Masking vecotor for sa (uint32_t, 8 LSB).
 ********************************************************************/
/*
inline void __attribute__ ((always_inline)) sprw_ctrl_sa_set(uint32_t mask) {
   mask = mask << 5;
  asm volatile ("csrc %[reg], %[val]":: [reg] "i" (CSR_SPRW), [val] "r" (SPRW_SA_MSK) );
  asm volatile ("csrs %[reg], %[val]":: [reg] "i" (CSR_SPRW), [val] "r" (mask) );
}
*/

/*******************************************************************//*
 * Clear sa to (0, 1, 2, 3).
 *
 ********************************************************************/
inline void __attribute__ ((always_inline)) sprw_ctrl_sa_clear() {
  uint32_t mask = 0b11100100 << 5;
  asm volatile ("csrc %[reg], %[val]":: [reg] "i" (CSR_SPRW), [val] "r" (SPRW_SA_MSK) );
  asm volatile ("csrs %[reg], %[val]":: [reg] "i" (CSR_SPRW), [val] "r" (mask) );
}

/*******************************************************************//*
 * Set sb.
 *
 * @param[in] sa Masking vector for sb (union swizz_t).
 ********************************************************************/
inline void __attribute__ ((always_inline)) sprw_ctrl_sb_set(union swizz_t sb) {
  uint32_t mask = sb.vector << 13;
  asm volatile ("csrc %[reg], %[val]":: [reg] "i" (CSR_SPRW), [val] "r" (SPRW_SB_MSK) );
  asm volatile ("csrs %[reg], %[val]":: [reg] "i" (CSR_SPRW), [val] "r" (mask) );
}

/*******************************************************************//* ↩

 * Set sb.
 *
 * @param[in] mask Masking vecotor for sb (uint32_t, 8 LSB).
 *******************************************************************/ ↩

/*
inline void __attribute__ ((always_inline)) sprw_ctrl_sb_set( ↩
   uint32_t mask) {
   mask = mask << 13;
  asm volatile ("csrc %[reg], %[val]":: [reg] "i" (CSR_SPRW), [val] ↩
      "r" (SPRW_SB_MSK) );
  asm volatile ("csrs %[reg], %[val]":: [reg] "i" (CSR_SPRW), [val] ↩
      "r" (mask) );
}
*/

/*******************************************************************//* ↩

 * Clear sb to (0, 1, 2, 3).
 *
 *******************************************************************/ ↩

inline void __attribute__ ((always_inline)) sprw_ctrl_sb_clear() {
  uint32_t mask = 0b11100100 << 13;
  asm volatile ("csrc %[reg], %[val]":: [reg] "i" (CSR_SPRW), [val] ↩
      "r" (SPRW_SB_MSK) );
  asm volatile ("csrs %[reg], %[val]":: [reg] "i" (CSR_SPRW), [val] ↩
      "r" (mask) );
}


#ifdef __cplusplus
}
#endif
#endif // neorv32_sprw_h
```