Implementación de una plataforma para tests de inyección de fallos mediante electromagnetismo contra SoCs basados en RISC-V

Por
Pedro Javier Fernández Fernández

UNIVERSIDAD COMPLUTENSE
MADRID

Grado en Ingeniería Informática (Bilingüe)
Facultad de Informática

Dirigido por
Juan Antonio Clemente, Juan Carlos Fabero

Implementation of an electromagnetic fault injection platform for a RISCV-based SoC

Madrid, Mayo 2022

Resumen

El mercado de los microcontroladores, CPUs, ordenadores de escritorio y servidores ha alcanzado nuevas cotas y superado numerosos retos técnicos durante la última década. Con la aparición del conjunto de instrucciones RISC-V en 2010, llegó un nuevo mundo de posibilidades y libertades. Sin embargo, la necesidad creciente de ordenadores seguros y confiables también ha aumentado, tanto de cara al consumidor, como en otras partes de la industria. En numerosas ocasiones, los componentes hardware son los grandes olvidados a la hora de evaluar la seguridad de un sistema, debido a razones tales como la dificultad de acceder o manipular estos componentes, o el coste prohibitivo que conlleva modificar e investigar dichas partes. En este trabajo, se plantea la pregunta: «¿Cómo de bien resiste la arquitectura RISC-V frente a peligros físicos?». Para evaluar posibles respuestas, se desarrolla un dispositivo nóvel capaz de llevar a cabo ataques de inyección de fallos mediante electromagnetismo, con énfasis en obtener un dispositivo cuya fabricación sea asequible.

Palabras clave: Inyección de fallos, Single Event Upset, electromagnetismo, ataques de glitcheo, seguridad de la arquitectura, explotación de fallos, hardware.

Abstract

The market of microcontrollers, CPUs, desktop and server computers has seen both numerous milestones achieved and new challenges arise in the last decade. With the RISC-V ISA being introduced in 2010, a new set of possibilities and freedoms was unlocked. However, the overall necessity for security and resilient computers has increased, not only for consumer grade devices, but also for every other field. Hardware is oftentimes one of the most forgotten attack surfaces, due to several reasons like lack of ease-of-access, or the cost of research. In this document, we ask the question: "how well does the RISC-V architecture stand against physical harms?". We also develop a novel device capable of doing Electromagnetic Fault Injection attacks while being a very affordable solution to build.

Keywords: Fault Injection, Single Event Upset, electromagnetism, glitch attack, architectural security, exploit, hardware.

Contents

Chapter 1

Introduction, motivation and contributions

For a long time, the hardware security realm has received significantly less public attention than the software counterpart, due to the technical difficulties and expenses derived from hardware experimentation and research. Most modern-day devices, like smartphones, computers, Internet of Things (IoT), infamous crypto-wallets, and even industrial systems, are exposed to physical hazards, either from the environment they work on, or from malicious attackers trying to obtain unfair advantages through the device manipulation.

However, at the same time this difficulty to secure the hardware of our devices has increased, lots of new technological innovations have been recently proposed. The rise of open-source software has permeated into the developer culture and has extended to the hardware, electronics, Central Processing Unit (CPU) design, Field Programmable Gate Arrays (FPGAs), and (Register-Transfer Level) RTL fields. The appearance of the RISC-V Instruction Set Architecture (ISA) in 2010 was one of many significant milestones, whose success is being perceived nowadays.

The question this research aims to answer is: do these technological advances constitute an improvement in the fight against the increasing hardware challenges? While some of these challenges have been known for a long time (such as electromagnetic fields and radiation affecting circuits in space computers), now the world is seeing a rise of malicious attacks to everyday devices based on the same principles.

This work has 2 primary objectives: The first is to evaluate the reliability against EMFI attacks of several Systems-on-Chip (SoCs) that implement different versions of the previously mentioned RISC-V ISA. The second, to develop an economically affordable EMFI device suited for academia research. Both contributions will be developed in the subsequent chapters of this document.

An affordable electromagnetic fault injection device suited for academia research is going to be built, and used to explore devices that are based on different versions and implementations of the previously mentioned RISC-V ISA.

Thus, Chapter 2 explores the state-of-the-art techniques and experiments in Fault Injection, and introduces important terminology. Chapter 3 introduces the two EMFI devices

that have been designed, while Chapter 4 presents the RISC-V devices used for experiments in this work, and reviews RISC-V ISA concepts. Chapter 5 showcases the results of EMFI experiments performed with the presented RISC-V based devices, and Chapter 6 explores future research and innovations that can be derived of this work.

This work was financially supported in part by the Spanish Ministry of Science and Innovation (MICINN) under grant "PID2020-112916GB-I00". This project is conducted by the *grupo en Gestión del Hardware Dinámicamente Reconfigurable* research group, at the UCM Computer Science faculty (Facultad de Informática).

Chapter 2

State of the art

In this chapter, the most modern and effective fault injection methodologies, research and devices are reviewed. Focus is placed on the electromagnetic fault injection techniques, which are of the uttermost interest.

2.1 Fault Injection

The use of controlled fault injection experiments has been a persistent subject in academia for the last two decades. The objective of these tests may be one of the following:

- Determine how resilient a system is against real spontaneous faults.
- Stress-test the System Under Evaluation (SUE).
- Change normal program behaviour into a path that usually would not be taken.
- Test the physical resistance of hardware components.
- Introduce non-persistent (temporary) vulnerabilities into secure hardware or secure software.

In software solutions, fault injection is performed by introducing unexpected/malformed datum into the input streams of a program, and afterwards, probing both the internal status and the outputs of the program, to observe for indicators of problems.

When it comes to hardware, the methodologies change significantly. Providing incorrect inputs is no longer the only way to proceed. The inputs might be correct or expected information, similar to what the system may receive during normal operation. Instead, physical parameters of the SUE are altered to influence and inject faults into the internal state of the system. This internal state is then observed and compared against a known correct state. The outputs of the system can also be monitored for errors, like it is done with software.

This is possible because the internal status of a hardware device is stored in components like latches, flip-flops, and memories. These may be susceptible in varying degrees to physical changes, depending on the technology they were built with. Different technologies could be classified, for example, by the properties of the transistors used. For instance, transistors' size (nm), their manufacturing material (silicon, germanium, etc.),

or technology (FET, MOSFET, BJT). Not only do transistors' properties affect the reliability against physical variations, but also the properties of other components affect the overall design (like capacitors, which are a vital part of DRAM memories). For these components, parameters to consider might be the capacitor's internal dielectric material being used, its size, or its capacitance value, as parameters that might influence how susceptible to faults the component is. Any component that is involved in maintaining information of the system might be influenced in one way or another.

The existence of faults in one or more of these electronic components can cause a change in the perceived internal status of the hardware device.

Not only is the internal status affected by physical changes, but also the *combinational* pieces of logic in a circuit. These are built on top of logic gates, which are manufactured using transistors. Thus, a fault can occur in a part of a circuit that does not store the machine's state, but rather that computes the next state of the machine, or that computes the output, or in many other relevant information paths of the system. Clock signals and their paths can also be disrupted by a fault,

Examples of physical parameters that can be altered to produce faults in a hardware device comprehend:

- Temperature
- Clock speed
- Operating voltage of the circuit

These terms are oftentimes used in conjunction with the term *glitching*, rather than fault injection, and other research may refer to techniques like clock glitching, voltage glitching, and so on.

Fault Injection attacks have repeatedly been used to defeat the security of commercial products. For instance, in 2015, several researchers [1] independently obtained *boot ROM level* unsigned code execution in the popular Nintendo 3DS family of consoles through a glitching attack, thus allowing for complete takeover of the system. Later, other researchers [2] leveraged the obtained knowledge to find software-level vulnerabilities in the boot ROM. Other critical devices were targeted at the popular CCC (Chaos Computer Club) conference in Europe, where a team of security researchers defeated protection measures of various popular *crypto wallets* in the market [3]. Similarly, in 2019 researcher YifanLu [4] injected software vulnerabilities into the Sony's PlayStation Vita through a controlled and accurate voltage fault injection attack. The same year, Gauvain Roussel et al. [5], defeated the security in the Nvidia TEGRA x1, the security processor in the Nintendo Switch console released two years before. They used a hardware fault injection attack vector to extract the CPU's secrets. Later, in 2020, researchers Ákos Hajdu et al. were able to defeat the security of smart contracts in hardware wallets through means of fault injection [6]. The following year, researchers Robert Buhren, Hans Niklas, Thilo Krachenfels and Jean-Pierre Seifert were able to defeat the security mechanisms of AMD's Secure Processor (AMD-SP, formerly known as PSP) [7] just using voltage fault injection, compromising AMD's Secure Encrypted Virtualization (SEV) technology.

In the security industry, most advanced solutions and equipment for Fault Injection experiments are developed and provided by companies. For example, the company NewAE

Technology Inc. is known for offering a range of products that stand out, like the *ChipWhisperer* family of devices, which are able to introduce operating voltage glitches through the use of crowbar circuits and fast transistors. For illustrative purposes, this was showcased in Colin O'Flynn's paper [8] which presented and showed this device in action, attacking embedded devices and causing the targets to execute instructions incorrectly or to store faulty data. He was able to obtain high reproduction rate of the fault effects, both with microcontrollers and FPGAs. Higher end solutions include the *ChipShouter*, a device capable of generating controlled and position-accurate electromagnetic pulses, to create currents in the target device through induction. It includes several safety measures to deal with the high voltages, and it is a costly device (+3000€)[1] suited for industrial and governmental research.

The involved physical phenomenons will be briefly reviewed in the next chapters as we develop a device that achieves the same results but operates in a much simpler fashion and is cheaper, thus being suited for academic scenarios.

2.2 Electromagnetic Fault Injection (EMFI)

For each of the physical parameters mentioned earlier, there exist several ways to alter or modify them. Some of them are semi-permanent or permanent, altering the original circuit or board, for example, replacing the clock source of the system with an attacker controlled clock source. This could involve soldering and replacing components or cables on the boards. Although in some cases it would be possible to de-solder and undo any modifications, the board would unlikely be in the same state as at the beginning, and the warranty of the device will be voided. Others solutions are non-permanent, like using a laser to inject controlled charges into the on-chip networks of the Device Under Test (DUT), or using electromagnetic fields to induce a charge into the power or data rails of the device. The latter is the technique that will be explored more in depth, Electromagnetic Fault Injection (EMFI). Most of these techniques can be permanently damaging to the DUT if executed incorrectly.

In short, this technique consists in the use of attacker-controlled electromagnetic fields generated in the area in which the device under test is located. The fault injection is produced not by the existence of the electromagnetic field itself, but rather by the electromotive forces (voltage) that are generated in the conductor materials that the chip is made of. If the conductor material is part of a closed circuit, then the voltage flows, and it can be called a *current* moving through the circuit. These unexpected currents in parts of the chip or in electrical paths where they should not be, are what are considered as faults in the DUT, and are the culprit of altering the internal status of the whole system or of many of its individual components.

Since the time work started on this project, a plethora of new research papers have been published regarding fault injection on RISC-V targets, with some of these works employing EMFI devices specifically, rather than other techniques like voltage fault injection.

Researches from the Department of Electrical and Computer Engineering University of Waterloo used an EMFI platform [9] to attack RISC-V and ARM Cortex-M0 CPUs,

[1] https://store.newae.com/chipshouter-kit/

demonstrating instruction skipping with high reproduction rate. They employed both JTAG and custom fault handlers to extract state information from the DUTs.

At the RTL and simulation level, research by Johan Laurent et al. [10] showed how the microarchitecture is relevant when evaluating the faults and countermeasures that can be implemented in a RISC-V core. The popular Rocket Processor core was selected for the research, as its RTL source code can be analysed and studied.

Similarly, in 2020 student Mahmoud A. Elmohr explored electromagnetic fault injection in RISC-V architecture cores, with an emphasis in FPGA designs and implementations [11]. Furthermore, papers exploring the security concerns of fault injection in RISC-V systems reveal the concerns and possibilities of malicious attackers employing these techniques to bypass security measures. Shoei Nashimoto et al. [12] were able to bypass the memory protections of popular RISC-V Trusted Execution Environments (TEE) like *Keystone* and *MultiZone (Hex Five)* by targetting specific instructions.

2.3 Devices to measure EM fields

Although not the main objective of this work, having the means to measure the actual strength of our device would be a valuable addition to the knowledge of the EMFI set-up. This way, it would be possible to empirically measure the strength of a given EMFI device, and create approximate comparisons between it and different devices or implementations. This could be important, as the strength of the device is directly related to the appearance of faults, as explored in later chapters.

For instance, in 2021 the hacker known as *PoroCYon* managed to dump the Nintendo DSi's protected boot region through an electromagnetic fault injection attack [13]. In the initial setup, an expensive, commercial product for EMFI experiments was employed. However, as the author narrates, the results were not good, no faults were observed, and the experiment failed, because the strength generated by the device was not sufficient to provoke any fault on the system. However, when employing a custom research device developed at their university, the results were successful.

Real cases like these show it might be important to understand not only the fault injection technique being used, and whereas it is effective in disrupting the DUT, but also the equipment and its inner design, as well as how to fine-tune it. In the particular case of EMFI, one of the many parameters for success is the strength of the electromagnetic fields generated by the injector. For this reason, a few solutions were developed and tested to try to measure and represent the strength of the electromagnetic field generated by our proposed EMFI platform in a given area of space.

Initially, hall-effect based sensors were explored to measure the magnetic component of the fields and two prototypes were assembled and software was developed, but these turned out to be ineffective in detecting the electromagnetic fields generated by the proposed device.

2.3.1 Passive electromagnetic receiver

A different device was built aiming at detecting the electromagnetic field generated by the device. It consisted of a basic passive electronic circuit, capable of detecting radio

signals in the [1.8 , 30] MHz range. This circuit was designed, built and tested by José J. Barceló Sarria[2] who published the circuit schematic shown in Fig.2.2 with build instructions for the device.

Figure 2.1: Implementation of the passive electromagnetic receiver.

This device works as a very generic electromagnetic field detector, and the frequencies it can work with are highly dependent on the type of antenna used, similarly to popular hobbyist AM crystal radio receivers[3]. A set of diodes and capacitors rectify the Alternate Current (AC) induced in the antenna, to a Direct Current (DC) signal, which is then fed into a micro-amperimeter, to provide an approximate visual representation of the intensity of the signal perceived. This is by no means calibrated laboratory equipment, but rather a simplistic field tool usable to determine the existence or not of certain electromagnetic noise.

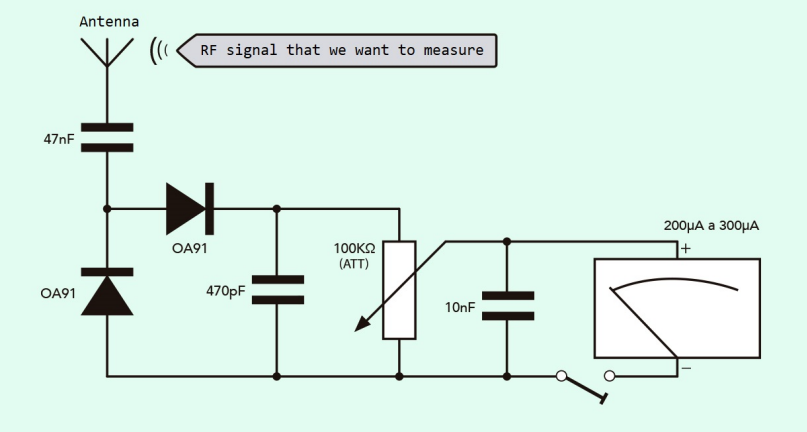


Figure 2.2: Diagram of the electronic circuit. Source: https://electroclinica.org/2018/03/23/fabricacion-de-un-sencillo-medidor-de-campo-de-rf-diy/

[2]https://electroclinica.org/2018/03/23/fabricacion-de-un-sencillo-medidor-de-campo-de-rf-diy/

[3]https://electronics.stackexchange.com/questions/616496/is-this-circuit-measuring-the-electrical-field/616497616497

Unfortunately, neither this circuitry was able to detect the electromagnetic fields generated by the two devices proposed in the next chapter, probably due to the high-frequency oscillating fields generated, but also due to the lesser amounts of energy that these injectors employ, if compared to a radio emitter. Our portable implementation of the device is shown in Fig. 2.1.

The induced current in the antenna is not big enough to overcome the diode's forward bias threshold. An improvement to the circuit and solution to this issue, would be to add an amplifier component after the antenna and before the diodes. This receiver as-is, may be more suited for traditional, low-frequency radio signals.

Chapter 3

The proposed EMFI attacker device

To conduct EMFI experiments and evaluate the security and reliability implications, an appropriate hardware set-up that can be used to reproduce the experiments and their conditions is required. The implementation of this device is discussed in this chapter. Following this development, aiming to obtain some preliminary results with the device and simultaneously aid to their research, I teamed up with Mohammadreza Rezaei, a PhD student at the UCM Computer Science faculty, to test the system on several COTS SRAM memories.

3.1 Implementation details

Aiming to perform the experiments, an EMFI device was created in order to generate the disruptions explained earlier. The bottom line is that a high voltage needs to be generated and routed through a set of spires, a coil. This will generate an electromagnetic field. The field will produce the induction phenomena on the conductor elements in its range of movement. A current will be induced on such conductors, disrupting their normal operation conditions. Moreover, a key point in its design was to produce a cheap and easy-to-replicate device that can be set up with minimal materials. This is in contrast to commercial solutions, which are often sold for thousands of dollars [7].

In this work, two ways to obtain a working EMFI device are explored. Firstly, beginning by simply modifying and 'hacking' a cheap, widespread mosquito killer racket, to turn it into an inexpensive EMFI device for academic research. Secondly, a completely controllable EMFI device is designed from the schematics, equally suited for academic research, but highly customizable, repairable, and able to produce more repeatable results.

To do so, a novel method is proposed, which consists in re-purposing an existing device which contains most of the components and circuitry needed to generate the required voltage levels.

Specifically, a kind of circuit called "*Mosquito Zapper*" was selected. This type of circuit is often found in anti-insect electrical devices, which are used to prevent pests and other sanitary problems. An example can be seen in Fig. 3.1. Although there are many variations of the circuit, they all follow the same idea. Firstly, one of these devices was acquired from a local store, and then it was disassembled. Fig. 3.2 shows circuitry inside the Mosquito Zapper device that was disassembled.

Figure 3.1: Example of commercial high-voltage mosquito zapper.

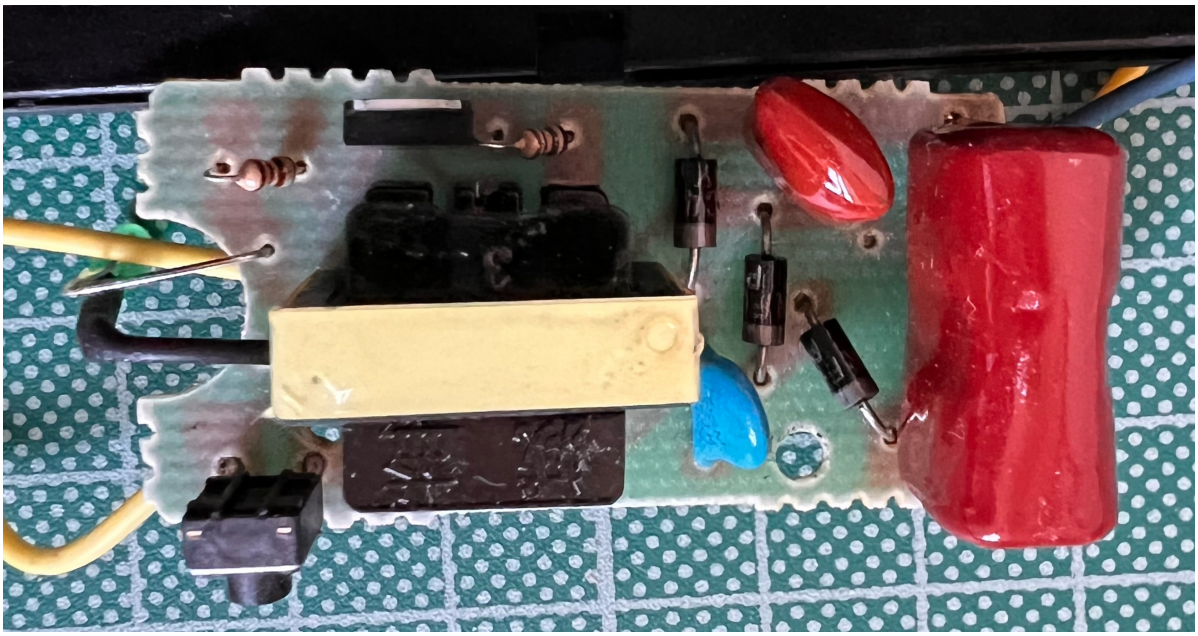


Figure 3.2: Printed Circuit Board (PCB) of the commercial zapper device.

From the observation of this circuit, schematics were reverse-engineered and drawn. This made possible to develop a custom device, the possibility to understand all the components and stages required to generate an electromagnetic field with simple electronics, and manipulate parameters of such generated electromagnetic field.

The schematic depicted in Fig. 3.3 accurately represents the circuit in one of the devices that was acquired. It can be divided into three functional blocks. From left to right, firstly the user input (push button) and the power supply (batteries) can be observed. When the user presses the button, current starts flowing and a green LED diode (D1) indicates this. The current then flows into the transformer, and its feedback output to the MOSFET transistor. Said circuit behaves as an oscillator or self-oscillator, by switching on and off repeatedly. This converts the Direct Current (DC) produced by the batteries into Alternate Current (AC). This is important, as the high-voltage transformer will only function with AC, not DC. The transformer itself relies on magnetic fields and induction to produce a high-voltage AC from the low-voltage AC on its input. The right-side part

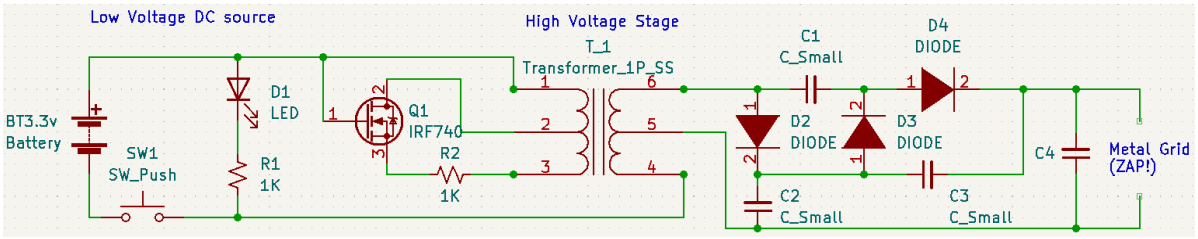


Figure 3.3: Reverse-engineered circuit schematic in KiCad.

of the circuit is a voltage rectifier, which converts the AC back to DC. It is combined with several capacitors (C1,C2,C3) to store and increase the voltage even more, likely to an order of kiloVolts. This charge is stored in a final capacitor (C4) and connected to the metal grid that is used to *zap* mosquitoes and other bugs.

3.1.1 The mosquito EMFI injector

Once the original circuit is understood, it becomes trivial to modify it to turn it into an EMFI device. The bottom-line is reusing all the existing electronic circuitry that generates a high-voltage current. However, instead of connecting this current to a metal grid like the one depicted in Fig. 3.1 and in the circuit in Fig. 3.3, it will be passed through a spire, all at once, thus generating a strong electromagnetic field. The required modifications to the base circuit can be reproduced as follows:

Manufacturing of a spire to create the electromagnetic field.

The most significant change is the replacement of the metal grid by a spire (observed in Fig. 3.4), which will be used to generate the electromagnetic field, causing induction in nearby conductors as the field increases and decreases. This is insufficient to provoke a

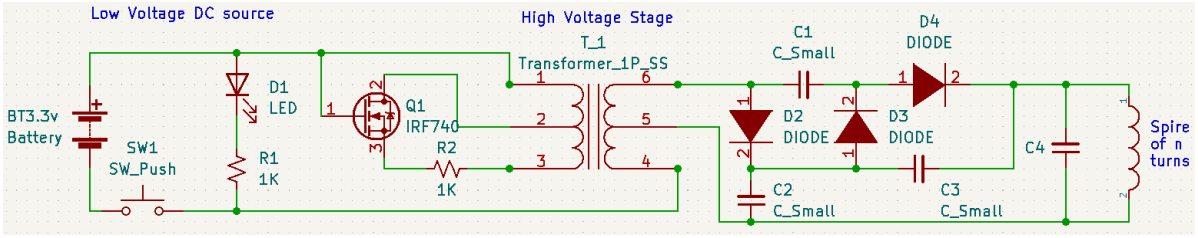


Figure 3.4: The mosquito-killer grid is replaced by a spire.

strong and moving field because the capacitor barely has time to accumulate energy, as its energy is being drained while it is being charged, in a continuous manner. Instead, it is desired to charge the big C4 capacitor as much as possible, and discharge it at all once. The capacitance is the measure of a capacitor's ability to store an electrical charge onto its plates. In Fig. 3.2 observe capacitor C4 located on the right-hand side of the circuit board, of a size considerably larger than the other capacitors.

Introduction of a spark gap to fire the high voltage circuit when charged.

The most affordable and straightforward solution to considerably charge the capacitor is to introduce a spark gap. This is an air gap in the circuit, through which current can

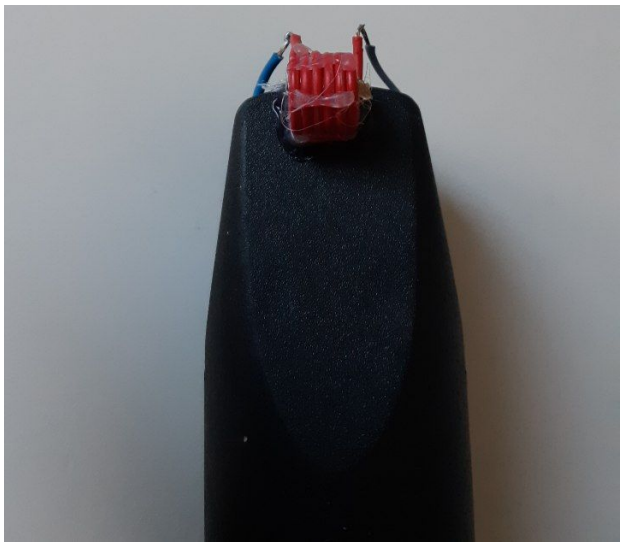


Figure 3.5: Homemade 6-turn spire installed into the modified mosquito device.

only pass when it is high enough to break the air's electrical resistance. This phenomenon is what produces a visible spark from one terminal of the gap to the other. For this to happen, the charge on the capacitor has to be sufficiently high.

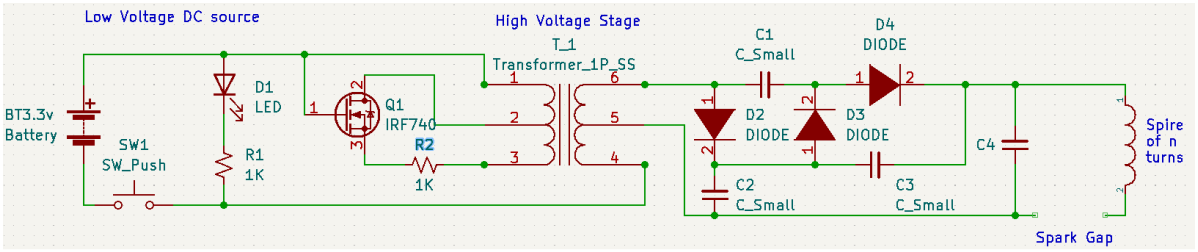


Figure 3.6: A spark gap is added to force the discharge of the capacitor C4 all at once.

With this simple modification, all the charge in the capacitor will be released at once, once enough charge was accumulated to break the air's resistance. This simultaneously ensures that the current going through the spire is high enough to produce a significant field during a brief period of time.

Empirically, as it will be showcased in point 3.3, it is observed that this setup is enough to produce noticeable faults. However, it is interesting to compare the experimental results to a theoretical framework. The spire introduced into the circuit is a solenoid, and the magnetic field generated by this kind of structure has some peculiarity. The greater the longitude of the solenoid, the closer to zero the magnetic field outside the solenoid is. In a supposedly infinite solenoid, the field outside the structure would be zero, as the fields generated from each of the solenoid coil turns cancel each-other out. This topic is extensively discussed in Aritro Pathak argument [14]. As can be observed in Fig. 3.4 the proposed coil is far from being an infinite solenoid, and magnetic field lines escape the structure, affecting surrounding objects. There is no magnetic field without an existing electrical field, which is also present.

(Optional) Adding a discharge resistor to safely discharge the circuit.

Optionally, it is possible to include additional modifications to increase the safety of the circuit. Although the high voltages produced by the modified device are not lethal to humans, it is recommended to add a discharge resistor so that any remaining energy in the capacitor naturally discharges after the device has been fired or partially charged. This modification is observed in Fig. 3.7.

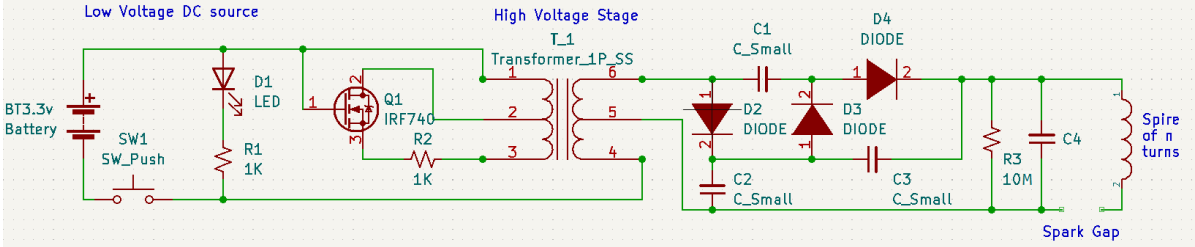


Figure 3.7: A resistor (R3) can be added to slowly discharge capacitor C4 over time.

3.1.2 cheapSHOUTER: advanced injector from scratch

Although correct, the reverse-engineered schematic above is a not complete, functional description of the circuit, as it was not possible to find the specific part numbers for several components like the capacitors or the transformer. Designing our own circuitry that can be actually used to manufacture and test a prototype, proved to be more complex than expected. It was not possible to just clone or completely derive the new design from the reverse-engineered one, although in general terms it is divided in the same three stages that the reverse-engineered circuit had: firstly, user input and oscillator followed by a step-up transformer, next a voltage rectification and step-up circuit composed of diodes and capacitors, and finally a bigger capacitor to store the charge connected to a discharge system.

Additionally, the schematics include circuitry to measure the charge status in the last capacitor, although this was not implemented in the first prototype.

The process of putting together a design started by choosing a high-voltage transformer to be the keystone of the design. Once a transformer model was chosen, the rest of the circuit was designed around it. Initially, both 749196501 [15] and 749196118 [16] models from *Wurth Elektronik* were selected and ordered, as these were one of the few widely available on retail suppliers at the moment, and they worked with high voltages. Unfortunately, these models turned out to not be chosen correctly nor useful, as their primary coil to secondary coil turn ratio was 1 to 1, thus acting as a filter, and not achieving the voltage step-up effect that was desired.

Other components were not immediately available due to the current semiconductor shortage that is nowadays affecting the world, but the following could be sourced. The MOSFET transistor chosen for the oscillator part of the circuit was the IRFZ44ZPBF. The circuit uses 1K resistors, diodes (of which part number UF5408 were used), and capacitors (MPP-39nr15) to rectify the AC into DC before sending it to a high-capacity 1000v capacitor (ECW-H10753JVB) to store the electric charge, that is then rapidly discharged over a coil producing the electromagnetic field. Fig. 3.8 depicts the schematic of the final

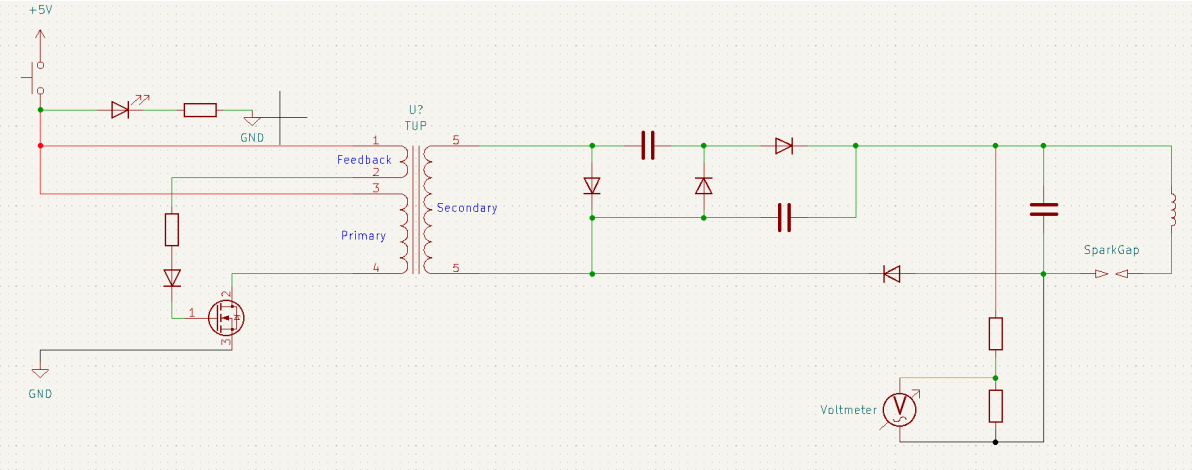


Figure 3.8: Schematic of the circuit of the proposed device, designed in KiCad.

design, which was attempted to be produced into a working prototype thorugh the use of proto-boards.

This schematic can be obtained attached to this memory, as a project file for the popular open source and free Electronic Design Automation (EDA) software, KiCad 6.0. The complete list of components and their part numbers can be consulted in the Bill Of Materials (BOM) .xls file. The device was named *cheapSHOUTER* as a funny allusion in contrast to the more expensive and professional *chipSHOUTER* device mentioned in earlier chapters.

Ultimately, it was only possible to completely assemble and verify the complete design, with a different transformer. This was because erroneous transformers were ordered, which did not produce sufficiently high voltage on their output. However, verification of the rest of the circuit was successful. A first attempt to solve this was winding our own transformer, as shown in Figs. 3.9 and 3.10. An inductor from a high-voltage low-consumption COTS light bulb was used as a secondary coil with a very high number of turns, and the crafted transformer had a primary coil of 6 turns, and a primary feedback coil of 6 turns, which was empirically shown to be able to output a (not very stable) average output of 200 to 500 V, when measured with a multimeter.

Because this manually winded transformer was not accurately built, it was eventually replaced with a donor transformer from one of the mosquito devices, which helped to verify both the oscillator circuit and the rectifier circuit. The selected capacitor was charged and a strong spark could be generated by connecting both terminals of the capacitor.

To conclude the prototype build, an SMD inductor component was extracted from a recycled motherboard, and used as the coil/probe of the device. This coil is much smaller and has more turns than the manually fabricated one used for the repurposed mosquito injector. Furthermore, these devices use a ferrite core, instead of the air core used in the previous device, which significantly increases the magnetic flux.

Thus, all parts of the circuit, except for the custom transformer, were empirically verified. Because the device was considerably more prototypical and bare-bones, only a few preliminary tests with the cheapSHOUTER were performed, and are described at the

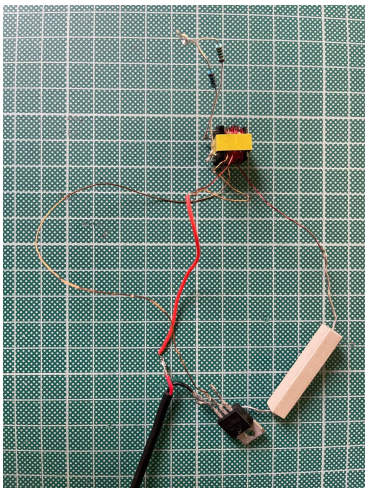


Figure 3.9: Testing circuit.

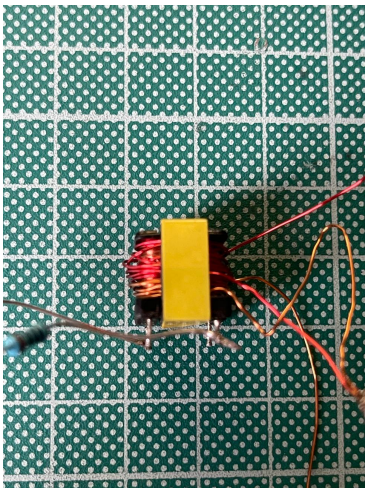


Figure 3.10: Manually winded transformer.

end of the next section.

3.2 Preliminary EMFI results

Using the first injector version which was described in section 3.1.1, the modified mosquito zaper-based device, several tests and experiments were run. It was rapidly evidenced that the device was working, and producing considerable faults on the DUTs. These preliminary experiments were not oriented to be specially reproducible.

In subsection 3.2.1, the preliminary results with memories and the mosquito injector are showcased. Following, section 3.2.2 describes and shows the experiments with microcontroller, using the same injector. Finally, section 3.2.3 describes some basic tests performed with the cheapSHOUTER design prototype hardware.

3.2.1 Preliminary results with memories

COTS SRAM memories were the first target of our newly assembled device. The induction generated by the injector's electromagnetic field, in several experiments, was shown to produce both temporary and permanent faults. A circuit board with a socket, seen in Fig. 3.11 was employed to interface with the SRAM memories from an Arduino board. These memories have a TSOP48 footprint, as depicted in Fig.3.12. To read and write the contents of the memories, a PC host software utility developed by Mohammadreza was used.

Description of the experiments:

Experiments were performed using two different memory components:

Firstly, a brand new CY62167DV30LL-55ZXI SRAM memory [18], manufactured by Infineon Technologies in 130-nm bulk CMOS process, was used for the first tests. This is a 16-Mbit memory, which the manufacturer claims has high-performance transistors, and automatic power-down features that reduce power consumption by 99% if addresses are not being toggled. The models used in this research were the 48-pin TSOP packages, although the same memories are also available in a 48-ball VFBGA package. This SRAM

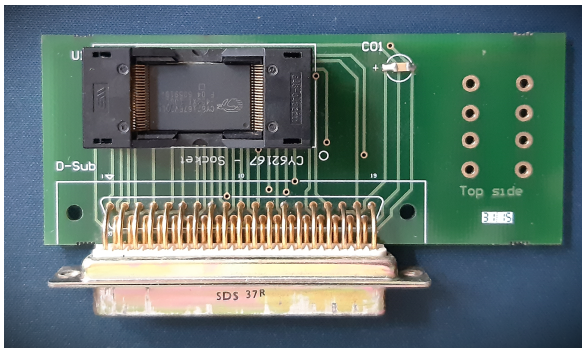


Figure 3.11: SRAM memory socket board used by Mohammadreza.

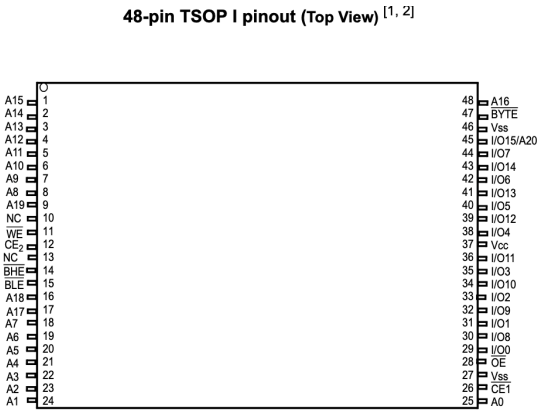


Figure 3.12: SRAM memory layout. Source: Infineon CY62167DV30 datasheet [17].

memory is essentially a parallel memory, where multiple bits can be read at once, unlike in memories with serial protocols (i.e.: i2c, SPI, etc.). The tests run were the following:

- Test 1.1: The memory was initially filled with a static pattern of 0xFF bytes. After the whole memory was written, x1 shot was done from a 5 cm distance away from the SRAM. Afterwards, the memory was read back. No bit-flips were detected.
- Test 1.2: The memory was initially filled with a static pattern of 0xFF bytes. x2 shots were done from a 5 cm distance away from the SRAM, in dynamic fashion: while reading. Single bit-flips appeared exactly when the pulses occurred. We do not attribute it to bit-flips, but to perturbations in the I/O lines which were transporting the read data.
- Test 1.3: The memory was initially filled with a static pattern of 0xFF bytes. Afterwards, in a dynamic fashion, while reading, x1 shot was done from a distance of 1 cm from the SRAM. The software started reading 0x00 bytes since the moment the shot was received. The memory stopped responding, even if turning it off and on again. Apparent physical damage was caused to the chip, and the memory no longer worked. It is believed that this might have been a consequence of a high voltage current being induced in the chip, destroying some connections inside.

In a second test, another different SRAM chip of the same model was inserted, although this time it was not brand new, as it had been irradiated by Mohammadreza in a previous radiation-ground based campaign [19], but it was a working non-damaged memory.

- Test 2.1: x2 shots from 5 cm away from the SRAM in a dynamic manner (i.e.: while the memory was being read). No apparent errors.
- Test 2.2: x3 new shots from 3 cm away in a dynamic manner. No apparent errors.
- Test 2.3: x3 shots from 3 cm away, but to the cables, and another 3 shots to the connector, in a dynamic manner. No apparent errors.
- Test 2.4: After finishing this dynamic reading cycle, the SRAM contents were read again from the beginning. 7 addresses affected by Multiple Bit Upsets (MBUs) [1] and Multiple Cell Upsets (MCUs) [2] were discovered at the beginning of the memory space. The direct x2 and x3 shots in Test 1) and Test 2) might have caused this.
- Test 2.5: x3 shots from 1 cm away, in static manner. No errors observed.
- Test 2.6: x2 shots from 1 cm away, in a static manner, a whopping quantity of 520 addresses presented upsets.
- Test 2.7: x1 final shot from 1 cm away, in a static manner. No errors observed.

Finally, using a different CY62167DV30LL-55ZXI SRAM, the same pattern of 0xFF bytes was written, and a dynamic test was performed, issuing 5 shots while the software was reading the memory in real time. All the observed bit-flips are shown in Table 3.1.

Address	Data
0x003800	0x0F
0x003820	0x1F
0x00B800	0x3F
0x00B820	0x1F
0x01B800	0x3F
0x01B820	0x7F
0x01F820	0x7F
0x111111	0x11

Table 3.1: Address and data of the observed bitflips

Finally, a static test was carried at 1 cm of distance, previously filling the memories with the same linear pattern of 0xFF bytes. This experiment produced a surprising and considerable number (+100) of upsets, which can be found in the companion file 20211125-Static_1cm_130-nm_520adr.csv

Analysis and hypotheses:

To further gain understanding of the faults that had been observed, these results were forwarded to the *Grupo en Hardware DInámicamente Reconfigurable* (GHADIR), who are the developers of the tool LAELAPS[3] (Lists of All Events for Locating Anomalies by Preparing Statistics). This statistical analysis[20][21] revealed that errors in the physical

[1] An MBU occurs when a single external agent provokes errors in several bits in the same memory word.

[2] An MCU occurs when a single external agent provokes errors in several bits in different same memory words. This is very common since SRAM words are interleaved, so logically adjacent bits are physically distant to each other in the device.

[3] https://github.com/fjfrancopelaez/LELAPE

layout of the memory appeared in vertical lines. More information can be found in the companion file CY62167D_by_StatisticalAnalysis.pdf.

The cause of these observed errors in vertical lines could be due to several factors. For example, three hypotheses could be: true Multiple Cell Upsets (MCUs), single event functional interrupts (SEFIs) at reading, or micro-latchups[22]. Extensive analysis of faults induced by EMFI on SRAM memories was not performed and remains a very interesting line of future research to explore.

3.2.2 Preliminary results with microcontrollers

Arduino UNO

The Arduino UNO is an inexpensive and widely used board with an 8-bit ATmega328P microcontroller, employed in a plethora of Do-It-Yourself (DIY) projects. A simple code was created which always performed the exact same arithmetic operations in a loop. It will be used with all the next targets as well, and it is quite straightforward:

```
#include <stdio.h>
int main(void){
    int i,j,k,cnt;
    k = 0;
    while(1){
     cnt = 0;
     for(i=0; i<5000; i++){
       for(j=0; j<5000; j++){
          cnt++; } }
     printf("%d %d %d %d\n", cnt, i, j,k++);  }
}
```

In theory, this code will run indefinitely on the Arduino UNO, as it does not have an exit condition. The arithmetic never changes in the code, therefore the result of such operations will always be the same. Finally, the results of the arithmetic operation and the value of the loop's counter is sent over a serial connection to the PC.

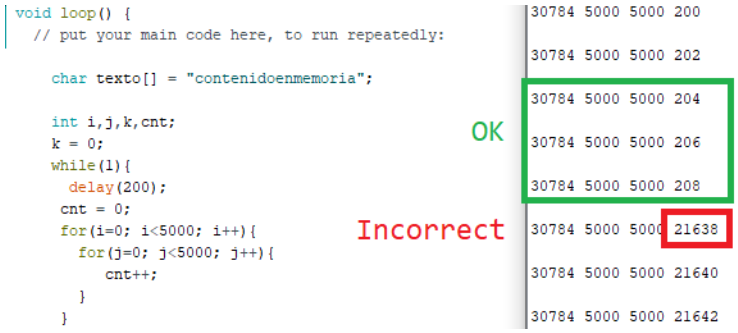


Figure 3.13: Example of software malfunction on the Arduino UNO after an injector burst.

However, when using our injector to shoot atop the surface of the ATmega328P Integrated Circuit, multiple faults were immediately observed:

- A first shot at 4 cm crashed the device. It had to be rebooted.

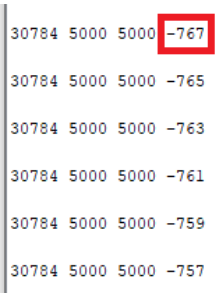


Figure 3.14: Second software malfunction on the Arduino UNO after an injector burst. The counter value became negative.

- A shot at 4 cm caused an immediate change in the variable k, which increased, as seen on Fig. 3.13.
- A shot at 4 cm caused an immediate change in the counter, which became a negative number, as seen on Fig. 3.14.
- Shots at greater distances either did not produce observable faults nor crashed the device.
- Shots at closer distances crashed or rebooted the device most of the time.

Arduino MEGA

Similarly, the Arduino MEGA is a slightly more powerful board than the Arduino UNO, packing an ATmega2560 AVR microcontroller, with greater memory and I/O capabilities.

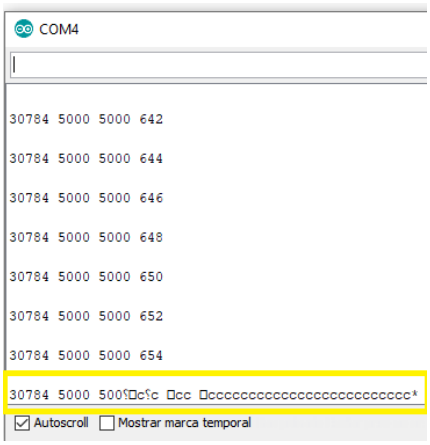


Figure 3.15: Software malfunction on the Arduino MEGA after an injector burst.

As in the previous example, arithmetic errors were produced when sending burst with the injector. Additionally, some print-related routine was affected by a fault, causing garbage to be dumped on the serial terminal, as can be observed in Fig. 3.15 and Fig. 3.16.

Figure 3.16: Second software malfunction on the Arduino MEGA affecting the serial output.

Raspberry Pi B+

Finally, the Single Board Computer (SBC) Raspberry Pi model B+ was targeted. It employs a BCM2835 ARM CPU and has 512 MB of available RAM memory.

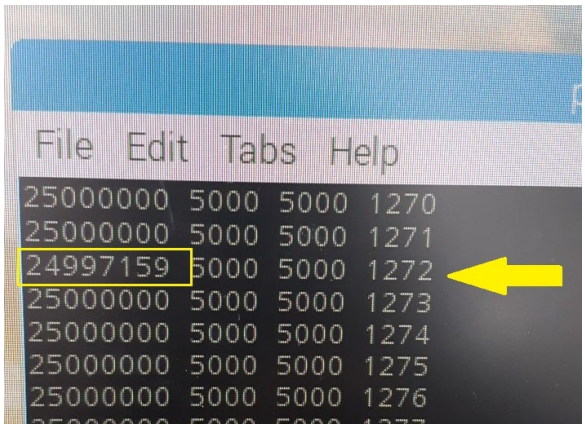


Figure 3.17: Arithmetic fault on the Raspberry Pi loop software, provoking an incorrect accumulator value.

Unlike the previous target, the Raspberry Pi B+ resembles more to a computer than to a microcontroller. It runs a Linux-based operating system called RaspbianOS, which has support for modern abstraction techniques like virtual-memory process isolation, use of cache memory, etc.

Several bursts of the EMFI device were enough to cause observable upsets and faults on the running software, like the arithmetic errors that can be seen in Fig. 3.17 or complete crashes and faults in the program, as observed in Fig. 3.18. At this point, even though demonstrated in an informal manner, it is obvious that our device is disrupting the normal operation of these microcontrollers and memory components in the boards.

3.2.3 Preliminary cheapSHOUTER results

After the successful confirmation that the first cheap and repurposed proposed device was working and producing faults, it was time to test the second design. Again, an Arduino UNO device was selected, and the code presented in the previous section was compiled and loaded into the device. The much smaller probe was placed at a very close distance of the ATMega32 microcontroller on the board, at about 0.1 cm. Said probe was placed centred atop the chip. The device was shot, and immediately the serial terminal stopped receiving any information. The 'L' LED light on the Arduino board usually blinks at start-up, indicating that the bootloader of the microcontroller is running. This blink pattern no longer appeared, and the device appeared to be permanently damaged. The

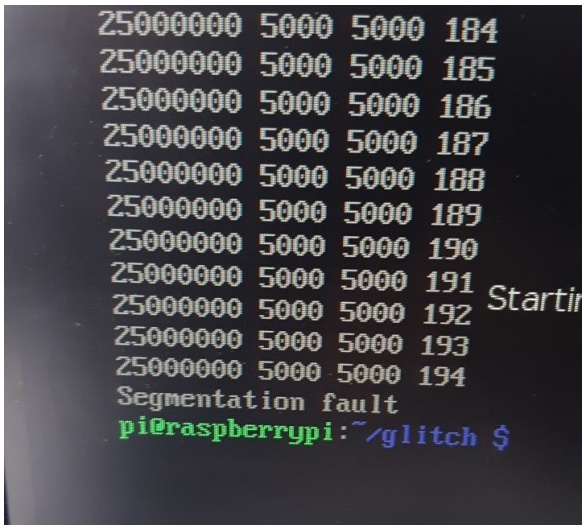


Figure 3.18: Segmentation fault on the Raspberry Pi software.

Figure 3.19: Reboot of the Arduino program.

combination of the new probe, which generates a stronger electromagnetic field because it has a ferrite core and a larger number of turns, with a very close distance to the DUT, resulted in an electromagnetic field strong enough to damage internal conductors of the device.

Figure 3.20: Counter increasing from 1038 to 1110 unexpectedly.

Using another brand-new Arduino UNO board, a set of 20 tests was run at 2 cm. This time, the device was not damaged. From 20 shots, only four produced alterations in the execution. Shots number 7,8 and 14 caused a reset of the device as seen in Fig. 3.19, while shot number 9 altered the arithmetic counter of the program, making it increase significantly, more than expected, as depicted in Fig. 3.20.

Chapter 4

The RISC-V-based DUTs used in this project

4.1 Description of the devices

For this project, two different COTS development boards with RISC-V ASIC cores were used, with the addition of a development board with an FPGA in which a RISC-V soft core was synthesized and deployed.

4.1.1 MAiX BiT with Kendryte K210 Core

The MAiX BiT4.1 is a low-cost development board based on the Kendryte K210 SoC [23]. It employs the RISC-V 64-bit RV64GC ISA with the IMAFDC set of extensions, has 8 MiB of on-chip SRAM, and includes two cache memories[24]. As for the instruction cache, the datasheet mentions: *'Cores 0 and 1 each have a 32 KiB instruction cache to improve dual-core instruction read performance'*[1]. These are two memories of 32 KiB each. As for the data cache, the datasheet again explains: *'Cores 0 and 1 each have a 32 KiB data cache to improve dual-core data read performance'*[2]. Again, these caches are 32 KiB each, one located in each core.

As it can be seen in the table in Fig. 4.2 from the Kendryte K210 datasheet, the general-purpose SRAM memory has two different interfaces that can be used to access its contents. One of them is routed to access the CPU caches mentioned earlier, while the other one accesses the memory directly without caching. It is also fragmented in two sections: a 6 MiB range is available for the CPUs and general purpose applications, while 2 MiB are allocated only for use of the KPU (a general purpose neural-network processing unit)[3]. Only the general-purpose SRAM areas will be explored in this work.

The MAiX Bit board exposes a JTAG interface that can be used to debug and probe the internal status of the CPU at any moment in time. Appendix A shows how to set up the appropriate software toolchain to interact with the Maix Bit over JTAG. The *Sipeed USB-*

[1] Bottom of page 13 of the cited datasheet.
[2] Page 14 of the cited datasheet.
[3] https://wiki.sipeed.com/soft/maixpy/en/api_reference/Maix/kpu.html

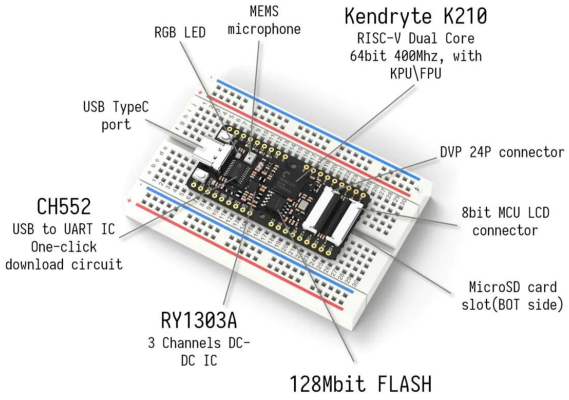


Figure 4.1: MAIx BiT board with Kendryte K210 RISC-V and other components. Source: https://www.mouser.es/images/marketingid/2019/microsites/0/Sipeed-Bit-intro.jpg

SRAM address map :

Region	Access	Start Address	End Address	Size
General-purpose SRAM	CPU cached	0x80000000	0x805FFFFF	0x600000
AI SRAM	CPU cached	0x80600000	0x807FFFFF	0x200000
General-purpose SRAM	CPU non-cached	0x40000000	0x405FFFFF	0x600000
AI SRAM	CPU non-cached	0x40600000	0x407FFFFF	0x200000

Figure 4.2: SRAM memory address map of the Kendryte K210. Source: Kendryte K210 datasheet[24].

JTAG/TTL RISC-V Debugger [4] was used as a USB - JTAG bridge. Previously, a generic USB-JTAG bridge based on the inexpensive FTDI FT232r chip[5] was considered and tested, but it was finally discarded due to its low speed, which became a bottleneck when dumping the 6 MiB of memory from the Kendryte K210 RAM memory over JTAG.

4.1.2 Longan Nano with GD32VF103C8T6

The Longan Nano4.3 is a development board manufactured by Sipeed, which packs a GD32VF103C8T6 RISC-V core with support for the RV32IMAC instruction set, plus support for so-called rapid interrupts [25]. It follows a Harvard architecture for memory organization, This DUT is much more limited, if compared with the previous target, having only 128 KiB Flash and 32 KiB SRAM memories, and a reduced instruction set.

As in the previous case, this board includes a populated JTAG header which allows for easy-to-access debugging capabilities, using the *Sipeed USB to JTAG adapter* that will be shown in the set-ups of the next chapter. Additionally, it has dedicated pins for UART Tx and Rx data lines.

[4]https://www.seeedstudio.com/Sipeed-USB-JTAG-TTL-RISC-V-Debugger-ST-Link-V2-STM8-STM32-Simulator-p-2910.html

[5]https://ftdichip.com/wp-content/uploads/2020/08/DS_FT232R.pdf

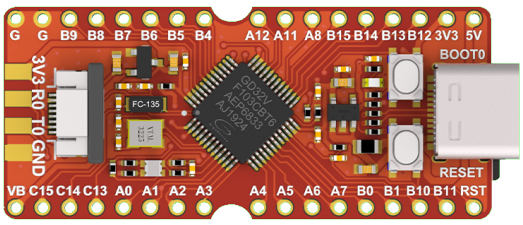


Figure 4.3: Longan nano board with GD32VF103C8T6 RISC-V plus peripherals. Source: https://es.aliexpress.com/item/4000368549335.html?channel=twinner

4.1.3 iCE40HX4K-TQ144 FPGA running a soft-core

For the third device under test in this project, focus is shifted from ASIC implementations of RISC-V CPUs to implementations that can be deployed in FPGA devices. FPGAs are reprogrammable components, and this is tremendously important, as the construction and structure of an FPGA device is considerably different to that of ASIC components. Therefore, it can be theorized that internal faults may occur or materialize differently than on ASIC chips.

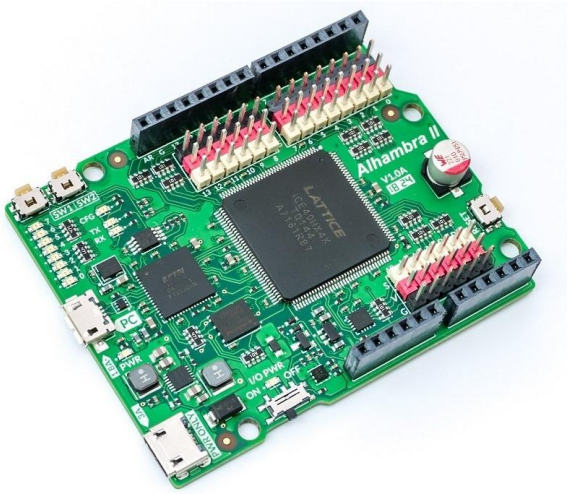


Figure 4.4: Alhambra II FPGA mainboard. Source: https://robotlandia.es/1194-large_default/placa-alhambra-ii-fpga-de-codigo-abierto.jpg

Specifically, the RISC-V-FPGA implementation for Alhambra II[6] was used, which synthesizes into a core implementing an RV32I[7] RISC-V CPU. In comparison, this CPU is simpler than the others, implementing only the base integer ISA with no extensions at all. It does not contain a JTAG controller, unlike the previous targets, so empirical experiments will rely on observing faults on a running program in the soft CPU, rather than memory analysis.

[6]https://github.com/Obijuan/RISC-V-FPGA
[7]https://github.com/Obijuan/RISC-V-FPGA/issues/7

A specific firmware was explicitly designed to serve as a target for the fault injection experiments, and its code can be found in appendix C. The code of this firmware is similar to the code shown in Chapter 3. It runs an infinite loop performing arithmetic operations, which in normal conditions must always produce the same results.

4.2 Instruction Set Architecture review

The RISC-V (pronounced risk five) architecture was born in May 2010 at UC Berkeley[26] as part of the Parallel Computing Laboratory (Par Lab). Later, in 2015, the RISC-V International Foundation was created, to '*build an open, collaborative community of software and hardware innovators based on the RISC-V ISA*'. This foundation is a non-profit corporation and has controlled the development of the architecture, promoting initial adoption of the RISC-V ISA. The ISA or Instruction Set Architecture is a document that describes the model of the CPU and what instructions it must execute, as well as the software or hardware involved in the process.

RISC (Reduced Instruction Set Computer) is a term which encompasses all microprocessor architectures that employ reduced and optimized sets of instructions, instead of the specialized and huge sets of instructions typically found in Complex Instructions Set Computer (CISC) architectures.

The basic keystones of the RISC-V ISA are reviewed here, with special emphasis on those design decisions that could, in the author's view, be relevant to maintaining the status of the program, and which could be more easily be affected by a fault injection attack.

Currently, the RISC-V ISA is subdivided in three major specifications:

- **The ISA specification**: which contains the bulk of the details about the instruction set and operation of a RISC-V machine.
- **The Debug specification**: containing the details about debugging hardware that can be optionally included in a RISC-V core.
- **The Trace specification**: containing the details of the instruction tracing hardware that can be optionally included in a RISC-V core.

In this document, focus is placed to review the ISA specification. It defines a few different sets of instructions called the bases, which only contain integer instructions support, upon which extensions can be added. These are:

- RV32I: Base integer ISA with 32-bit instructions and address space.
- RV32E: Base integer ISA with 32-bit instructions oriented to embedded systems.
- RV64I: Base integer ISA with 64-bit instructions and address space.
- RV128I: Base integer ISA with 128-bit address space.

Some examples of extensions to these bases are:

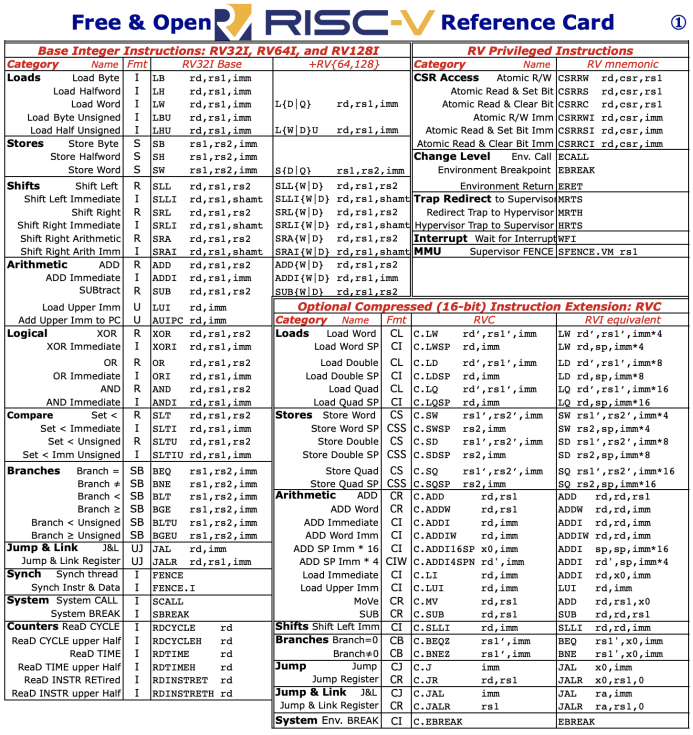
- M Extension: for integer multiplication/division instructions.
- A Extension: for Atomic instructions.
- F Extension: for Single Precision ("Float" datatype) floating point support.

- D Extension: for Double Precision ("Double" datatype) floating point support.
- B Extension: for Bit manipulation instructions.
- H Extension: for *Hypervisor* support instructions.
- etc.

It is important to take into account the different extensions a RISC-V core may have, as these directly influence which capabilities, computing power, and use cases the core may have. The underlying hardware will also change accordingly, to provide logic blocks with which to build the required logic units to support the new instructions.

Instructions can be classified in groups or categories and it can be expected that instructions from the same category may use the same hardware resources and units.

The RISC-V Reference Card seen in Fig. 4.5 illustrates the different categories of instructions that can be found in a RISC-V core.



Free & Open RISC-V Reference Card

Base Integer Instructions: RV32I, RV64I, and RV128I

Category	Name	Fmt	RV32I Base	+RV{64,128}
Loads	Load Byte	I	LB rd,rs1,imm	
	Load Halfword	I	LH rd,rs1,imm	
	Load Word	I	LW rd,rs1,imm	L{D|Q} rd,rs1,imm
	Load Byte Unsigned	I	LBU rd,rs1,imm	
	Load Half Unsigned	I	LHU rd,rs1,imm	L{W|D}U rd,rs1,imm
Stores	Store Byte	S	SB rs1,rs2,imm	
	Store Halfword	S	SH rs1,rs2,imm	
	Store Word	S	SW rs1,rs2,imm	S{D|Q} rs1,rs2,imm
Shifts	Shift Left	R	SLL rd,rs1,rs2	SLL{W|D} rd,rs1,rs2
	Shift Left Immediate	I	SLLI rd,rs1,shamt	SLLI{W|D} rd,rs1,shamt
	Shift Right	R	SRL rd,rs1,rs2	SRL{W|D} rd,rs1,rs2
	Shift Right Immediate	I	SRLI rd,rs1,shamt	SRLI{W|D} rd,rs1,shamt
	Shift Right Arithmetic	R	SRA rd,rs1,rs2	SRA{W|D} rd,rs1,rs2
	Shift Right Arith Imm	I	SRAI rd,rs1,shamt	SRAI{W|D} rd,rs1,shamt
Arithmetic	ADD	R	ADD rd,rs1,rs2	ADD{W|D} rd,rs1,rs2
	ADD Immediate	I	ADDI rd,rs1,imm	ADDI{W|D} rd,rs1,imm
	SUBtract	R	SUB rd,rs1,rs2	SUB{W|D} rd,rs1,rs2
	Load Upper Imm	U	LUI rd,imm	
	Add Upper Imm to PC	U	AUIPC rd,imm	
Logical	XOR	R	XOR rd,rs1,rs2	
	XOR Immediate	I	XORI rd,rs1,imm	
	OR	R	OR rd,rs1,rs2	
	OR Immediate	I	ORI rd,rs1,imm	
	AND	R	AND rd,rs1,rs2	
	AND Immediate	I	ANDI rd,rs1,imm	
Compare	Set <	R	SLT rd,rs1,rs2	
	Set < Immediate	I	SLTI rd,rs1,imm	
	Set < Unsigned	R	SLTU rd,rs1,rs2	
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm	
Branches	Branch =	SB	BEQ rs1,rs2,imm	
	Branch ≠	SB	BNE rs1,rs2,imm	
	Branch <	SB	BLT rs1,rs2,imm	
	Branch ≥	SB	BGE rs1,rs2,imm	
	Branch < Unsigned	SB	BLTU rs1,rs2,imm	
	Branch ≥ Unsigned	SB	BGEU rs1,rs2,imm	
Jump & Link	J&L	UJ	JAL rd,imm	
	Jump & Link Register	UJ	JALR rd,rs1,imm	
Synch	Synch thread	I	FENCE	
	Synch Instr & Data	I	FENCE.I	
System	System CALL	I	SCALL	
	System BREAK	I	SBREAK	
Counters	ReaD CYCLE	I	RDCYCLE rd	
	ReaD CYCLE upper Half	I	RDCYCLEH rd	
	ReaD TIME	I	RDTIME rd	
	ReaD TIME upper Half	I	RDTIMEH rd	
	ReaD INSTR RETired	I	RDINSTRET rd	
	ReaD INSTR upper Half	I	RDINSTRETH rd	

RV Privileged Instructions

Category	Name	RV mnemonic
CSR Access	Atomic R/W	CSRRW rd,csr,rs1
	Atomic Read & Set Bit	CSRRS rd,csr,rs1
	Atomic Read & Clear Bit	CSRRC rd,csr,rs1
	Atomic R/W Imm	CSRRWI rd,csr,imm
	Atomic Read & Set Bit Imm	CSRRSI rd,csr,imm
	Atomic Read & Clear Bit Imm	CSRRCI rd,csr,imm
Change Level	Env. Call	ECALL
	Environment Breakpoint	EBREAK
	Environment Return	ERET
Trap Redirect	to Supervisor	MRTS
	Redirect Trap to Hypervisor	MRTH
	Hypervisor Trap to Supervisor	HRTS
Interrupt	Wait for Interrupt	WFI
MMU	Supervisor FENCE	SFENCE.VM rs1

Optional Compressed (16-bit) Instruction Extension: RVC

Category	Name	Fmt	RVC		RVI equivalent	
Loads	Load Word	CL	C.LW	rd',rs1',imm	LW	rd',rs1',imm*4
	Load Word SP	CI	C.LWSP	rd,imm	LW	rd,sp,imm*4
	Load Double	CL	C.LD	rd',rs1',imm	LD	rd',rs1',imm*8
	Load Double SP	CI	C.LDSP	rd,imm	LD	rd,sp,imm*8
	Load Quad	CL	C.LQ	rd',rs1',imm	LQ	rd',rs1',imm*16
	Load Quad SP	CI	C.LQSP	rd,imm	LQ	rd,sp,imm*16
Stores	Store Word	CS	C.SW	rs1',rs2',imm	SW	rs1',rs2',imm*4
	Store Word SP	CSS	C.SWSP	rs2,imm	SW	rs2,sp,imm*4
	Store Double	CS	C.SD	rs1',rs2',imm	SD	rs1',rs2',imm*8
	Store Double SP	CSS	C.SDSP	rs2,imm	SD	rs2,sp,imm*8
	Store Quad	CS	C.SQ	rs1',rs2',imm	SQ	rs1',rs2',imm*16
	Store Quad SP	CSS	C.SQSP	rs2,imm	SQ	rs2,sp,imm*16
Arithmetic	ADD	CR	C.ADD	rd,rs1	ADD	rd,rd,rs1
	ADD Word	CR	C.ADDW	rd,rs1	ADDW	rd,rd,imm
	ADD Immediate	CI	C.ADDI	rd,imm	ADDI	rd,rd,imm
	ADD Word Imm	CI	C.ADDIW	rd,imm	ADDIW	rd,rd,imm
	ADD SP Imm * 16	CI	C.ADDI16SP	x0,imm	ADDI	sp,sp,imm*16
	ADD SP Imm * 4	CIW	C.ADDI4SPN	rd',imm	ADDI	rd',sp,imm*4
	Load Immediate	CI	C.LI	rd,imm	ADDI	rd,x0,imm
	Load Upper Imm	CI	C.LUI	rd,imm	LUI	rd,imm
	MoVe	CR	C.MV	rd,rs1	ADD	rd,rs1,x0
	SUB	CR	C.SUB	rd,rs1	SUB	rd,rd,rs1
Shifts	Shift Left Imm	CI	C.SLLI	rd,imm	SLLI	rd,rd,imm
Branches	Branch=0	CB	C.BEQZ	rs1',imm	BEQ	rs1',x0,imm
	Branch≠0	CB	C.BNEZ	rs1',imm	BNE	rs1',x0,imm
Jump	Jump	CJ	C.J	imm	JAL	x0,imm
	Jump Register	CR	C.JR	rd,rs1	JALR	x0,rs1,0
Jump & Link	J&L	CJ	C.JAL	imm	JAL	ra,imm
	Jump & Link Register	CR	C.JALR	rs1	JALR	ra,rs1,0
System	Env. BREAK	CI	C.EBREAK		EBREAK	

Figure 4.5: RISC-V Instruction reference sheet. Source: https://www.cl.cam.ac.uk/teaching/1617/ECAD+Arch/files/docs/RISCVGreenCardv8-20151013.pdf

For example, if a RISC-V were designed which implemented these instructions, it can be expected that all Load-type instructions use similar Load Queues or Load Buffers in the hardware, while Addition/Subtraction-type instructions could share related functional units in hardware, like an integer Arithmetic Logic Units (ALUs).

A fault that is produced in a functional unit may affect the execution of the program if that functional unit has been reserved or is in use by a given instruction, but may go initially unnoticed or be completely benign if no instruction is using or will use the hardware affected by the fault. Consequently, it is principal to understand the different

types of instructions the ISA defines, to better analyse observed faults and pinpoint parts of the design that might have been affected by a SEU (Single Event Upset).

Chapter 5

Fault Injection Experiments

As seen in Chapter 3, now a proposed device is known to work, being able to produce faults in components like memories, microcontrollers and CPU-based systems. Next, a set-up and procedures to realize further EMFI experiments are described, with emphasis on ensuring that the experiments and their conditions can be reproduced by other researchers in a laboratory environment.

5.1 Considered experiment parameters

The parameters that need to be controlled to perform a reproducible fault injection experiment are:

- Relative vertical distance from the DUT to the EMFI device's probe or coil.
- Implementation details of the EMFI device.
- Relative horizontal position of the DUT.
- Environmental conditions of the experiment.
- Logical state of the DUT.

5.1.1 Relative vertical distance from the DUT

As explained in Chapter 3, the distance from the electromagnetic field source of an EMFI device to the DUT is important, as the electromagnetic field strength is inversely proportional to the distance. Thus, the closer the EMFI injector is placed to the DUT, the greater the effects of the field on the DUT will be.

A distance far away from the DUT may result in no faults being observed in the device, while a distance too short could result in permanent damage of the DUT, like in the preliminary experiments with memories in Chapter 3. Henceforth, a researcher shall run several tests to calibrate their setup and find a reasonable distance, depending on the DUTs in place, their dimensions, and the EMFI device's strength.

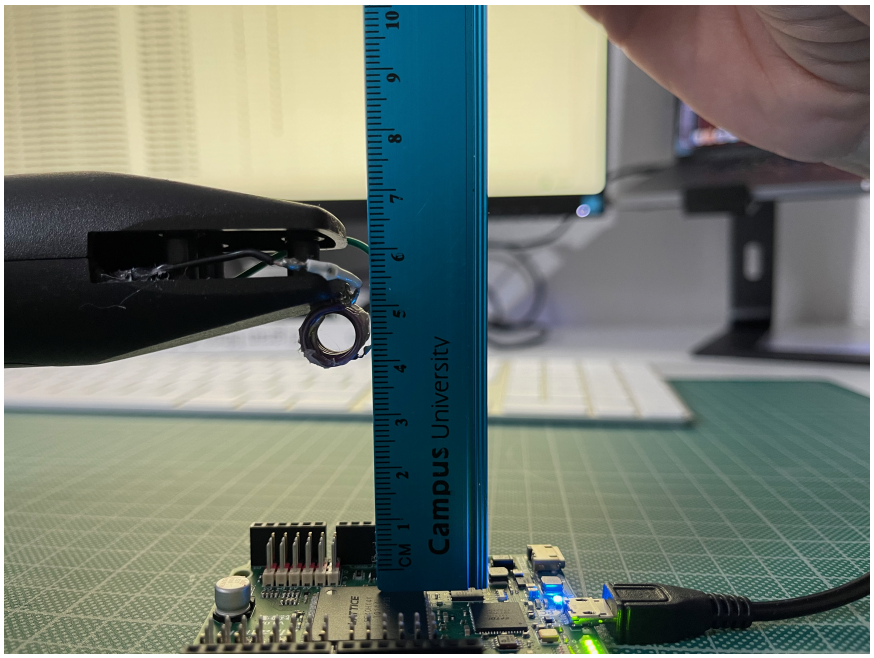


Figure 5.1: Example of test run with Alhambra II board at 4 cm

5.1.2 Implementation details of the EMFI device

The strength of the electromagnetic field generated by our device can be expressed in terms of different parameters. Therefore, it is possible to roughly compare two different EMFI devices, should the need arise. For example, a pair of EMFI devices with the same electronic circuitry, but a different number of turns in their coils, would not produce the same results. The device with a larger number of turns in its coil will have a larger inductance, and thus a larger electromagnetic flux. This could be perceived such as one device being able to produce observable faults at a given distance, while the other injector would not produce any observable faults at the same distance. A similar example would be two EMFI devices with the same electronic circuitry, and the same number of turns in their probes, but with different materials in the core. In this case, the magnetic permeability changes, and so does the magnetic flux, in the end influencing the electromagnetic field that is being generated.

Depending on the objective of the experiments, it may not be essential to consider measuring the exact strength of the fields generated, for example, if the researcher wants to investigate the resilience of a device under these fault-prone situations, regardless of the exact type and strength of the faults occurring. Small manufacturing differences in the electronic circuit of the EMFI devices may generate variations and energy losses, but these errors are often negligible if compared to all the other parameters.

5.1.3 Relative horizontal position of the DUT

As important as the previous parameters is the relative position of the DUT under the injector. The part of the DUT that is immediately under the EMFI device in the vertical axis will be the most probable to be affected by a fault. Therefore, a researcher might want to know which part of the chip's die is located under the probe or the coil of our device. For experiments targeting the resilience of the memory components, the on-chip storage

and memory elements would need to be pinpointed. For experiments aiming at analysing the effects of faults in the component's logic, the areas with the desired components (decoders, encoders, multiplexers, logic gates and logic circuits, latches, etc.) would need to be located. Depending on the target and the precision required, smaller and more accurate coils and probes will have to be fabricated.

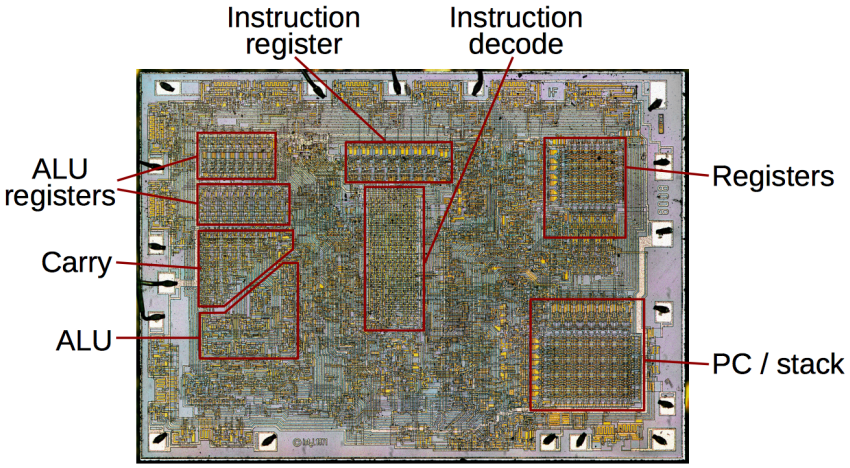


Figure 5.2: Labeled die photograph of the historic 8008 Intel CPU, by Ken Shirriff. Source: http://www.righto.com/2016/12/die-photos-and-analysis-of$_2$4.*html*

If additional details about the DUT were available, like the internal layout (which can be obtained through proprietary information provided by the manufacturer, through x-ray images of the device, or through package *decaping*) it becomes possible to pinpoint specific areas of the chip (e.g.: cache memories, CPU core, interconnection networks, etc.) and locate the EMFI device's probe or spire atop that part of the target. Fig. 5.2 shows Intel's 8008 processor die, with several logic blocks identified and labelled, making it possible to cluster different sections of the CPU depending on which area of the die they are located. For example, an attacker or researcher who knows where the Arithmetic Logic Unit (ALU) is located could inject specific faults in that area of the die, to have a higher chance of causing faults in arithmetic operations.

5.1.4 Environmental conditions of the experiment

It is important to keep a constant and stable physical state of the DUT, which can be reproduced. This includes values such as the surface temperature of the DUT, the room temperature, the lightning levels of the room or area where the experiment is performed (and if direct sunlight is hitting the device), humidity level of the room, and existence of any other environmental conditions that might affect the experiments.

While running the experiments, it is preferable to keep the device away from other electronic devices which may be running to discard any kind of electromagnetic or radio-frequency noise.

5.1.5 Logical state of the DUT

The last parameter that has to be controlled is the actual logical state of the DUT. This comprehends whether the device is powered off, in standby mode (if existing) or if it's running software and which one (in the case of a core), if it is actively storing data (in the case of a memory), or if it has a working bitstream flashed (in the case of an FPGA).

In the case of a core, it is relevant to include the source code or release number of the program that was being run, in addition to details about what Operating System (if any) was running at the same time.

For memories, it is enough to include a written description which explains the memory state, in the experiments' report (e.g.: *"The memory was written with 0xDE bytes"*, *"Program A was loaded in the system"*) or provide a binary copy of the contents of the memory before the experiment is run. For FPGAs, both the Hardware Description Language (HDL) source code and the used bitstream can be included. The bitstream needs to be included because the synthesis process is not always deterministic, and different tools or versions of a synthesis tool can produce different bitstreams.

5.1.6 About experiment logs

For the static experiments with MAiX Bit and Longan Nano, the CPU was halted immediately after it is booted using JTAG and OpenOCD, and the devices were manually probed, registering the results from the experiments in their respective comma-separated value (CSV) file. The following data was recorded: device under test, type of injector, number of injector shots issued, distance from the coil to the DUT in each shot, number of faults detected and their relative addresses in the dump file, md5 hash of the golden file, and md5 hash of the posterior dump (when relevant). For dynamic experiments with MAiX BiT and Alhambra II FPGA, the software running on the device and its specific version were noted, in addition to what stage of the software was running (boot code, specific path of execution, etc.).

5.2 Experimental results

To perform static experiments with RISC-V boards, the next procedure was employed: Firstly we set up the device by flashing a program to the onboard memory. On a normal device reset, the CPU jumps to the code and starts executing it.

However, for the experiments the CPU is immediately halted after boot. The OpenOCD tool reset configuration[1] ensures that the cores are halted *'at the reset vector before the 1st instruction is execute'*.

At this point, JTAG is used to take a memory dump of the system. This first dump will be known as the *golden* dump, which will be used as a baseline to compare.

Next, while the cores are still halted, the injector is used to provoke faults on the device.

Finally, a second memory dump is taken, and its contents are compared to the *golden* dump, looking for changes. The addresses in which the differences are found (if any) are

[1] https://openocd.org/doc/html/Reset-Configuration.html

recorded and included in the appropriate CSV file. In this way, we look for faults being induced mainly in the memory components, but also in the mechanisms and buses used for reading the memory (i.e.: the JTAG controller, I/O mechanisms, etc.) . Hardware-wise, the set-up, which is depicted in Fig.5.3, proved to be portable and easy to use.

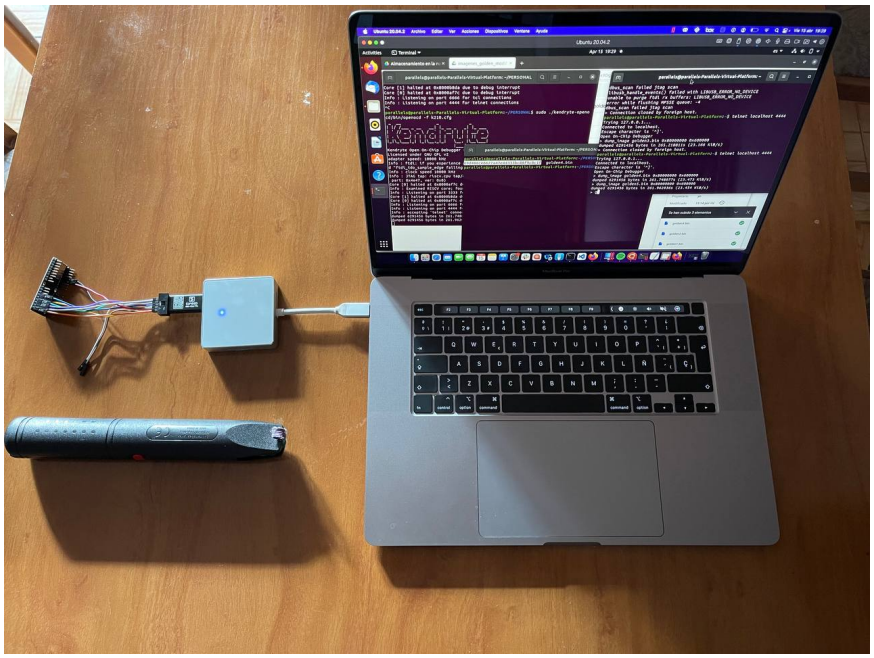


Figure 5.3: Portable EMFI JTAG setup for static tests: Injector, DUT, JTAG USB adapter, and laptop.

For dynamic tests, a program of choice is loaded and let run. While the program is running, EMFI bursts are issued and the behaviour of the program is observed, recording any unusual results or outputs.

5.2.1 MAiX BiT

The procedures described in Appendix A and Appendix B were employed to set up a JTAG connection to the SoC's JTAG controller. This offered control over the system, and a way to reset or clear the state of the device. Additionally and equally important, it provides access to observe the state of the JTAG controller, which in itself can be affected by faults, as it is part of the SoC. The device was flashed with MicroPython firmware v0.6.2 for the MAiX BiT[2]. Per-experiment details of the environment of the experiments can be found in the *experiments_batch1.xlsx* and *experiments_batch2.xlsx* companion files to this work.

Static Tests (Batch 1)

Experiment Description: Using the initial mosquito injector, 30 static experiments were performed using the MAiX BiT development board. In these static experiments, only the JTAG interface of the device was being observed. The Serial-over-USB terminal of the device was not observed. The injector was located atop the centre of the device,

[2]https://github.com/sipeed/MaixPy/releases/tag/v0.6.2

and with the probe coil completely parallel to the surface of the DUT. All the following tests used the JTAG cached memory interface to dump the contents of the SRAM.

Analysis and hypotheses: Of these 30 experiments, the first three ones were not relevant, as due to an oversight, only Core 0 was halted, while Core 1 was running, thus the state of the system was different for each experiment. Of the remaining 27 experiments, 7 were performed at a distance of 4 cm, 10 were performed at 3 cm, and 10 were performed at 2 cm. Only the experiments at 2 cm produced observable results, with apparent faults in 9 out of the 10 experiments.

Figure 5.4: OpenOCD reset-like errors observed in static experiment no. 26 (batch 1).

The OpenOCD error message shown in Fig. 5.4 was present in all the static experiments which showed observable results. It should only be observed in normal conditions when the CPU is manually reset pushing the onboard reset button, hence resetting the JTAG controller as well.

Three hypothesis are presented, as to why these JTAG interference may be occurring and affecting the communication.The first hypothesis involves the communication channel. To talk with the JTAG controller, a USB to JTAG adapter is used, which at the same time has several wires connected to the device. It is possible that the EMFI is not only affecting

the conductors inside the DUT, but also these wires, thus effectively injecting nonsensical signals in the communication channels. The cables may be acting as antennas for the electromagnetic noise the device generates, disrupting the connection.

The second hypothesis proposed would be that the faults are being, as ideally desired, inside the microcontroller's die. Because the device's probe is being place atop the center of the DUT, if the JTAG controller logic is located in the center of the die (for instance, designers might have place it there so it is at the same distance of every other part of the silicon) it could be a recurrent part of the DUT being affected by the experiments, and provoking the errors observed.

A third hypothesis is that, regardless of the JTAG controller position, faults can occur on interconnection networks going to this controller. Specially, since said controller would need to have access to most of the hardware of the CPU, it's connection could be spanning all over the die's surface. Therefore, any fault, regardless of its position, could possibly affect communications going to the JTAG controller, propagating into it. An unexpected input arriving to this debugging state-machine could likely put it into an invalid state or cause a reset of the system.

Conclusions: It is observed that the EMFI device produced faults in the device which affected the communication between OpenOCD and the JTAG controller, either resetting the controller itself or altering its status, so that the connection with OpenOCD was restarted. These errors will be observed again in the following experiments. The last two hypotheses could be easily strengthened (or completely discarded) if the internal die layout of the device was known, either obtained through x-ray imaging of the die, chip *decapping*, or through other techniques, allowing us to know the position of the JTAG controller logic and its interconnection networks. The detailed results for the described tests can be found in the experimBatch1.xcls file.

Dynamic Tests (Batch 1)

Experiment Description: Using the same device, 20 dynamic experiments were performed. The key difference in these experiments is that now focus is placed on observing the behaviour of the system under fault injection conditions while it is executing real software, rather than only analysing the effects of EMFI on memory components. To do so, the MAiX BiT was connected to both the JTAG controller, using the Sipeed JTAG adapter and over USB to a separate laptop, in which a serial terminal was listening for incoming log messages using the Linux 'screen' software[3]. This set-up can be observed in Fig. 5.5. The injector was located on top of the center of the device, and with the probe coil completely parallel to the surface of the DUT.

Analysis and hypotheses: In these experiments, several outcomes were observed, with the DUT being forced into different faulty paths of execution. It was observed that the injector was able to produce, at least, five different kinds of faults into the CPU, specifically: faulty fetches (experiment no. 35), illegal instruction execution in both Core 0 and Core 1 (experiments no. 36, 42, 44, 46, 48 and 50), misaligned stores (experiment no. 34), faulty loads (experiment no. 39 and 45) and misaligned loads (experiment no. 49). These were cases in which specific parts of the CPU were affected by a glitch,

[3]command used: screen /dev/ttyUSB0 115200

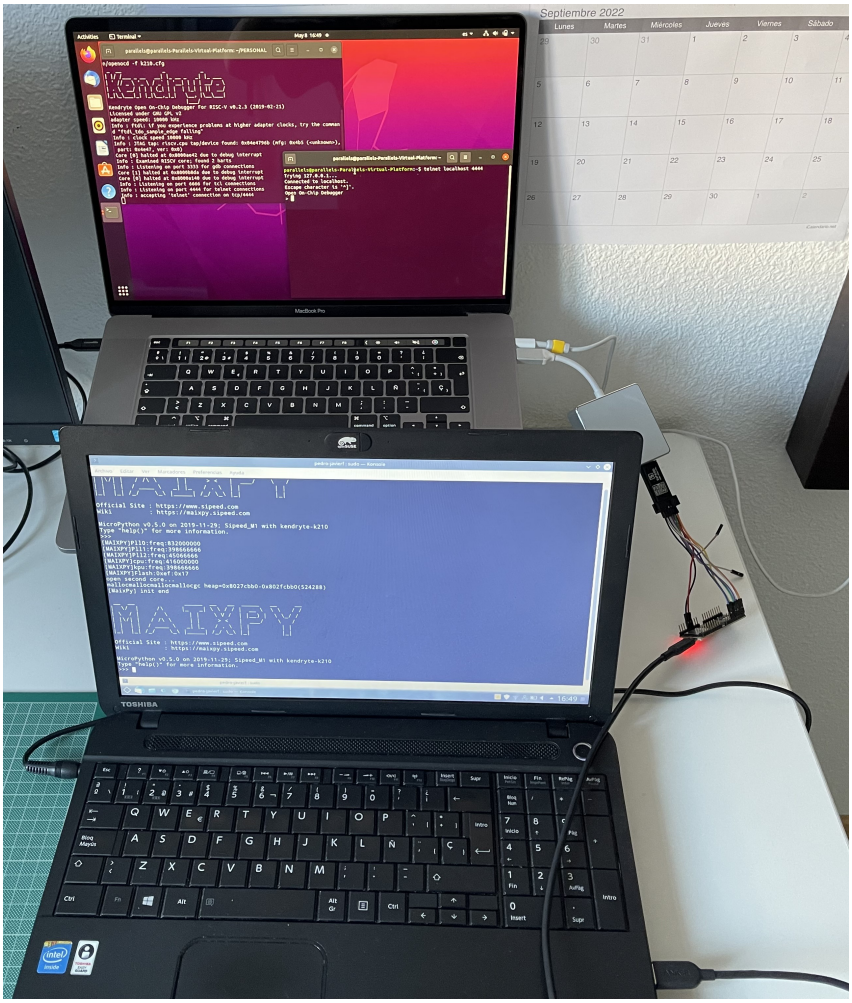


Figure 5.5: Dual laptop setup for JTAG and UART over USB. Used for dynamic tests.

resulting in an exception being raised and the associated exception handler routine being executed.

While the debugging mechanisms available do not have the mechanisms to trace the exact micro-architectural components that may have been affected by the faults, it is possible to make general assumptions of which components could have been affected. For example, the ultimate cause of a misaligned store exception, is a store instruction trying to write to a memory address which does not follow the alignment requirements of the processor. The origin of such an invalid memory address is unknown, although there can be several sources for this faulty value. It could have been caused by a memory address stored in SRAM, which was corrupted by our EMFI interference, and later loaded into the CPU's registers and used to access the memory. However, a different possibility could be one in which the address was correctly stored in SRAM and correctly loaded into a CPU register, but it was corrupted by the EMFI attack once in a register in the CPU store buffers or store queues. Similarly, a different possibility would encompass the situation in which the address was correctly stored in a register, and the CPU was operating in said register, for instance, by adding an offset to the register value. If a bit flip affected the logic units in charge of such an arithmetic operation, the result of such addition could be erroneous and go undetected initially. As the error propagates to the store unit / memory unit, an

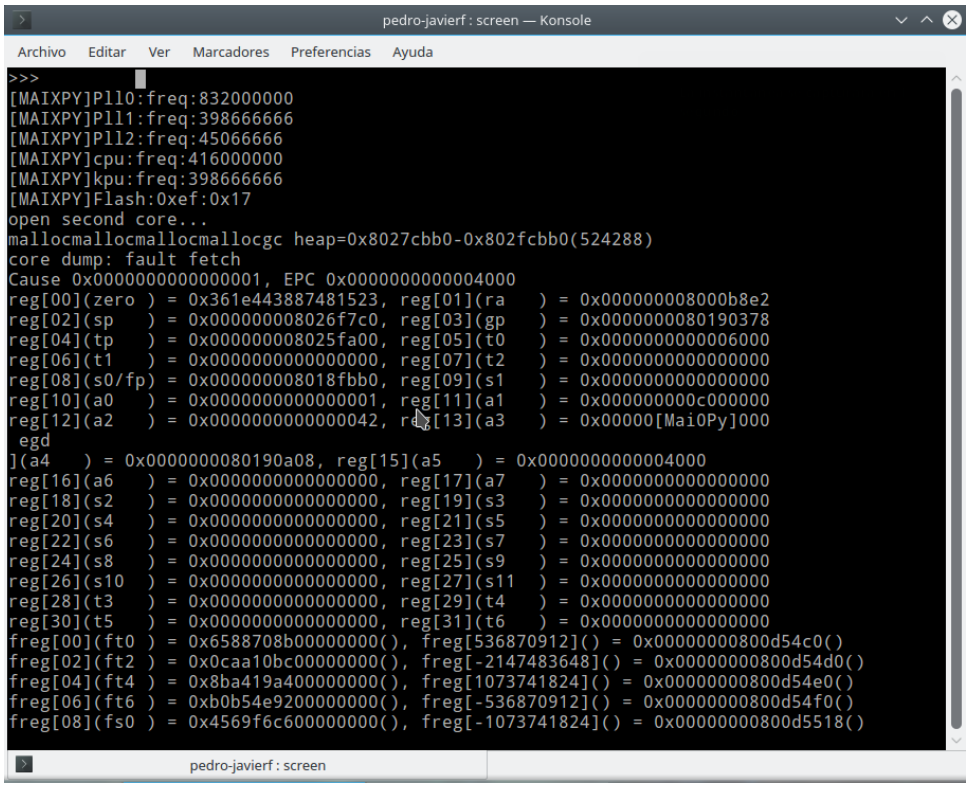


Figure 5.6: Fetch fault. Dynamic experiment no. 35 (batch 1).

exception is raised as the address is not one from an aligned memory address. This is an example of a situation in which an error is observed through a different component rather than in the one directly affected by the initial fault.

Static Tests (Batch 2)

Experiment Description: A second batch of 30 tests was performed on a different day, following the same procedure. The injector was located on top of the centre of the device, and with the probe coil completely parallel to the surface of the DUT. However, this time the non-cached memory interface of the device was used, as described in Appendix B. The exact same set-up and number of shots was used, as in the first batch. Interestingly, the results were different from the ones obtained using the cached interface. In 10 out of 30 experiments, unexpected behaviour was observed. In 6 of them, OpenOCD errors were observed, but no issues were observed in the SRAM memory dumps taken after the fault injection occurred. In two other experiments, the faults injected not only caused OpenOCD reset errors, but also resulted in incorrect SRAM dumps completely filled with 0x00 bytes. Experiment no. 29 also suffered from excessively slow SRAM dumping time. The average time that it took to dump the 6 MiB of SRAM memory in normal operation is of around 260-280 s, as can be seen in Fig. 5.8. However, experiment no. 29 reported a time of 797.9 s, and the previously mentioned zero-filled dump file.

Finally, two experiments presented SRAM dumps taken after fault injection which presented different MD5 hashes than the golden dump taken before the EMFI attack, indicating upsets. Of these two experiments, experiment no. 1, at 4 cm of distance, did not showcase any OpenOCD error log, while experiment no. 30, at 0.5 cm of distance, not

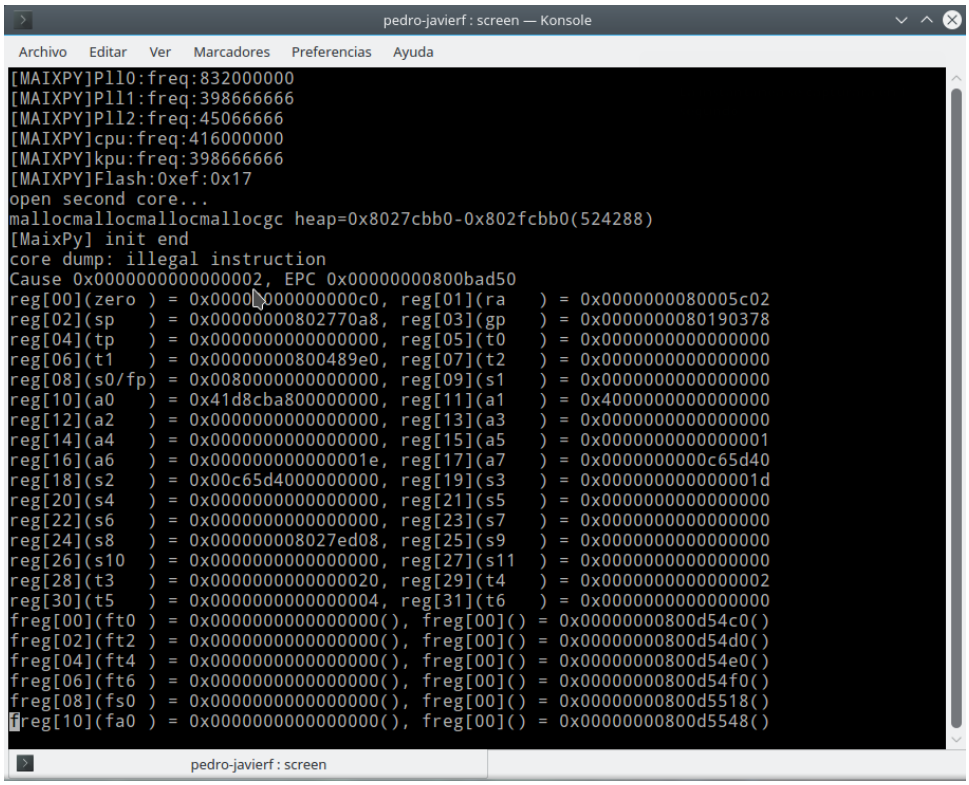


Figure 5.7: Illegal instruction fault. Dynamic experiment no. 46 (batch 1).

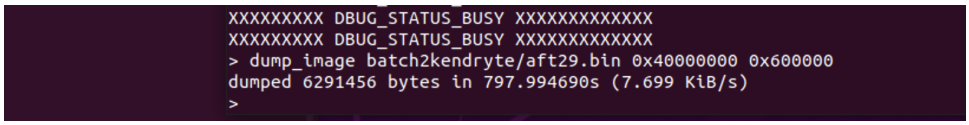


Figure 5.8: Large JTAG SRAM dumping time. Dynamic experiment no. 29 (batch 2)

only presented OpenOCD reset error logs, but also printed a "Core 0 halted" error when the EMFI device was used. In this last experiment, when trying to dump the SRAM again, the system presented an error due to an exception in the core, which seems to happen during a read operation, the one issued to dump the memory. This prevented to dump the SRAM on the first attempt, as seen in Fig. 5.9. However, a second attempt re-issuing the dump_image OpenOCD command completed successfully, obtaining the SRAM image which presented a different hash than the golden image.

Analysis: The two SRAM dumps taken after fault injection, from experiments 1 and 30, were analysed and compared against their relative golden dumps. To do so, firstly, the VBinDiff utility was used. This tool compares two files for differences (often known as a diff operation) and, unlike other tools, shows the results in a visual manner, highlighting the bytes in which changes have occurred from one file to another.

In experiment 1, only two affected bytes were observed as seen in Fig. 5.10. Bytes at contiguous relative file addresses 0x0018FBA8 and 0x0018FBA9 suffered changes. In the golden dump, the value was 0x5A38 whereas after fault injection these bytes changed to 0x7C83. No more differences were observed between the two images.

Conversely, the analyzed files from experiment 30 revealed a substantial amount of bytes

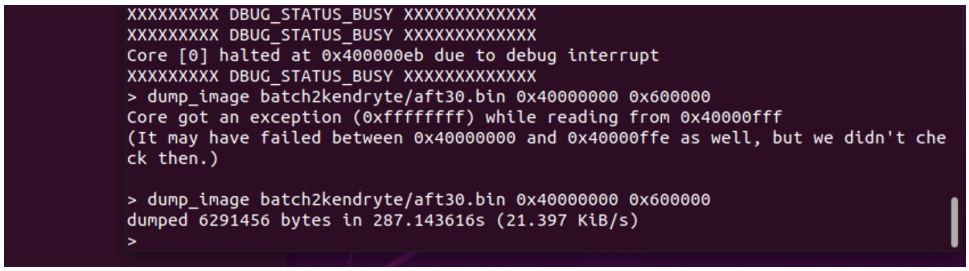


Figure 5.9: Previously never seen errors on the JTAG log terminal, indicating Core exception, and preventing initial SRAM dumping attempt. Dynamic experiment no. 30 (batch 2)

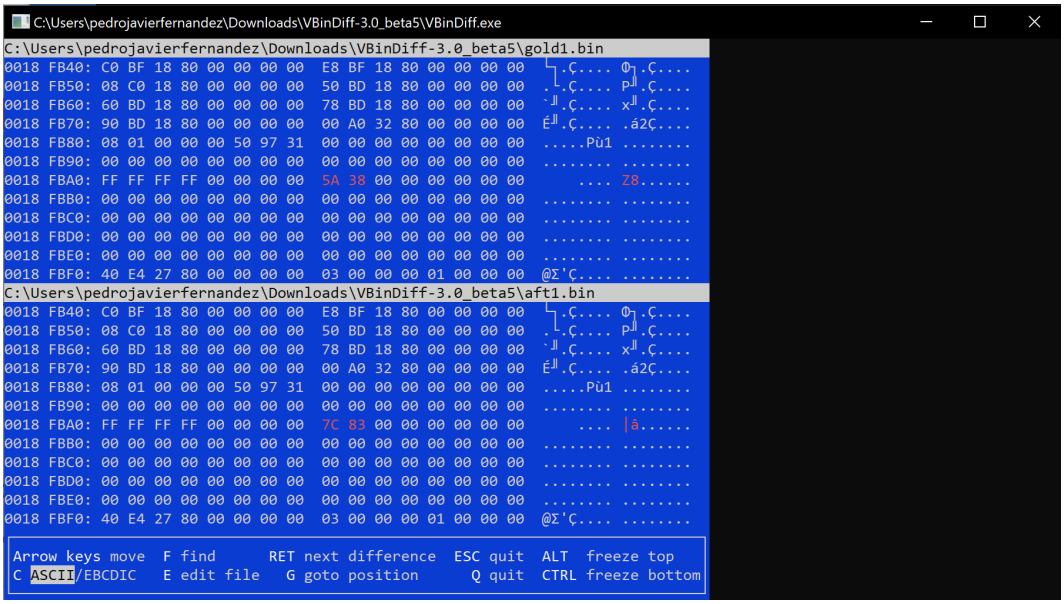


Figure 5.10: Highlighted changes between gold1.bin and aft1.bin.

whose values had changed after the fault injection experiment. The amount of faults is such that it is not feasible to visually analyzed them individually. Instead, the approach selected is to cluster and categorize them, based on the memory addresses at which they are found.

A group of 3 bytes, plus a separate byte, were affected at 0x0018EE30 addresses and can be observed in Fig. 5.11.

For instance, a set of two bytes were affected at addresses 0x0018FBA8 and 0x0018FBA9, like in experiment one. This time, the bytes changed from 0xD900 to 0xC4D1. Furthermore, a set of four contiguous bytes changed at address 0x0018FD50, depicted in Fig. 5.12.

In Fig. 5.13 Another affected region can be spotted starting at addresses 0x0018FEC0 with multiple contiguous and non-contiguous bytes that change in value.

Continuing toward upper memory addresses, three bytes are affected at address 0x00197C40. A cluster of six bytes is also modified, with 0xFF bytes changing their value, at address 0x0019D300. Two groups of 32 bits (4 bytes) which were zeroed (0x00) in the golden file, changed their value around address 0x0019E2E0. A byte was affected at 0x0019EB00,

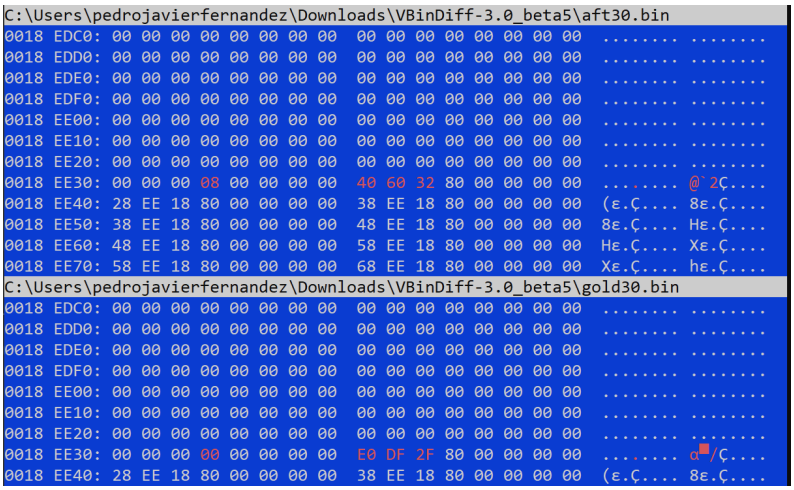


Figure 5.11: Observed changes when comparing gold30.bin and aft30.bin.

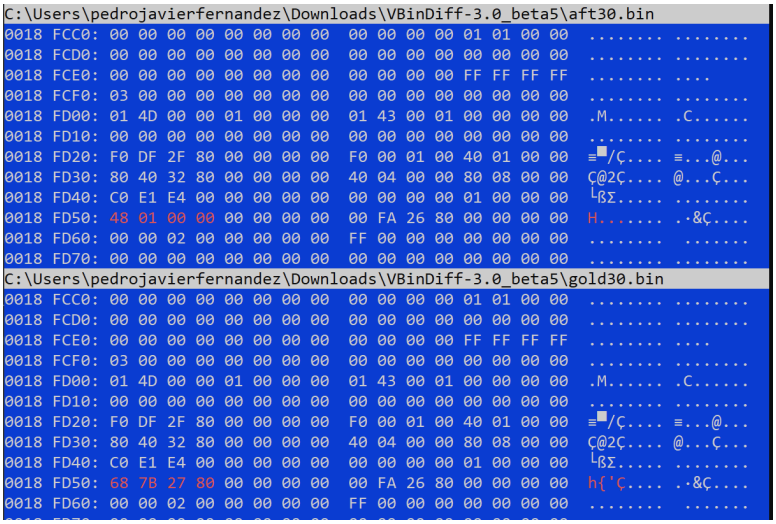


Figure 5.12: Group of 4 affected bytes at 0x0018FD50.

followed by another changed byte just 16 bytes forward in memory, at 0x0019EB10.

Considerably large 31 Kb clusters of memory positions starting at 0x0025F9E0 are affected, with seemingly random bytes suffering changes. Similar clusters of around the same size appear modified through higher memory positions.

To analyse and categorize the results, a software tool was developed to look for SBU (Single Bit Upsets) and MBU (Multiple Bit Upsets) comparing the golden files with the dumps obtained after performing the fault injection procedure. The code for this tool written in the C language can be found in Appendix D and receives as input the golden image and the image taken after the experiment.

For the assets from experiment no. 1, the tool showcased what already had been observed visually. There were two upsets, one of multiplicity 3, and another of multiplicity 6. For experiment no. 30, the tool helped to count the total number of upsets produced and to categorize them based on their types, as observed on Fig. 5.14. A representation of the data can be observed in Fig. 5.15 observing that the majority of upsets observed were SBUs. which is reasonable statistically speaking.

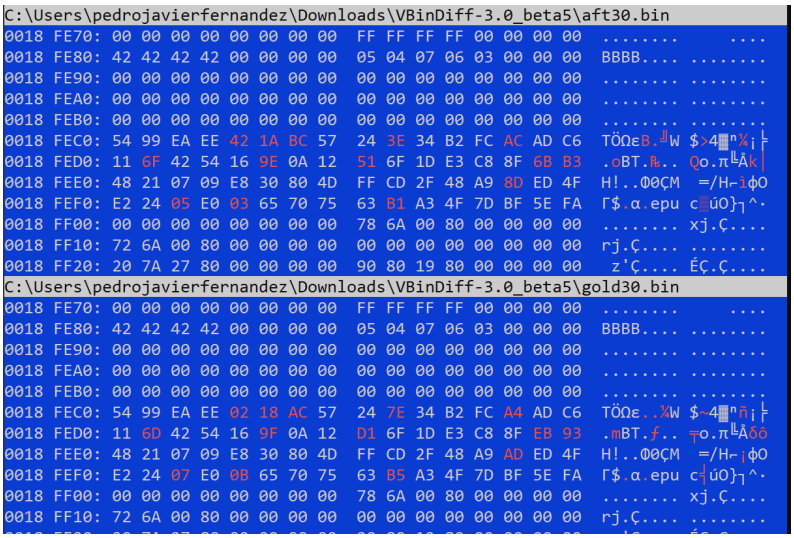


Figure 5.13: Region with multiple affected memory positions.

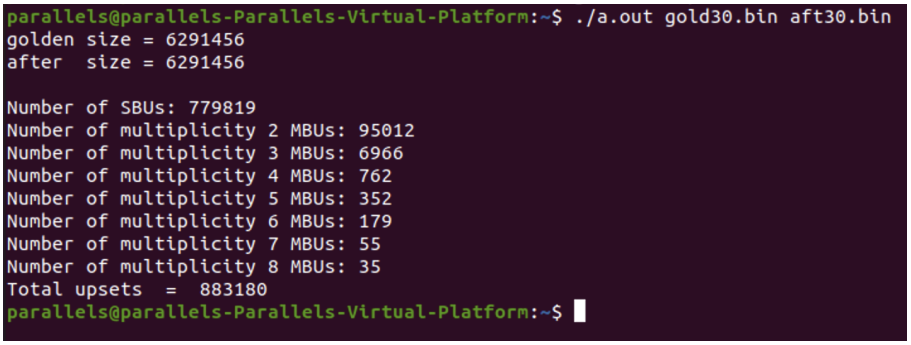


Figure 5.14: Result of the tool with the set of data from experiment 30.

Hypotheses: The observations from experiment 30 are partially satisfactory, as it is evident that the state of the system was heavily impacted by the EMFI device, as is observed both in the OpenOCD logs and through memory analysis. The first hypothesis is that these changes could have been produced solely by the injector device, affecting the SRAM memory.

Unfortunately, even though it was observed that our injector device is able to introduce faults in memories, as shown in Chapter 3, there is a chance that the disturbed memory positions affected in the last static experiment were not as a direct consequence of faults caused on the memory. As depicted in Fig. 5.9 a CPU exception was somehow detected by OpenOCD. However, this was not expected, as both CPU cores were halted by the JTAG controller, at the beginning of the OpenOCD set-up. This might suggest that the fault injected altered the JTAG controller, awakening one of the cores, which started to run again, just to crash into an exception later. This rampant CPU core could have executed part of its expected code and issued memory writes, altering the state of the memory, and thus generating the clusters with changes that were seen in previous figures. This is the second hypothesis. Nevertheless, there are some caveats to these suppositions. For instance, the changes in the big clusters appeared to occur on random and interleaved positions (while CPU code is more likely to process buffers or arrays linearly) and the data in the golden dumps appear to have high-entropy, suggesting machine code (instructions)

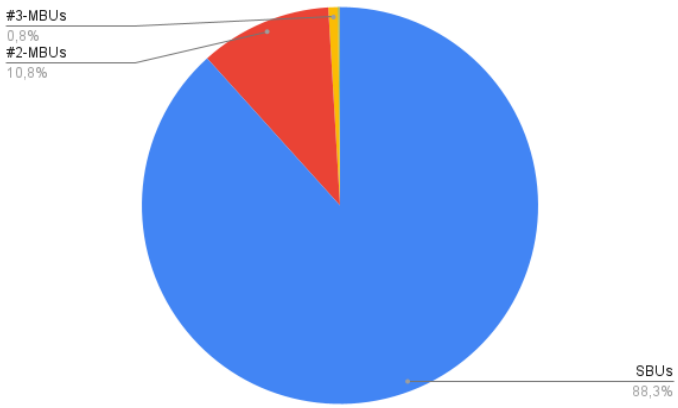


Figure 5.15: Pie chart showing the predominant type of upsets discovered: An 88,3% were SBUs, while the 10,8% were multiplicity 2 MBUs, followed by an 0,8% of multiplicity 3 MBUs.

or other high-density information, which a CPU in a microcontroller would typically not modify, unlike with clear-text data, like strings in a program, that may be formatted or muted by a program. Unfortunately, this is not a sufficient proof to discard that this hypothesis of changes were provoked by the CPU.

The third hypothesis is that both the injector affected the memory directly, and that the CPU in a faulty state trashed certain memory regions before running into an exception.

Conclusions: Ultimately, the conclusion is that there is not enough information to discern and categorically affirm which was the exact root cause of the issue. Three reasonable hypothesis are presented, all within the bounds of the capabilities of the employed EMFI device. The table with the experiment conditions of this batch can be found in included file *maixbitnocache.xcls*

A note on security concerns: Exception registers poisoning-like attacks

Dynamic tests revealed that our injector, despite being somewhat simple, is able to cause, at least, five different types of exceptions in the normal execution of the RISC-V core: misaligned stores, faulty fetches, illegal instruction, faulty loads and misaligned loads. From the point of view of a malicious attacker who is trying to extract secrets or code from a RISC-V device, this is a likely indicator that EMFI can be used to perform more complex attacks, which not only disrupt the state of the system, but exploit that incoherent state to bypass security mechanisms at the architectural or Operating System level.

One attack that stands out in this situation is the so-called "Exception Vector Poisoning" attack, for ARM architectures. This attack, which has been actively used [27] for years, exploits the fact that it is possible to trigger exceptions and thus redirect the flow of execution of the CPU to the exception handler routines, which often execute in an *exception* or *interrupt* context. Therefore, with preparation and knowledge of the target device, it is possible to modify the memory addresses where these vectors point, making them point to an area of memory controlled by the attacker. Afterwards, using fault

injection, the attacker can trigger the exception handler, thus forcing the CPU to jump and execute whatever code stored in the mentioned attacker-controlled memory address. A successful attack of this kind was performed by researcher "derrek"[1] as cited in Chapter 2. A glitch attack was used to halt the execution of the Nintendo 3DS's bootloader before it erased its own code from memory, redirecting the flow of execution to a routine programmed by the researcher to dump the bootloader's binary code from memory to an external storage device.

Similarly, the RISC-V architecture employs mechanisms to locate the routines that shall be executed in the scenario of an exception arising in the system. If these mechanisms can be tampered to redirect the flow of execution to an attacker-controlled memory region, and if it is possible to generate arbitrary exceptions through fault injection, the security of the device can be compromised. It has been observed that our injector device can provoke these exceptions. Recent previous research for RISC-V architecture showcased on DEF CON 29[28] indicates that mechanisms like the *mtvec*[29] can be altered and abused to perform similar code hijacking attacks.

5.2.2 Longan Nano

The Longan Nano, although using a different RISC-V microcontroller, has a PCB designed by the company Sipeed, and thus is compatible with the USB to JTAG debugger employed previously for the MAiX BiT experiments. Unfortunately, the current official Software Development Kit (SDK) contains broken debugging tools, like OpenOCD, which made it impossible to successfully debug the device. Therefore, only dynamic tests were performed.

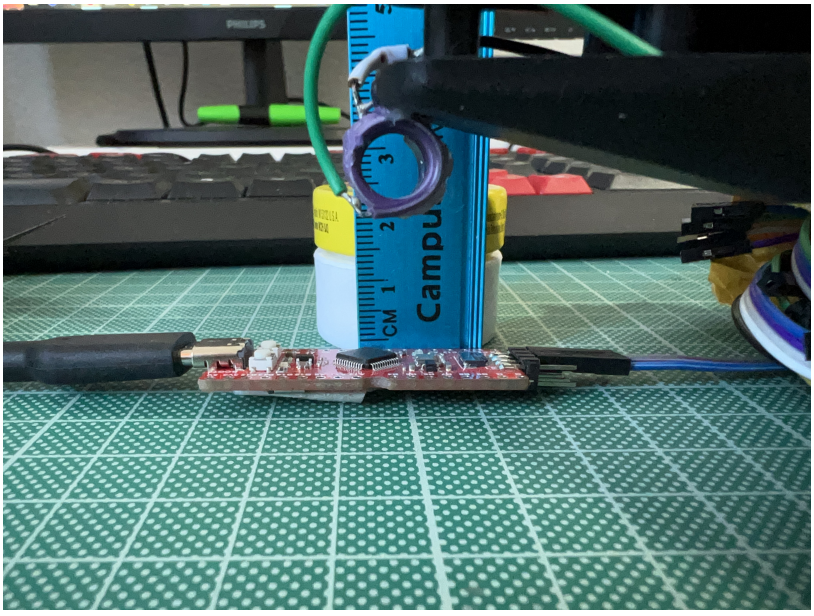


Figure 5.16: Injector centred atop the Longan Nano at a 2 cm distance.

Experiment Description: A set of 50 dynamic experiments was performed. The device was loaded with a firmware that uses UART to log messages and then runs the same arithmetic loop code shown in the preliminary results of Chapter 3. As in previous experiments, the mosquito device was used and the probe was place centred atop the DUT.

Analysis: Several types of faults were observed in the experiments. A chart showing the percentages for each type of fault can be seen in Fig. 5.17. Most experiments where no faults were observed correspond to larger distances from the injector to the DUT, while the cases with a higher number of observable faults correspond to tests in which the injector was located closer to the device.

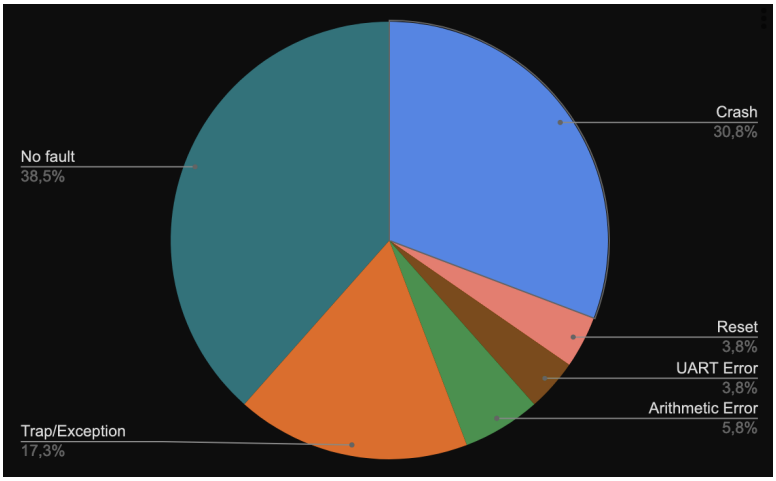


Figure 5.17: Pie chart of the types of faults and their occurence in the 50 tests.

```
25000000 5000 5000 8522
25000000 5000 5000 8523
25000000 5000 5000 8524
25000000 5000 5000 8525
25000000 5000 5000 8526
25000000 5000 5000 8527
25000000 5000 5000 8528
25000000 5000 5000 8529
25000000 5000 5000 8530
25000000 5000 5000 8531
25000000 500trap

Program has exited with code:0x30000002
```

Figure 5.18: Longan Nano trap exception example in dynamic experiment no. 39

Fig. 5.18 shows the result of experiment no. 39, in which the fault injection experiment produces a *trap* or exception, leading to the activation and execution of the default exception handler on the device, printing a numeric code representing the type of exception that occurred. The code of this default exception handler in C can be seen in Fig. 5.19. The RISC-V standard exception codes are defined in the privileged specification of the ISA[4]. When an exception occurs, the specific error values are saved in the architectural register called the Machine Cause Register or *mcause* register. The mentioned handler reads these codes and prints them through the default output configured, serial in this case.

Hypotheses:

For the experiments that resulted in an exception being handled in the core (17,3% of the total) there is some more information than in those in which the device just crashed. Particularly, two different *mcause* codes were obtained: 0x30000002 and 0x30000001.

[4]https://riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf Page 35.

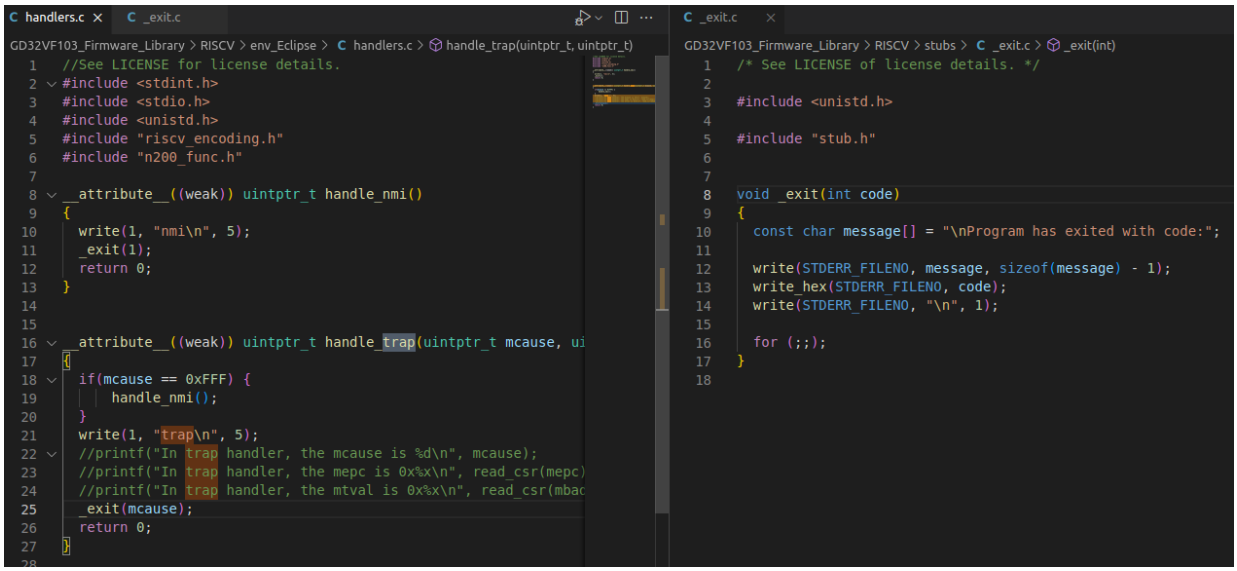


Figure 5.19: GD32V low-level exception Handler code in SDK files *handlers.c* and *_exit.c*

According to the privileged RISC-V specification mentioned earlier, the most significant bit (MSB) of the register shall be 1 if the trap was caused by an interrupt, and 0 otherwise. Both error codes have a 0 as the MSB, thus indicating the exception was not produced by a planned interrupt, but rather by some malfunction or error.

- Error 0x30000002, Illegal instruction: This is the most common exception observed in the experiments of this work. There are several paths that, if suffering from a fault, could lead to this type of exception. Several possible hypotheses are presented. Firstly, a fault in the instruction decoding logic, resulting in invalid decoding of a correct set of bytes. The other possibility is a fault in the address value of the instruction fetch operation, resulting in an architecturally-correct fetch (i.e.: a valid memory access that doesn't raise any exception) but that is logically incorrect, fetching instructions from a different place than expected (e.g.: from a data region, from zeroed memory etc.) thus resulting in invalid data being forwarded to the instruction decoding logic.
- Error 0x30000001, Instruction access fault: In this case, the error code is slightly more specific, allowing to hypothesize with more insight. In this case, the instruction fetch failed. This is caused by an invalid read operation, possibly due to a fault or error in the memory address, which could be pointing to invalid memory regions (out of the memory map), protected regions or pages, or simply data pages.

In both cases, there is not enough information to pinpoint the exact component that was originally affected by the fault, as the error could have propagated, and with the proposed experiment settings it is only possible to observe the final outcome and which logic is catastrophically failing. Statistically, it is possible that a larger number of experiments would have revealed other types of exceptions being caused, similarly to the experiments with the MAiX BiT DUT.

Conclusions:

It can be concluded that this 32-bit RISC-V microcontroller seems to be equally susceptible to being affected by the EMFI using the proposed basic injector. Knowledge of the

RISC-V ISA can be employed to gather information about the type of exception or fault that has occurred, always taking into account that errors might have propagated from different parts of the DUT before reaching the observed affected logic.

The results of all the experiments can be found in included file $experiments_{l}ongan_{b}atch1$.

5.2.3 Alhambra II FPGA RISC-V soft core

Dynamic Tests

Unfortunately, the selected RISC-V soft-core available for this board did not include a JTAG state machine, therefore, it is not possible to perform purely static experiments like memory analysis after fault injection, at least without altering the CPU state or employing it to analyse the memory. For these experiments, aim was put to looking for the appearance of faults in a program running arithmetic operations on the core, and observing if it was interrupted, halted or if it had crashed, or if alterations in the outputs of the program were caused. The source code for the C firmware designed to be glitched can be found in Appendix C. Limitations like a reduced serial output functionality existed in the base firmware used, thus limiting the ability to print verbose dynamic debug messages.

Experiment Description:

A first batch of 50 runs of the program were carried, trying to induce glitches in the arithmetic loop of the firmware. 42 tests resulted in disruption of the system, which automatically rebooted itself. This is, an 84% of the executions were altered by the injector. However, no arithmetic faults were observed.

Hypotheses and Conclusions:

The internal composition of FPGAs is different from that of ASIC parts. However, even though in the batch of experiments that was performed, no faults other than reboots were observed, and no arithmetic flaws were observed, this is not enough reason to affirm that FPGAs are more immune to EMFI attacks. The hypothesis is that with a larger number of tests, it may be possible to observe exceptions and arithmetic faults, like with the previous devices. It can be concluded that further experimentation needs to be done with this FPGA device, solving firstly the firmware limitations, or switching to a more advanced and complex soft-core design.

Chapter 6

Conclusions and future work

In previous chapters, the advantages and caveats of an affordable electromagnetic fault injector have been explored. The initial objectives are reviewed and evaluated to see if the main milestones have been achieved. Finally, future lines of work that have spawned as a result of this work are discussed.

The main objective of this work was successfully achieved, which consisted in finding an affordable solution to conduct EMFI experiments in the academic context. Available hardware resources were widely explored, such as off-the-self appliances like a mosquito killer device, which were then re-purposed to serve a nobler duty. The bottom-line was affordability, which was obtained with success. Furthermore, an intermediate solution in terms of cost-usability ratio was developed completely anew. The second injector design does not require to re-purpose any previously existing device. However, results are only partially satisfactory, as it was not possible to verify the circuit completely due to the difficulty to source electronic components, which is affecting the whole industry at the time of writing.

Nevertheless, the development of this work and the desire to improve the different designs (mechanical, electronic, etc.) described in it, yielded new ideas for future work and research. Some of the most relevant are:

- **Injector Printed Circuit Board (PCB) Design:** In chapter 3, an EMFI device's circuit was designed from scratch. This initial prototype was built using scratch-boards. To improve the design, make it easier for other people to build it, and make it suitable for mass-production up to an extent, the next logical step is to design a PCB. Overseas hobbyist PCB manufacturing has become cheaper in the last years, thus making creating high-quality PCBs a reasonable goal for student research. This was not explored due to a lack of time.
- **3D printed injector support**: The aim of keeping development and research costs at a minimum while obtaining reliable results and a safe, newbie-friendly platform for fault injection tests, can be achieved by using the now popular 3D printing techniques. Our in-house EMFI device, *cheapSHOUTHER*, can be built and soldered in a scratch-board, or in a PCB like mentioned above. However, in any of these cases, the bare circuit will be exposed to the people handling the device. High voltages always have to be considered dangerous, and thus, the need for casing

or insulation arises. In addition, a case or shell for our device would provide the researches with a more comfortable device, easier to use, and easier to store.

- **3D printed injector arm:** Another possible improvement or future line of work would be the design of 3D models for mechanical arms which would hold the device in place. These arms would allow regulating height and position of the device with significantly greater precision, while at the same time, ensuring that the device can be left unattended without worrying of subtle changes in its position or location, as the device would be attached or glued to the mechanical scaffolding.
- **Use of a CNC machine for increased accuracy:** If used in conjunction with the 3D printed arm, a CNC turning machine would provide extremely precise movements to relocate the injector with accuracy in space. This can be used to effectively target specific areas of the die or chip under test, and to reproduce the experiments with more fidelity, by being able to position the injector's probe in space with increased precision.
- **Improvements for the reverse engineered circuit:** The cheapSHOUTER device developed in this work presents room for improvement as well. The first stage of the circuit, the analogue oscillator, presents several caveats, most notably a lack of frequency control. A better analogue oscillator circuit would be possible, but the cost would increase significantly. While this work was being developed, company NewAE released a new middle-end EMFI device, called the picoEMP[1]. This device is peculiar, as it replaces the oscillator circuitry with an affordable COTS microcontroller, the Raspberry Pi Pico ©. This allows for software-controlled frequencies, through the generation of an output Pulse Width Modulation (PWM) signal, which is used to drive the high-voltage transformer. A similar digital control system could be integrated into the design proposed in Chapter 3. Furthermore, safety improvements can be introduced into the injector, using different electronic components. For example, a new iteration of the circuit could be created, replacing the spark gap with an optocoupler switch (specifically, a variety often referred to as phototransistor). This way, the circuit and the logic that triggers the shot of the device becomes electrically isolated from the circuit that works at a high voltage. A basic diagram of an optocoupler switch is depicted in Fig. 6.1 and an example application is depicted in Fig. 6.2. Using this component would allow triggering the shot of the injector in a safer way and more controlled way, although the price of the hardware might increment.

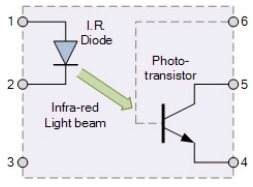


Figure 6.1: Diagram of a photo-transistor. Source: https://www.electronics-tutorials.ws/blog/optocoupler.html

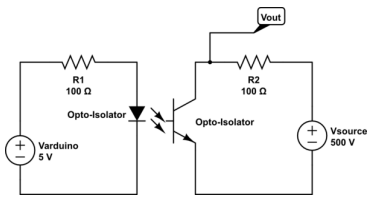


Figure 6.2: Photo-transistor application. Source: https://i.stack.imgur.com/gMjwr.png

[1]https://github.com/newaetech/chipshouter-picoemp

- **Evaluation of RISC-V building blocks:** Individual commonly-used building blocks (i.e: ALUs, register banks, cache memories, and even manufacturer's Intellectual Property (IP) blocks) can be studied to test their resilience or behaviour under fault injection situations. To approach this, a possible solution would be having an FPGA-controlled baseline design, in which blocks can be altered or even completely replaced as the researcher considers, being able to switch their low-level implementations. Then, the performance and tolerance of the whole system can be evaluated, to evaluate and compare how the modification of a given component affects the whole system. A candidate for this could be the popular RISC-V generator rocket-chip[2].
- **Evaluation of RISC-V security and virtualization extensions:** Should a research be designed to dive deeper into the RISC-V architecture and the effects of EMFI in all the possible implementations and variations, it would be of interest to analyse the virtualization and security-oriented extensions of the ISA, as these extensions are likely to be found in secure devices, more susceptible to suffering from malicious fault injection attacks.
- **Creation or study of fault models:** Fault models are used in engineering to try modelling and predicting the situations in which a fault occur, and its consequences. For a given DUT, a future study could elaborate a fault model to evaluate and predict how they respond to EMFI.

Despite the endless possible targets that could be tested under EMFI conditions, and the enhancements that our devices could go through, the results of this work are satisfactory, as most objectives and milestones were reached, even if with varying degrees of success. It is crystal clear that the RISC-V architecture is no stranger to being susceptible to fault injection attacks, even if the architecture itself is not yet as widespread as other RISC flavours. More research needs to be developed to study the potential vulnerabilities or weaknesses of the architecture, specially when facing low-cost solutions that could allow for easy hardware disruption to attackers all over the world, with physical access to their devices.

In an increasingly connected world, both reliability and security are concerns which manufacturers and designers must address. Real-world data is going to be needed to create models, simulations, and to improve the quality of the products of tomorrow. RISC-V keeps gaining followers and support, and more microchips implementing this ISA will be released to the market, henceforth reinforcing the need for experimentation and research about all the aspects and corners of this architecture.

[2]https://github.com/chipsalliance/rocket-chip

List of Figures

List of Tables

Bibliography

[1] Derrek, "Exploiting nintendo 3ds's arm9/arm11 bootrom vectors pointing at uninitialized ram."

[2] M. Scire, M. Mears, D. Maloney, M. Norman, S. Tux, and P. Monroe, "Attacking the nintendo 3ds boot roms," 2018.

[3] J. D. Thomas Roth, Dmitry Nedospasov, "wallet.fail hacking the most popular cryptocurrency hardware wallets."

[4] Y. Lu, "Injecting software vulnerabilities with voltage glitching," 2019.

[5] G. Roussel-Tarbouriech, N. Menard, T. True, T. Vi, and Reisyukaku, "Methodically defeating Nintendo Switch security," 2019.

[6] Hajdu, N. Ivaki, I. Kocsis, A. Klenik, L. Gönczy, N. Laranjeiro, H. Madeira, and A. Pataricza, "Using fault injection to assess blockchain systems in presence of faulty smart contracts," *IEEE Access*, vol. 8, pp. 190760–190783, 2020.

[7] R. Buhren, H. N. Jacob, T. Krachenfels, and J.-P. Seifert, "One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization," 2021.

[8] C. O'Flynn, "Fault Injection using Crowbars on Embedded Systems."

[9] M. A. Elmohr, H. Liao, and C. H. Gebotys, "Em," in *2020 21st International Symposium on Quality Electronic Design (ISQED)*.

[10] J. Laurent, V. Beroulle, C. Deleuze, and F. Pebay-Peyroula, "Fault injection on hidden registers in a RISC-V Rocket processor and software countermeasures," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 252–255, 2019.

[11] Elmohr, Mahmoud A., "Embedded systems security: On EM fault injection on RISC-V and BR/TBR PUF Design on FPGA," Master's thesis, 2020.

[12] S. Nashimoto, D. Suzuki, R. Ueno, and N. Homma, "Bypassing isolated execution on RISC-V with fault injection." Cryptology ePrint Archive, Report 2020/1193, 2020. `https://ia.cr/2020/1193`.

[13] PoroCYon, "Nintendo hacking - Dumping the DSi boot ROMs: Uncovering 13 year old hardware secrets."

[14] A. Pathak, "An elementary argument for the magnetic field outside a solenoid."

[15] W. Elektronik, "749196501 transformer datasheet."

[16] W. Elektronik, "749196118 transformer datasheet."

[17] I. Technologies, "CY62167DV30 datasheet."

[18] Infineon Technologies, "Infineon cy62167dv30ll-55zxi datasheet."

[19] M. Rezaei, G. Hubert, P. Martín-Holgado, Y. Morilla, J. C. Fabero, H. Mecha, F. J. Franco, H. Puchner, and J. A. Clemente, "Impact of Dynamic Voltage Scaling on SEU Sensitivity of COTS Bulk SRAMs and A-LPSRAMs against Proton Radiation," *IEEE Trans. Nucl. Sci.*, vol. 69, no. 2, pp. 126–133, 2022.

[20] F. J. Franco, J. A. Clemente, M. Baylac, S. Rey, F. Villa, H. Mecha, J. A. Agapito, H. Puchner, G. Hubert, and R. Velazco, "Statistical deviations from the theoretical only-sbu model to estimate mcu rates in srams," *IEEE Transactions on Nuclear Science*, vol. 64, no. 8, pp. 2152–2160, 2017.

[21] F. J. Franco, J. A. Clemente, H. Mecha, and R. Velazco, "Influence of randomness during the interpretation of results from single-event experiments on srams," *IEEE Transactions on Device and Materials Reliability*, vol. 19, no. 1, pp. 104–111, 2019.

[22] C. Palomar Trives, "Inducción de sucesos aislados en memoria SRAM (Induced single events in SRAMs)," 2012.

[23] Seeed Technology Co, Ltd SIPEED, "Sipeed maix bit reference page."

[24] C.-C. Inc., "Kendryte K210 English datasheet."

[25] Seeed Technology Co, Ltd SIPEED, "Sipeed Longan Nano reference page."

[26] R.-V. Foundation, "RISC-V History."

[27] A. 'acez' Cama, "Corrupting the ARM exception vector table."

[28] M. Zabrocki, "Glitching RISC-V chips: MTVEC corruption for hardening ISA."

[29] R. Foundation, "The RISC-V instruction set manual, volume ii: Privileged architecture."

Appendix A

RISC-V OpenOCD Linux set up procedure for Kendryte K210

OpenOCD configuration for KendryteK210 in Debian based Linux distributions The following files need to be created with the following contents: **k210.cfg**

```
# debug adapter
source [find ft2232c.cfg]

transport select jtag
adapter_khz 10000

# server port
gdb_port 3333
telnet_port 4444

# add cpu target
set _CHIPNAME riscv

jtag newtap $_CHIPNAME cpu -irlen 5 -expected-id 0x04e4796b

set _TARGETNAME $_CHIPNAME.cpu
target create $_TARGETNAME riscv -chain-position $_TARGETNAME

# command
init
halt
```

ft2232c.cfg

```
interface ftdi
ftdi_vid_pid 0x0403 0x6010
ftdi_layout_init 0xfff8 0xfffb
```

```
ftdi_layout_signal nTRST -data 0x0100 -oe 0x0100
ftdi_layout_signal nSRST -data 0x0200 -oe 0x0200
```

Instalation:

```
$ wget https://github.com/kendryte/openocd-kendryte/releases/
   download/v0.2.3/kendryte-openocd-0.2.3-ubuntu64.tar.gz
$ tar xvf kendryte-openocd-0.1.3-ubuntu64.tar.gz
$ ./kendryte-openocd/bin/openocd -f k210.cfg
```

Appendix B

Commands to dump memory over JTAG

Dumping memory over JTAG with OpenOCD: Connect to the OpenOCD local server over telnet:

```
$ ./kendryte-openocd/bin/openocd -f k210.cfg
$ telnet localhost 4444
```

Once connected, the memory space of the MAiX BiT can be dumped (using the cached interface) using the following command:

```
> dump_image RAMdump.bin 0x80000000 0x600000
```

For the MAiX BiT non-cached interface, the following range of addresses must be used:

```
> dump_image RAMdump.bin 0x40000000 0x600000
```

Appendix C

FPGA set-up and firmware source code

Alhambra II PicoSoC core set-up

a). Set up icestudio and the toolchain.

b). Flash PicoSoC design to FPGA

c). Download pre-built RISC-V GNU toolchain binaries for riscv32i from the link below C

d). Modify Makefile at firmware/soc-demo/src-c to point to the prebuilt binaries

e). Run 'make' while the FPGA is connected. This will flash the firmware to the SoC

Prebuilt Toolchain

https://github.com/stnolting/riscv-gcc-prebuilt

GlitchMe firmware source code

```
// GlitchMe C firmware
// Pedro Javier Fernández - 2022
// Based on RISC-V-FPGA-master /soc-demo/src-c/test.c
#include <stdint.h>

//-- Registros mapeados
#define reg_uart_data (*(volatile uint32_t*)0x02000008)
#define reg_leds (*(volatile uint32_t*)0x03000000)

// ------------------------------------------------------

void putchar(char c)
{
        if (c == '\n')
                putchar('\r');
        reg_uart_data = c;
}
```

```
void print(const char *p)
{
        while (*p)
                putchar(*(p++));
}

char getchar_prompt(char *prompt)
{
        int32_t c = -1;
        int32_t count = 0;

        uint32_t cycles_begin, cycles_now, cycles;
        __asm__ volatile ("rdcycle %0" : "=r"(cycles_begin));

        reg_leds = ~0;
        count = 0;

        if (prompt)
                print(prompt);

        while (c == -1) {
                __asm__ volatile ("rdcycle %0" : "=r"(cycles_now));
                cycles = cycles_now - cycles_begin;
                if (cycles > 2000000) {
                        if (prompt)
                                print(prompt);
                        cycles_begin = cycles_now;
                        count += 1;
                        reg_leds = count;
                }
                c = reg_uart_data;
        }

        reg_leds = 0;
        return c;
}

char getchar()
{
        return getchar_prompt(0);
}

void menu()
{
        print("\n");
        print("  ____  _          ____         ____\n");
        print(" |  _ \\(_) ___ ___/ ___|  ___  / ___|\n");
```

```
        print(" | |_) | |/ __/ _ \\___ \\ / _ \\| |\n");
        print(" |  __/| | (_| (_) |__) | (_) | |___\n");
        print(" |_|   |_|\\___\\___/____/ \\___/ \\____|\n");
        print("\nRunning on the Alhambra II board :-)\n");
        print("\n");
}

void sleep()
{
        int i = 0;
        while(i != 100)
                i++;
}

// This function is the target of the glitch attacks
void glitchMe()
{
        int counter = 0;

        while(1)
        {
                counter++;
                if(counter==1) print("1\n");
                else if(counter==2) print("2\n");
                else if(counter==3) print("3\n");
                else if(counter==4) print("4\n");
                else if(counter==5) print("5\n");
                else if(counter==6) print("6\n");
                else if(counter==7) print("7\n");
                else if(counter==8) print("8\n");
                else if(counter==9) print("9\n");
                else if(counter==10){print("10\n"); counter = 0;}
                sleep();
        }
}

void crashMe()
{
        // Unaligned memory read
        int* pointer = 0x00000001;

        // Dereference
        int value = *pointer;
}

// --------------------------------------------------------

void main()
```

```
{
        char c;

        reg_leds = 0x1F;
        print("Booting...\n\n ");

        reg_leds = 0x7F;
        print("Press ENTER to continue...");

        //-- Wait until /n or /r is received
        do {
                c = getchar_prompt(0);
        } while (c != '\n' && c != '\r');

        menu();

        while (1)
        {
                        print("Command> ");
                        char cmd = getchar();
                        if (cmd > 32 && cmd < 127)
                                putchar(cmd);
                        print("\n");

                        switch (cmd)
                        {
                        case '1':
                          menu();
                                break;
                        case '2':
                                glitchMe();
                                break;
                        case '3':
                                crashMe();
                                break;
                        default:
                                continue;
                        }

        }
}
```

Appendix D

Source code for the developed tool to count SBUs and MBUs

```
/*
* countUpsets.c
* Compare two files and find the number
* of SBUs and MBUs that are detected from
* one to the other.
*
* Pedro Javier Fernández (c) 2022
*/


#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>

// Note 1:
// For MBUs, we keep a counter for
// the different multiplicity upsets
// found. Because this tool compares
// at a byte (8-bit) level, the
// maximum multiplicity of an MBU
// is 8. (i.e.: no need to keep
// track of multiplicity >= 8 MBUs).

// Note 2:
// Multiplicity 0 means no upsets.
// Multiplicity 1 is the same as an SBU.

typedef struct MBUCounter
{
  #define MAX_MULTIPLICITY 8
```

```
  unsigned int counter[MAX_MULTIPLICITY+1];
} MBUCounter;


////////////////////////////////
// Helper functions
////////////////////////////////
void do_xor(uint8_t* output, uint8_t a, uint8_t b)
{
    *output = a ^ b;
}



void findUpsets(unsigned char* golden, unsigned char* after,
                unsigned int  size,
                unsigned int* sbu_count,
                MBUCounter*   mbu_count)
{
    // Local variables
    uint8_t xor_result = 0;
    int    local_count = 0;

    // Improvement: for large files, unsigned will warp around...
    for(unsigned int x = 0; x < size; x++)
    {
            local_count = 0;

            // Compute number of upsets
            do_xor(&xor_result, golden[x], after[x]);

            // If both bytes are equal, the result of the xor will be 0
            // Otherwise, count the number of upsets (which is equal to
            // the number of bits that are set)
            if(xor_result!=0)
            {

                // Brian Kernighan's Algorithm
                // The loop is executed as many times as set bits (1's)
                // are in the byte.
                while(xor_result)
                {
                xor_result &= (xor_result - 1);
                local_count++;
                }

                // If only one bit was set, then there was only one upset
                if(local_count==1)
```

```
                    (*sbu_count)++;
                else if(local_count > 1) // only count multiplicity >=2
                    mbu_count->counter[local_count]++;
            }
    }
}

// Expected arguments
// ./sbucount <goldenFile> <afterFile>
int main(int argc, char *argv[])
{
    // Local variables
    FILE* golden_file;
    FILE* after_file;
    struct stat golden_stats;
    struct stat after_stats;
    unsigned char* golden_data;
    unsigned char* after_data;
    unsigned int sbu_num = 0;
    MBUCounter mbu_counts;

    // Init counters
    for(int i = 0; i < MAX_MULTIPLICITY+1; i++)
            mbu_counts.counter[i] = 0;

    // Check parameters
    if(argc!=3)
    {
        printf("[!] invalid number of parameters\n");
            printf("[i] usage: ./sbucount <goldenFile> <afterFile>\n");
        return 1;
    }

    // Open golden & after files
    golden_file = fopen(argv[1], "rb");
    if (!golden_file) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }

    after_file = fopen(argv[2], "rb");
    if (!after_file) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }

    // Sanity check both files are equal in size
    if (stat(argv[1], &golden_stats) == -1) {
```

```
        perror("stat");
        exit(EXIT_FAILURE);
    }

    if (stat(argv[2], &after_stats) == -1) {
        perror("stat");
        exit(EXIT_FAILURE);
    }

    printf("golden size = %ld\n",golden_stats.st_size);
    printf("after  size = %ld\n\n",after_stats.st_size);
    if(golden_stats.st_size != after_stats.st_size)
    {
        printf("[!] Files have unequal sizes\n");
        return 1;
    }

    // Load both files into memory
    golden_data = malloc(golden_stats.st_size);
    after_data = malloc(after_stats.st_size);

    fread(golden_data, golden_stats.st_size, 1, golden_file);
    fread(after_data, after_stats.st_size, 1, after_file);


    // Analyze
    findUpsets(golden_data, after_data, golden_stats.st_size, &sbu_num, &mbu_counts);

    // Print results
    printf("Number of SBUs: %d\n", sbu_num);

    for(int i = 2; i < MAX_MULTIPLICITY+1; i++)
            printf("Number of multiplicity %d MBUs: %d\n", i, mbu_counts.counter[i]);

    // Cleanup
    free(golden_data);
    free(after_data);

    return 0;
}
```