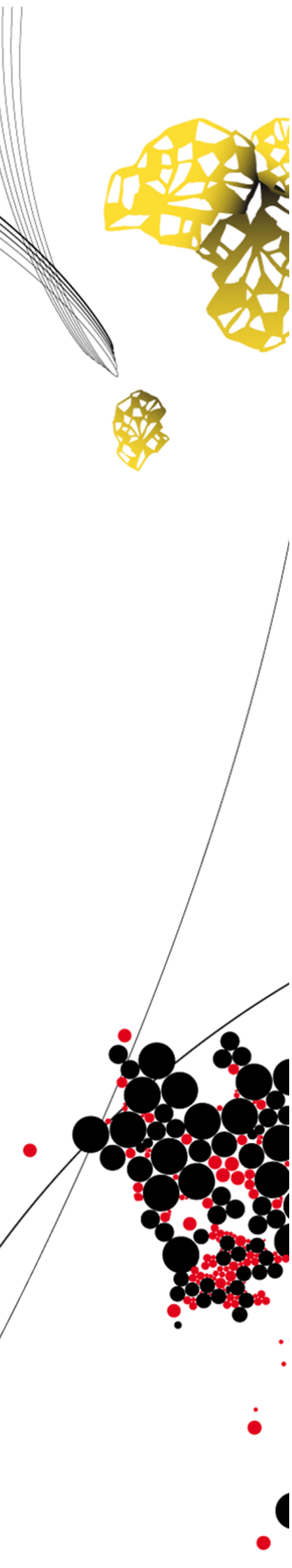


UNIVERSITY OF TWENTE.

**Faculty of Electrical Engineering,
Mathematics & Computer Science**

Preventing soft-errors and hardware trojans in embedded RISC-V cores

Edian B. Annink

MSc. Thesis
May 9, 2022

Supervisor:
Dr. ir. M. Ottavi
Committee:
Dr. ir. G. Furano
Dr. ir. E. Hakkennes
ir. S. Di Mascio
Dr. ir. A. Menicucci
ir. E. Molenkamp
Dr. ir. G. Rauwerda

Computer Architecture for Embedded Systems (CAES)
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

Single-event upsets and multiple-bit upsets that are part of single-event effects, cause bit-flips and hence lead to data corruption. Therefore, devices that are deployed in harsh environments such as in space use fault-tolerant processors or redundancy methods to ensure hardware reliability. Another serious vulnerability is the introduction of hardware trojans. Besides environmental side-effects, an adversary that has injected a malicious mechanism e.g., in the processor or memory can trigger unwanted behavior or leak sensitive information. Techniques to prevent or mitigate hardware trojans are important to ensure hardware security. Proprietary solutions exist in the market that introduces fault-tolerance or security extensions to establish this. Openness is important to prevent monopolistic proprietary solutions and create alternative solutions, such as an analogy of what happened in the world of Operating Systems; Windows NT (proprietary OS and kernel) versus Linux (open-source kernel). This is where the open RISC-V instruction set architecture becomes relevant. A novel solution to improve the security and reliability of RISC-V soft-cores with a low area and latency overhead was introduced in this thesis. The *instruction validator* which is the first part of this solution can effectively detect hardware trojans and multiple-bit upsets in the instruction memory by checking instruction/address pairs using a Bloom filter probabilistic data structure. The second part of the solution is the proposal of an error correction code instruction memory using Hamming single-error correction to detect and correct single-event upsets. It has also been proven that the Hamming decoder improves the detection performance of the instruction validator. An automation framework was introduced to generate, simulate and synthesize the instruction validator for different configurations which presents the designer with different options based on the application requirements. Besides this automation framework, two BF optimizations were proposed that decrease the BF area overhead. The instruction validator and error correction code instruction memory were successfully tested and integrated with the FreNox RISC-V core on an FPGA fabric. This resulted in a low area and latency overhead which makes it suitable to use with embedded RISC-V soft-cores that have strict security and reliability requirements.

Contents

1 Introduction **8**
- 1.1 Organization . . . 8
- 1.2 Background . . . 9
- 1.3 Fault/Threat model . . . 11
- 1.4 Report outline . . . 13

2 Related work **15**
- 2.1 The relevance of RISC-V . . . 15
- 2.2 SEU/MBU correction and detection techniques . . . 17
- 2.3 HWT detection and prevention techniques . . . 18
- 2.4 Objectives . . . 19

3 Theoretical background **21**
- 3.1 Probabilistic data-structures . . . 21
  - 3.1.1 Bloom filter . . . 22
  - 3.1.2 Quotient filter . . . 23
  - 3.1.3 Cuckoo filter . . . 24
  - 3.1.4 Concluding summary . . . 26
- 3.2 Error correction codes . . . 27
  - 3.2.1 Hamming SEC/SEC-DED codes . . . 28
  - 3.2.2 Hsiao SEC-DED codes . . . 29
  - 3.2.3 Bose–Chaudhuri–Hocquenghem (BCH) codes . . . 30
  - 3.2.4 Concluding summary . . . 31

4 Design **33**
- 4.1 Criteria . . . 34
- 4.2 Instruction validator . . . 36
  - 4.2.1 Non-cryptographic hash functions . . . 37
  - 4.2.2 Instruction validator abstraction . . . 39
  - 4.2.3 Hardware design . . . 42
  - 4.2.4 Reconfigurable hardware . . . 46

4.3 ECC instruction memory 48
4.4 Concluding summary 49

5 Simulation **51**
5.1 No faults 54
5.2 Injecting faults 55
5.3 Triggering HWTs 62
5.3.1 Injecting HWT 63
5.3.2 Modifying HWT 65
5.4 Concluding summary 66

6 Implementation **68**
6.1 Synthesis results 68
6.2 Isolated implementation 71
6.2.1 Test setup 72
6.2.2 Test results 73
6.3 Integration with FreNox 77
6.3.1 FreNox fault-injection setup 80
6.3.2 Testing without introducing faults and HWTs 81
6.3.3 Introducing a sequence of faults 81
6.3.4 Introducing single-event faults 84
6.3.5 Introducing HWTs 84
6.4 Concluding summary 85

7 Conclusion and future work **87**
7.1 Conclusion 87
7.2 Future work 89
7.2.1 Handling the illegal signal 89
7.2.2 Further research on the effect of Hamming decoding on the instruction validator 89
7.2.3 Adding multiplier stages for high-performance embedded systems 89
7.2.4 Researching the effectiveness of this proposal in ASICs 89

Bibliography **91**

Appendices

A Injecting faults without Hamming decoder **97**

B Diagrams **100**
B.1 Negative slack diagram 100
B.2 Instruction validator schematic 102
B.3 Instruction validator synthesized 104
B.4 Instruction validator debug 106

Acronyms

ALU	**A**rithmetic **L**ogic **U**nit
ASIC	**A**pplication-**S**pecific **I**ntegrated **C**ircuit
AXI	**A**dvanced e**X**tensible **I**nterface
BBF	**B**uffered **B**loom **F**ilter
BCH	**B**ose-**C**haudhuri-**H**ocquenghem
BF	**B**loom **F**ilter
BOOM	**B**erkeley **O**ut-of-**O**rder **M**achine
BRAM	**B**lock **R**andom **A**ccess **M**emory
CAES	**C**omputer **A**rchitecture for **E**mbedded **S**ystems
CBF	**C**ounting **B**loom **F**ilter
CCF	**C**as**c**ade **F**ilter
CF	**C**uckoo **F**ilter
CPU	**C**entral **P**rocessing **U**nit
CRC	**C**yclic **R**edundancy **C**heck
DSP	**D**igital **S**ignal **P**rocessing
DEC	**D**ouble-**E**rror-**C**orrection
DED	**D**ouble-**E**rror-**D**etection
DPRAM	**D**ual-**P**ort **R**andom-**A**ccess **M**emory
DSP	**D**igital-**S**ignal **P**rocessing
DUT	**D**esign-**U**nder-**T**est
DoS	**D**enial-**o**f-**S**ervice
ECC	**E**rror-**C**orrection **C**odes
EEMCS	**E**lectrical **E**ngineering, **M**athematics and **C**omputer **S**cience
FBF	**F**orest-structured **B**loom **F**ilter
FF	**F**lip**F**lop
FPP	**F**alse **P**ositive **P**robability
FPR	**F**alse **P**ositive **R**ate
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
HDL	**H**ardware **D**escription **L**anguage
HWT	**H**ard**w**are **T**rojan

IC	**I**ntegrated **C**ircuit
ICT	**I**nformation and **C**ommunication **T**echnology
IF	**I**nstruction **F**etch
IM	**I**nstruction **M**emory
I/O	**I**nput/**O**utput
ILA	**I**nternal **L**ogic **A**nalyzer
ISA	**I**nstruction **S**et **A**rchitecture
JTAG	**J**oint **T**est **A**ction **Group**
LFSR	**L**inear **F**eedback **S**hift **R**egister
LSB	**L**east **S**ignificant **B**it
LUT	**L**ook**U**p **T**able
MBU	**M**ultiple **B**it **U**psets
MSB	**M**ost **S**ignificant **B**it
PC	**P**rogram **C**ounter
PCB	**P**rinted **C**ircuit **B**oard
QEC	**Q**uadruple-**E**rror-**C**orrection
QED	**Q**uadruple-**E**rror-**D**etection
QF	**Q**uotient **F**ilter
RAM	**R**andom **A**ccess **M**emory
RISC-V	**R**educed **I**nstruction **S**et **C**omputer-**Five**
ROM	**R**ead **O**nly **M**emory
RDL	**R**egister **D**escription **L**anguage
RTL	**R**egister-**T**ransfer **L**evel
R&D	**R**esearch & **D**evelopment
SDC	**S**ilent **D**ata **C**orruption
SEC	**S**ingle-**E**rror-**C**orrection
SED	**S**ingle-**E**rror-**D**etection
SEE	**S**ingle **E**vent **E**ffects
SEU	**S**ingle **E**vent **U**psets
SoC	**S**ystem-**o**n-**C**hip
SRAM	**S**tatic **R**andom-**A**ccess **M**emory
SSD	**S**olid **S**tate **D**rive
TMR	**T**riple **M**odular **R**edundancy
TEC	**T**riple-**E**rror-**C**orrection
TED	**T**riple-**E**rror-**D**etection
UART	**U**niversal **A**synchronous **R**eceiver-**T**ransmitter
ULSI	**U**ltra **L**arge-scale **I**ntegrated **C**ircuits
USB	**U**niversal **S**erial **B**us
VHDL	**VHSIC H**ardware **D**escription **L**anguage

VLSI **V**ery **L**arge-scale **I**ntegrated **C**ircuits
WNS **W**orst **N**egative **S**lack

List of Figures

1.1 Harvard architecture diagram 11
1.2 Instruction memory HWTs classified by the HWT taxonomy [1] 12

3.1 BF example 22
3.2 Hamming(7, 4) SEC 29
3.3 Hamming(8, 4) SEC-DED 29

4.1 Proposed solution in a Harvard architecture 33
4.2 CF and BF area overhead compared to IM 35
4.3 High-level hardware design of the instruction validator and ECC instruction memory with the RISC-V core 36
4.4 Hamming distance and normal distribution of four hash functions . . . 38
4.5 Instruction validator with CRC-32C hash 42
4.6 Instruction validator with MultiplyShift hash 42
4.7 Pipeline with a latency of three clock cycles 45
4.8 Pipeline with a latency of four clock cycles 45
4.9 One VHDL generation execution 47
4.10 ECC flow 49

5.1 Generating all program/hash/optimization configurations 51
5.2 Cocotb simulation setup 52
5.3 Testing all program/hash/optimization configurations 53
5.4 Flipping single bits in the 38-bit codeword 55
5.5 Flow of method that introduces MBUs 56
5.6 Hamming distance of single output bits 61
5.7 Sample set #1 62
5.8 Sample set #2 62
5.9 Cocotb flow of HWT test cases 63
5.10 RV32I instruction types [2] 65

6.1 Isolated implementation diagram 72
6.2 Isolated implementation bus layout diagram 73
6.3 Isolated implementation test flow 74

6.4 Fetching legal instructions 75
6.5 Fetching illegal instructions 75
6.6 Fetching one illegal instruction from the first address 75
6.7 Illegal signal zoomed in 76
6.8 Fetching one illegal instruction from the first address 76
6.9 Illegal signal zoomed in 76
6.10 FreNox hardware overview 77
6.11 FreNox fault-injection with saboteur and PRNG 80
6.12 FreNox test setup 81
6.13 Instruction validator counter register value after executing program . . 82
6.14 FreNox single-bit error injection 82
6.15 FreNox double-bit error injection 82
6.16 FreNox triple-bit error injection 83
6.17 FreNox trap entry infinite loop 83
6.18 FreNox trap entry infinite loop remains after disabling fault-injection . . 83
6.19 FreNox trap entry infinite loop after injecting a sequence of double-bit errors 83
6.20 Double- and triple-bit errors trap entry 84
6.21 Double- and triple-bit errors trap handling 84
6.22 Hardware Trojan resulting in QuickSort malfunctioning 85
6.23 Hardware Trojan waves 85

List of Tables

3.1 Performance comparison 26
3.2 ECC comparison for 32-bit data word 32

4.1 Design criteria 34
4.2 Testing elements in a BF executing 1000 runs using different hash functions. 41

5.1 Double-bit errors test case results of 10 runs with $\epsilon = 0.05$ without optimization using CRC-32C 57
5.2 Triple-bit errors test case results of 10 runs with $\epsilon = 0.05$ without optimization using CRC-32C 57
5.3 Hamming decoder introducing an extra fault 58
5.4 Double-bit errors test case results of 10 runs with $\epsilon = 0.05$ and m-k optimization using the MultiplyShift hash 59
5.5 Triple-bit errors test case results of 10 runs with $\epsilon = 0.05$ and m-k optimization using the MultiplyShift hash 59
5.6 Double-bit errors test case results of 10 runs with $\epsilon = 0.05$ and rounding optimization using the MultiplyShift hash 60
5.7 Triple-bit errors test case results of 10 runs with $\epsilon = 0.05$ and rounding optimization using the MultiplyShift hash 60
5.8 Results of injecting HWT test case using CRC-32C 64
5.9 Results of the injecting HWT test case using MultiplyShift and the m-k optimization 65
5.10 Results of the injecting HWT test case using MultiplyShift and the rounding optimization 65
5.11 Results of modifying HWT test case using CRC-32C 66
5.12 Results of the modifying HWT test case using MultiplyShift and the m-k optimization 66
5.13 Results of the modifying HWT test case using MultiplyShift and the rounding optimization 66

6.1 Rijndael AES synthesis results with 100MHz constraint, all configurations, and $\epsilon = 0.05$ 70
6.2 Blowfish synthesis results with 100MHz constraint, all configurations, and $\epsilon = 0.05$ 70
6.3 Dijkstra synthesis results with 100MHz constraint, all configurations, and $\epsilon = 0.05$ 70
6.4 FFT synthesis results with 100MHz constraint, all configurations and $\epsilon = 0.05$ 70
6.5 Patricia synthesis results with 100MHz constraint, all configurations and $\epsilon = 0.05$ 70
6.6 Quicksort synthesis results with 100MHz constraint, all configurations, and $\epsilon = 0.05$ 71
6.7 SHA synthesis results with 100MHz constraint, all configurations, and $\epsilon = 0.05$ 71
6.8 Comparing synthesis results to other checkers with Dijkstra and $\epsilon = 0.01$ 71
6.9 Synthesis results of FreNox SoC-e 79

A.1 Double-bit errors test case results of 10 runs with $\epsilon = 0.05$ without optimization and Hamming decoder using CRC-32C 97
A.2 Triple-bit errors test case results of 10 runs with $\epsilon = 0.05$ without optimization and Hamming decoder using CRC-32C 97
A.3 Double-bit errors test case results of 10 runs with $\epsilon = 0.05$ and m-k optimization using the MultiplyShift hash and without Hamming decoder 98
A.4 Triple-bit errors test case results of 10 runs with $\epsilon = 0.05$ and m-k optimization using the MultiplyShift hash and without Hamming decoder 98
A.5 Double-bit errors test case results of 10 runs with $\epsilon = 0.05$ and rounding optimization using the MultiplyShift hash and without Hamming decoder 98
A.6 Triple-bit errors test case results of 10 runs with $\epsilon = 0.05$ and rounding optimization using the MultiplyShift hash and without Hamming decoder 99

Chapter 1

Introduction

This chapter represents the introduction and the plan of approach. The first section *Organization* discusses where the master thesis took place and who was involved. The second section *Background* presents the general topics and ideas that led to this thesis. The third section *Fault/Threat model* covers the fault model of single-event upsets (SEU) and multiple-bit upsets (MBU) and the threat model of HWTs. What effect do SEUs, MBUs, and HWTs have on a Harvard CPU architecture? Finally, the thesis outline is discussed.

1.1 Organization

This thesis was executed by Embedded Systems Master's student ing. Edian B. Annink at the University of Twente located in Enschede and Technolution B.V. located in Gouda. At the University of Twente, the research group *Computer Architecture for Embedded Systems* (CAES) is involved which is part of the Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS). The thesis was supervised by dr. ir. M. Ottavi who is an associate professor at the University of Rome Tor Vergata and an associate professor in the CAES group at the University of Twente. The committee consists of the following examiners:

- Dr. ir. M. Ottavi (University of Twente and the University of Rome Tor Vergata)
- ir. E. Molenkamp (University of Twente)
- Dr. ir. A. Menicucci (Delft University of Technology)

The committee consists of the following advisors:

- ir. S. Di Mascio (Delft University of Technology and European Space Agency)
- Dr. ir. G. Furano (European Space Agency)

- Dr. ir. G. Rauwerda (Technolution B.V.)
- Dr. ir. E. Hakkennes (Technolution B.V.)

Currently, CAES employs 34 people. The research group started with energy efficiency as the main research area concerning *energy-efficient processing and communication sub-systems for battery-powered embedded systems, such as mobile phones and wireless sensor networks* [3]. The research has been extended to the following two main research areas [3]:

- Efficient architectures and tools for streaming applications
- ICT for energy management in buildings and smart grids

Technolution B.V. currently employs 250 people. Technolution was founded by four engineers in 1987 who saw more potential in software and electronics than the current state-of-the-art at the time.

The Technolution team is passionate about technology and works on innovative products, systems, and technologies for a wide range of clients. They are skilled in many disciplines (e.g. electronics, embedded hardware/software, programmable logic, and application software). Besides these skills, they master all aspects of data chains in almost any context, from data acquisition to access and process management.

Technolution contributes its expertise to customers in multiple industries, such as mobility (traffic management and *Smart Cities*), energy (*Smart Grids*, energy-transition projects), high-tech and big science (advanced meteorology equipment), manufacturing (automation, artificial intelligence, and computer vision) and high assurance (preventive security for classified data) [4].

1.2 Background

Moore's law states that the number of transistors in an integrated circuit (IC) doubles every 24 months. While this is no longer true [5], this exponential growth resulted in more transistors per die. ICs became more sophisticated over time and very large-scale integrated circuits (VLSI) were introduced followed by ultra-large-scale integrated circuits (ULSI). This increase in capacity resulted in more hardware per die and an increased complexity which introduced negative side-effects such as security issues (HWTs) and reliability issues such as single-event upsets (SEUs) and multiple-bit upsets (MBUs). More hardware per area and smaller noise margins mean that SEUs and MBUs occur earlier with a smaller charge. A more detailed explanation will be presented later.

A well-known hardware security issue is a hardware trojan (HWT) which is a *malicious, intentional modification of a circuit design that results in undesired behavior when the circuit is deployed* [6]. HWTs can lead to catastrophic system failures depending on the type of HWT [1]. The first HWT was discovered in a Syrian radar system. A suspected Syrian nuclear installation in the northeast was bombed in September 2007. The Syrian radar system didn't detect this attack. The Syrian radar consisted of commercially available microprocessors that supposedly contained a hidden backdoor and temporarily disabled the system. A United States defense contractor didn't confirm this specific event but confirmed that European chip manufacturers built microprocessors containing kill switches. This kill switch could disable microprocessors remotely when falling into the wrong hands [7]. According to [6], the fabrication of an IC contains many steps in the following sequential order: specifications, register-transfer level (RTL) design, netlist, physical Design, fabrication, assembly, and market. The complexity of each step in the fabrication process makes it difficult to prevent HWTs. Reducing cost and a fast time to market (TTM) often forces research and development (R&D) departments to buy intellectual property circuits from other companies which increases the risk of HWTs even further [8]. Other important phenomena that affect hardware reliability are SEUs and MBUs.

SEUs and MBUs are both subsets of Single-Event Effects (SEE) and both cause a *temporary change of memory contents or commands in an instruction stream* [9]. SEUs and MBUs in space originate from heavy ions coming from cosmic rays or high-energy protons coming from solar flares. SEUs and MBUs can also occur from secondary cosmic rays which can reach the Earth's surface. A famous example of a SEU caused by a secondary cosmic ray is the occurrence of a bit-flip in an electronic voting system in Belgium which resulted in 4096 more votes [10]. MBUs cause two or more bit-errors per word. According to [9] four types of MBUs exist:

- An incoming particle passing through adjacent cells
- Diffusion of charge to closely spaced junctions upsetting more than one bit
- An ionization cloud overlapping two or more sensitive regions
- Two random particle hits occurring in different bits in the word during a given time period

SEUs and MBUs often result in data corruption which may lead to system malfunctioning. While radiation hardening leads to fewer SEU and MBU cases in spacecraft, it is important to find other ways to mitigate or decrease SEUs and MBUs in digital circuits. Especially systems that can have a big impact on the environment and human lives such as space, missile, and avionics systems [9].

1.3 Fault/Threat model

Fault model

It is unlikely that one solution covers all HWTs, SEUs, and MBUs. A fault model and threat model must be introduced to get an overview of the behavior of HWTs, SEUs, and MBUs. Consider the Harvard CPU architecture displayed in Figure 1.1.

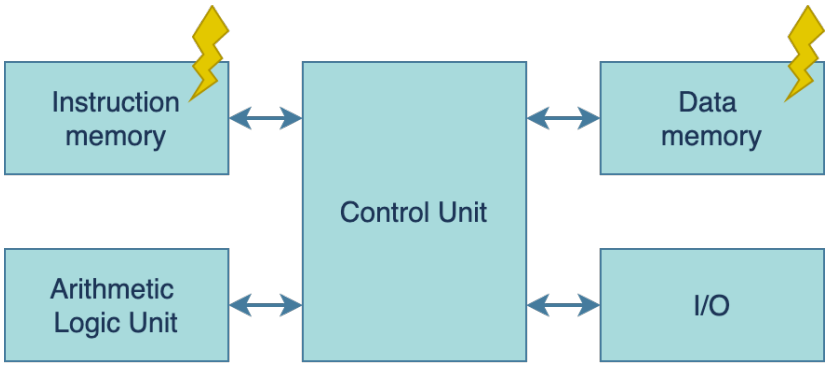


Figure 1.1: Harvard architecture diagram

An example of a Harvard-based architecture is RISC-V which stands for *Reduced Instruction Set Computer 5*. RISC-V is an open instruction set architecture (ISA) that was introduced in 2010 by the University of California located in Berkeley. The RISC-V ISA is also subjective to reliability and security issues. Current security and reliability problems in RISC-V will be discussed later on as well as the relevance of RISC-V in this thesis.

Two components in the Harvard architecture can be influenced by SEUs and MBUs: Instruction memory and data memory. This means that the reliability of the instruction and data memory cannot be guaranteed in this case.

While the data memory might also be affected by SEUs, MBUs, and HWTs, this thesis focuses on SEUs, MBUs, and HWTs that occur in the instruction memory.

Threat model

In [1] it is shown that different types of HWTs exist. The presented taxonomy shows that HWTs can be classified by looking at the *insertion phase*, *abstraction level*, *activation mechanism*, *effects*, and *location*. As mentioned before, this thesis focuses on the presence of HWTs in the instruction memory. This is an important scope as HWTs could also be implemented in other components of the processor. For example, an HWT can modify the functionality of the registers or the arithmetic logic unit (ALU). This means that HWT detection located in the instruction memory can be bypassed as instructions are changed after the instruction fetch (IF) phase of the

processor. Figure 1.2 displays the proposed threat model of this type of HWT. This so-called instruction memory HWT is part of the class ($\in$) or is not part ($\notin$) of the class in each attribute.

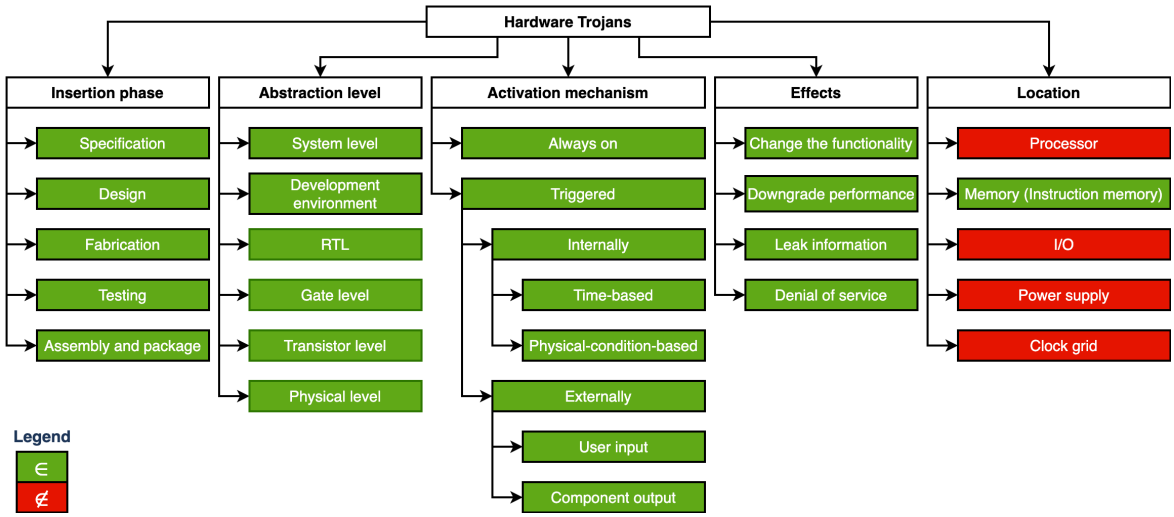


Figure 1.2: Instruction memory HWTs classified by the HWT taxonomy [1]

An HWT that resides in instruction memory can be inserted in every phase. A requirement in the specification phase can be added that simplifies adding HWTs in a later phase. A possible example is that a third-party memory IP block is used in the design phase that injects malicious instructions into the instruction memory. Another example is that a developer can add malicious HDL code that implements an HWT in the instruction memory. In the worst case, an alternative photomask can be used to change or replace the instruction memory. If the testing phase is also modified to prevent the detection of this malicious change, an HWT can be introduced in the fabrication phase of the IC. The same is the case for assembly and packaging. If the memory is separated from the processor on a PCB, a malicious memory component can be used.

The HWT can also be implemented at every abstraction level. This is the case because the instruction memory can be maliciously modified or replaced on every level.

This specific HWT also supports every activation mechanism or *trigger mechanism*. The HWT can be always-on, internally triggered, and externally triggered.

All effects, also called *payloads*, are supported. The functionality can be changed such as changing instructions or injecting malicious instructions. Performance can be downgraded by spamming instructions or repeating instructions. Information can be leaked by injecting instructions that copy sensitive data to memory addresses that can be read by the adversary. Denial-of-Service (DoS) is also a possibility, e.g., completely disabling the instruction memory. No instructions can be written to the

instruction memory and reading the instruction memory results in undefined signals which completely halts the pipeline.

This HWT only resides in the instruction memory and hence the other locations are not part of the classification.

1.4 Report outline

Chapter 2 *Related work* discusses the related work. The related work is divided into state-of-the-art regarding the prevention of SEUs, MBUs, and HWTs in the RISC-V architecture. This creates a clear gap in current research that ultimately leads to a discussion and the definition of the objectives of this thesis.

Chapter 3 *Theoretical background* discusses the theoretical background of the main techniques that were used in this thesis: Probabilistic data structures and error correction codes (ECC).

Chapter 4 *Design* discusses the design proposal to achieve the objectives using criteria and by weighing different options. This chapter introduces the instruction validator and ECC high-level design and hardware design which checks instruction/address pairs. Besides the instruction validator, the automation framework which can generate the instruction validator using disassembled RISC-V programs is introduced. The ECC instruction memory is also discussed which is implemented in both software and hardware.

Chapter 5 *Simulation* discusses the simulations using automated cocotb simulations. A test harness is introduced to simulate both the instruction validator and ECC separately and combined. Benchmark programs were used and compiled with the RISC-V toolchain and used in all the simulations. HWT, SEU, and MBU were injected in different simulations to evaluate the instruction validator and ECC efficiency regarding improving the security and reliability in RISC-V soft-cores.

Chapter 6 *Implementation* discusses the implementation of the instruction validator. Firstly, the synthesis results of the instruction validator configurations are discussed. Secondly, an isolated implementation of the instruction validator was created to test the instruction validator separately from the RISC-V on an FPGA fabric. This isolated implementation was tested using a logic analyzer. The implementation also discusses the synthesis results of different instruction validator configurations using different hashes, optimizations, and benchmark programs. Finally, the instruction validator was integrated into the FreNox SoC-e system-on-chip (SoC) with the FreNox RISC-V soft-core.

Chapter 7 *Conclusion and future work* conclude the thesis. The main results of the thesis are discussed and a reflection on the objectives is presented. This chapter also discusses the future work of this thesis: What can be improved and

what are the next steps? Possible new research topics can be materialized based on this future work.

Chapter 2

Related work

This chapter describes the related work and objectives of this thesis. The state-of-the-art is discussed to be able to conclude the current gap in research that is partially filled by this thesis. First, the relevance of RISC-V to this thesis is discussed. Secondly, SEU/MBU correction and detection techniques related to RISC-V and other architectures are discussed. Thirdly, HWT detection and prevention techniques related to RISC-V and other architectures are discussed. Finally, the objectives are discussed based on the current gap in state-of-the-art techniques to mitigate SEUs, MBUs, and HWTs.

2.1 The relevance of RISC-V

This thesis focuses on the RISC-V open ISA. As mentioned previously, RISC-V standing for Reduced Instruction Set Computer 5 is an open instruction set architecture (ISA) that was introduced in 2010 by the University of California, Berkeley. Later, the RISC-V foundation was founded in 2015 with Technolution being one of the founding members. The main goal of the RISC-V Foundation is to promote RISC-V [11]. An open ISA means that companies and academia can create hardware implementations based on this ISA specification without paying royalties or using proprietary tools. Instead, companies and academia are free to create their RISC-V core or System-on-Chip (SoC) based on the RISC-V specification and can contribute to RISC-V itself and the software tools that come with it [12].

The goals behind RISC-V are as follows [13]:

- Must be open and freely available to academia and industry.
- Must be suitable for direct native hardware implementation.
- Must be specified independently of microarchitecture styles or implementation technology.

- Must be separated into a base integer ISA that can be extended with custom accelerators and standard extensions.
- Must support the revised 2008 IEEE-754 floating-point standard.
- Must support extensive ISA extensions and specialized variants.
- Must have 32-bit and 64-bit address space variants for applications, operating system kernels, and hardware implementations.
- Must have support for highly parallel multicore or manycore implementations, including heterogeneous multiprocessors.
- Must have optional variable-length instructions to both expand available instruction encoding space and to support an optional dense instruction encoding for improved performance, static code size, and energy efficiency.
- Must be fully virtualizable to ease hypervisor development.
- Must simplify experiments with new privileged architecture designs.

RISC-V has an unprivileged and privileged specification. According to the specification [13]: *The RISC-V privileged architecture covers all aspects of RISC-V systems beyond the unprivileged ISA, including privileged instructions as well as additional functionality required for running operating systems and attaching external devices*. RISC-V consists of 32- and 64-bit base integer ISA and extensions. By the time of writing this thesis, RISC-V consists of the following ratified base integer ISA:

- RV32I Version 2.1
- RV64I Version 2.1

RISC-V consists of the following ratified standard extensions that can be used to extend the base ISA [13]:

- M: Integer Multiplication and Division Version 2.0
- A: Atomic Instructions Version 2.1
- F: Single-Precision Floating-Point Version 2.2
- D: Double-Precision Floating-Point Version 2.2
- Q: Quad-Precision Floating-Point Version 2.2
- C: Compressed Instructions Version 2.0

The main reason to focus on RISC-V is its openness and RISC-V's future in industries that require computer systems that provide high reliability and security such as the aerospace industry [14].

Like any other ISA, RISC-V is also subject to research that improves its security and reliability. It has been proven that HWTs, SEUs, and MBUs cause major security vulnerabilities and reliability issues [6] [9].

2.2 SEU/MBU correction and detection techniques

Recent studies [15]–[17] show that the number of fault-tolerant RISC-V cores that prevent SEUs and MBUs is still limited. While this is still true, the number of researchers and the industry that is developing fault-tolerance solutions for RISC-V is growing. The first example of fault-tolerant RISC-V cores is the RISC-V core protected by Triple Modular Redundancy (TMR) and Hamming codes based on the unprivileged specification proposed by Santos et al. [16]. A second example is an addition of ECC-protected memory to the out-of-order Rocket core *BOOM* by Berkeley proposed by Dörflinger et al. [15]. Gaisler Cobham [18] who is known for developing fault-tolerant processors and fault-tolerant IPs for space recently released NOEL-V which is their implementation of the RISC-V specification. While this core currently is not fault-tolerant by design, it will soon get support for fault-tolerance. Ramos et al. [19] researched the impact of SEUs on multiple soft processors using SRAM-based FPGA implementations including the lowRISC SoC. SEUs were introduced using the Soft Error Mitigation (SEM) IP of Xilinx. The conclusion was that *Application Output Mismatches* caused by Silent Data Corruptions (SDC) and Hangs (infinite loop) were the most common faults besides hard faults (exception) and *Architecture Internal Failures* which means that the output is correct, but the internal state of the architecture isn't. They ultimately claim that *fault-tolerant techniques should be applied to lowRISC if it is going to be used in space missions*. A study by A. E. Wilson et al. [20] tested the fault-tolerance of RISC-V soft-cores on Xilinx SRAM-based FPGAs. They have proven that while reliability is improved when using TMR, the reliability of the core is still limited by multiple factors including MBUs affecting two or three TMR domains. Another study by M. Ottavi et al. [21] investigates a signature-based checker that mitigates SEUs in a complex instruction set computer (CISC): The Intel 8051 8-bit microcontroller. This checker checks the control flow integrity by analyzing the signature that is created for every sequence of instructions before every program branch. This signature is generated by linear feedback shift registers (LFSR) and is compared with pre-loaded signatures. An error is raised if the signature doesn't exist. This checker provided an average of 98.86% coverage, a high level of protection against freezes, and a correlation of

50% between control flow errors and wrong computations. A comparison will be presented later which includes this checker.

2.3 HWT detection and prevention techniques

The development of defense mechanisms against Hardware Trojans is relatively lagging behind according to a recent survey on RISC-V security regarding hardware and architecture [22]. This survey features multiple proposals that try to detect HWTs in RISC-V. Linscott et al. [23] focus on HWTs that are introduced in the fabrication process of silicon. The proposal is to mitigate HWTs by *mapping the security-critical portions of a processor design to a one-time programmable, LUT-free fabric*. This results in an area overhead of 27% when using the Rocket BOOM RISC-V core. Takahashi et al. [24] propose two detection methods based on machine learning and side-channel analysis. The methods were successful in detecting HWTs in PicoRV and Freedom RISC-V cores. The third proposal by Bolat et al. [25] introduces a protection architecture to detect HWTs in the instruction and data memory in RISC-V using a Bloom filter (BF). Hoque et al. [26] introduce a new HWT class that targets SRAM arrays. They conclude that these HWTs can evade *industry-standard post-manufacturing testing*. A study by A. Palumbo et al. [27] introduces a protection architecture like the architecture proposed by Bolat et al. [25]. This checker fragments instruction/address pairs into multiple data chunks. So for example fragmenting a 64-bit instruction/address pair into four 16-bit vectors that can serve as 16-bit addresses to a bit array. The instruction/address pairs that are stored in the instruction memory are stored in these bit arrays. When instructions are being fetched, the checker determines if each address corresponding to the fragmented instruction/address pair is '1' in their respective bit array.

A general idea besides the mentioned checkers would be to introduce a CRC-based checker that functions as a checksum for the instruction memory. This however causes several issues. The first issue is that the checksum can only be checked after all the instructions are fetched by the processor. A solution would be to preload the instruction memory and calculate and validate the checksum, initially and after every fetched instruction. This proves the continuous integrity of the instruction memory. However, this still doesn't solve the first issue and results in a large overhead in terms of latency.

The number of HWT, SEU, and MBU countermeasures in RISC-V is still limited and results in a large overhead in terms of area and latency according to recent studies and surveys. A lot of work must yet be done to ensure that RISC-V-based ASICs and softcores are fault-tolerant and resistant to HWTs. As RISC-V is becoming an industry standard and even the standard in future aerospace systems [14]

this must become the norm. The paper from Di Mascio et al. [28] focuses on the future of space systems and how RISC-V can fit in with current systems. Besides that, they portray the importance of alternatives to current monopolistic proprietary architectures in aerospace in which RISC-V plays an important role.

2.4 Objectives

To conclude, the current techniques that are used to mitigate SEUs, MBUs, and HWTs are still limited and result in large overhead in terms of area and latency. The objective of this thesis is to materialize a multi-purpose solution that mitigates SEUs, MBUs, and HWTs in the instruction memory of an embedded RISC-V core considering the proposed fault and threat model. Another goal is to achieve an overhead that is as low as possible in terms of area and latency.

Redundancy techniques can be divided into the following two categories that are relevant for the proposed problem: Hardware redundancy and information redundancy.

Hardware redundancy techniques to mitigate SEUs and MBUs such as dual modular redundancy and triple modular redundancy need double or triple the amount of hardware which results in large area overhead and high manufacturing cost. Error correction codes (ECC) that mitigate SEUs in RISC-V cores do not result in a large area overhead as mentioned previously.

Information redundancy adds information to the original data to be able to detect and possibly correct single- and/or multiple-bit errors. Multiple checkers that check instruction/address pairs of the sequence of instructions were discussed earlier and can be categorized as information redundant solutions. These checkers result in the lowest amount of area overhead with the introduction of a small amount of latency or ideally no latency at all and will be evaluated and compared next.

First, let's consider the signature-based checker introduced by M. Ottavi et al. [21]. While the signature-based checker's goal was to mitigate SEUs, its effectiveness against HWTs will also be evaluated to be able to compare this checker against checkers that mitigate HWTs. The signature-based checker is only able to check the program flow by analyzing a sequence of addresses pointing to instructions that are fetched from the instruction memory. This means that instructions can still be injected. The checker only verifies the signature when a conditional instruction is fetched. This means that its possible to inject instructions that are not conditional until the watchdog is triggered. It can be concluded that this checker is not resilient against HWTs. To improve the detection of HWTs and SEUs, a copy of the instruction/address pairs must be stored in a redundant memory and every instruction/address pair must be evaluated when fetched by the core.

Secondly, two checkers were introduced by Bolat et al. [25] and A. Palumbo et al. [27] that store instruction/address pairs using different methods. The checker introduced by Bolat et al. introduces a checker that stores a compressed copy of the instruction memory in a BF probabilistic data structure and checks the instruction/address pair with every instruction fetch. The checker introduced by A. Palumbo et al. uses fragmentation and multiple data chunks to store instruction/address pairs. This checker was able to detect all instructions injected by an HWT outside of the legal program memory space. However, an HWT that changes instructions during the IF phase, was detected with a false positive rate of 0.4% until 3.91%. This checker consumes a smaller number of lookup tables (LUTs) (0.49% vs 5.83%/10.19%) and flip-flops (FFs) (0.31% vs 0.85%/0.90%) than the BF-based checker.

However, it must be noted that this checker consumes significantly more Block RAM (BRAM) (208 vs 32 and 64 kbit) than the BF-based checker. The reason for this significant difference is that the BF-based checker compresses the instruction/address pairs while the fragmentation checker allocates a fixed memory size to store instruction/address pairs.

While considering the mentioned redundancy techniques, the following main research question proposes a multi-purpose solution to mitigate SEUs, MBUs, and HWTs:

How can HWTs and MBUs be detected and SEUs be detected and corrected in the instruction memory of an embedded RISC-V core using redundant memory in combination with ECC and instruction/address hashing?

To find a solution that answers this main question, multiple sub-questions were formulated:

- What ECC implementation is the most effective in detecting/correcting SEUs and results in the lowest amount of overhead in terms of area and latency as possible while taking the used probabilistic data structure into account?
- What is the most effective way to store the hashed instruction/address pairs to be able to detect HWTs and MBUs while creating as less overhead in terms of area and latency as possible?
- What information redundancy configuration using ECC and hashed instruction/address pairs creates the least amount of overhead in terms of area and latency while being able to detect HWTs and MBUs and detect/correct SEUs?

Chapter 3

Theoretical background

This chapter covers the theoretical background of the thesis. To know how HWTs can be detected and SEUs/MBUs can be detected and possibly corrected, relevant ECC techniques and probabilistic data structures must be researched. First, probabilistic data structures will be analyzed and finally, error correction codes will be discussed.

3.1 Probabilistic data-structures

Hashed instruction/address pairs can be generated by a hash function. The idea behind hashed instruction/address pairs is to create a lossy redundant database that holds these hashes which function as identifiers of instruction/address pairs. This database can be used to check if the instruction/address pair that is fetched in the IF phase either contains faults (SEUs or MBUs) or is an instruction triggered by an HWT. Checking if an element is part of a dataset is also known as the *membership problem* [29].

Among all the probabilistic data structures the following most common data structures and their variants will be analyzed:

- Bloom filter
- Quotient filter
- Cuckoo filter

Besides that, a concluding summary will be presented that concludes which probabilistic data structure would be the most fitting to use in the context of checking instruction/address pairs.

3.1.1 Bloom filter

The Bloom filter (BF), introduced by Howard Bloom in 1970 [30], is the most used data structure that solves the previously mentioned *membership problem*.

The BF has been invented as a space-efficient probabilistic data structure. This means that this BF can determine if an element is part of the dataset using a much smaller area than other conventional methods such as hash tables or linked lists. This makes it very suitable for embedded applications as resources are often limited.

The BF is represented by a bit array. The classical BF only supports insertion and testing. This means that elements can only be inserted and can be tested if they are present in the BF. Before the elements are added to the BF, each element is hashed using multiple hash functions. To insert the element, all bit positions in the bit array that match the hash outputs (modulo the bit array size) are set to 1. To check if an element is in the BF, all bit position that corresponds to the hash outputs (modulo the bit array size) of that element are checked if they are 1. Figure 3.1 displays a visual example of a BF. This BF has a bit array size of 16 and uses three hash functions. In this case, element x is hashed three times using different hash functions and is inserted in the bit array by changing the pointed bit array positions to 1. These positions can again be checked to check if element x is present in the BF. The modulo operation is used to stay within bounds as hash functions often output a decimal number that is higher than the bit array size.

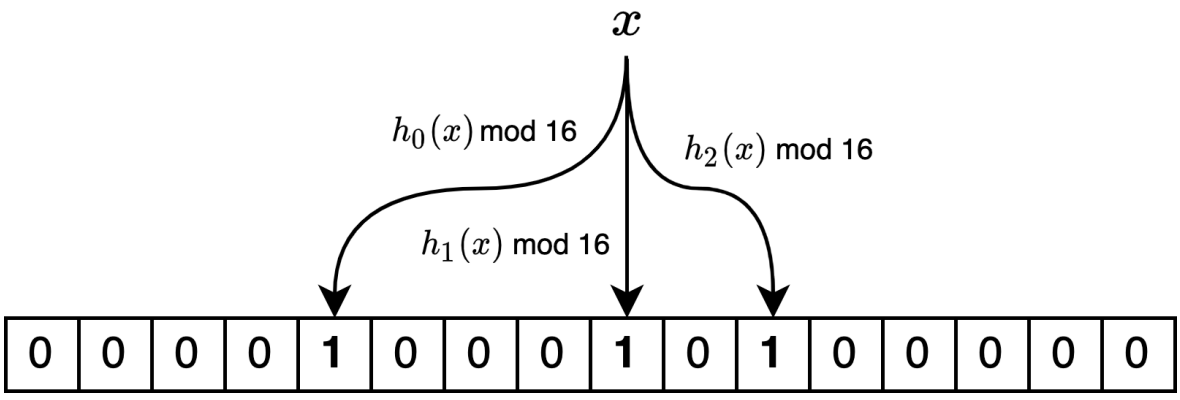


Figure 3.1: BF example

The trade-off of this technique is that the BF introduces a small percentage of errors increasing with the number of elements in the filter, also known as the false positive probability (FPP) (i.e. the BF returns that an element is part of the set while it was not inserted). Therefore it's important that the number of hash functions used and the length of the BF are big enough for the expected number of elements that will be added.

Equations 3.1, 3.2 and 3.3 specify the how to compute the FPP ϵ, the optimal total bit array size m based on the FPP and the number of elements and the optimal number of hash functions k based on the total bit array size and number of elements [29]:

$$\epsilon \approx \left(1 - e^{-\frac{kn}{m}}\right)^k \tag{3.1}$$

$$m = -\frac{n \ln(\epsilon)}{\ln(2)^2} \tag{3.2}$$

$$k = \frac{m}{n} \ln(2) \tag{3.3}$$

Counting Bloom filter

The counting Bloom filter introduced by L. Fan et al. in 2000 [31] is a modification of the Bloom filter introducing an array of counters per bit instead of just an array of bits like the Bloom filter. Using counters introduces the possibility to delete elements. This comes however with the important condition that the element must exist in the dataset. This means that it might be necessary to test if the element is present before deleting it. The negative side-effect of using counters instead of single bits is that a lot more space is needed that holds the counter bits. Another problem is that the counters can overflow if the number of assigned bits per counter is too small. This can be prevented by picking a large enough counter, e.g. 4 bits per counter.

A wide range of other variants of the Bloom filter exists. For example, the buffered Bloom filter (BBF) proposed by M. Canim et al. [32] which uses a buffer space in RAM to decrease I/O to and from storage. Another variant is the forest-structured Bloom filter (FBF) proposed by G. Lu et al. [33] which also uses a combination of RAM and flash memory, mainly to improve the lookup performance on flash storage. Many other Bloom filters exist that serve one optimization purpose or serve multiple purposes in one implementation [34].

3.1.2 Quotient filter

An alternative probabilistic data structure to the BF and its variants is the Quotient filter (QF) introduced by M. Bender et al. in 2012 [35].

The QF stores a p-bit fingerprint f for each element. It is called the *Quotient* filter as it uses Donald Knuth's *quotienting*. The least significant bits (LSB) of the fingerprint are used as the remainder $f_r = f \mod 2^r$ and the $q = p - r$ most significant bits (MSB) are used to construct the quotient $f_q = \lfloor \frac{f}{2^r} \rfloor$. The fingerprint is stored by storing f_r into bucket $T[f_q]$. The fingerprint can be restored as follows: $f = f_q 2^r + f_r$ [35].

The Quotient filter offers *comparable performance to the Bloom filter in terms of space and time, but with better data locality*. The Quotient filter offers multiple advantages over the classic Bloom filter such as *cache friendliness, in-order hash traversal, resizing, merging, and deletion* [35]. Multiple variants of the Quotient filter are:

- Quotient filter (QF) - Designed to run on RAM
- Buffered quotient filter (BQF) - Designed to run on flash memory
- Cascade filter (CCF) - Designed to run on flash memory

To compare the Bloom filter and the Quotient filter, in-RAM and SSD experiments were conducted by M. Bender et al. It is important to note that QF can only be filled for 75% to remain efficient. This was also reflected in the experiments. The in-RAM experiments in [35] showed that *QFs outperform BFS by factors of 1.3$\times$ to 2.5$\times$, depending on the false-positive rates. For uniform random lookups, BFs are 1.4$\times$-1.6$\times$ faster. For successful lookups, there was no clear winner*.

The SSD experiments in [35] showed that when using a *RAM-to-filter* ratio of 1:4, *both BQF and CCF insert at least 4 times faster than other data structures and that BQF is at least twice as fast for lookups as the BFs. In fact, on successful lookups, it runs roughly 11 times better than EBF and BBF.*

3.1.3 Cuckoo filter

The Cuckoo filter proposed by B. Fan et al. in 2014 [36] also stores fingerprints like the QF. First, a fingerprint f of the element is created. The primary bucket location is derived by hashing the element. Relocation is essential for Cuckoo hashing. The Cuckoo filter only stores fingerprints and there is no way to restore the original elements and re-hash them to find their new bucket in the hash table. The solution is to use ”Partial-key Cuckoo hashing”. The Cuckoo filter computes two candidate buckets $h_1(x) = \text{hash}(x)$ and $h_2(x) = h_1(x) \oplus \text{hash}(f)$. If the candidates are occupied, an alternate location is computed as displayed in eq. 3.4 using the current element's index i and fingerprint f [36].

$$j = i \oplus \text{hash}(f) \tag{3.4}$$

The Cuckoo filter does lookup and deletion by checking the two candidate buckets. If and only if the element x was previously inserted, a copy of the fingerprint f can be removed.

The following paragraphs evaluate the performance of the CF compared to the other probabilistic data structures.

False-positive rate ϵ w.r.t. bits per element

The QF uses extra meta-bits to navigate through the data structure which leads to 10-25% more memory than the space-efficient BF while achieving the same false-positive rate [36]. This means that the BF can achieve a lower false-positive rate while using the same amount of memory.

The semi-sort CF and CF both can achieve a higher false-positive rate than the considered BF implementations while using fewer bits per item for low false-positive rate applications. The semi-sort CF uses fewer bits per item when $\epsilon \leq 3\%$ and the CF when $\epsilon \leq 0.39\%$ (see Figure 4.2 when using 2 candidate buckets and 4 fingerprints per bucket. The CF implementations even achieve a higher space-efficiency than the *d-left Counting Bloom filter* which uses a similar approach as the CF.

Another research conducted by P. Reviriego et al. [37] compares the false positive rate w.r.t. table occupancy of both the Bloom filter and Cuckoo filter. They conclude that while the false positive rate of the Cuckoo filter increases linearly, the Bloom filter increases more steeply and still performs better when the table occupancy is lower than 80%, 85%, and 90% corresponding to fingerprint sizes 12, 15, and 18. This corresponds to a false positive rate range of 0.2% - 0.003%.

The performance of the Cuckoo filter compared to the previously mentioned probabilistic data structures was evaluated by B. Fan et al. by the means of a benchmark. Multiple aspects were evaluated that will be discussed onwards. The benchmarks were executed on a general-purpose CPU: The Intel Xeon L5640 running at 2.27 GHz with 12MB L3 cache and 32 GB RAM [36].

Space efficiency and construction speed

It has been proven by experiments that the semi-sort CF was able to contain the highest number of elements while using the lowest number of bits per element. The semi-sort CF also consisted of the lowest false positive rate compared to the other probabilistic data structures. The blocked BF however had the highest construction speed as the blocked BF *operates on a single cache line for each query* [36].

Insertion

BFs have a constant insertion rate as the length of the filter is fixed and the bit array is overwritten with the hashed element. This is why the BFs have a constant insertion rate. The CF and semi-sort CF both decrease in insertion performance as the occupancy increases. This has to do with the characteristic that more elements in the CF result in more relocations of buckets to insert elements in their designated positions [36].

Testing/Lookup

The CF has a slightly higher throughput on existing elements lookup than the blocked BF. The CFs remain constant on lookups regardless of the occupancy as the same number of elements are checked on every query. The CF was outperformed by the blocked BF when checking for items that do not exist in the dataset. The BF outperformed the semi-sort CF in the same test. However, the CF outperformed the blocked BF and the semi-sort CF outperformed the BF when checking for items that do exist in the dataset [36]. This means that based on the application, the CF or the blocked BF is the best choice when lookup performance is key.

Deletion

The CF provided the highest deletion throughput. The QF and counting BF performed similarly with a low occupancy but diverged with increased occupancy. The QF performed better than the semi-sort CF until $\approx 45\%$ of the filter was filled with elements and the throughput declined significantly. The CF is the clear winner in this case [36].

3.1.4 Concluding summary

This thesis focuses on embedded RISC-V cores. Embedded systems often run one dedicated program for a certain process or system. This means that the instruction memory layout remains static. This is an important factor to compare the different probabilistic data structures and conclude which probabilistic data structures would be the best candidate for checking instruction/address pairs for this type of application.

Table 3.1 shows a table containing a general overview of the performance per property of every probabilistic data structure. As mostly existing instructions/pairs will be tested apart from corrupt or injected instructions part of instruction/address pairs, only the testing of existing elements is considered in the table.

	Properties				
Filter	**Space efficiency**	**Construction speed**	**Insertion**	**Testing (existing elements)**	**Deletion**
BF	++	+	+	++	n.a.
Counting BF	– –	+	–	+	+
Blocked BF	–	++	++	++	n.a.
CF	++	+	+	++	++
semi-sort CF	++	–	–	–	–
QF	– –	– –	– –	– –	– –

Table 3.1: Performance comparison

The most important factor in embedded systems is space efficiency. Resources in terms of area and power are often scarce and must be compensated by algorithms or data structures that are designed for embedded systems. When looking at the table the most space-efficient data structures are the Cuckoo filter, the semi-sort Cuckoo filter, and the Bloom filter.

As mentioned before, it is assumed that the instruction memory doesn't change during runtime. This means that the insertion of instruction/address pairs in the probabilistic data structure only must be done once. This is similar to loading the instructions in instruction memory before the first instruction is fetched and executed. This fact makes the construction speed and insertion less important as inserting the elements can be done before implementation.

Testing if an instruction/address pair exists is evident to check for HWTs, MBUs, and possibly SEUs. This must be checked as fast as possible to minimize the damage or faults that HWTs, MBUs, and SEUs introduce. The checking procedure must not influence the achievable clock speed of the core and must not influence the core at all. The blocked BF and the Cuckoo filter are the fastest in terms of testing them for existing elements. The Bloom filter isn't fast in software implementations as multiple hash functions must be computed before the element can be tested in the bit-array. However, hardware implementations can benefit from parallelism and the hashes can be computed concurrently while introducing slightly more area and power as shown in the study of A. Bolat et al. [25].

The deletion of instruction/address pairs is unnecessary as it is assumed that the same instruction/address pairs are used throughout the whole lifetime of the embedded system. The probabilistic data structure must however be easily reconfigurable to be able to change the instruction/address pairs, e.g. when the program has been modified.

To conclude, when taking all previously mentioned factors into account, the most suitable probabilistic data structure would be the Cuckoo filter or the Bloom filter. They both have the best space efficiency and testing performance. Both filters will be analyzed and compared in more detail in the chapter 4: *Design* which discusses the proposed design.

3.2 Error correction codes

As mentioned previously, bit-flips in memory can be caused by SEUs and MBUs. So-called "Error Correction Codes" (ECC) exist that can detect and correct single or multiple bit-flips. This section evaluates several ECC techniques that are suitable for use with embedded RISC-V cores. Small descriptions of each ECC will be provided. The section will be concluded with a comparison that compares the different ECCs.

3.2.1 Hamming SEC/SEC-DED codes

The first error correction code was invented by Richard Hamming at Bell Labs in 1950 [38]. The Bell Telephone Laboratories' *Model 5 Relay Computers* consisted of 8900 relays. Two to three relays failed per day [38]. This was a problem as the computers ran programs overnight and were unable to recover the errors on their own which required user intervention. This led to the Hamming error correction codes that can detect and correct errors without any user intervention. Hamming introduced two variants of his error correction code called the *single error correction* (SEC) codes which detect and correct one single-bit in a n-bit *codeword*. Hamming also introduced a variant that can correct and detect a single-bit error and can detect if a double-bit error occurred called *single error-correcting + double error detecting codes* (SEC-DED). Both variants will be discussed in more detail.

Single Error Correcting codes (SEC)

As mentioned previously, parity bits are used to check if the codeword contains errors. The algorithm uses so-called *parity checks* to check the data and the parity bits for errors. Hamming uses a parity bit to check a group of bits. The sum of data bits m and parity bits k equal the n-bit *codeword* (eq. 3.5).

$$n = m + k \tag{3.5}$$

The maximum number of data bits can be calculated using the number of parity bits that are used as displayed in eq. 3.6 [38].

$$m \leq 2^k - k - 1 \tag{3.6}$$

Substituting eq. 3.5 into eq. 3.6 results in the equation displayed in eq. 3.7 which represents the maximum n-bit codeword.

$$n \leq 2^k - 1 \tag{3.7}$$

Let's consider the following example: When using 3 parity bits, the codeword is $\leq 2^3 - 1 = 7$ bits. The number of data bits are $\leq 2^3 - 3 - 1 = 4$ bits. Which can also be written as Hamming(7, 4).

The parity bit has positions that are part of the set $\{2^x | x \in \mathbb{N}, 0 \leq x \leq (k-1)\}$ [38]. This position is used to check the parity of groups of data bits that are '1' in the select position. For example in Hamming(7, 4), the three parity bits have positions $\{1, 2, 4\}$ which means that 3 parity checks are executed as displayed in Figure 3.2. The positions correspond to **1**, **1**0, **1**00 in binary. This means that the parity bit on position 1 checks positions 1, 3, 5, and 7 as their bits are 1 on the first position. For

position 2, positions 2, 3, 6, and 7 are checked as their second bit is 1. Finally, for position 3, positions 4, 5, 6, and 7 are checked.

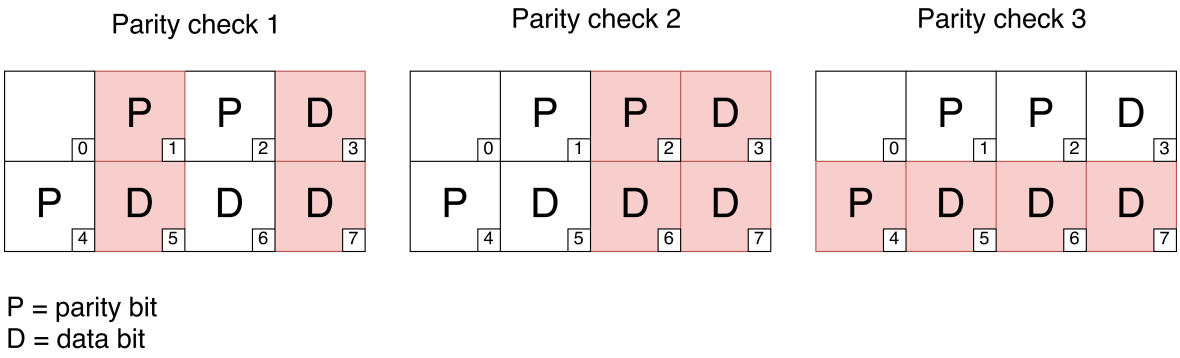


Figure 3.2: Hamming(7, 4) SEC

Single Error Correcting + Double Error Detecting codes (SEC-DED)

A double-bit error can be detected but can't be corrected. Remember that the parity bits started at position 1. A double-bit error is corrected by using position 0 of the bit-array as block parity bit and doing an extra parity-check that checks the complete block as displayed in Figure 3.3. So Hamming(8, 4) uses a total of 8 bits when using SEC-DED instead of 7 bits that were used for SEC.

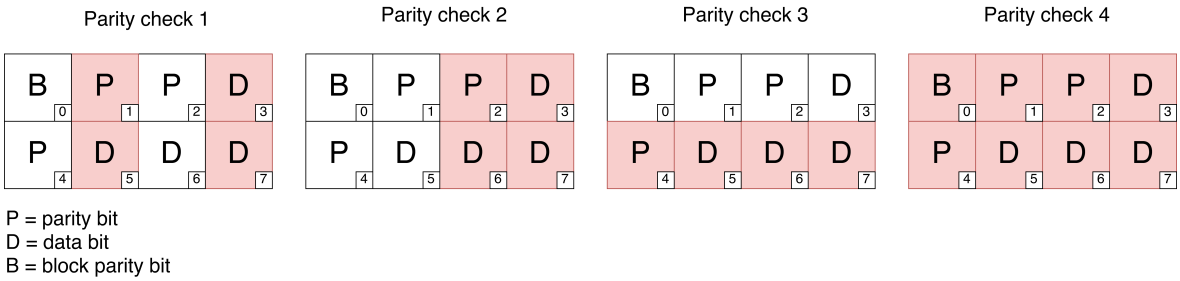


Figure 3.3: Hamming(8, 4) SEC-DED

3.2.2 Hsiao SEC-DED codes

Optimization of the Hamming SEC-DED codes was introduced by M. Y. Hsiao in 1970 [39]. This optimization uses the same relation between the number of parity bits and data bits with computer memory in mind. First, a small coding theory introduction is given before presenting this optimization. The $\boldsymbol{G}$-matrix also called the generator matrix, maps the data into a space that contains the codewords denoted as $\boldsymbol{G} = [I_k|P]$ where I_k is the $k \times k$ matrix and P is the $k \times (n-k)$ matrix. The $\boldsymbol{H}$-matrix also called the *parity-check matrix* can be denoted as $\boldsymbol{H} = [-P^T|I_{n-k}]$.

$\boldsymbol{G}$ is a $k \times n$ matrix while $\boldsymbol{H}$ is a $(n-k) \times n$ matrix [40]. With Hsiao codes, $\boldsymbol{H}$ must meet the following requirements [39]:

- There are no all-0 columns
- Every column is distinct
- Every column contains an odd number of 1's which implies an odd weight

This leads to the fact that the parity-check matrix consists of less 1's than conventional Hamming codes which implies faster decoding as was proven by experiments in a study by G. Tshagharyan et al. [41]. A trade-off can be observed from this study between Hsiao and Hamming codes which is based on the size of the codeword that is used. Hsiao became more effective in terms of logic levels and area from a codeword size of 72 bits consisting of 64 data bits and 8 parity bits. This means that Hamming(8, 4) remains more effective than Hsiao, e.g. when checking every 4-bit to be able to correct more than one bit-flip in a 16-bit word.

3.2.3 Bose–Chaudhuri–Hocquenghem (BCH) codes

BCH codes were invented by Bose, Ray-Chaudhuri, and Hocquenghem and were published in 1959 and 1960 [42] [43]. BCH codes are cyclic codes that are generated using polynomials over Galois fields. This is different compared to Hamming and Hsiao which are generated using vectors. BCH codes can be binary and non-binary by nature. This background will only focus on binary BCH codes.

Binary BCH codes are defined as follows [40]:

1. Determine the smallest *m* such that $GF(2^m)$ has a primitive n-th root of unity β.
2. Select a non-negative integer b. Frequently, $b = 1$.
3. Write down a list of $2t$ *consecutive* powers of β:

   $$\beta^b, \beta^{b+1}, ..., \beta^{b+2t-1}$$

   Determine the minimal polynomial with respect to $GF(2^m)$ of each of these powers of β. Because of conjugacy, frequently these minimal polynomials are not distinct.
4. The generator polynomial $g(x)$ is the least common multiple (LCM) of these minimal polynomials. The code is a $(n, n - \deg(g(x)))$ cyclic code.

A BCH code is called *narrow sense* when $b = 1$. When $n = 2^m - 1$ implies that the BCH code is *primitive*.

BCH codes can be generated that not only support single-error correction but can correct multiple errors which are different from SEC-DED codes which only support single-error correction.

For example, $m = 3 \implies n = 2^3 - 1 = 7$. Let β be a root of the primitive narrow-sense polynomial $x^3 + x + 1$ in $GF(2^3)$. To construct a binary single-error correcting BCH codes ($t = 1$) implies that the *consecutive powers* are β, β^2. The minimal polynomials of $GF(2^3)$ w.r.t. $GF(2)$ are equal ($m_1(x) = m_2(x) = x^3 + x + 1$). This implies that $g(x) = M_1(x) = x^3 + x + 1$. This is a 3rd-degree polynomial which implies that this is a (7, 4) code.

BCH codes can theoretically correct many more errors than just one. These codes can be constructed with t being equal to the number of error corrections. This results in more *consecutive powers*. A comparison will be presented in the conclusion that displays the results of BCH codes that correct more than one error.

Messages can be encoded by first creating a polynomial of the message m which is smaller or equal to $n-\deg(g(x))$. E.g. to encode 1010 in a (7, 4) code, the data bits can be transformed into a polynomial over $GF(2)$ called $p(x)$. The corresponding polynomial is $p(x) = x + x^3$. The codeword can be constructed by multiplying $p(x)$ with $g(x)$. Codeword $c(x) = p(x)g(x) = x^6 + x^3 + x^2 + x$ which is equal to binary 1001110.

A parallel BCH double-error correction (DEC) and DEC with triple-error correction decoder (DEC-TED) was introduced by R. Naseer et al. [44]. This decoder overcomes the multi-cycle decoding latency introduced by conventional BCH decoders. This allows for BCH codes that support multi-error correction and multi-error detection to be used in memory applications. While this proposal introduces performance improvements for mainly the decoder, the area grows significantly. This makes it less suitable for embedded systems that have strict area requirements.

A technique introduced by P. Reviriego et al. [45] allows for DEC using BCH with reduced power consumption by dynamically power gating the decoder. Experiments showed that up to 43% of the power consumption and up to 40% of the area could be reduced. While a reduction in the area could be achieved, the BCH decoder still consumes significantly more area and power than a Hamming implementation [44].

3.2.4 Concluding summary

This thesis focuses on embedded RISC-V cores. Currently, the most common embedded RISC-V soft-cores use the 32-bit *Base Integer Instruction Set* called *RV32I* [2]. This architecture uses 32-bit instructions. Table 3.2 displays a com-

parison of the different ECC with a 32-bit data word size.

ECC	Type	Codeword (bit)	Code rate
Hamming	SED	33	96.9%
	SEC	38	84.2%
	SEC-DED	39	82.1%
Hsiao	SEC-DED	39	82.1%
BCH	SEC	38	84.2%
	SEC-DED	39	82.1%
	DEC	44	72.7%
	DEC-TED	45	71.7%
	TEC	50	64%
	TEC-QED	51	62.7%

Table 3.2: ECC comparison for 32-bit data word

As mentioned previously, BCH encoders and decoders use more area than Hamming or Hsiao implementations which makes them less suitable for embedded systems that have strict area requirements. The comparison between Hsiao and Hamming depends on the double-error detection requirement. Now that the theoretical background is established, the criteria and design can be discussed. Based on these criteria and this theoretical background, a suitable ECC can be selected.

Chapter 4

Design

This chapter presents the design that introduces criteria and multi-level designs for the defined objectives and sub-objectives. Section 4.1 discusses the criteria of the design. Section 4.2 introduces the instruction validator design. Section 4.3 introduces the ECC instruction memory design.

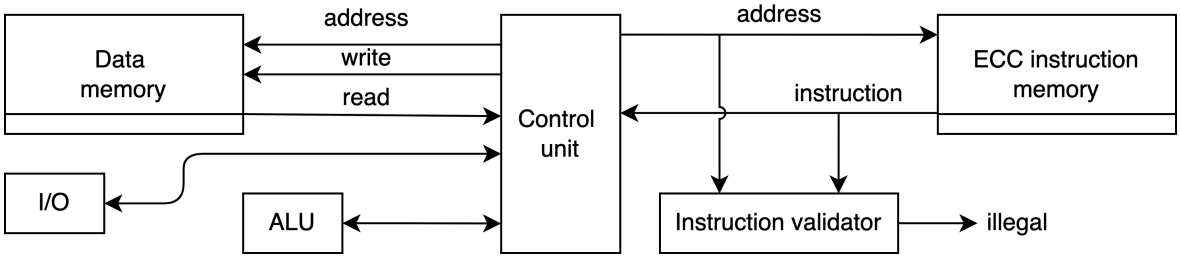


Figure 4.1: Proposed solution in a Harvard architecture

Figure 4.1 displays a high-level overview of the proposed solution. First, all the instructions are pre-loaded into the ECC instruction memory with their respective parity bits. Secondly, the instruction/address pairs are mapped onto their respective hashes generated by the hash function(s) and are also pre-loaded in memory. Finally, the processor starts fetching, decoding, and executing instructions and the instruction validator concurrently tests every instruction/address pair. After every instruction fetch, it must be checked if the instruction/address pair present on the bus is an element of the set of instructions that were initially loaded into the ECC instruction memory. If the instruction/address pair is not an element of the set, this can have the following causes:

- The instruction is either injected or manipulated by an HWT
- The instruction was manipulated by an MBU and hence couldn't be corrected by the Hamming SEC code

4.1 Criteria

Criterion	Description	Goal
ECC	Error-correction code that will be used for the instruction memory	Hamming SEC
Maximum area overhead of ECC and instruction validator	Maximum area overhead of ECC and instruction validator compared to the RISC-V core, instruction memory and data memory expressed as percentage	35%
Instruction validator latency overhead	Instruction validator affects the pipeline critical path	False
Probabilistic data-structure	Probabilistic data-structure that will be used to validate instructions	BF
ISA	Instruction set architecture that the design must comply with.	RISC-V
ISA base	ISA base that the design must comply with.	Base Integer Instruction Set RV32I - Version 2.1
ISA extension(s)	ISA extension(s) that the design must comply with.	Multiplication and Division RV32M - Version 2.0
RISC-V core	The softcore that will be used in conjuction with the design	FreNox RISC-V core
SoC	The SoC that will be used	FreNox SoC-e

Table 4.1: Design criteria

Table 4.1 displays the criteria of the design.

The instruction validator validates all the instruction/address pairs and checks for MBUs on top of the single-error correction and detection that the ECC instruction memory is handling. This means that detecting double errors using Hamming SEC-DED is redundant and using SEC has the same effect and results in less area overhead.

Hamming ECC adds check bits to correct and detect bit errors. Those check bits introduce an area overhead. A criterion was added that limits the overhead that the instruction validator and Hamming SEC introduce.

The instruction validator must not influence the timing of the RISC-V core. Checking the instruction/address pairs before decoding and executing them makes them part of the critical path which is unacceptable. The processor must not be dependent on this checking mechanism to prevent an increasing critical path.

The probabilistic data structure that will be used depends mainly on the false positive rate (FPR) as both filters perform well as mentioned previously. The FPR must be as low as possible to prevent instruction/address pairs from being marked as positive while being negative. As mentioned previously, the BF and CF are very similar in performance for low FPRs.

The performance of the probabilistic data structure performs the best when implemented in hardware is another significant factor. Most BF and CF implementations are software-based. The feasibility of using a BF hardware implementation in a RISC-V pipeline has been proven by Bolat et al. [25]. While this is the case, the

CF hardware implementation must be evaluated to be certain which filter is the most suitable to use while meeting the application criteria.

As the focus of this thesis is embedded RISC-V cores, area overhead is an important criterion. It was previously mentioned in the theoretical background, that the BF false positive probability curve is lower than the CF FPP curve for a growing table occupancy. This means that in practice, the BF has a better FPR with a growing table occupancy. While this is true, the CF consumes less memory than the BF with a maximum table occupancy of 95% and an FPP $\leq 0.39\%$. The CF and BF can both be evaluated by plotting the theoretical overhead compared to the instruction memory when varying the FPP. Figure 4.2 displays the amount of area overhead for the CF and BF compared to the instruction memory (IM) for multiple configurations. The CF starts using fewer bits per item than the BF at an FPP threshold of 0.39%. This threshold corresponds to an overhead of 35.98% without using ECC, 30.30% when using a Hamming(38, 32) SEC code, and 29.52% when using a Hamming(39, 32) SEC-DED code (see Table 3.2). An FPP threshold of 0.39% results in a large overhead of up to 35.98%. It must be concluded that for this project, the Cuckoo filter is at a disadvantage. To conclude, the BF was used as an FPP this small is outside of the scope of this project and results in an overhead that is unacceptable when considering strict memory requirements.

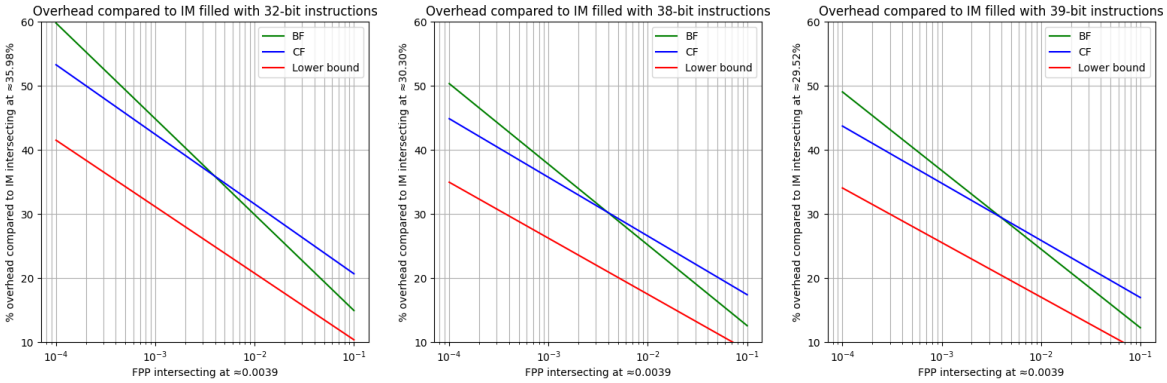


Figure 4.2: CF and BF area overhead compared to IM

The ISA that will be used is RISC-V for the reasons mentioned in Chapter 2: *Related work*.

The RISC-V ISA has different bases and extensions. For this design the *Base Integer Instruction Set* Version 2.1 with the *Standard Extension for Integer Multiplication and Division* - Version 2.0 will be the minimum criterion as the combination of this base and extension is often used in embedded RISC-V cores.

The instruction validator must be implemented with a RISC-V core to prove its effectiveness. The Technolution FreNox RISC-V core will be used for this purpose.

The FreNox SoC-e will be used to provide peripherals to the FreNox RISC-V core which is instantiated by this SoC.

Now an overview of the design is introduced and criteria are established, the next section focuses on the detailed designs of the instruction validator and ECC instruction memory.

4.2 Instruction validator

Figure 4.3 displays the instruction validator design using the BF. The first step is to hash the instruction/address pair using a non-cryptographic hash function. The second step is testing the instruction/address hashes in the BF. As proposed in the study by Bolat et al. [25], each hash function can have its own memory element that consists of a part of the total BF bit array. This separation allows for concurrently reading all the memory elements instead of reading the memory elements sequentially. A NAND gate can be connected to all outputs of the memory elements. This gate becomes high when one of the memory elements tests negative when testing instruction/address pairs.

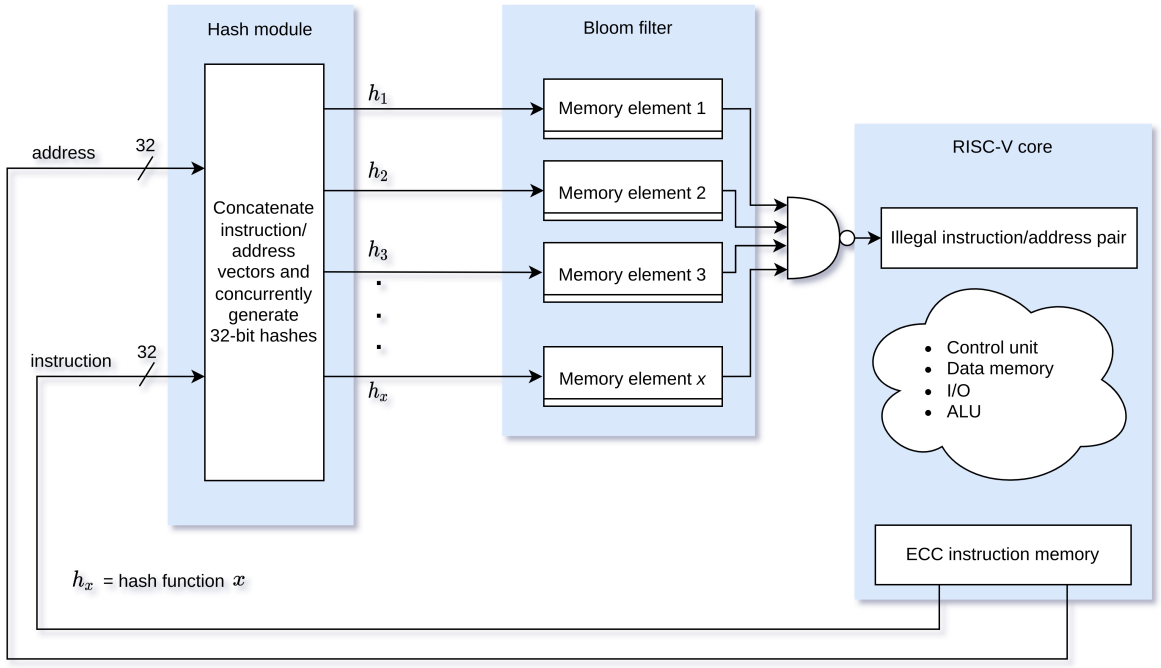


Figure 4.3: High-level hardware design of the instruction validator and ECC instruction memory with the RISC-V core

4.2.1 Non-cryptographic hash functions

Hash functions in general are functions that map and generally compress n-bits to an often smaller fixed amount of bits, commonly powers of two. The big advantage of non-cryptographic hash functions is that they are significantly faster than cryptographic hash functions as they do not need strict randomness properties. However, a non-cryptographic hash must not have a bias toward certain groups of bits and must have a sufficient uniform distribution to prevent hash collisions. Hash collisions occur when two elements output the same hash. This is problematic as each hashed element must have its own unique hash. Non-cryptographic are not collision-resistant like cryptographic hash functions are. This means that it's easier for attackers to find collisions in non-cryptographic hash functions which can lead to malicious activities such as digital signatures that appear to be legitimate but are forged [29].

This project mainly focuses on non-cryptographic hashes to achieve minimal latency and area overhead. Cryptographic hashes need solid randomization properties and hence are often complex in terms of mathematical operations and often consist of multiple stages which result in multiple cycles per hash computation. This leads to a larger area and latency overhead.

Latency overhead must be minimized as checking the instruction/address pairs should not take multiple pipeline cycles to prevent major damage caused by faults or HWTs. For this reason, hash techniques that are used in network-based FPGA applications become relevant. Two studies from R. Dobai et al. show that while CRC is not designed as a hash function, it is often used in hardware applications. A CRC-based implementation was used for a hardware implementation on an FPGA that allows for fast lookups in dynamic packet filtering [46] [47] which is comparable to a fast lookup of instruction/address pairs concurrently to a RISC-V pipeline. Another study from M. J. Lyons et al. [48] evaluates the design of a BF for ultra-low-power systems by proposing a hardware accelerator for compressed BFs. In this hardware accelerator, the Multiply and Shift hash is used which was originally introduced by Dietzfelbinger et al. [49]. The MultiplyShift hash is a universal hashing scheme and can be computed as displayed in eq. 4.1.

$$h_a(x) = \left\lceil \frac{ax \mod 2^k}{2^{k-l}} \right\rceil, \quad \text{for } 0 \leq x, a < 2^k \tag{4.1}$$

Variable a is a random k-bit odd integer and l denotes the number of output bits [50]. E.g. when hashing a 64-bit instruction/address pair as 32-bit, x denotes the 64-bit instruction/address pair and a is a random odd integer and are both within the range $0 \leq a < 2^{64}$. The modulo and division are powers of two and can be computed efficiently in hardware. The modulo limits the overflow of the multiplication. The

division can be written as a right shift as displayed in eq. 4.2 which shows the equation of hashing 64-bit instruction/address pairs resulting in a 32-bit output.

$$h_a(x) = \lceil (ax \mod 2^{64}) >> 32 \rceil \tag{4.2}$$

Avalanche effect analysis

The Avalanche effect first introduced by Horst Feistel [51] is a desired property that a hash function must comply with to be considered a hash function that consists of good randomization. The Avalanche effect states that when one input bit is changed, half the output bits should change. MurmurHash3, MultiplyShift, CRC-32C(astagnoli), and CRC-32 were analyzed using 10 million randomly generated 64-bit numbers. MurmurHash3 is a fast non-cryptographic hash function that was invented by Austin Appleby [52]. It is used in many popular software applications such as Elasticsearch [53], the Apache Commons Codec, [54] and PHP [55].

The output Hamming distances of all the hash functions are plotted together with a normal distribution in Figure 4.4. It can be observed that MurmurHash3 and MultiplyShift indeed meets the Avalanche property as the Hamming distances match the normal distribution. The CRC class functions weren't designed to be used as functions. However, it can be observed that CRC-32C and CRC-32 both roughly match the normal distribution while CRC-32C has a mean that is closer to 50% of 32-bit: 16-bit. CRC-32C uses a different generator polynomial
$x^{32}+x^{28}+x^{27}+x^{26}+x^{25}+x^{23}+x^{22}+x^{20}+x^{19}+x^{18}+x^{14}+x^{13}+x^{11}+x^{10}+x^{9}+x^{8}+x^{6}+1$
than CRC-32 which uses generator polynomial
$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1.$

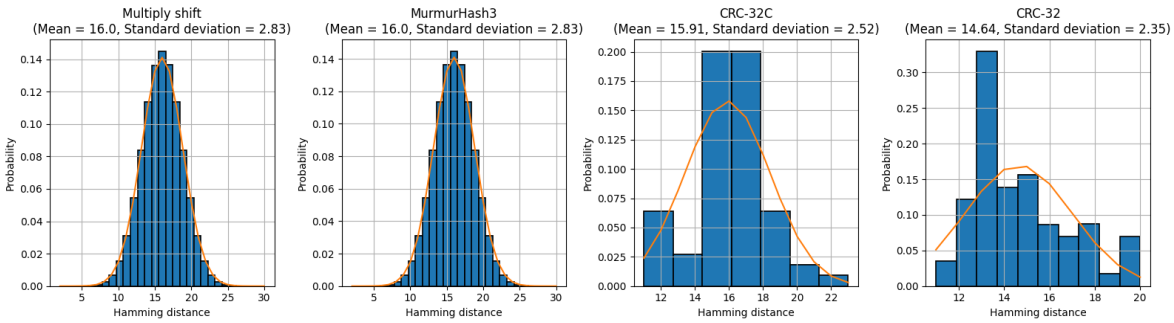


Figure 4.4: Hamming distance and normal distribution of four hash functions

The difference between these hash functions will be evaluated further in a software implementation of the instruction validator to be able to observe the distribution in the bit arrays and hence what effect the hash functions have on the FPR when using a BF.

4.2.2 Instruction validator abstraction

This subsection focuses on the software implementation of the instruction validator using the four previously mentioned hash functions. A software abstraction can be used to evaluate the performance of the hash functions together with the BF used by the instruction validator and to get a feeling for the BF and hash function parameters.

With modular programming in mind, a BF Python class was written. This same class is used throughout all Python scripts that use or analyze the instruction validator.

This BF class consists of the following methods:

- `def __init__(self, items_count, fp_prob, selecthash, optimization, crcinit=0xF)`
- `def checkcollisions(self)`
- `def insert(self, present)`
- `def test(self, listabsent)`
- `def info(self)`

and the following class methods:

- `def crc32(self, crc, p, len)`
- `def crc32c(self, crc, p, len)`
- `def multiplyshift(self, init, key, l)`
- `def get_randomkeys(self, keys)`
- `def get_fpp(self, n, m, k)`
- `def get_size(self, n, p)`
- `def get_hash_count(self, m, n)`
- `def get_optimal_mk(self, n, p)`

The `__init__()` method initializes the class. The method parameters are used to initialize the class with all necessary parameters to construct the probabilistic data structure. The `selecthash` integer selects the same hashes that were used in the Avalanche effect analysis: `mmh3.hash()`, `crc32c()`, `crc32()` or `multiplyshift()`. The `items_count` integer represents the number of elements n that will be added to the BF with desired false positive probability `fp_prob`. The `optimization` integer is

used to disable memory optimization, use m-k optimization, or round the individual bit array size to the nearest higher power of two. If the hardware optimization is set to 0, the total bit array size m is computed using `get_size()`. Based on the total bit array size, the number of hash functions k is computed using `get_hash_count()`. If the optimization is set to 1, the optimal m and k are computed using the optimization algorithm `get_optimal_mk()`. This algorithm is explained in more detail in subsection 4.2.3: *Hardware design*. Setting the optimization to 2 results in rounding the single bit array sizes to the nearest higher power of two. The idea behind this optimization is to prevent the use of the modulo operation and just use a part of the LSBs. This significantly reduces the hardware that is needed for the modulo operation while staying within the address space of the individual memory elements. This will also be explained in more detail in subsection 4.2.3: *Hardware design*. The `multiplyshift()` hash is initialized with k separate random keys using the `get_randomkeys()` method. The `get_randomkeys()` method returns random odd keys from 0 to $2^{64} - 1$ as required by the MultiplyShift hash (eq. 4.1).

The `checkcollisions()` method prints the number of hash collisions that occurred per individual bit array.

The `insert()` method inserts the 64-bit elements into the separate bit arrays by using one of the four hash functions. The hash output modulo the individual bit array sizes result in the bit array position that is changed to '1' as displayed in Figure 3.1.

The `test()` method tests if the element is present in the BF. The `listabsent` parameter contains a list of elements that were not inserted in the BF. This list is used to track the number of false positives that occur when testing elements, i.e. to compute the FPR. An example of a false positive is an element part of `listabsent` which is flagged as present in the BF.

The `info()` prints information about the BF object such as the set FPP ϵ, the theoretical FPP (eq. 3.1) based on n (the number of added elements), m (the total bit array size) and k (the number of hash functions) and the hash that is used.

Another Python script was created which creates multiple BF objects and runs simulations based on the high-level design displayed in Figure 4.3. To test if the FPP is respected, a n-amount of random 64-bit elements is inserted into the BF. A n-amount of elements is tested that were inserted in the filter to check if the element may exist in the BF. Another n-amount of elements that are different from the inserted elements are tested. There are two possibilities: the element is not present in the BF or a false positive occurs. The number of false positives is counted and divided by n to determine the FPR. The performance of the BF with every hash function was measured by taking the mean FPR over 1000 runs. An FPP of 0.01 was chosen and a varying number of elements amounting to different powers of two were inserted starting from 128 elements until 2048 elements. The results are displayed

in Table 4.2.

		Properties			
n	Hash	Mean hash collisions	m	Mean FPR	k
128	MH3	0	1226	0.0097890625	7
	CRC-32C			0.0098828125	
	CRC-32			0.010515625	
	MultiplyShift			0.0096328125	
256	MH3		2453	0.01014453125	
	CRC-32C			0.0106484375	
	CRC-32			0.01051953125	
	MultiplyShift			0.010109375	
512	MH3		4907	0.010328125	
	CRC-32C			0.010091796875	
	CRC-32			0.01012890625	
	MultiplyShift			0.01029296875	
1024	MH3		9815	0.0100205078125	
	CRC-32C			0.0103291015625	
	CRC-32			0.010068359375	
	MultiplyShift	0.001		0.010025390625	
2048	MH3	0.002	19630	0.01007568359375	
	CRC-32C	0		0.0102548828125	
	CRC-32	0.007		0.01022509765625	
	MultiplyShift	0.003		0.0101064453125	

Table 4.2: Testing elements in a BF executing 1000 runs using different hash functions.

From this table, it can be observed that the mean FPR is very close to the defined value and most of the time even below it which proves that the hash outputs have a good distribution. Hash collisions start appearing at 1024 elements for the MultiplyShift hash. The average lowest amount of hash collisions occur when using CRC-32C. This is surprising as CRC-32C is not designed as a hash function as mentioned before. The MultiplyShift hash also performs very well in terms of FPR and hash collisions compared to the other hash functions. These results show that the CRC class functions and MultiplyShift hash are indeed suitable non-cryptographic hash functions for this type of application.

It must be noted that the FPR depends on the elements that were inserted and tested. This was observed when executing multiple runs after inserting and testing random 64-bit elements, hence results were evaluated by taking the average over multiple runs. To evaluate the instruction validator with instruction/address pairs

originating from RISC-V binaries, the to-be-executed program assembly consisting of the instruction/address pairs must be extracted and models must be used that describe different HWTs. HWT models and the introduction of real instruction/address pairs parsed from disassembled RISC-V programs will be discussed in section 5: *Simulation*.

4.2.3 Hardware design

Two different hardware designs are proposed. Figure 4.5 displays the hardware design of the instruction validator based on the CRC-32C hash. Figure 4.6 displays the instruction validator hardware design based on the MultiplyShift hash.

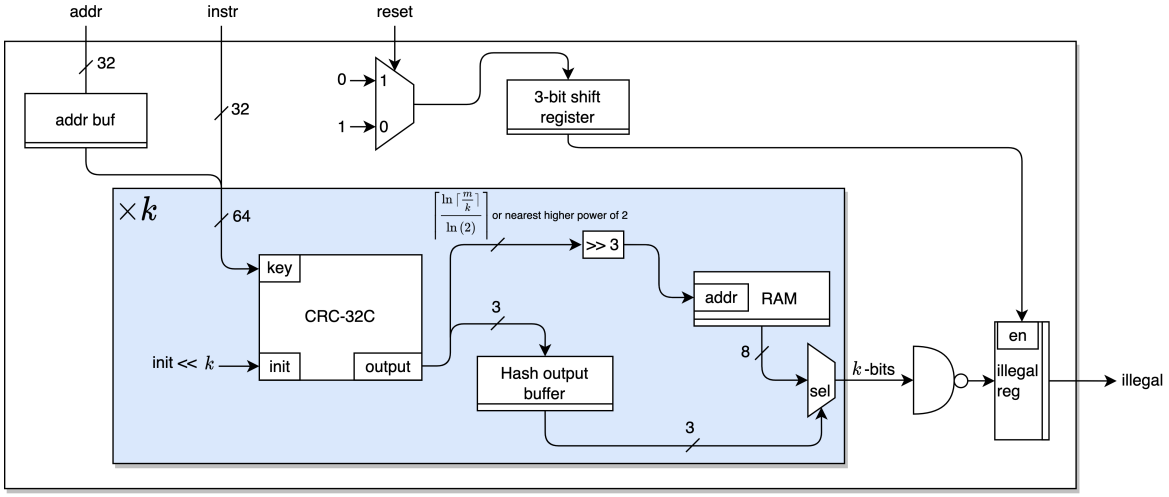


Figure 4.5: Instruction validator with CRC-32C hash

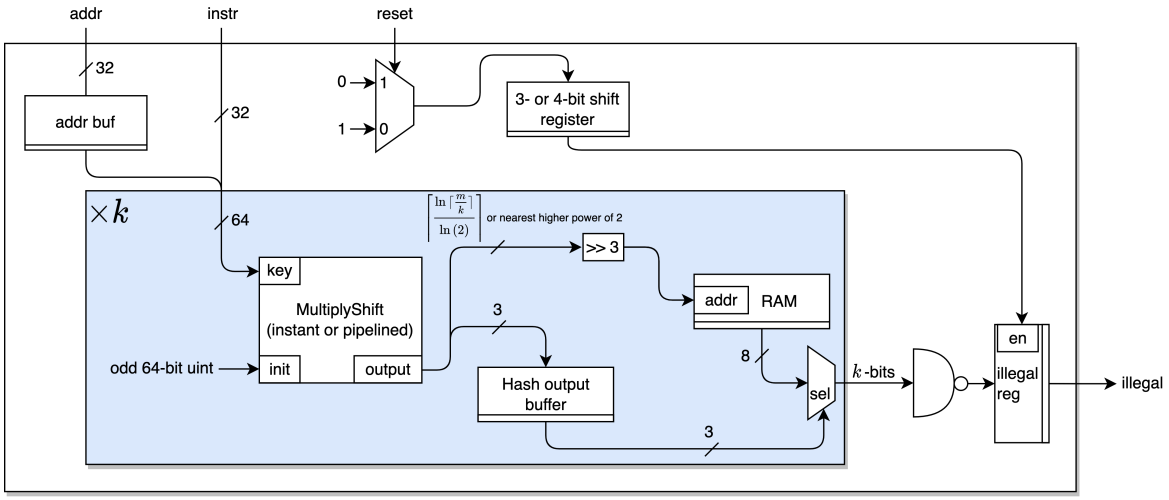


Figure 4.6: Instruction validator with MultiplyShift hash

The instruction validator has the following inputs and outputs:

- Input: Clock (clk)
- Input: Reset (reset)
- Input: 32-bit address (addr)
- Input: 32-bit instruction (instr)
- Output: Illegal instruction bit (illegal)

It is important to note that the blue block in both figures is duplicated k times. This is done to describe hashes with their corresponding memory elements in parallel as described in the high-level design displayed in Figure 4.3.

To take the one clock cycle latency of fetching the instruction from the instruction memory into account, a register `addr_buf` is used to store the address. This register can be concatenated with the respective instruction as the hash key.

The CRC-32C block contains a hash function based on CRC-32C. This hash accepts two inputs: A key and an initialization value. The key accepts 64-bit inputs and hence the address and instruction inputs must be concatenated. The initialization value is a 32-bit vector and is shifted k times to the left. This enables a good output distribution amongst the different individual hash functions. This number k portrays the number of hash functions each consisting of their own memory elements with their respective bit arrays. This block contains a clocked process that implements a linear-feedback shift register (LFSR) with the CRC-32C generator polynomial. Each bit array size is the ceiling of the total size divided by the number of hash functions. This bit array size is used to compute the modulo on the negated hash output. The modulo is computed to stay within the individual bit array address space and is assigned to the output.

The MultiplyShift block uses a clocked process. The MultiplyShift hardware can be very elegantly described. The hash has two inputs: A 64-bit key and a 64-bit vector ”init” which can be any odd 64-bit vector. The key and init vectors are multiplied while ignoring the overflow as multiplying both numbers without ignoring overflow results in a 128-bit vector. The MultiplyShift comes with two variants: Instant multiplication or pipelined. The instant variant uses only one register that stores the hash output and takes one clock cycle. The pipelined version uses more registers to relax the timing which takes two clock cycles. Instead of shifting the hash output as displayed in eq. 4.2, the hash output is sliced and assigned to the output.

As the modulo operation is expensive to realize in hardware in terms of latency and area, the best scenario is the modulo being a power of two. When this is the case, the modulo can be computed by picking the LSBs instead of using adders which significantly reduces overhead in terms of area and latency. This modulo is dependent on the total bit array size divided by the number of hash functions.

Ideally, the total bit array fits perfectly in all the separate RAM. This can be achieved by choosing an optimal m and k while respecting the FPP.

The `get_optimal_mk()` method of the BF class is used to compute the optimal m and k. This method implements the algorithm displayed in Algorithm 1.

Algorithm 1 m and k optimization with $\frac{m}{k}$ being a power of two

Require: $\epsilon \leq 1$
Require: $n \leq 2^{32}$
  $k_{opt} \leftarrow 0$
  $m_{opt} \leftarrow \infty$
  $\epsilon_{opt} \leftarrow 0$
  while $k \leq 7$ **do**
    while $x \leq 18$ **do**
      $p \leftarrow \left(1 - e^{-\frac{k \cdot n}{k \cdot 2^x}}\right)^k$
      if $p < (\epsilon \cdot 1.05) \ \wedge \ k \cdot 2^x < m_{opt}$ **then**
        $k_{opt} \leftarrow k$
        $m_{opt} \leftarrow k \cdot 2^x$
        $\epsilon_{opt} \leftarrow p$
      end if
    end while
  end while

This algorithm computes the most optimal number of hash functions and bit array size with $\frac{m}{k}$ being a power of two based on specified n and ϵ. The minimum in this optimization is the lowest total bit array size while p is lower than $\epsilon \cdot 1.05$ which represents allowing a 5% deviation. The constraints for k and $2^x = m$ can be set accordingly which are 7 and 18 in this case. Variables k_{opt}, m_{opt} and ϵ_{opt} hold the most optimal k, m and ϵ after executing this algorithm.

After the hash output with the respective modulo is computed, one byte can be fetched from the RAM. The address that stores the check bit can be determined by shifting the bit three positions to the right. After the byte is fetched, the check bit position can be determined by again taking the modulo over the check bit position. E.g., when bit 42 of 128 bits needs to be checked, the corresponding address when using one byte per address is $42 >> 3 = 5$, and the bit-position of this byte in the corresponding byte is $42 \mod 8 = 2$ which are the three LSBs of the binary representation of 42. The hash output is stored in a register to use in the next clock cycle when the byte is fetched from the RAM.

Fetching the instruction from the instruction memory and fetching the respective bit array positions from RAM results in a two-clock cycle latency. Calculating

the hashes results in one clock cycle latency. To prevent an incorrect illegal signal caused by this latency, a 3- or 4-bit shift register is used. The 4-bit shift register is only used when using the pipelined MultiplyShift hash implementation. After the reset, the shift register input becomes high and eventually, the illegal register is enabled. Figure 4.7 displays the pipeline with a three-clock cycle latency and Figure 4.8 displays the pipeline when using the pipelined MultiplyShift hash which results in a latency of four clock cycles.

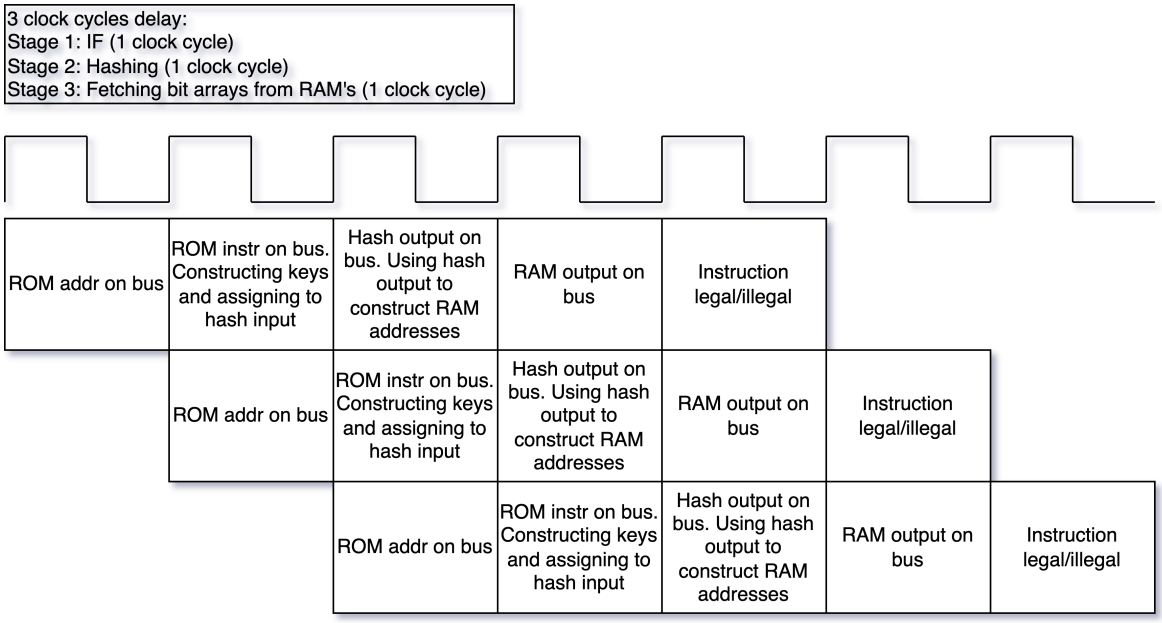


Figure 4.7: Pipeline with a latency of three clock cycles

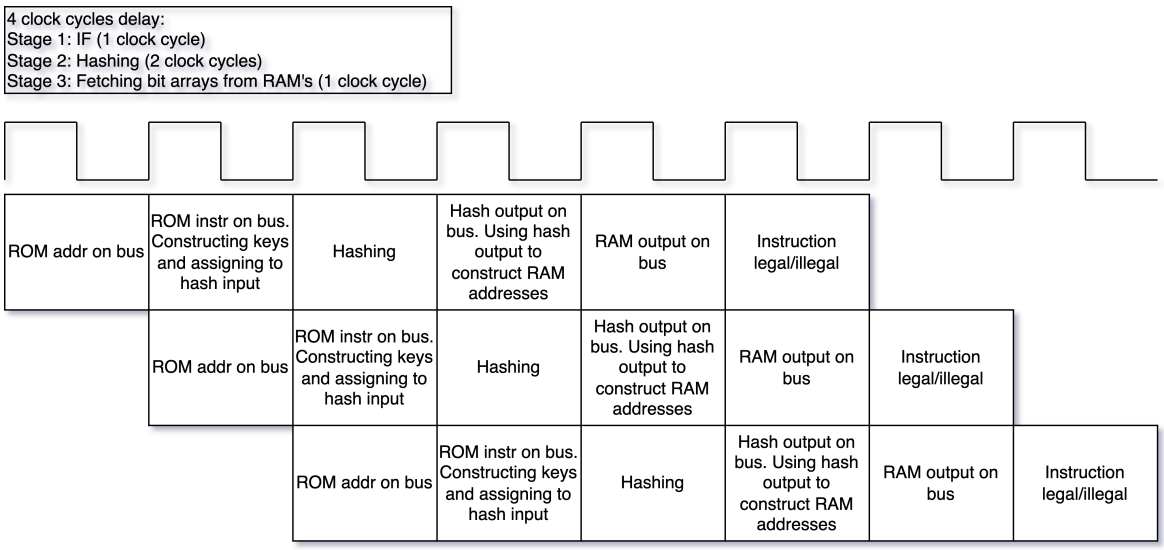


Figure 4.8: Pipeline with a latency of four clock cycles

Each hash function and its corresponding memory element output if the check

bit is indeed high. They are all connected to one NAND gate which becomes high if one of the check bits is low. This implies that the instruction/address pair is illegal and the value is propagated to the illegal register.

4.2.4 Reconfigurable hardware

The instruction validator has many parameters that can be tweaked as mentioned previously in the instruction validator software abstraction. It is for example important to be able to tweak the FPP, specify the number of elements that need to be added, and the ability to preload the bit arrays in the separate RAM based on the instruction/address pairs. Doing this manually is a tedious task and can be automated. Therefore a Python script was developed which generates all the hardware descriptions needed for the simulation and implementation. The same BF class was reused in this script which proves that the software abstraction is not only useful for doing quick simulations but this abstraction can also be used to again specify a part of the hardware behavior. Another benefit is the quick reconfiguration of the instruction validator. This script is also very useful when simulating multiple instruction validator configurations and analyzing synthesis results for different configurations which will be discussed later.

The following VHDL component files are generated by the script:

- The instruction validator itself
- Different RAM elements which store the BF bit arrays
- Instruction ROM
- Hash based on CRC-32C or MultiplyShift
- Test harness which instantiates all components mentioned above and is used as top-level in cocotb simulations

The hardware description that must be generated can be dynamically described based on all the parameters that can be tweaked in the software abstraction. The following parameters can be defined as system arguments for the generator script:

- Parse assembly? (y/n) (default=y)
- Program filename (default = primenumbers)
- Desired false positive probability 0.0 - 1.0 (default = 0.05)
- Modulo optimization mode (0 = none, 1 = m-k optimization, 2 = rounding) (default = 0)

- Number of instructions (default = 128)
- Select hash (0 = CRC-32C, 1 = MultiplyShift, 2 = MultiplyShiftPipelined) (default = 0)
- CRC initialization (default = 0xF)

The reason why the parameters are used as system arguments is to make it possible to use this script in build tools. The parameters can be specified in e.g. a Makefile and the generator script can be called with `make`. Input validation is used to ensure that the right parameters are selected and are within limits.

The generator script can generate 64-bit instruction/address pairs based on random instructions. To extend the generator script and analyze the instruction validator with RISC-V programs, a method was created that can generate a file with 64-bit instruction/address pairs per line based on a disassembled RISC-V executable. First, the source code is compiled into a binary. The next step is to disassemble this binary using `elf-objdump`. The instruction/address pairs are extracted from the assembly and are written to a file in a 64-bit binary format which can be read by the generator script.

Many configurations can be generated depending on the used program, a hash function (CRC-32C, MultiplyShift, or MultiplyShiftPipelined), and the used optimization (no optimization, m-k optimization or rounding). To simplify simulating and synthesizing the different configurations, every VHDL file belonging to that configuration has its own entity. The entity has the format `component_program_hash#_optimization#`. The hash and optimization parameters are represented as numbers matching the mentioned system argument options. To give an example, the instruction validator generated for the Rijndael AES benchmark program using the MultiplyShift hash and m-k optimization has the following entity: `instruction_validator_aes_1_1`.

The generator script also has an option to preload the instruction ROM with Hamming(38, 32) encoded instructions instead of 32-bit instructions. This is important to simulate the instruction validator with an ECC instruction ROM. This Hamming encoder implementation will be discussed in more detail in the next section.

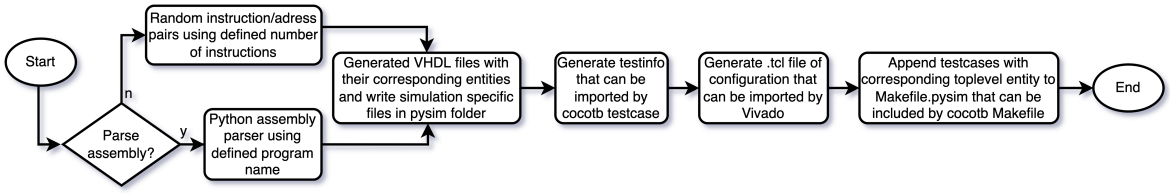


Figure 4.9: One VHDL generation execution

Figure 4.9 displays the flow of one generation execution. The user can specify

to use random instruction or parse real programs. As mentioned before, all entities have a specific format to distinguish the program/hash/optimization configuration. Simulation-specific files such as the test harnesses will be written to the pysim folder. This is done for structural reasons to separate the files needed for synthesis and additional files needed for simulation purposes. The script also generates `Makefile.pysim` which contains variables that specify all the test cases per configuration. This Makefile can later be included in the cocotb automation flow which automatically executes all test cases and writes the results to LaTeX format. This will be explained in more detail in chapter 5: *Simulation*. To automate synthesis, `.tcl` files are generated per configuration that can be used to automate the synthesis of all generated configurations. This `.tcl` file contains all necessary dependencies to synthesize every configuration. This automation was used in chapter 6: *Implementation* to synthesize all different configurations.

4.3 ECC instruction memory

As mentioned previously, Hamming SEC can be used to encode and decode 32-bit instructions. Using 5 parity bits, the number of data bits must be less or equal to $26 \leq 2^5 - 5 - 1$ (see eq. 3.6). This means that 6 parity bits must be used ($57 \leq 2^6 - 6 - 1$) denoted as a Hamming(38, 32) code. The Hamming encoding was developed as part of the Python generator script for simulation purposes. A Hamming SEC encoder method was implemented that works for every data size. This encoder works on the following basic steps that follow the Hamming principles:

- Write parity bit positions to list
- Write data bit positions to list
- Construct the codeword using the data/parity bit positions and instruction bits
- Write parity bits to codeword based on corresponding data bit groups using an XOR reduction function (see Figure 3.2)

As the instruction memory hardware description is generated and pre-loaded with the codewords resulting from the Python Hamming encoding, the Hamming decoder was developed in VHDL. This Hamming decoder consists of the following steps:

- Computing syndrome bits of the codeword and constructing syndrome bit vector
- Comparing syndrome bit vector to parity bit vector

- Syndrome bit vector different than syndrome bit vector $\Longrightarrow$ flipping bit on position (parity bit vector $\oplus$ syndrome bit vector).
- Slicing codeword to return data bits only

Figure 4.10 displays a complete overview of the Hamming encoding/decoding flow.

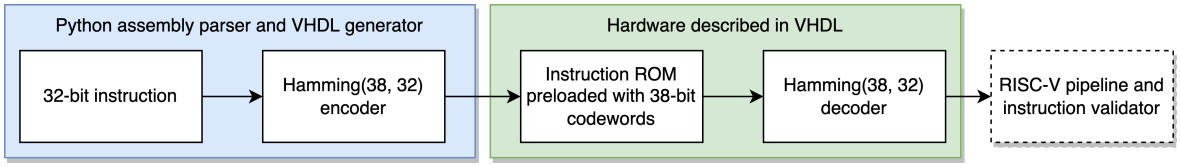


Figure 4.10: ECC flow

It must be noted that this flow is used for simulation purposes only. For the implementation with FreNox, Hamming encoder and decoder IPs provided by Xilinx were instantiated.

4.4 Concluding summary

To conclude, in this chapter the design criteria, instruction validator hardware design and software abstraction, generator script, and ECC instruction memory design were introduced.

In section 4.1, the design criteria were introduced. Based on the overhead of the CF and BF compared to the IM, the BF was chosen. As the instruction validator is able to detect MBUs, Hamming SEC-DED is redundant and Hamming SEC suffices. The minimum RISC-V standard that the design must comply with is RV32IM. The instruction validator and ECC must be integrated with the FreNox SoC-e which instantiates the FreNox RISC-V core.

In section 4.2, the instruction validator design was introduced. The instruction validator instantiates separate hash and memory elements to be able to concurrently read all bit positions in the bit array. In subsection 4.2.1, an introduction to non-cryptographic hash functions was presented as well as an Avalanche effect analysis which analyzes them in more depth. From this analysis, it could be observed that the MultiplyShift meets the Avalanche property while the CRC class functions roughly match the normal distribution with CRC-32C having a better mean. In subsection 4.2.2, a software abstraction of the instruction validator was created. This software abstraction was used to analyze the BF performance with the introduced non-cryptographic hash functions. All hash functions stayed below or close to the defined false positive probability. In subsection 4.2.3, two hardware designs

instantiating the MultiplyShift hash and CRC-32C hash were introduced. Both hardware designs were analyzed in-depth and their latency and area overhead were considered. To minimize area overhead and respect strict memory requirements, the m-k optimization algorithm was introduced. In subsection 4.2.4, the generator script was introduced. This generator script uses the instruction validator software abstraction to generate all the hardware descriptions of all the needed components to simulate and synthesize the instruction validator. Also, a parser was developed which parses instruction/address pairs from disassembled RISC-V programs which can be used as input for this generator script.

In section 4.3, the ECC instruction memory design was introduced for simulation purposes. The Hamming SEC encoding was developed in Python and is part of the generator script. As the generator script also generates the instruction memory for simulation purposes, the Hamming decoding had to be described in VHDL. In the implementation, the Xilinx ECC IP was used to provide Hamming encoding and decoding.

Chapter 5

Simulation

This chapter focuses on simulating the hardware design. The instruction validator was simulated using multiple test cases. The test cases can be categorized into three categories:

1. Testing without injecting faults
2. Testing while injecting SEUs and MBUs
3. Testing while triggering different types of HWTs

The VHDL generation and testing were unified and automated to make the VHDL generation and testing flow easier and faster. The first step is to generate all VHDL files of all specified program/hash/optimization configurations. As mentioned before, the generator script also enables the user to use randomly generated 64-bit instruction/address pairs. However, it must be noted that for all test cases in this chapter, only disassembled benchmark programs were used. The used programs and their details are discussed later.

Figure 5.1: Generating all program/hash/optimization configurations

Figure 5.1 displays the automation flow of generating all different configurations as preparation for simulation and implementation (remember the tcl generation). This flow is written in a Makefile and can be executed by executing `make`. The first step is to set the RISC-V toolchain environment variables as they are not present in the default PATH. A list of programs can be specified which are automatically built. Before the Python generation script can be executed, a Python3 virtual environment must be installed. The Python package installer `pip` is used to install the packages needed by the BF class that is imported in the generator script. The next option is

to generate all configurations. This is done by iterating through all programs and calling the generator script with all different program, hash, and optimization system arguments. The last step is to run RDL. This is not part of the simulation but is used to generate the register files used for the isolated instruction validator test setup which is discussed in detail in chapter 6: Implementation.

Figure 5.2 displays the simulation setup. The Python module ***CO****routine based* ***CO****simulation* ***T****est****B****ench* cocotb was used to automate the simulation using the test cases discussed below [56].

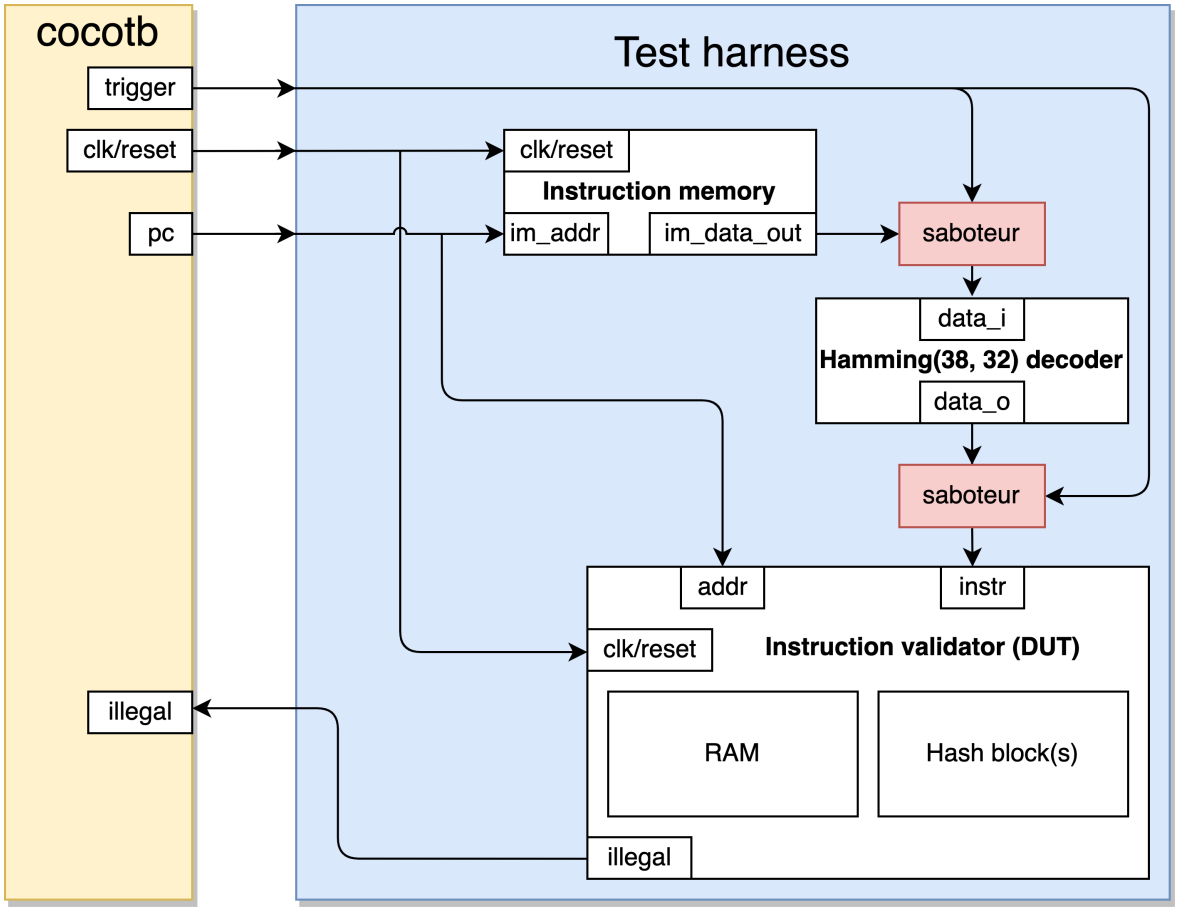


Figure 5.2: Cocotb simulation setup

The test harness which is also generated by the generator script instantiates the components generated by the Python generator script, the Hamming decoder, and saboteurs. The saboteurs can be triggered from cocotb and can trigger different faults such as single/multiple bit-flips and can also trigger HWTs. No saboteur is introduced between the PC and the test harness components as this is outside of the scope of this thesis. However, while HWTs that directly manipulate the PC are not considered in this thesis, the PC can also be manipulated by HWTs that inject branch instructions which are discussed later. The `pc` signal is initialized to the initial

address of the parsed RISC-V program or zero when using random instruction/address pairs and is incremented with one each clock cycle. This way, the memory layout is respected by the generated instruction memory which contains a 38-bit codeword per address. The next sections discuss the test cases in depth.

Figure 5.3: Testing all program/hash/optimization configurations

Figure 5.3 displays the automation flow of executing all test cases that are mentioned in the next sections. Again this automation was written in a Makefile. The first step is to import the generated VHDL files which are in the directory where the generator script is located and the pysim sub-directory. The random seed can be set in the Makefile that is adopted by all the test cases. The next step is to import the `Makefile.pysim` which includes all the test case specifications denoted as variables. An example of a test specification is displayed in Listing 5.1.

Listing 5.1: Test case specification

```
MODULE                  := instruction_validator_tc_nofaults_aes_0_0
MODULES                 := $(MODULES) $(MODULE)
$(MODULE)PY_MODULE      := instruction_validator_tc_nofaults
$(MODULE)TESTCASE       := main
$(MODULE)TOPLEVEL       := testharness_aes_0_0
$(MODULE)RUN_TIME       := 100000 ns
$(MODULE)SIM_ARGS       :=
$(MODULE)DO_GUI_BEFORE  := log -rec *
$(MODULE)DO_GUI_AFTER   := add wave *
```

Each test case has its own module name and specifies multiple parameters such as the test harness top level that was generated by the generator script. The maximum run time is also specified.

The last step is to run all the tests present in `Makefile.pysim`. To simplify the flow even further, the results are generated in categorized files in LaTeX table format and were inserted in this document.

All the tests discussed in the next sections were executed using instruction/address pairs from the following benchmark programs:

- Rijndael AES
- Blowfish
- Dijkstra

- FFT
- Patricia
- SHA
- Quicksort

The mentioned benchmarks are programs based on common algorithms provided by MiBench: *A free, commercially representative embedded benchmark suite* [57]. The Rijndael AES benchmark belongs is part of the security category. The MiBench security category represents different algorithms for data encryption/decryption and hashing. Blowfish and SHA are both part of the MiBench network and security categories. Dijkstra and Patricia belong to the MiBench network category. The MiBench network category represents algorithms that are used in embedded systems found in network equipment such as switches and routers. FFT belongs to the MiBench telecom category which includes telecommunication algorithms used in wireless equipment. QSort is part of the MiBench automotive and industrial control category which represents embedded control systems.

5.1 No faults

The first step was to test the instruction validator without injecting faults. The expected behavior should be that when only legal instructions are fetched, the instruction validator should not trigger the illegal signal as false negatives are not possible. A cocotb test case class was created for this category to test the instruction validator without injecting faults.

The test executes multiple asynchronous Python methods which check the instruction validator while incrementing the program counter. All the test cases are initialized with a 100MHz clock generator. An asynchronous method `async def pc()` is used to increment the program counter (PC) at every rising edge of the clock.

Another asynchronous method called `async def checkillegal()` asserts the illegal signal to remain low which is expected behavior without injecting faults. The illegal register must be enabled by the `instr_valid` signal as displayed in Figures 4.5 and 4.6. This means that it's important to make sure that the `instr_valid` signal is high at every address change. This is checked using the `async def checkvalid()` method. The methods are called in `async def test_001_No_Faults()`. All tests passed with all different program/hash/optimization configurations. Listing 5.2 displays the result of one of the test cases.

```
*************************************************************************************************
** TEST                                  STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
*************************************************************************************************
** instruction_validator_tc_nofaults.main  PASS        18516.01           6.06       3055.53  **
*************************************************************************************************
** TESTS=1 PASS=1 FAIL=0 SKIP=0                        18516.01           6.07       3049.48  **
*************************************************************************************************
```

Listing 5.2: No faults test case result of Rijndael AES using the CRC-32C hash without optimization

5.2 Injecting faults

The next test case introduces faults in the instruction memory. As mentioned previously, single- and multiple-bit errors can occur in the instruction memory. The saboteur positioned between the instruction memory and Hamming decoder was used to emulate those faults in the instruction memory. Multiple tests were written to test the instruction validator and ECC instruction memory.

The instruction ROM and bit arrays are filled with instruction/address pairs from programs compiled with the FreNOX RISC-V toolchain. It must be noted that the programs aren't executed like on a real RISC-V core. The PC is just incrementing i.e. jumps are fetched but the PC isn't influenced by them. This doesn't influence the functional testing of the instruction validator as jumps in the program jump to instruction/address pairs that are legal. This was proven in the implementation with FreNox in Chapter 6. The reason for using real instruction/address pairs from disassembled programs is to fill the bit arrays with real instruction/address pairs instead of randomly generated instruction/address pairs which result in more realistic results in terms of false positives.

The first test `async def test_1_Single_Bit_Error_ECC_Test()` injects single-bit faults (SEUs) in all instruction codewords. Again, an asynchronous method `async def pc()` is used to increment the program counter at every rising edge of the clock. Figure 5.4 displays the simulation flow that was used to flip bits in the codeword.

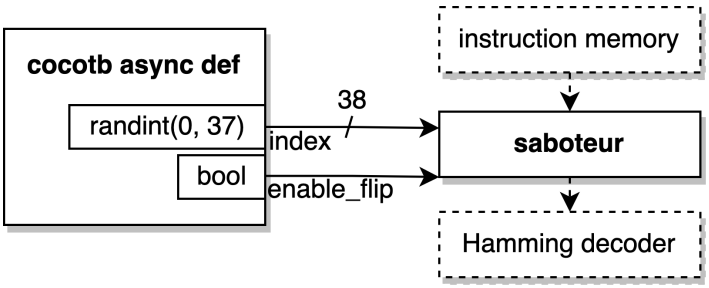


Figure 5.4: Flipping single bits in the 38-bit codeword

This index is used as input to the saboteur which randomly picks and flips one

of the 38 bits per codeword. An enable signal triggers the flipping of bits. Single-bit faults are corrected by the Hamming SEC decoder. Hence, the expected behavior is that no illegal instructions are detected by the instruction validator. The `async def checkecc()` method contains two asserts: The first assert checks this behavior and the second assert was used to make sure that all instruction codewords are flipped. Like the no faults test, all tests passed with different program/hash/optimization configurations which proves that the Hamming SEC decoder passed the test.

The saboteur in the second test `async def test_2_Double_Bit_Errors_Test()` and third test `async def test_3_Triple_Bit_Errors_Test()` introduces double- and triple-bit errors (MBUs). This test uses an enable signal to trigger the multiple-bit errors in the saboteur with cocotb together with a class method in the test case that flips bits. The method which inputs the codeword from the instruction memory is displayed in Figure 5.5.

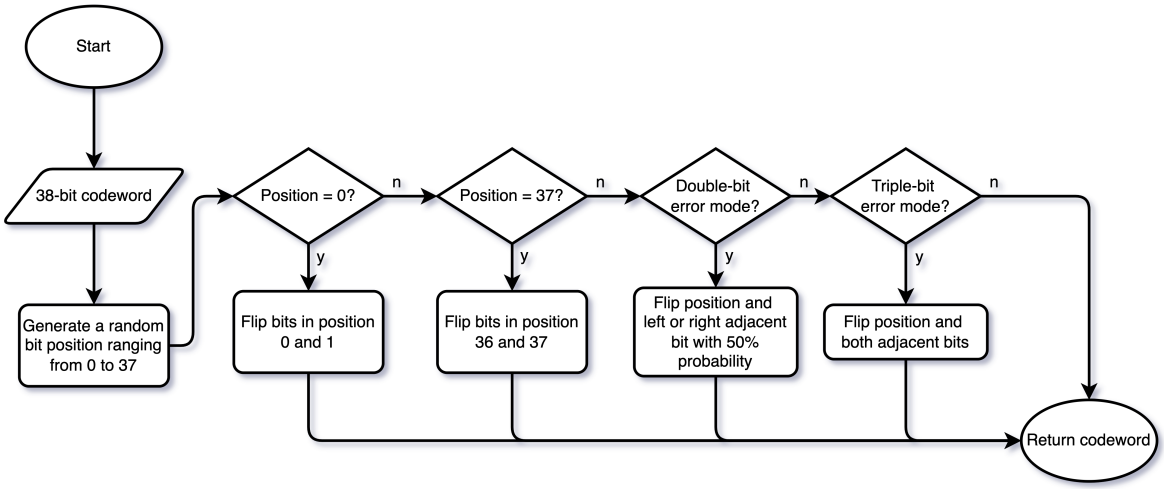


Figure 5.5: Flow of method that introduces MBUs

The `async def checkiv()` spawns the `async def checkfp()` method when a new instruction is output by the instruction memory and triggers MBUs using the `def flipbits()` class method. The `async def checkfp()` method checks the illegal signal after four or five clock cycles depending on the used hash and increments a counter when the illegal signal is low which implies a false positive. The end of the test case contains an assert to make sure that the FPR is below the set FPP.

Table 5.1 and 5.2 display the double- and triple-bit error CRC-32C test results for every benchmark with a set FPP of 0.05, without m and k optimizations, with a fixed random seed of 10110010597110 and an initial CRC value of 0xF. The initial value is shifted three times per hash. So if the first CRC hash has an initial value of 0xF, the second hash has an initial value of 0x78, the third hash 0x3C0, etc.

Table 5.4, 5.5, 5.6 and 5.7 display the double- and triple-bit error MultiplyShift

test results for every benchmark, again with a set FPP of 0.05 when doing both optimizations and with a fixed random seed of 10110010597110.

The benchmark column denotes the used benchmark program in the test. The number of instructions/elements is denoted by n. The total bit array size is denoted by m. The amount of hash functions is denoted by k. The FPP column denotes the calculated FPP based on the determined n, m, and k parameters defined in the BF class. Remember that the optimizations ensure that the hash output can be sliced, hence every individual bit array size is a power of two. The FPR denotes the false positive rate: An instruction/address pair that is illegal but is marked as legal by the BF. The result denotes if the test passed or failed. Every FPR must stay under the FPP (0.05) with a deviation of 10% to pass the test. A small FPR deviation of 10% is allowed to take the small FPR error into account when computing m and k. The following tests contain an averaged FPR over 10 runs.

Benchmark	$\mathbf{n}$	$\mathbf{m}$	$\mathbf{k}$	**FPP**	**FPR**	**Result**
aes	6171	38480	5	0.051018	0.052115	PASS
blowfish	24710	154075	5	0.051026	0.05106	PASS
dijkstra	451	2815	5	0.050857	0.052328	PASS
fft	26004	162140	5	0.051029	0.049862	PASS
patricia	765	4770	5	0.051027	0.04915	PASS
sha	627	3910	5	0.051007	0.053748	PASS
qsort	333	2080	5	0.050736	0.054655	PASS

Table 5.1: Double-bit errors test case results of 10 runs with $\epsilon = 0.05$ without optimization using CRC-32C

Benchmark	$\mathbf{n}$	$\mathbf{m}$	$\mathbf{k}$	**FPP**	**FPR**	**Result**
aes	6171	38480	5	0.051018	0.050267	PASS
blowfish	24710	154075	5	0.051026	0.051238	PASS
dijkstra	451	2815	5	0.050857	0.050554	PASS
fft	26004	162140	5	0.051029	0.051315	PASS
patricia	765	4770	5	0.051027	0.053464	PASS
sha	627	3910	5	0.051007	0.054864	PASS
qsort	333	2080	5	0.050736	0.061261	FAIL

Table 5.2: Triple-bit errors test case results of 10 runs with $\epsilon = 0.05$ without optimization using CRC-32C

When comparing the double- and triple-bit error CRC-32C tests displayed in Table 5.1 and 5.2, it can be observed that the FPR is generally higher when introducing

triple-bit errors. Further research was done to explain this behavior. Observing the simulation waves resulted in an interesting finding. The Hamming decoder has a positive side-effect on the instruction validator. The introduced faults sometimes result in parity mismatches which lead to the Hamming SEC decoder correcting the wrong bits and unintentionally adding an extra fault to the instruction. This behavior has a positive effect on the FPR which implies that the combination of ECC and the instruction validator improves the detection performance. The triple-bit error FPR is sometimes higher than the FPR with double-bit errors. The reason is that the parity is correct more often with triple-bit errors than double-bit errors. Hence, faults are added to the double-bit error instructions which result in a better FPR. Also, the positions of the bit-flips by the Hamming encoder are more distributed over the instruction than triple-bit errors which are clusters of triple-bit errors. Let's consider the example in Table 5.3 taken from one of the tests. The red-colored bits denote the bits that are affected by the fault which is a triple-bit error in this case. The blue bits denote the extra fault that is introduced by the Hamming decoder. Remember that the codeword contains both parity and data bits. The Hamming decoder data bits are eventually output and checked by the instruction validator. In this case, the blue bit in the Hamming decoder is a data bit as displayed in the data bit column and hence this instruction contains an extra fault and is less likely to be marked as a false positive.

Output	Codeword	Data bits
Instruction ROM	11111101011000100001**011**01100010010110	11111110110001000010**01**0110000011
Saboteur	11111101011000100001**100**01100010010110	11111110110001000010**10**0110000011
Hamming decoder	11111101011000100001**1001**1100010010110	11111110110001000010**101**110000011

Table 5.3: Hamming decoder introducing an extra fault

This effect was proven by bypassing the Hamming decoder and running all the tests again. Those respective test results are displayed in Table A.1, A.2, A.3, A.4, A.5 and A.6 which clearly shows a higher overall FPR and a significantly increased amount of failed tests. This proves that using Hamming ECC results in a lower overall FPR and hence better detection of illegal instruction/address pairs.

It can be observed in Table 5.2 that the Quicksort benchmark failed. There could be two possible explanations: The sample size is too small which leads to a less precise FPR as a false positive has a higher effect on the FPR. This seems a possible explanation as the benchmark that failed has the lowest amount of instructions denoted by n. Another explanation could be that the instructions with faults and their respective addresses are similar for this specific random seed to the instruction/address pairs that were initially added to the BF. This could lead to more false positives as the hash function maps those instruction/address pairs to existing elements. It

must however be noted that this test passed when using different combinations of initial values and shifts per hash.

Benchmark	n	m	k	FPP	FPR	Result
aes	6171	40960	5	0.0415	0.033949	PASS
blowfish	24710	163840	5	0.041647	0.04208	PASS
dijkstra	451	3072	3	0.045209	0.043237	PASS
fft	26004	163840	5	0.049319	0.0485	PASS
patricia	765	5120	5	0.04036	0.039216	PASS
sha	627	4096	4	0.043962	0.042424	PASS
qsort	333	2048	4	0.052274	0.052553	PASS

Table 5.4: Double-bit errors test case results of 10 runs with $\epsilon = 0.05$ and m-k optimization using the MultiplyShift hash

Benchmark	n	m	k	FPP	FPR	Result
aes	6171	40960	5	0.0415	0.034776	PASS
blowfish	24710	163840	5	0.041647	0.041056	PASS
dijkstra	451	3072	3	0.045209	0.04878	PASS
fft	26004	163840	5	0.049319	0.049392	PASS
patricia	765	5120	5	0.04036	0.037647	PASS
sha	627	4096	4	0.043962	0.044338	PASS
qsort	333	2048	4	0.052274	0.053153	PASS

Table 5.5: Triple-bit errors test case results of 10 runs with $\epsilon = 0.05$ and m-k optimization using the MultiplyShift hash

All the MultiplyShift double- and triple-bit error tests using m-k optimization (see Algorithm 1) passed. The results are displayed in Table 5.4 and 5.5. An interesting observation when looking at the CRC-32C and rounding optimization test results is that the total bit array size is significantly lower than the total bit array sizes of the rounding optimization. Also, the total bit array size is approximately like the original total bit array sizes as displayed in the CRC-32C test results. This proves that the m-k optimization has a positive effect on the area overhead while respecting the FPP in this case.

Benchmark	n	m	k	FPP	FPR	Result
aes	6171	40960	5	0.0415	0.033949	PASS
blowfish	24710	163840	5	0.041647	0.04208	PASS
dijkstra	451	5120	5	0.005737	0.003991	PASS
fft	26004	163840	5	0.049319	0.0485	PASS
patricia	765	5120	5	0.04036	0.039216	PASS
sha	627	5120	5	0.02013	0.021372	PASS
qsort	333	2560	5	0.024995	0.026126	PASS

Table 5.6: Double-bit errors test case results of 10 runs with $\epsilon = 0.05$ and rounding optimization using the MultiplyShift hash

Benchmark	n	m	k	FPP	FPR	Result
aes	6171	40960	5	0.0415	0.034776	PASS
blowfish	24710	163840	5	0.041647	0.041056	PASS
dijkstra	451	5120	5	0.005737	0.007095	PASS
fft	26004	163840	5	0.049319	0.049392	PASS
patricia	765	5120	5	0.04036	0.037647	PASS
sha	627	5120	5	0.02013	0.021053	PASS
qsort	333	2560	5	0.024995	0.022523	PASS

Table 5.7: Triple-bit errors test case results of 10 runs with $\epsilon = 0.05$ and rounding optimization using the MultiplyShift hash

All the MultiplyShift double- and triple-bit error tests while using rounding optimization passed. The results are displayed in Table 5.6 and 5.7. The CRC-32C double- and triple-bit error test results display the original total bit array sizes. When comparing the results with CRC-32C, it can be observed that using the rounding optimization results in a higher total bit array size. This has to do with the fact that the rounding optimization rounds the individual bit array sizes to the nearest higher power of two. This leads to a much lower FPP and hence a lower FPR. The disadvantage is that this optimization results in a higher area overhead in some cases (m-k sometimes chooses the same parameters) as proven in the synthesis results presented in Chapter 6: *Implementation*. However, the advantage is that this rounding optimization ensures that the set FPP isn't violated which is reflected in the test results and FPR.

It can be concluded that when comparing the CRC-32C and MultiplyShift results for both optimizations using MultiplyShift results in a better overall FPR. This is even the case when using m-k optimization which approximates the CRC-32C parameters. While CRC-32C doesn't perform much worse than MultiplyShift, it can be

observed that MultiplyShift clearly performs better for these specific benchmarks.

It must be noted that relaxing the FPP deviation of the m-k optimization can be very useful for certain applications. E.g., increasing the FPP deviation might be a suitable approach for embedded systems that have very strict memory or power requirements and can suffice with a higher FPR than the set FPP.

Another interesting observation is that all tests failed when combining the CRC-32C hash with the m-k optimization algorithm. The reason for the failing test was the explosive FPR. An additional investigation was performed on the CRC-32C hash to find the cause of this problem. Remember that the m-k optimization results in a power of two modulo. Besides checking the Avalanche effect, the single output bits were analyzed when again changing one bit in the hash input. Again, generating 10 million random 64-bit pairs and changing one bit in the input resulted in Figure 5.6.

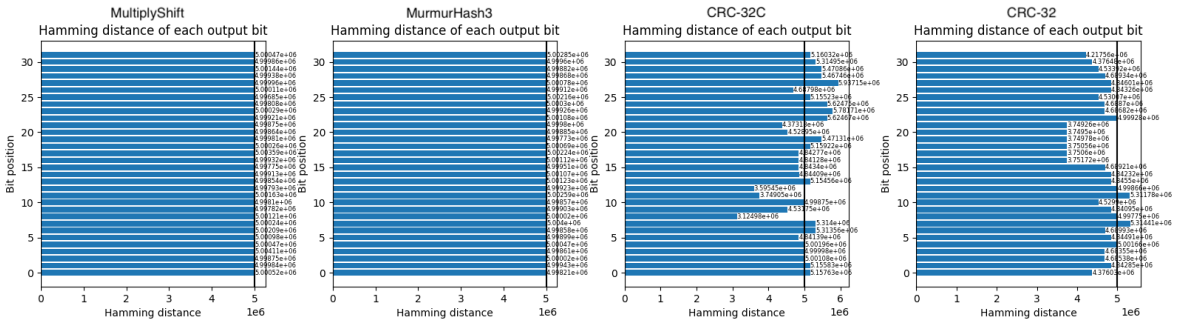


Figure 5.6: Hamming distance of single output bits

The most optimal result would be that the Hamming distance of each output bit is exactly or approximately half the number of samples. It can be observed that output bits of MurmurHash3 and MultiplyShift have a very even Hamming distance distribution. The CRC-32C and CRC-32 hashes have a less even distribution. While the CRC-32C and CRC-32 can be used as a hash as proven in the Avalanche test, they are weaker than strong hashes like MurmurHash3 and MultiplyShift. Too much information is lost when ignoring a part of the CRC output bits or when slicing single CRC output bits.

Instruction/address pairs that are not part of the bit arrays result in a significant amount of hash collisions with existing instruction/address pairs when using a modulo power of two which in turn results in a high FPR. This is the case when using a power of two as modulo which results in a part of the bits being ignored. Picking individual output bits is an alternative method to picking a modulo with a power of two but results in the same information loss and hence an FPR which by far doesn't meet the set FPP.

The inconsistency in the single output bits can be observed in more detail in Figure 5.7 and 5.8. Each sample set consists of 128 random 64-bit words. Again,

one bit is changed in one of the words in the pair and the Hamming distance of the output of both words in the pair is compared.

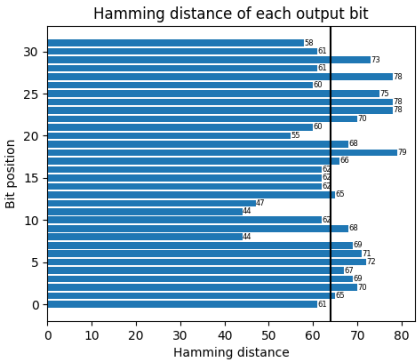


Figure 5.7: Sample set #1

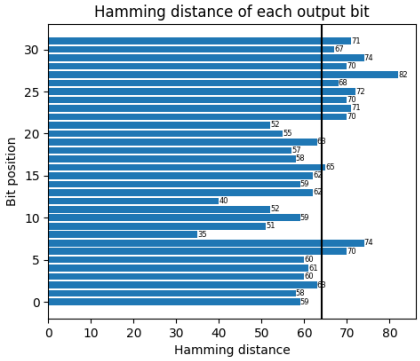


Figure 5.8: Sample set #2

To conclude, the complete output of the CRC-class hashes must be used to get a distribution that is performing well enough to achieve FPRs that meet the set FPP.

5.3 Triggering HWTs

The saboteur connected to the output of the Hamming decoder displayed in Figure 5.2 was used to trigger HWTs. Two test cases were written that trigger HWTs in the instruction memory. The following HWTs can be activated by an adversary which can lead to malicious program execution, accessing illegal memory locations, data corruption, and unexpected behavior in the program output or flow:

- `async def test_1_HWT_Injecting()`: HWT that injects instructions
- `async def test_2_HWT_Modifying()`: HWT that modifies instructions

The following subsections discuss hose test cases in more detail. Both test cases adapt the same flow displayed in Figure 5.9.

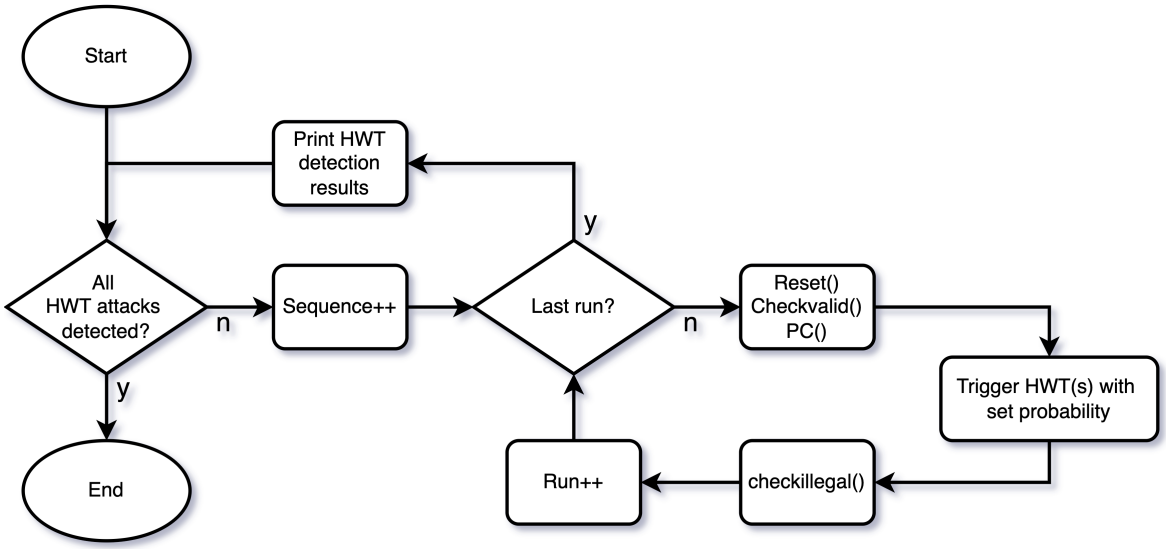


Figure 5.9: Cocotb flow of HWT test cases

This flow consists of two loops: Sequences and runs. The sequence denotes the number of injected/affected instructions per HWT. The runs denote the number of different timed HWT attacks.

A run is initialized by executing `async def pc()` which asynchronously starts the PC and by executing `async def reset()` which resets all the components in the test harness. Again, the `instr_valid` signal is checked by executing `async def checkvalid()`. Every HWT can be triggered once or multiple times at a random interval in the simulation using a set probability. For every injected instruction it is checked and logged if the instruction validator detected this instruction/address pair using `async def checkillegal()`. The sequence loop is terminated when all HWT attacks are detected. The test results are printed after every sequence.

The first test case focuses on HWTs that inject instructions and is discussed in the next subsection.

5.3.1 Injecting HWT

For this type of HWT, it is assumed that the attacker has access to the instruction memory. As discussed previously in the HWT taxonomy, this type of HWT can be inserted in every phase. Injecting instructions can lead to malicious program execution as injected instructions can jump to instruction memory locations that contain malicious instructions that were inserted by the adversary. It's also possible that those malicious instructions can access illegal parts of the data memory that contains sensitive data. Injected instructions can also lead to data corruption and can influence the program output or flow that is supposed to be executed. Malicious

instructions can alter data in registers and affect the program's integrity. It's also possible to inject branches that compromise the program flow.

This test case injects random 32-bit instructions and overrides the to-be-executed instructions. The PC isn't influenced by this instruction and just keeps incrementing as usual. A probability of 0.1 is used to trigger HWTs randomly in a run for a specified amount of instructions denoted by the sequence. This resulted in the results displayed in Table 5.8 using 1000 runs and using the CRC-32C hash. The results using the MultiplyShift hash with the m-k and rounding optimization are displayed in Table 5.9 and 5.10. The Dijkstra benchmark was used consisting of 121 instructions with an FPP of 0.05 and a fixed random seed of 10110010597110. Like the fault injection results, the m-k optimization in combination with the MultiplyShift hash was also used in this test as this optimization approximates the m and k parameters used by the CRC-32C hash. This makes for a better comparison between the MultiplyShift and CRC-32C hashes.

Looking at the results show that the attack becomes fully detected at four instructions per attack for the CRC-32C hash and five for the MultiplyShift hash using m-k optimization. The MultiplyShift hash with rounding optimization can detect all HWT attacks at three instructions per attack. This is expected as the FPP is significantly lower than the set FPP when using the rounding optimization as discussed previously in the fault injection tests. It can also be observed that the CRC-32C hash and MultiplyShift hash perform very similarly. Another observation is that the MultiplyShift hash can detect the HWT inject attacks slightly better with a growing sequence length than CRC-32C when looking at the ratio. It could also be observed that the number of undetected attacks was declining with an increasing sequence. The reason for this behavior is the product of probabilities. With an FPP of 0.05, the probability of two instructions being undetected by the instruction validator is $0.05 \cdot 0.05 = 0.0025$, for three instructions $0.05 \cdot 0.05 \cdot 0.05 = 0.000125$, etc. This means that with a higher sequence, the detection becomes significantly better. HWTs often consist of multiple instructions. This means that the more sophisticated the attack, the more capable the instruction validator becomes to detect them.

Sequence length	Undetected	Nr. of attacks	Undetected ratio
1	504	9950	0.050653
2	59	9981	0.005911
3	2	10058	0.000199
4	0	10049	0.0

Table 5.8: Results of injecting HWT test case using CRC-32C

Sequence length	Undetected	Nr. of attacks	Undetected ratio
1	506	9950	0.050854
2	52	9981	0.00521
3	1	10058	0.0001
4	1	10049	0.0001
5	0	9848	0.0

Table 5.9: Results of the injecting HWT test case using MultiplyShift and the m-k optimization

Sequence length	Undetected	Nr. of attacks	Undetected ratio
1	115	9950	0.011558
2	33	9981	0.003306
3	0	10058	0.0

Table 5.10: Results of the injecting HWT test case using MultiplyShift and the rounding optimization

5.3.2 Modifying HWT

This test case focuses on HWTs that modify legal instructions. A possible HWT that modifies instructions is an HWT that triggers an AND operation on the instruction and a 32-bit mask that sets all the fields to zero except for the opcode field and the funct3 field (see Figure 5.10).

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

Figure 5.10: RV32I instruction types [2]

The opcode field is set to ”0010011” which is the opcode for an immediate add (ADDI) instruction. The funct3 field is set to ”111” which denotes the ADDI instruction. The idea behind this attack is to skip instructions by overwriting them with this mask. Table 5.11 displays the results of modifying instructions varying from one to three instructions using the CRC-32C and using a fixed random seed of 28061997. Table 5.12 and 5.13 display the results of this HWT using the MultiplyShift hash

with m-k and rounding optimization. The CRC-32C and MultiplyShift can both detect all HWT attacks at three instructions per attack. Again, the MultiplyShift hash can detect HWT attacks slightly better with sequence length one than the CRC-32C hash. It can also be observed that the CRC-32C detection becomes better than MultiplyShift with a growing sequence length like with the injecting HWT test case. Again, the rounding optimization results in a significantly better detection for the reasons mentioned in the HWT injection attack tests.

Sequence length	Undetected	Nr. of attacks	Undetected ratio
1	522	9992	0.052242
2	39	9881	0.003947
3	0	9936	0.0

Table 5.11: Results of modifying HWT test case using CRC-32C

Sequence length	Undetected	Nr. of attacks	Undetected ratio
1	378	9971	0.03791
2	72	10053	0.007162
3	0	9947	0.0

Table 5.12: Results of the modifying HWT test case using MultiplyShift and the m-k optimization

Sequence length	Undetected	Nr. of attacks	Undetected ratio
1	169	9918	0.01704
2	19	10024	0.001895
3	0	10069	0.0

Table 5.13: Results of the modifying HWT test case using MultiplyShift and the rounding optimization

5.4 Concluding summary

To conclude, a test harness, automation flow, and multiple test cases were introduced to simulate the instruction validator and ECC instruction memory. An automation flow was used to simulate all different hash/optimization/program configurations for different test cases. Each hash/optimization/program configuration was tested without injecting faults and while injecting single-bit errors (SEUs), double- and triple-bit errors (MBUs), and HWTs.

Section 5.1 discussed the no faults simulation. This test case verified the correct functioning of the ECC instruction memory and instruction validator. Running the test resulted in all program/hash/optimization configurations passing the test.

Section 5.2 discussed the simulation when injecting single-, double- and triple-bit errors. Multiple cocotb methods were developed to test the instruction validator and Hamming decoder. All single-bit errors were successfully detected and corrected by the Hamming decoder. When injecting double- and triple-bit errors, the FPR of all configurations using the MultiplyShift hash stayed below the set FPP with a small deviation while one test failed when using the CRC-32C hash. However, the test did pass when using a different combination of initial values and shifts per individual hash function. Observing the simulation waves in more detail resulted in an interesting finding. Introducing MBUs results in the Hamming decoder occasionally miscorrecting bits which results in a better overall FPR for the instruction validator. This effect was proven by running all the MBU simulations without the Hamming decoder. This resulted in an overall higher FPR and hence more failed tests. Another interesting observation was when applying a modulo with a power of two on the CRC-32C hash output to decrease area overhead, all tests failed. An additional analysis was executed on the CRC-32C hash by looking at single output bits. This resulted in the conclusion that the complete hash output of CRC-32C must be used to get a strong distribution.

Section 5.3 discussed injecting two types of HWTs. A flow was developed that dynamically injects these two types using a set probability. The *injecting HWT* attack discussed in subsection 5.3.1 became fully detected with a sequence length of four instructions with the CRC-32C hash. The MultiplyShift with m-k optimization which approximates the performance of CRC-32C detected all HWT attacks at a sequence length of five instructions. The MultiplyShift with rounding optimization was able to detect all HWT attacks at a sequence length of three instructions. However, this was the case as the rounding optimization increased the total bit array sizes which results in a better FPR. The *modifying HWT* attack discussed in subsection 5.3.2 became fully detected at an instruction sequence length of three for all hash/optimization configurations.

Chapter 6

Implementation

This chapter discusses the implementation of the instruction validator and ECC instruction memory. Firstly, the synthesis results of the instruction validator configurations are discussed. Secondly, the implementation is discussed which is divided into two parts: An isolated implementation and an implementation with the FreNox RISC-V core and FreNox SoC-e. The isolated implementation discusses the implementation of the instruction validator with some peripherals in an isolated test environment. The instruction validator was tested on an FPGA fabric before integrating it with the FreNox SoC-e and FreNox RISC-V core. The implementation with FreNox discusses the implementation of the instruction validator and ECC instruction memory on an FPGA fabric with the FreNox SoC-e which instantiates the FreNox RISC-V core.

6.1 Synthesis results

Synthesizing all the different instruction validator configurations and benchmark programs with a 100MHz clock constraint for the Arty A7-100T containing the Xilinx Artix-7 XC7A100TCSG324-1 FPGA resulted in the synthesis results displayed in Table 6.1, 6.2, 6.3, 6.4, 6.5, 6.6 and 6.7. The RTL and synthesis schematics of an instruction validator configuration are displayed in Figure B.2 and B.3. A small Python script was developed to parse all the synthesis results from the Vivado work folder using regex. The first notable observation is the CRC-32C hash failing the timing requirements, even with post-route physical optimization. All synthesis results consist of a negative slack. Doing further research on why this was the case resulted in interesting observations. When looking at the report and the schematic (displayed in Appendix B.1) of the path with the worst slack, it can be concluded that the modulo operation results in a long path of carry- and LUT-blocks which results in a large amount of slack which doesn't meet the timing requirement of 10 ns. This

proves that a modulo operation in hardware can be very expensive in terms of time overhead.

Looking at the general differences between the hashes shows that the CRC-32C hash does use significantly more LUTs and FFs than the MultiplyShift hash. This makes sense as the modulo operation takes many LUTs and FFs when looking at the utilization reports. Another observation is that the MultiplyShift hash consumes significantly less dynamic and static power than the CRC-32C hash. When looking at the amount of BRAMs shows that all hash/optimization combinations roughly consume the same amount of BRAMs. This is the case because roughly they have the same bit array sizes. The simulation results showed a big difference in the Dijkstra benchmark between MultiplyShift rounding and m-k. This is also reflected in the number of consumed BRAMs. As already discussed, the MultiplyShift hash offers many improvements over the CRC-32C hash in terms of power consumption, slack, LUTs, and FFs. A notable difference is the amount of consumed digital signal processing (DSP) blocks. The DSP blocks are used for the MultiplyShift multiplications. It must be noted however that the used FPGA offers a maximum of 240 DSP blocks. This means that with a large program such as Blowfish, with a text section of 103460 bytes, the maximum DSP utilization percentage is 12.5 % which leaves a very reasonable 87.5 % for other hardware. However, when looking at smaller-sized programs, the lowest amount of DSP blocks is 9 which results in a utilization percentage of just 3.75%. When comparing the MultiplyShift and MultiplyShiftPipelined, it can be observed that MultiplyShiftPipelined has an overall better worst negative slack (WNS) and consumes fewer DSP blocks. While the MultiplyShiftPipelined has an effect on some configurations in terms of the amount of DSP blocks and a higher WNS when looking at large programs the amount of DSP blocks is equal to the MultiplyShift variant. The reason for this behavior was studied and resulted in an interesting finding. The design rule check reports mention that DSP blocks are also pipelined by the synthesis tool to meet the timing requirements. A possible recommendation to improve the design further is to create more multiplier pipeline stages. This is out of the scope for this project as the latency is minimized to only three clock cycles and the timing requirements are met. However, when strict area requirements must be met and a higher latency is acceptable, more multiplier pipeline stages would be a good solution and improvement to the current design.

Hash	Optimization	Dynamic P (W)	Static P (W)	WNS (ns)	LUTs	FFs	BRAMs	DSP blocks
CRC-32C	-	0.066	0.091	-5.133	1501	113	2.5	0
MultiplyShift	m-k	0.033	0.091	0.941	84	39	2.5	29
MultiplyShift	rounding	0.033	0.091	0.941	84	39	2.5	29
MultiplyShiftPipelined	m-k	0.033	0.091	2.511	66	22	2.5	30
MultiplyShiftPipelined	rounding	0.033	0.091	2.511	66	22	2.5	30

Table 6.1: Rijndael AES synthesis results with 100MHz constraint, all configurations, and $\epsilon = 0.05$

Hash	Optimization	Dynamic P (W)	Static P (W)	WNS (ns)	LUTs	FFs	BRAMs	DSP blocks
CRC-32C	-	0.078	0.092	-5.489	1545	113	5	0
MultiplyShift	m-k	0.040	0.091	0.513	76	21	5	30
MultiplyShift	rounding	0.040	0.091	0.513	76	21	5	30
MultiplyShiftPipelined	m-k	0.040	0.091	2.613	76	22	5	30
MultiplyShiftPipelined	rounding	0.040	0.091	2.613	76	22	5	30

Table 6.2: Blowfish synthesis results with 100MHz constraint, all configurations, and $\epsilon = 0.05$

Hash	Optimization	Dynamic P (W)	Static P (W)	WNS (ns)	LUTs	FFs	BRAMs	DSP blocks
CRC-32C	-	0.086	0.092	-5.256	1865	113	2.5	0
MultiplyShift	m-k	0.017	0.091	3.109	154	65	1.5	10
MultiplyShift	rounding	0.028	0.091	2.284	256	123	2.5	18
MultiplyShiftPipelined	m-k	0.016	0.091	2.391	170	72	1.5	9
MultiplyShiftPipelined	rounding	0.026	0.091	2.525	272	94	2.5	15

Table 6.3: Dijkstra synthesis results with 100MHz constraint, all configurations, and $\epsilon = 0.05$

Hash	Optimization	Dynamic P (W)	Static P (W)	WNS (ns)	LUTs	FFs	BRAMs	DSP blocks
CRC-32C	-	0.080	0.092	-5.146	1632	113	5	0
MultiplyShift	m-k	0.040	0.091	0.672	76	21	5	30
MultiplyShift	rounding	0.040	0.091	0.672	76	21	5	30
MultiplyShiftPipelined	m-k	0.040	0.091	2.613	76	22	5	30
MultiplyShiftPipelined	rounding	0.040	0.091	2.613	76	22	5	30

Table 6.4: FFT synthesis results with 100MHz constraint, all configurations and $\epsilon = 0.05$

Hash	Optimization	Dynamic P (W)	Static P (W)	WNS (ns)	LUTs	FFs	BRAMs	DSP blocks
CRC-32C	-	0.072	0.091	-3.293	1572	113	2.5	0
MultiplyShift	m-k	0.028	0.091	1.751	267	105	2.5	17
MultiplyShift	rounding	0.028	0.091	1.751	267	105	2.5	17
MultiplyShiftPipelined	m-k	0.026	0.091	2.608	278	94	2.5	15
MultiplyShiftPipelined	rounding	0.026	0.091	2.608	278	94	2.5	15

Table 6.5: Patricia synthesis results with 100MHz constraint, all configurations and $\epsilon = 0.05$

Hash	Optimization	Dynamic P (W)	Static P (W)	WNS (ns)	LUTs	FFs	BRAMs	DSP blocks
CRC-32C	-	0.063	0.084	-1.841	1401	153	0	0
MultiplyShift	m-k	0.016	0.084	3.771	191	157	0	12
MultiplyShift	rounding	0.020	0.084	2.990	239	193	0	15
MultiplyShiftPipelined	m-k	0.017	0.084	3.168	190	165	0	12
MultiplyShiftPipelined	rounding	0.021	0.084	3.058	237	201	0	15

Table 6.6: Quicksort synthesis results with 100MHz constraint, all configurations, and $\epsilon = 0.05$

Hash	Optimization	Dynamic P (W)	Static P (W)	WNS (ns)	LUTs	FFs	BRAMs	DSP blocks
CRC-32C	-	0.103	0.092	-4.451	2166	113	2.5	0
MultiplyShift	m-k	0.023	0.091	3.384	183	112	2	15
MultiplyShift	rounding	0.028	0.091	2.405	238	123	2.5	18
MultiplyShiftPipelined	m-k	0.021	0.091	2.744	196	82	2	12
MultiplyShiftPipelined	rounding	0.025	0.091	2.403	249	93	2.5	15

Table 6.7: SHA synthesis results with 100MHz constraint, all configurations, and $\epsilon = 0.05$

Table 6.8 displays the synthesis results of the instruction validator and the checkers introduced in [25] and [27]. The Sudoku Solver program synthesis results from [25] and [27] are displayed in the table. The Sudoku Solver program consists of 475 instructions which is similar to the number of instructions in the Dijkstra program which consists of 451 instructions. It can be observed that the amount of LUTs is higher than the proposal in [25] while being significantly lower than the proposal in [27]. The amount of FFs is higher than both proposals which are consumed by the instruction validator hash functions. The BRAM size is slightly bigger than [27] while being significantly lower than [25]. This has to do with the fact that the proposal in [25] allocates a fixed amount of memory to store instruction/address pairs. The F_{max} is between the proposal in [25] and [27]. The biggest difference between the instruction validator and both proposals is the amount of DSP blocks.

Checker	LUTs	FFs	BRAM size (kbit)	F_{max} (MHz)	DSP Blocks
MultiplyShift with m-k	268	127	90	175	18
MultiplyShiftPipelined with m-k	272	94	90	175	15
Proposal in [25]	1539	89	64	106	0
Proposal in [27]	75	31	208	275	0

Table 6.8: Comparing synthesis results to other checkers with Dijkstra and $\epsilon = 0.01$

6.2 Isolated implementation

This section discusses the isolated implementation. The first subsection *Test setup* discusses the isolated instruction validator test setup on an FPGA. Finally, the sec-

ond subsection *Test results* discuss the results of the isolated test setup and discuss how the tests were executed.

6.2.1 Test setup

Figure 6.1 displays the diagram of the isolated implementation.

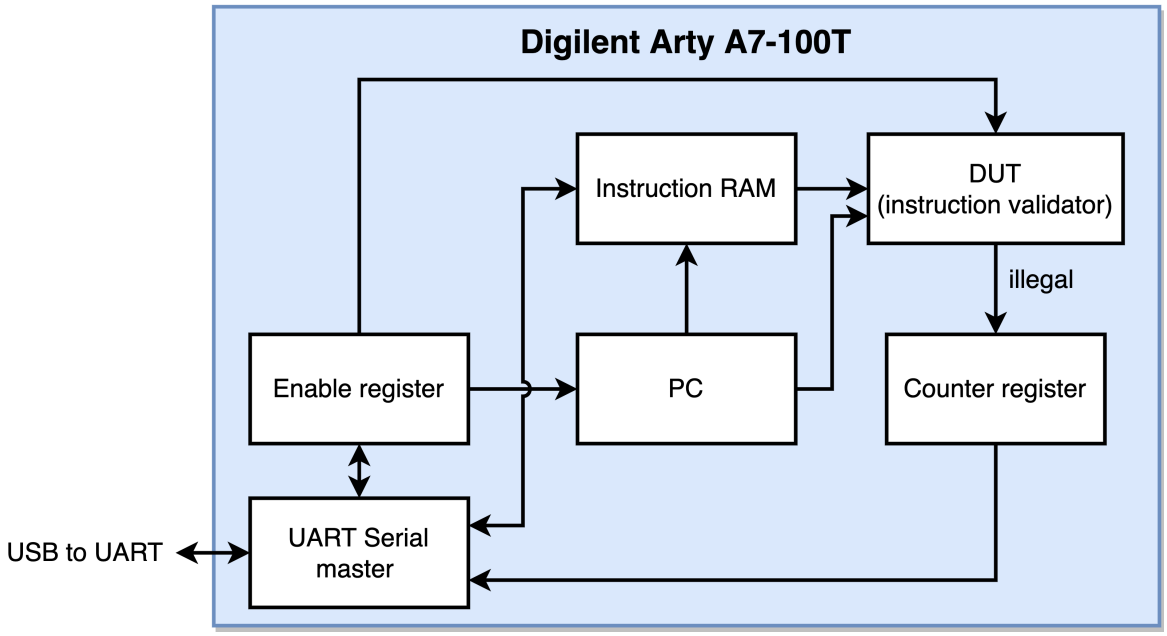


Figure 6.1: Isolated implementation diagram

The isolated implementation was again synthesized for the Digilent Arty A7-100T board which uses a Xilinx Artix-7 XC7A100TCSG324-1 FPGA [58]. The implementation consists of several peripherals to drive and verify the design-under-test (DUT). Technolution developed their own language called RDL standing for Register Description Language. RDL is a language to describe registers and specify the bus structure for different targets including FPGAs. The enable register is a control register that was described using RDL and was used to enable the PC and instruction RAM using the AXI4-lite bus. The AXI4-lite bus layout is displayed in Figure 6.2. The serial master is used to execute read and write operations on the AXI4-lite bus using UART. This was used to read and write to and from the registers and the dual-port RAM (DPRAM). The maximum value of the PC was limited in hardware to prevent going out of bounds in the instruction RAM address space. The reset input of the instruction validator was connected to the negated signal of the enable register. This was important as the instruction RAM initially doesn't contain any instructions and this results in a run of illegal instructions. The idea was to be able to dynamically write and read instructions from and to the instruction RAM and to enable the PC

and DUT. The ability to change instructions after synthesis made it easy to test if the instruction validator triggers the illegal signal when illegal instructions/address pairs are present. The instruction validator is connected to the instruction RAM, the enable register, the FPGA clock, and the illegal signal is connected to the counter register. The counter register is another register that is described using RDL. This counter register increments its content by 1 when the input signal is high at every rising edge of the clock. Hence, this register can be used to count the number of illegal instructions in one run. The reset button on the Arty 7 board can be used to reset the registers to do another test run.

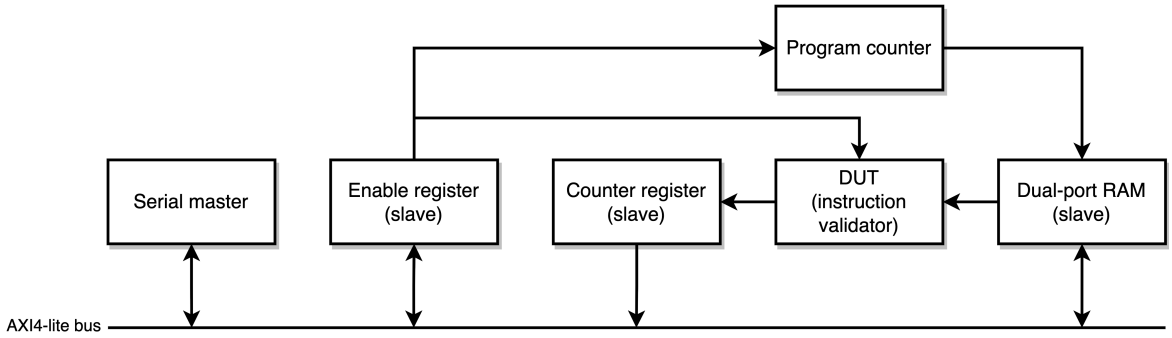


Figure 6.2: Isolated implementation bus layout diagram

6.2.2 Test results

The test setup displayed in Figure 6.1 was driven using another tool developed by Technolution called Pyte. Pyte is a Python-based tool that is used to interface with numerous types of embedded systems using different protocols such as I2C, UART, and Ethernet. To test the isolated implementation the so-called *pysciimaster* was used. This is a class that can interface with the AXI4-lite bus using the UART serial master over UART. Using UART was the most convenient protocol to use as the Digilent Arty A7 consists of USB-JTAG (which also provides the board's power) and USB-UART circuitry which means that JTAG, UART, and the board's power can be transported using a micro-USB cable. The first step before the instruction could be tested on hardware was to add nets to the ILA debugging core. ILA stands for Integrated Logic Analyzer and is an in-system debugger provided by Xilinx Vivado to test and debug hardware. The following signals were added to analyze the instruction validator:

- Address coming from The PC
- Instruction coming from instruction RAM
- Illegal output signal from the instruction validator

- PC enable signal that triggers the PC and instruction validator

The design could be synthesized after the nets corresponding to the mentioned signals were added. The instruction validator components were generated using the qsort program, the MultiplyShift hash, rounding optimization, and an FPP of 0.05. This resulted in the synthesis schematic displayed in Appendix B.4. The bitstream and `ltx` file containing the debug nets were also generated as a product of synthesis.

After synthesizing the design, the FPGA was programmed with the bitstream and the `ltx` file needed for ILA was loaded in Vivado Lab Edition. As mentioned earlier, Pyte was used to interface with the UART serial master that controls all the peripherals using the AXI4-lite bus. First, Pyte was configured using the right UART port and baud rate. A small test class was written which can drive the peripherals, e.g., enabling the PC and reading/writing to and from the instruction RAM. The class initialization is used to open the serial monitor with the port and baud rate. A list containing instructions from the disassembled qsort program was also read and imported. This instruction list was generated by the generator script. The instruction list is used by four Pyte class test methods that have the following basic functionality:

- `async def write_instructions()`: Writing the instructions to the instruction RAM and enabling the PC using the `enable_pc` register.
- `async def write_faults()`: Introducing faults in all instructions and writing the instructions to the instruction RAM and enabling the PC using the `enable_pc` register.
- `async def write_beginfault()`: Introducing a fault in the first instructions and writing the remaining instructions to the instruction RAM and enabling the PC using the `enable_pc` register.
- `async def write_endfault()`: Introducing a fault in the last instructions and writing the remaining instructions to the instruction RAM and enabling the PC using the `enable_pc` register.

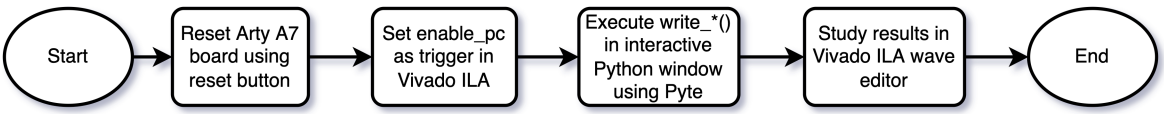


Figure 6.3: Isolated implementation test flow

The test flow that was used to test the instruction validator as displayed in Figure 6.3. This flow was used to test the instruction validator with the previously mentioned

methods. All tests were executed successfully and the behavior exactly matched the expected behavior. The different tests are discussed in detail below.

Executing the `async def write_instructions()` method resulted in the `instr_illegal` signal remaining low while fetching all the instructions from the instruction RAM as displayed in Figure 6.4. This was also reflected when reading the counter register which was 0. It can be observed that the `enable_pc` is triggered which is also indicated by the ”T” cursor. The PC starts incrementing and the instructions are fetched from the instruction RAM. This is the case for every following test.

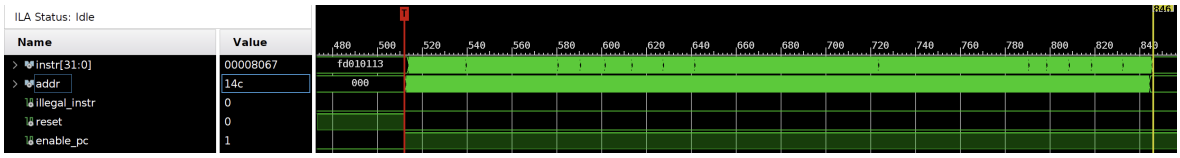


Figure 6.4: Fetching legal instructions

Executing the `async def write_faults()` method resulted in the `instr_illegal` signal remaining high most of the time with some false positives. This is displayed in Figure 6.5 and the false positives can be recognized by the notches.

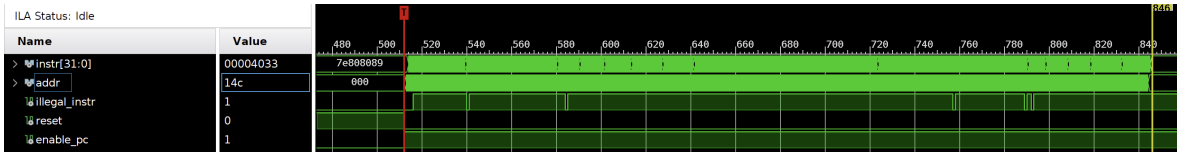


Figure 6.5: Fetching illegal instructions

Executing the `async def write_beginfault()` method resulted in the `instr_illegal` being high for one clock cycle as displayed in Figure 6.6. When looking at the zoomed-in waves in Figure 6.7 it can be observed that the `instr_illegal` becomes high with a latency of three clock cycles.

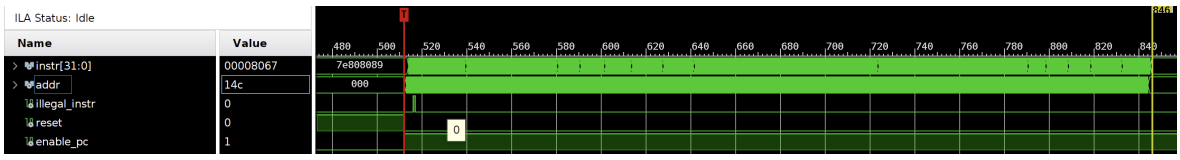


Figure 6.6: Fetching one illegal instruction from the first address

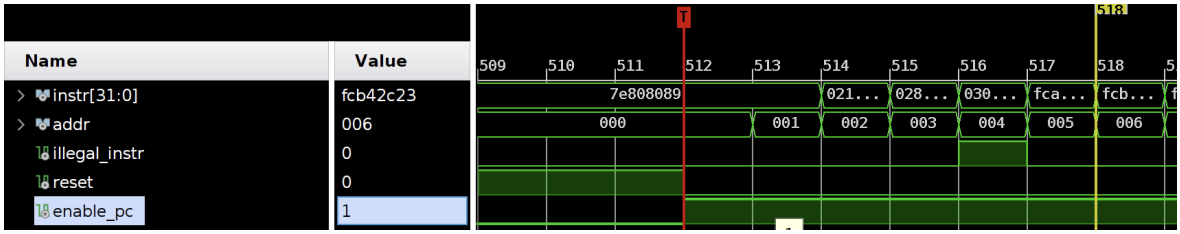


Figure 6.7: Illegal signal zoomed in

Executing the `async def write_endfault()` method resulted in the `instr_illegal` staying high as displayed in Figure 6.8. This is the case as the last instruction was fetched from the instruction RAM which resulted in the illegal instruction/address pair remaining unchanged, hence staying high. Another observation is the illegal signal becoming high with a latency of three clock cycles when looking at the zoomed-in waves in Figure 6.9.

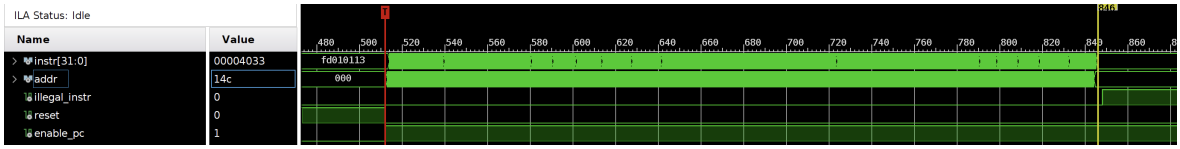


Figure 6.8: Fetching one illegal instruction from the first address

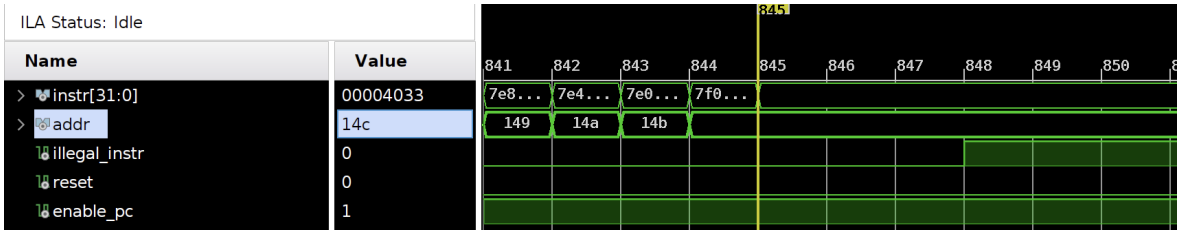


Figure 6.9: Illegal signal zoomed in

6.3 Integration with FreNox

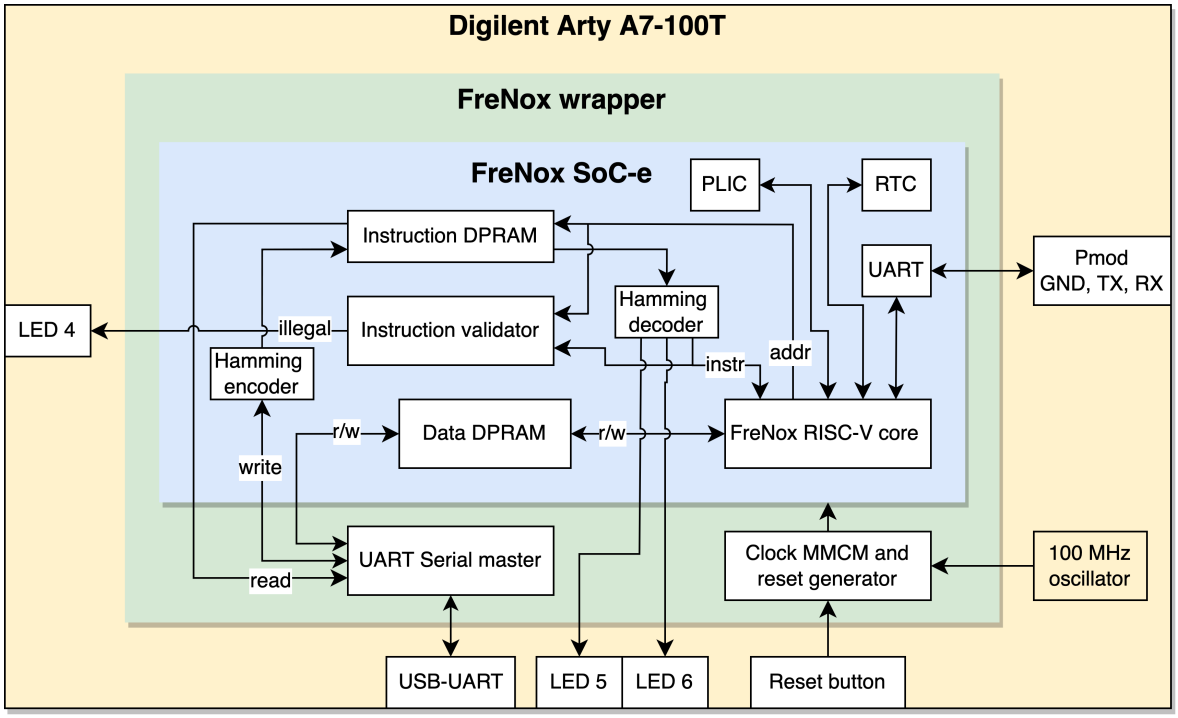


Figure 6.10: FreNox hardware overview

Figure 6.10 displays the hardware overview of the FreNox implementation. The implementation contains the FreNox SoC-e layer which instantiates the FreNox RISC-V core, peripherals (such as a platform-level interrupt controller (PLIC), and a real-time clock (RTC)), the instruction validator, ECC instruction memory, and data memory. The FreNox wrapper instantiates the FreNox SoC-e, a mixed-mode clock manager (MMCM), reset generator, and the UART serial master. The MMCM and reset generator provide the main clock and reset signals for all components. The FreNox wrapper is connected to several peripherals on the Arty board such as LEDs, a reset button, a USB-UART interface, and Pmod connector pins which serve as a secondary UART interface. The instruction validator illegal signal is connected to LED 4. The Hamming decoder single-bit error is connected to LED 5 and the double-bit error signal is connected to LED 6.

Xilinx provides BRAM IPs with ECC mode. This IP however comes with some disadvantages. The first disadvantage is portability. The Xilinx Artix-7 series FPGA supports ECC mode in BRAMs. However, this might be different for other FPGA series or brands. Another disadvantage is the fact that ECC is only supported in simple dual-port mode. This means that one DPRAM port is a dedicated write port and the other port is a dedicated read port. In that case, the instruction memory could only be written by the UART serial master and read by the FreNox RISC-V

core. This is a big disadvantage as the memory contents can't be verified using the UART serial master. Because of this, separate Hamming encoders and decoders IPs were used. While Hamming SEC is sufficient and results in an improved area overhead, Hamming SEC-DED was used for this implementation as the Xilinx ECC IP was used which only supports Hamming SEC-DED and Hsiao [59].

The design displayed in Figure 6.10 was synthesized. The FreNox SoC-e was synthesized with the RV32IM standard. An amount of 20480 kbyte was allocated to the FreNox SoC-e instruction memory and 16384 kbyte to the data memory. The instruction validator didn't have any impact on the maximum frequency of FreNox. This was however different for the ECC encoding/decoding. At the time of writing this thesis, the FreNox RISC-V core was not yet optimized for latency in the pipeline. This means that the ECC decoding is part of the instruction fetch and hence increases the critical path slack of the pipeline. The frequency had to be decreased to meet the timing requirements. Synthesizing this design with a clock frequency of 80 MHz resulted in a WNS of 0.009 ns. This means that adding the ECC encoding/decoding resulted in a frequency decrease of 20% compared to the FreNox SoC-e implementation without fault-tolerance and HWT mitigation which met timing requirements at 100 MHz.

The instruction validator was generated with the QuickSort program using the MultiplyShift hash and rounding optimization and an FPP of 0.05. The QuickSort program was compiled with the `-march=rv32im` parameter which selects the RISC-V base and extension and the `-O0` parameter which disables compiler optimizations. The QuickSort program repeatedly sorts an array generated by a Python script with 1000 random numbers between 0 and 1 million in an infinite while loop. This resulted in a text section size of 10276 bytes and a data section of 1076 bytes. The instruction validator was generated using MultiplyShift with m-k optimization. The instruction validator generator script reported 1460 instructions (n), a total bit array size of 10240 (m) and 5 hash functions (k). Table 6.9 displays the synthesis results of the FreNox RISC-V core, the ECC encoder/encoder, and the instruction validator configured with the QuickSort program. The power report showed that the total dynamic power consumption of the FreNox SoC-e was 0.078W. It can be observed that when looking at the dynamic power consumption, the ECC encoder/decoder and the instruction validator approximately consume half the power compared to the other components and FreNox SoC-e. This mainly has to do with the power that is consumed by the DSP blocks and the constant ECC decoding. The area overhead in terms of LUTs and FFs is low when looking at both overheads. While the LUT and FF area overhead is low, the BRAM and DSP block area overhead are large compared to the other components in the table and FreNox SoC-e overall. The BRAM area overhead of 38.9% has to do with the fact that each individual RAM element

is synthesized as BRAM. While one individual RAM element holds just 2 kbit, the RAM element is synthesized as half a BRAM which holds 18 kbit. All the RAM elements together result in a BRAM consumption of 2.5. The same was observed with the single BRAM that the FreNox RISC-V core is consuming which stores the core's register file with a size of 1024 bit. The precise area overhead of the instruction validator and Hamming SEC with the register file and the data memory is $\frac{10240+6\cdot1460}{32\cdot1460+1024+8608} = 33.7\%$. The DSP block area overhead is acceptable as FreNox only consumes four DSP blocks which together just consume 8% of the available DSP blocks. The DSP block area overhead is also acceptable for the other reasons discussed previously in the instruction validator synthesis results.

Component	Dynamic P (W)	LUTs	FFs	BRAMs	DSP blocks
FreNox RISC-V core	0.012	2363	1654	1	4
Instruction memory	0.019	0	0	4 (and 1 BRAM for check bits)	0
Data memory	0.013	38	0	4	0
ECC encoder (Hamming SEC-DED)	< 0.001	20	0	0	0
ECC decoder (Hamming SEC-DED)	0.009	97	0	0	0
Instruction validator	0.019	322	75	2.5	15
Overhead[1]	**63.6%**	**18.3%**	**4.5%**	**38.9%**	**375%**
Overhead[2]	**56%**	**11.2%**	**2.9%**	**38.9%**	**375%**

Table 6.9: Synthesis results of FreNox SoC-e

[1]Overhead of the instruction validator, ECC encoder/decoder and ECC check bits compared to the FreNox RISC-V core and the instruction/data memory

[2]Overhead of the instruction validator, ECC encoder/decoder and ECC check bits compared to the components in the table and other components part of the FreNox SoC-e

6.3.1 FreNox fault-injection setup

Figure 6.11 displays the hardware overview of introducing fault injection in the FreNox SoC-e. The saboteur can inject single-, double- or triple-bit errors in the codeword that are fetched from the instruction memory. The position where the bit error is injected changes every cycle and is determined by a pseudorandom number generator (PRNG) based on a 16-bit LFSR using the same polynomial that CRC-16 uses. This LFSR output is sliced to a 6-bit output. As the codeword is 39-bit, the LFSR output must be limited to stay within the limits of the codeword size. If the LFSR output is higher than 39, the output is decreased using subtraction to stay within limits. Besides the saboteur and the PRNG, counter, pulse, and control registers were described using RDL. The counter registers were used to track the number of illegal flags by the instruction validator. This will be used to verify that the instruction validator doesn't flag any instruction/address pairs after program execution when no faults are injected and when single-bit faults are injected. The control registers were used to trigger a sequence of single-, double- or triple-bit errors that are injected by the saboteur. The pulse registers were used to trigger single-, double- or triple-bit errors for the duration of one clock cycle.

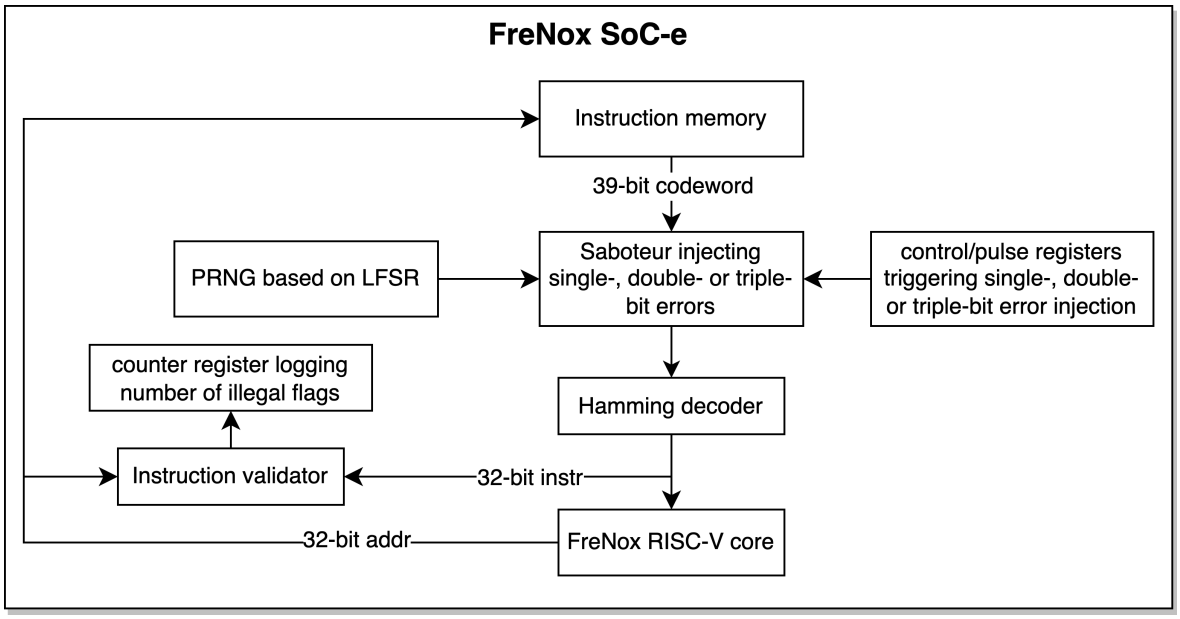


Figure 6.11: FreNox fault-injection with saboteur and PRNG

Figure 6.12 displays the test setup that was used to trigger and monitor fault-injection in FreNox. As mentioned previously, the LEDs are connected to the instruction validator and the Hamming decoder to get immediate feedback. The reset button was used to reset the FreNox SoC-e including all registers. The FreNox UART was connected to the test computer using a USB-TTL adapter. Linux screen was

used as a serial monitor. The UART serial master was connected to the computer using a micro-USB cable as the Arty has an onboard USB-UART chip. The pysciimaster of Pyte was again used like with the isolated test setup to read/write to and from the registers, upload/verify the program contents on the instruction memory, change instructions in the instruction memory and upload program data to the data memory. The same micro-USB cable was also connected to the USB-JTAG of the Arty board. JTAG was used to program the FPGA with the synthesized design displayed in Figure 6.10 and to use the internal logic analyzer (ILA) that captures the internal signals in the FPGA.

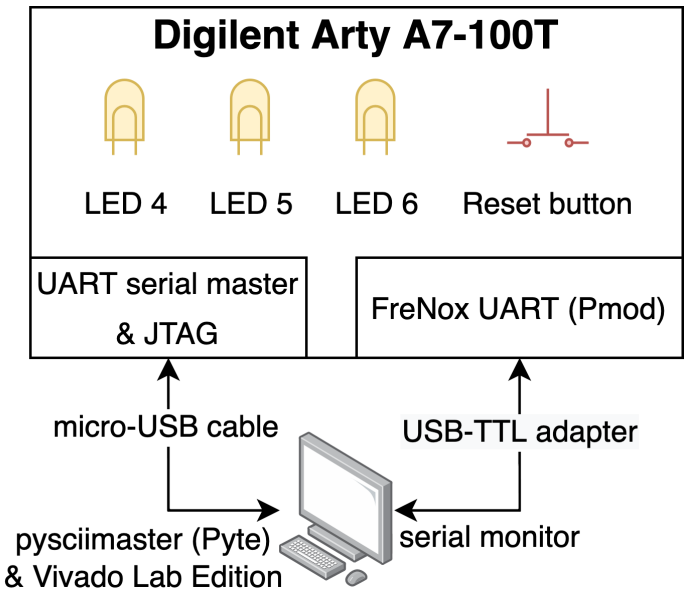


Figure 6.12: FreNox test setup

6.3.2 Testing without introducing faults and HWTs

First, the QuickSort program was uploaded to the FreNox SoC-e and the FreNox RISC-V core was enabled. After the core was enabled, the program output could be checked in the serial monitor connected to the UART of the FreNox RISC-V core. The illegal signal counter register was checked to verify the instruction validator functionality. This counter register resulted in 0 after executing the QuickSort program on FreNox. This means that the instruction validator successfully passed this test.

6.3.3 Introducing a sequence of faults

After testing without introducing faults, the instruction validator and ECC were tested by introducing a sequence of faults. The strategy from the previous subsection was re-used. However, instead of enabling the core directly, first, the control register that

enables single-bit errors was enabled. This resulted in LED 5 turning on. Again reading the illegal signal counter register resulted in 0 as displayed in Figure 6.13. This means that all single-bit faults were successfully corrected by the Hamming decoder. Figure 6.14 shows a screenshot of ILA when injecting single-bit errors. It can be observed that the single-bit error in the figure is corrected by the Hamming decoder and the instruction validator illegal signal stays low.

```
In [42]: frenox.registers.control_regs.iv_count
Out[42]: 0
```

Figure 6.13: Instruction validator counter register value after executing program

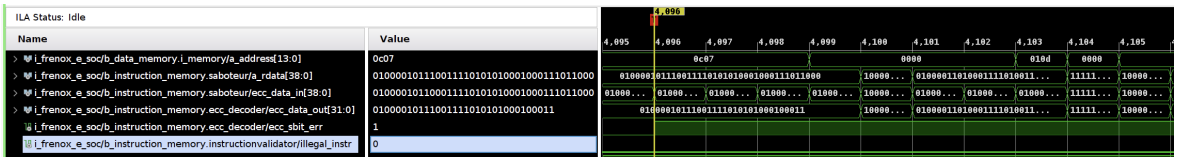


Figure 6.14: FreNox single-bit error injection

For the injection of double-bit errors, the same steps were followed but now the double-bit control register was enabled which resulted in LED 6 turning on. This resulted in the ILA screenshot displayed in Figure 6.15. It can be observed that the double-bit error isn't corrected by the Hamming decoder but the data bits on the input are forwarded to the output. This was the case because a Hamming SEC-DED decoder instead of a SEC decoder was used in the implementation. Studying the Xilinx ECC IP showed that the input is forwarded to the output when a double-bit error is detected. Next, triple-bit errors were introduced to prove the same effect that was observed in the simulation. If the Hamming decoder can't detect the error, the Hamming decoder should occasionally miscorrect bits which leads to a lower FPR for the instruction validator.

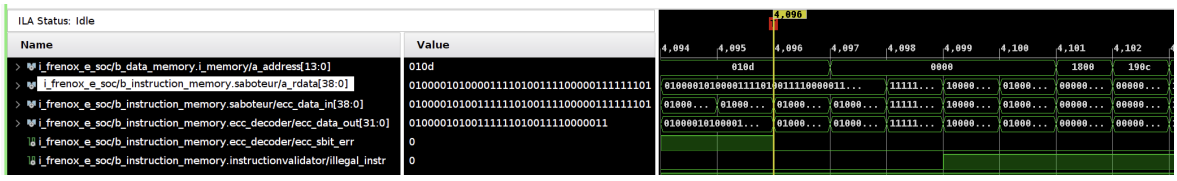


Figure 6.15: FreNox double-bit error injection

Figure 6.16 displays triple-bit error injection in FreNox. It can be observed that indeed the same effect was observed as in the simulation. The Hamming SEC-DED decoder couldn't detect triple-bit errors and miscorrected bits which resulted in four-bit errors. This failure of detecting triple-bit errors could also be observed by looking at the waves alternating between `ecc_sbit_err` and `ecc_dbit_err`.

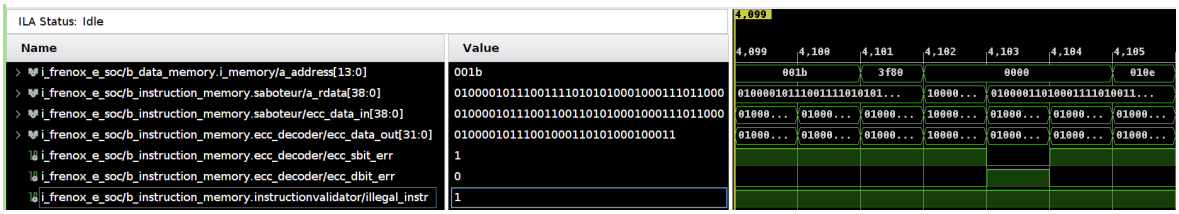


Figure 6.16: FreNox triple-bit error injection

Injecting double- and triple-bit errors resulted in an interesting observation in FreNox. When injecting errors, the FreNox jumped to the trap entry located at address 0x3a. The expected behavior would be that the trap entry is executed and jumps to the trap handler at address 0x64. However, this didn't happen. Instead, the FreNox got stuck in an infinite loop from address 0x3a to 0x40 when injecting a sequence of double- and triple-bit errors as displayed in Figure 6.17 and in some cases from address 0x3a to 0x3f as displayed in Figure 6.18. A sequence of 13 double-bit errors (displayed in Figure 6.19) lead to this behavior. This means that FreNox got stuck in an unrecoverable state, even after disabling the fault injection. This is an example where the instruction validator becomes relevant. The instruction validator can detect MBUs and hence can detect this unrecoverable error and trigger a mechanism that prevents the FreNox from hanging and crashing. This mechanism that handles these errors was however not part of the scope of this thesis. This thesis only focused on the detection and correction of SEUs and the detection of MBUs and HWTs. Error handling is part of the future work which is discussed in Chapter 7: *Conclusion and future work*.

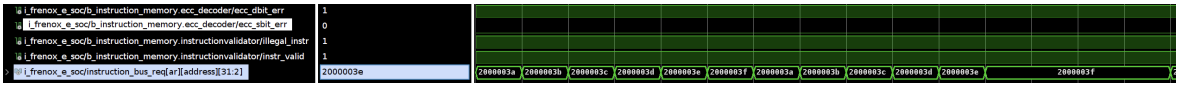


Figure 6.17: FreNox trap entry infinite loop

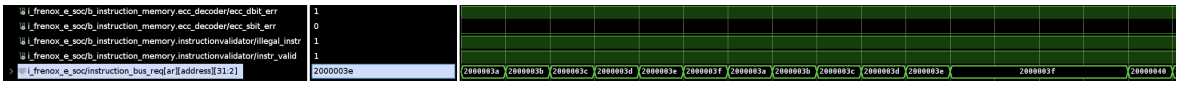


Figure 6.18: FreNox trap entry infinite loop remains after disabling fault-injection

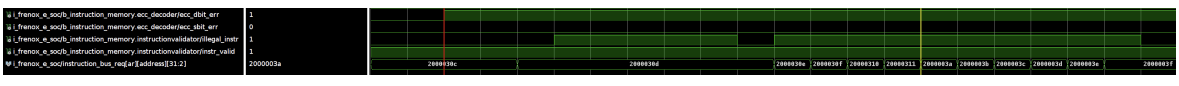


Figure 6.19: FreNox trap entry infinite loop after injecting a sequence of double-bit errors

6.3.4 Introducing single-event faults

After introducing a sequence of faults, single-event double- and triple-bit errors were injected using the pulse registers. Injecting both single-event double-bit and single-event triple-bit errors resulted in two different observations:

- The FreNox RISC-V core kept executing the QuickSort program
- An exception was raised by the trap handler and the FreNox RISC-V core was halted

As discussed previously, undetected faults while the processor is executing programs can lead to many different kinds of problems. This can for example lead to silent data corruption (SDC). The instruction validator again proves its usefulness by providing the ability to prevent these issues.

The second observation resulted in the program raising exceptions using a trap handler. First, the trap entry located at address 0x3a was called as displayed in Figure 6.20. The trap entry eventually jumped to the trap handler located at address 0x2e8 as displayed in Figure 6.21. Again, the instruction validator was able to detect this MBU which was a double-bit error in this case.

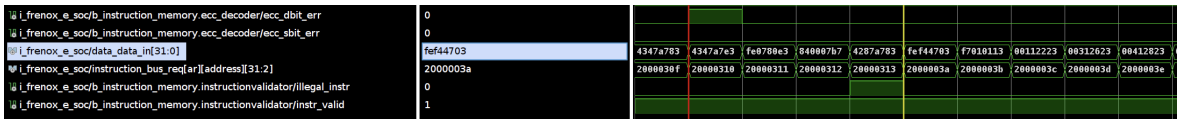


Figure 6.20: Double- and triple-bit errors trap entry

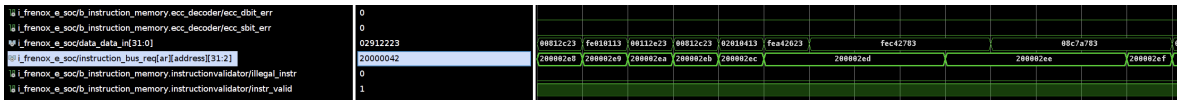


Figure 6.21: Double- and triple-bit errors trap handling

6.3.5 Introducing HWTs

The last step was injecting an HWT in FreNox. An interesting HWT attack was analyzed by studying the QuickSort assembly. Listing 6.1 displays a small part of the assembly of the QuickSort main function. The instruction at address 0x127c would be an interesting instruction to manipulate with an HWT. This jump and link instruction jumps to address 0x10b8 to execute the QuickSort algorithm. Let's consider the *modifying HWT* from the simulation chapter. This HWT overwrites the instruction with a mask to prevent the core from going into a trap state but causes

damage by skipping instructions. This mask 0x7013 was used to overwrite instruction 0xe3dff0ef. This resulted in the malfunctioning of the QuickSort program. The attack was successful and the numbers were still unsorted as displayed in the serial monitor screenshot in Figure 6.22. Figure 6.23 displays the ILA waves of the HWT which was also successfully detected by the instruction validator.

Listing 6.1: Part of the QuickSort assembly main function

```
800011e4 <main>:
    .
    .
80001270:       00070613                mv      a2,a4
80001274:       00000593                li      a1,0
80001278:       00078513                mv      a0,a5
8000127c:       e3dff0ef                jal     ra,800010b8 <quickSort>
80001280:       800027b7                lui     a5,0x80002
80001284:       86078513                addi    a0,a5,-1952 # 80001860 <_fstac..
80001288:       ab5ff0ef                jal     ra,80000d3c <UART_puts>
```

```
Sorted array in ascending order:
                                [88389, 200089, 547511, 682645, 710288, 533081,
819, 137693, 696780, 578592, 504698, 761456, 112101, 965946, 323118, 722383, 615
```

Figure 6.22: Hardware Trojan resulting in QuickSort malfunctioning

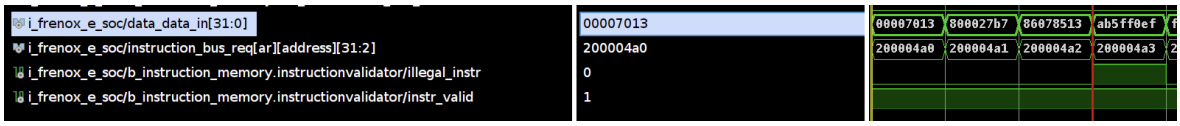


Figure 6.23: Hardware Trojan waves

6.4 Concluding summary

To conclude, this chapter presented the instruction validator synthesis results, implementation in an isolated test setup, and integration with the FreNox SoC-e and ECC instruction memory.

Section 6.1 discussed the instruction validator synthesis results of all program/hash/optimization configurations. The most notable observation was CRC-32C failing the timing requirements because the modulo operation resulted in too large a slack. The configurations using the MultiplyShift and MultiplyShiftPipelined resulted in the lowest area, power, and latency overhead while consuming DSP blocks. Comparing the synthesis results with the first proposal showed that the instruction validator consumes fewer LUTs, FFs, and BRAMs and achieves a higher maximum frequency.

Comparing the synthesis results to the second proposal showed that while using more LUTs and FFs and achieving a lower maximum frequency, fewer BRAMs were consumed.

Section 6.2 discussed the isolated implementation of the instruction validator. Peripherals were added and the instruction validator was tested and analyzed on an FPGA fabric using the internal logic analyzer and Pyte. An instruction memory, PC, registers, and a serial master were added. The instruction memory and PC were used as inputs to the instruction validator. The serial master was used to read/write to and from the registers that enabled the PC and the instruction validator. A counter register was used to count the number of illegal flags. Multiple tests were executed and analyzed using the internal logic analyzer. All tests proved that the behavior matched the expected behavior as defined in the hardware design.

Section 6.3 discussed the integration of the instruction validator with the FreNox SoC-e and ECC instruction memory. The FreNox SoC-e was synthesized with a separate Hamming SEC-DED encoder/decoder and the instruction validator was configured with the MultiplyShift hash and m-k optimization. The Hamming decoder being part of the pipeline critical path resulted in a frequency of 80 MHz, equal to a decrease of 20%. The instruction validator didn't introduce any latency overhead. Compared to all components instantiated by the FreNox SoC-e, the instruction validator and Hamming encoder/decoder introduced a power overhead of 56%, a LUT overhead of 11.2%, and a FF overhead of 2.9%. The area overhead resulted in 33.7% including the 6 check bits introduced by Hamming SEC encoding. A DSP block overhead of 375% was introduced which is acceptable as this makes up for just 7.9% of the available DSP blocks with the 4 DSP blocks consumed by the FreNox RISC-V core. Finally, the same tests that were introduced in the simulation were executed with FreNox. This again proved the functionality and advantages of both the ECC instruction memory and the instruction validator. All single-bit errors were detected and corrected by the Hamming decoder. Introducing double- and triple-bit faults resulted in hangs and crashes in the FreNox RISC-V core which was successfully detected by the instruction validator. An HWT attack was introduced based on the *modifying HWT* that overwrote a jump and link instruction. This resulted in the program bypassing a subroutine and failing its main task. While this attack only consisted of one instruction, it was also successfully detected by the instruction validator.

Chapter 7

Conclusion and future work

This chapter concludes this thesis. The results are discussed and a reflection on the objectives is discussed. The future work discusses possible improvements and further development of the techniques that were introduced in this thesis.

7.1 Conclusion

This thesis introduced a novel solution that can detect HWTs, SEUs, and MBUs in the instruction memory of an embedded RISC-V core. HWTs and MBUs can be detected by checking instruction/address pairs using to so-called *instruction validator* which uses a BF probabilistic data structure. To detect and correct SEUs, an ECC instruction memory using Hamming SEC was proposed.

With the detection capabilities of the instruction validator in mind and after extensively researching different ECC techniques, the Hamming SEC ECC implementation has been proven as the most effective technique to correct SEUs. It has been proven that Hamming SEC-DED is redundant and results in a higher area overhead. To conclude, Hamming SEC proved to provide the best performance in terms of area and latency overhead.

Extensive research was carried out to find the most effective way to store the hashed instruction/address pairs in hardware. Different probabilistic data structures were thoroughly analyzed, both theoretically and practically by analyzing their respective area and latency overhead. It has been proven that while the CF is a good BF contestant, the BF has an overall better area overhead and has been proven to have a low latency overhead.

Combining both the instruction validator and ECC proved the effectiveness of correcting and detecting SEUs and detecting MBUs and HWTs. Besides this, it has also been proven that Hamming SEC has a positive side-effect on the instruction validator which leads to better overall detection of HWTs and MBUs.

Using the MultiplyShift hash in the instruction validator resulted in the best overall FPR results when detecting MBUs and HWTs. The MultiplyShift hash also showed to have the best area and latency overhead compared to CRC-32C. The pipelined MultiplyShift introduces one extra clock latency while resulting in an improved WNS and some cases a lower amount of DSP blocks. Two different optimizations were introduced. The rounding optimization results in the best FPR while often using more area than optimal. This is the reason why an m-k optimization algorithm was introduced which approximates the most optimal bit array size and amount of hash functions while respecting the FPR with a small deviation. To conclude, different options were introduced to improve the area or latency overhead and it's up to the designer to determine the best setup based on the application requirements.

Introducing single- and multiple-bit errors in FreNox proved the importance of combining ECC with the instruction validator to prevent hangs, crashes, SDC, and HWT attacks. The instruction validator and ECC introduced a total area overhead of 33.7% compared to the FreNox SoC-e. The instruction validator didn't introduce any latency overhead. The Hamming decoder increased the critical path of the pipeline which lead to a 20% decrease in the maximum frequency from 100 MHz to 80 MHz.

Besides introducing the instruction validator design, an automation framework was developed in this thesis which was used to generate the hardware descriptions of all possible instruction validator configurations discussed in this thesis. Apart from generating hardware descriptions, this framework was also used to automate all the instruction validator cocotb simulations and generate Vivado tcl files which were used to synthesize the different configurations.

To summarize, a novel solution to improve the security and reliability of RISC-V soft-cores with a low area and latency overhead was introduced in this thesis. It has been proven that this so-called instruction validator can effectively detect HWTs and MBUs in the instruction memory by checking instruction/address pairs using a BF probabilistic data structure. ECC instruction memory using Hamming SEC was proposed to detect and correct SEUs which also besides error correction has proved to improve the detection performance of the instruction validator. An automation framework was developed to generate, simulate and synthesize the instruction validator for different configurations which presents the designer with different options based on the application requirements. Besides this automation framework, two BF optimizations were proposed that decrease the BF area overhead. To conclude, the instruction validator and ECC were successfully tested and integrated in the FreNox SoC-e with the FreNox RISC-V core on the Digilent Arty A7-100T development board using the Xilinx Artix-7 XC7A100TCSG324-1 FPGA. Integrating the instruction validator and ECC led to an area overhead of 33.7%. The introduction of ECC also resulted in a maximum frequency reduction of 20%.

This proves that the instruction validator and ECC instruction memory are suitable to use for embedded RISC-V soft-cores with strict security and reliability requirements.

7.2 Future work

7.2.1 Handling the illegal signal

The instruction validator flags illegal instruction/address pairs with a certain FPP using the illegal signal. This thesis only focused on analyzing this signal and not how to handle it. Handling this illegal signal in the RISC-V core can lead to many different research questions and error handling techniques that implement the ECC instruction memory and instruction validator.

7.2.2 Further research on the effect of Hamming decoding on the instruction validator

As discussed in the simulation with Hamming SEC decoding and in the implementation with Hamming SEC-DED decoding, the decoder introduces an extra bit error if the MBU can't be detected (Hamming SEC double-bit errors or more and SEC-DED, triple-bit errors or more). This leads to a better FPR as instructions that contain this extra bit error are more likely to be detected by the instruction validator. It would be interesting to research this behavior in more depth. An interesting approach would be to analyze the FPR and area overhead trade-off between Hamming SEC and Hamming SEC-DED using different FPPs.

7.2.3 Adding multiplier stages for high-performance embedded systems

While the instruction validator using MultiplyShift and MultiplyShitPipelined met the timing requirements, it would be interesting to research further performance improvements. A possible way to achieve this would be to add multiplier stages. This could lead to a higher maximum frequency with the cost of some latency.

7.2.4 Researching the effectiveness of this proposal in ASICs

This thesis focused on integrating the instruction validator with the FreNox RISC-V softcore and FreNox SoC-e on an FPGA fabric. The MultiplyShift hash uses

DSP blocks while using fewer LUTs and FFs. This may however be a disadvantage for application-specific integrated circuits (ASICs) as multipliers take a significant amount of area in ASICs. It might be interesting to research this and possible design modifications to optimize the instruction validator for use in ASICs.

Bibliography

[1] R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor, "Trustworthy hardware: Identifying and classifying hardware trojans," *Computer*, vol. 43, no. 10, pp. 39–46, 2010.

[2] E. A. Waterman and K. Asanovic, "The risc-v instruction set manual, volume i: User-level isa, document version 20191213," December 2019.

[3] "Research: Research areas: Caes." [Online]. Available: https://www.utwente.nl/en/eemcs/caes/Research/

[4] "About technolution," Feb 2021. [Online]. Available: https://www.technolution.com/about-technolution/

[5] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 2nd ed. Morgan Kaufmann, 2021.

[6] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor, "Hardware trojans: Lessons learned after one decade of research," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 22, no. 1, May 2016. [Online]. Available: https://doi.org/10.1145/2906147

[7] S. Adee, "The hunt for the kill switch," Jun 2021. [Online]. Available: https://spectrum.ieee.org/the-hunt-for-the-kill-switch

[8] X. Zhang and M. Tehranipoor, "Case study: Detecting hardware trojans in third-party digital ip cores," in *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*, 2011, pp. 67–70.

[9] E. Petersen, *Single event effects in aerospace*. IEEE Press, 2011.

[10] "Electronic voting random spontaneous bit inversion explained." [Online]. Available: https://web.archive.org/web/20070927185155/http://wiki.ael.be/index.php/ElectronicVotingRandomSpontaneousBitInversionExplained

[11] "About RISC-V." [Online]. Available: https://riscv.org/about/

[12] K. Asanović and D. A. Patterson, “Instruction sets should be free: The case for risc-v,” *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[13] A. Waterman and K. Asanovic, “The risc-v instruction set manual, volume i: Unprivileged isa, document version 20191213,” Dec 2019.

[14] S. Di Mascio, A. Menicucci, G. Furano, C. Monteleone, and M. Ottavi, “The case for risc-v in space,” in *Applications in Electronics Pervading Industry, Environment and Society*, S. Saponara and A. De Gloria, Eds. Cham: Springer International Publishing, 2019, pp. 319–325.

[15] A. Dörflinger, Y. Guan, S. Michalik, S. Michalik, J. Naghmouchi, and H. Michalik, “Ecc memory for fault tolerant risc-v processors,” in *Architecture of Computing Systems – ARCS 2020*, A. Brinkmann, W. Karl, S. Lankes, S. Tomforde, T. Pionteck, and C. Trinitis, Eds. Cham: Springer International Publishing, 2020, pp. 44–55.

[16] D. A. Santos, L. M. Luza, C. A. Zeferino, L. Dilillo, and D. R. Melo, “A low-cost fault-tolerant risc-v processor for space systems,” in *2020 15th Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2020, pp. 1–5.

[17] W. Heida, “Towards a fault tolerant risc-v softcore,” in *Master thesis*, 2016.

[18] [Online]. Available: https://www.gaisler.com/

[19] A. Ramos, J. A. Maestro, and P. Reviriego, “Characterizing a risc-v sram-based fpga implementation against single event upsets using fault injection,” *Microelectronics Reliability*, vol. 78, pp. 205–211, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0026271417304493

[20] A. E. Wilson and M. Wirthlin, “Neutron radiation testing of fault tolerant risc-v soft processor on xilinx sram-based fpgas,” in *2019 IEEE Space Computing Conference (SCC)*, 2019, pp. 25–32.

[21] M. Ottavi, S. Pontarelli, A. Leandri, and A. Salsano, “Design and evaluation of a hardware on-line program-flow checker for embedded microcontrollers,” in *2006 21st IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2006, pp. 371–379.

[22] T. Lu, “A survey on RISC-V security: Hardware and architecture,” *CoRR*, vol. abs/2107.04175, 2021. [Online]. Available: https://arxiv.org/abs/2107.04175

[23] T. Linscott, P. Ehrett, V. Bertacco, and T. Austin, “Swan: Mitigating hardware trojans with design ambiguity,” in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–7.

[24] J. Takahashi, K. Okabe, H. Itoh, X.-T. Ngo, S. Guilley, R.-R. Shrivastwa, M. Ahmed, and P. Lejoly, “Machine learning based hardware trojan detection using electromagnetic emanation,” in *Information and Communications Security*, W. Meng, D. Gollmann, C. D. Jensen, and J. Zhou, Eds. Cham: Springer International Publishing, 2020, pp. 3–19.

[25] A. Bolat, L. Cassano, P. Reviriego, O. Ergin, and M. Ottavi, “A microprocessor protection architecture against hardware trojans in memories,” in *2020 15th Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2020, pp. 1–6.

[26] T. Hoque, X. Wang, A. Basak, R. Karam, and S. Bhunia, “Hardware trojan attacks in embedded memory,” in *2018 IEEE 36th VLSI Test Symposium (VTS)*, 2018, pp. 1–6.

[27] A. Palumbo, L. Cassano, P. Reviriego, G. Bianchi, and M. Ottavi, “A lightweight security checking module to protect microprocessors against hardware trojan horses,” in *2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2021, pp. 1–6.

[28] S. Di Mascio, A. Menicucci, E. Gill, G. Furano, and C. Monteleone, “Leveraging the openness and modularity of risc-v in space,” *Journal of Aerospace Information Systems*, vol. 16, no. 11, pp. 454–472, 2019. [Online]. Available: https://doi.org/10.2514/1.I010735

[29] A. Gakhov, *Probabilistic data structures and algorithms for big data applications*. Books on Demand, 2019.

[30] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Commun. ACM*, vol. 13, no. 7, p. 422–426, jul 1970. [Online]. Available: https://doi-org.ezproxy2.utwente.nl/10.1145/362686.362692

[31] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, “Summary cache: A scalable wide-area web cache sharing protocol,” *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, p. 281–293, jun 2000. [Online]. Available: https://doi-org.ezproxy2.utwente.nl/10.1109/90.851975

[32] M. Canim, G. A. Mihaila, B. Bhattacharjee, C. A. Lang, and K. A. Ross, “Buffered bloom filters on solid state storage,” in *ADMS@VLDB*, 2010.

[33] G. Lu, B. Debnath, and D. H. Du, "A forest-structured bloom filter with flash memory," in *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, 2011, pp. 1–6.

[34] A. Abdennebi and K. Kaya, "A bloom filter survey: Variants for different domain applications," *CoRR*, vol. abs/2106.12189, 2021. [Online]. Available: https://arxiv.org/abs/2106.12189

[35] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok, "Don't thrash: How to cache your hash on flash," *Proc. VLDB Endow.*, vol. 5, no. 11, p. 1627–1637, jul 2012. [Online]. Available: https://doi-org.ezproxy2.utwente.nl/10.14778/2350229.2350275

[36] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 75–88. [Online]. Available: https://doi.org/10.1145/2674005.2674994

[37] P. Reviriego, J. Martínez, D. Larrabeiti, and S. Pontarelli, "Cuckoo filters and bloom filters: Comparison and application to packet classification," *IEEE Transactions on Network and Service Management*, vol. 17, no. 4, pp. 2690–2701, 2020.

[38] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[39] M. Y. Hsiao, "A class of optimal minimum odd-weight-column sec-ded codes," *Ibm Journal of Research and Development*, vol. 14, pp. 395–401, 1970.

[40] T. K. Moon, *Error correction coding: mathematical methods and algorithms*. John Wiley & Sons, Inc., 2021.

[41] G. Tshagharyan, G. Harutyunyan, S. Shoukourian, and Y. Zorian, "Experimental study on hamming and hsiao codes in the context of embedded applications," in *2017 IEEE East-West Design Test Symposium (EWDTS)*, 2017, pp. 1–4.

[42] A. Hocquenghem, "Codes correcteurs d'erreurs," *Chiffers*, vol. 2, pp. 147–156, 1959.

[43] R. C. Bose and D. K. Ray-Chaudhuri, "On a class of error correcting binary group codes," *Information and control*, vol. 3, no. 1, pp. 68–79, 1960.

[44] R. Naseer and J. Draper, “Parallel double error correcting code design to mitigate multi-bit upsets in srams,” in *ESSCIRC 2008-34th European Solid-State Circuits Conference*. IEEE, 2008, pp. 222–225.

[45] P. Reviriego, C. Argyrides, and J. A. Maestro, “Efficient error detection in double error correction bch codes for memory applications,” *Microelectronics Reliability*, vol. 52, no. 7, pp. 1528–1530, 2012.

[46] R. Dobai and J. Korenek, “Evolution of non-cryptographic hash function pairs for fpga-based network applications,” in *2015 IEEE Symposium Series on Computational Intelligence*, 2015, pp. 1214–1219.

[47] L. Kekely, M. Žádník, J. Matoušek, and J. Kořenek, “Fast lookup for dynamic packet filtering in fpga,” in *17th International Symposium on Design and Diagnostics of Electronic Circuits Systems*, 2014, pp. 219–222.

[48] M. J. Lyons and D. Brooks, “The design of a bloom filter hardware accelerator for ultra low power systems,” in *Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED ’09. New York, NY, USA: Association for Computing Machinery, 2009, p. 371–376. [Online]. Available: https://doi.org/10.1145/1594233.1594330

[49] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen, “A reliable randomized algorithm for the closest-pair problem,” *Journal of Algorithms*, vol. 25, no. 1, pp. 19–51, 1997. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0196677497908737

[50] M. Thorup, “High speed hashing for integers and strings,” *CoRR*, vol. abs/1504.06804, 2015. [Online]. Available: http://arxiv.org/abs/1504.06804

[51] H. Feistel, “Cryptography and Computer Privacy,” *Scientific American*, vol. 228, no. 5, pp. 15–23, May 1973.

[52] A. Appleby, “Murmurhash.” [Online]. Available: https://sites.google.com/site/murmurhash/

[53] “Using the murmur3 field: Elasticsearch plugins and integrations [8.1].” [Online]. Available: https://www.elastic.co/guide/en/elasticsearch/plugins/current/mapper-murmur3-usage.html

[54] “Murmurhash3 (apache commons codec 1.15 api).” [Online]. Available: https://commons.apache.org/proper/commons-codec/apidocs/org/apache/commons/codec/digest/MurmurHash3.html

[55] A. Karunaratne, “PHP 8.1: MurmurHash3 hash algorithm support.” [Online]. Available: https://php.watch/versions/8.1/MurmurHash3

[56] “Welcome to cocotb’s documentation!” 2022. [Online]. Available: https://docs.cocotb.org/en/stable/

[57] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, “Mibench: A free, commercially representative embedded benchmark suite,” in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001, pp. 3–14.

[58] “Arty A7 - Digilent Reference.” [Online]. Available: https://digilent.com/reference/programmable-logic/arty-a7/start

[59] Xilinx, “Xilinx documentation portal - ecc v2.0 product guide.” [Online]. Available: https://docs.xilinx.com/v/u/en-US/pg092-ecc

Appendix A

Injecting faults without Hamming decoder

Benchmark	$\mathbf{n}$	$\mathbf{m}$	$\mathbf{k}$	FPP	FPR	Result
aes	6171	38480	5	0.051018	0.089127	FAIL
blowfish	24710	154075	5	0.051026	0.088588	FAIL
dijkstra	451	2815	5	0.050857	0.101996	FAIL
fft	26004	162140	5	0.051029	0.088756	FAIL
patricia	765	4770	5	0.051027	0.091503	FAIL
sha	627	3910	5	0.051007	0.106858	FAIL
qsort	333	2080	5	0.050736	0.117117	FAIL

Table A.1: Double-bit errors test case results of 10 runs with $\epsilon = 0.05$ without optimization and Hamming decoder using CRC-32C

Benchmark	$\mathbf{n}$	$\mathbf{m}$	$\mathbf{k}$	FPP	FPR	Result
aes	6171	38480	5	0.051018	0.078418	FAIL
blowfish	24710	154075	5	0.051026	0.076482	FAIL
dijkstra	451	2815	5	0.050857	0.071796	FAIL
fft	26004	162140	5	0.051029	0.076001	FAIL
patricia	765	4770	5	0.051027	0.075033	FAIL
sha	627	3910	5	0.051007	0.073716	FAIL
qsort	333	2080	5	0.050736	0.086787	FAIL

Table A.2: Triple-bit errors test case results of 10 runs with $\epsilon = 0.05$ without optimization and Hamming decoder using CRC-32C

Benchmark	n	m	k	FPP	FPR	Result
aes	6171	40960	5	0.0415	0.069033	FAIL
blowfish	24710	163840	5	0.041647	0.080089	FAIL
dijkstra	451	3072	3	0.045209	0.110865	FAIL
fft	26004	163840	5	0.049319	0.08641	FAIL
patricia	765	5120	5	0.04036	0.082353	FAIL
sha	627	4096	4	0.043962	0.094099	FAIL
qsort	333	2048	4	0.052274	0.096096	FAIL

Table A.3: Double-bit errors test case results of 10 runs with $\epsilon = 0.05$ and m-k optimization using the MultiplyShift hash and without Hamming decoder

Benchmark	n	m	k	FPP	FPR	Result
aes	6171	40960	5	0.0415	0.059268	FAIL
blowfish	24710	163840	5	0.041647	0.06706	FAIL
dijkstra	451	3072	3	0.045209	0.072461	FAIL
fft	26004	163840	5	0.049319	0.073846	FAIL
patricia	765	5120	5	0.04036	0.06732	FAIL
sha	627	4096	4	0.043962	0.062807	FAIL
qsort	333	2048	4	0.052274	0.078919	FAIL

Table A.4: Triple-bit errors test case results of 10 runs with $\epsilon = 0.05$ and m-k optimization using the MultiplyShift hash and without Hamming decoder

Benchmark	n	m	k	FPP	FPR	Result
aes	6171	40960	5	0.0415	0.069033	FAIL
blowfish	24710	163840	5	0.041647	0.080089	FAIL
dijkstra	451	5120	5	0.005737	0.04878	PASS
fft	26004	163840	5	0.049319	0.08641	FAIL
patricia	765	5120	5	0.04036	0.082353	FAIL
sha	627	5120	5	0.02013	0.073365	FAIL
qsort	333	2560	5	0.024995	0.066066	FAIL

Table A.5: Double-bit errors test case results of 10 runs with $\epsilon = 0.05$ and rounding optimization using the MultiplyShift hash and without Hamming decoder

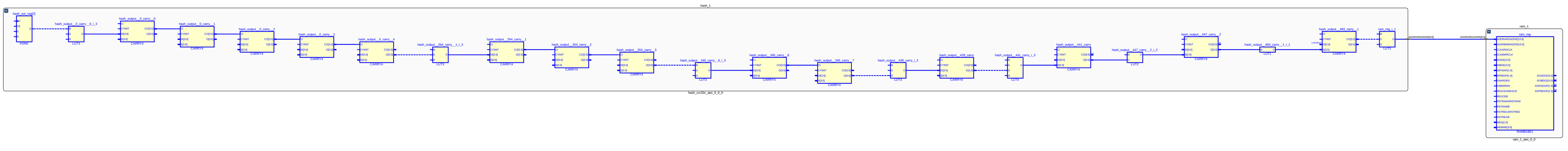
Benchmark	$\mathbf{n}$	$\mathbf{m}$	$\mathbf{k}$	**FPP**	**FPR**	**Result**
aes	6171	40960	5	0.0415	0.059268	FAIL
blowfish	24710	163840	5	0.041647	0.06706	FAIL
dijkstra	451	5120	5	0.005737	0.032062	PASS
fft	26004	163840	5	0.049319	0.073846	FAIL
patricia	765	5120	5	0.04036	0.06732	FAIL
sha	627	5120	5	0.02013	0.044817	PASS
qsort	333	2560	5	0.024995	0.05033	PASS

Table A.6: Triple-bit errors test case results of 10 runs with $\epsilon = 0.05$ and rounding optimization using the MultiplyShift hash and without Hamming decoder

Appendix B

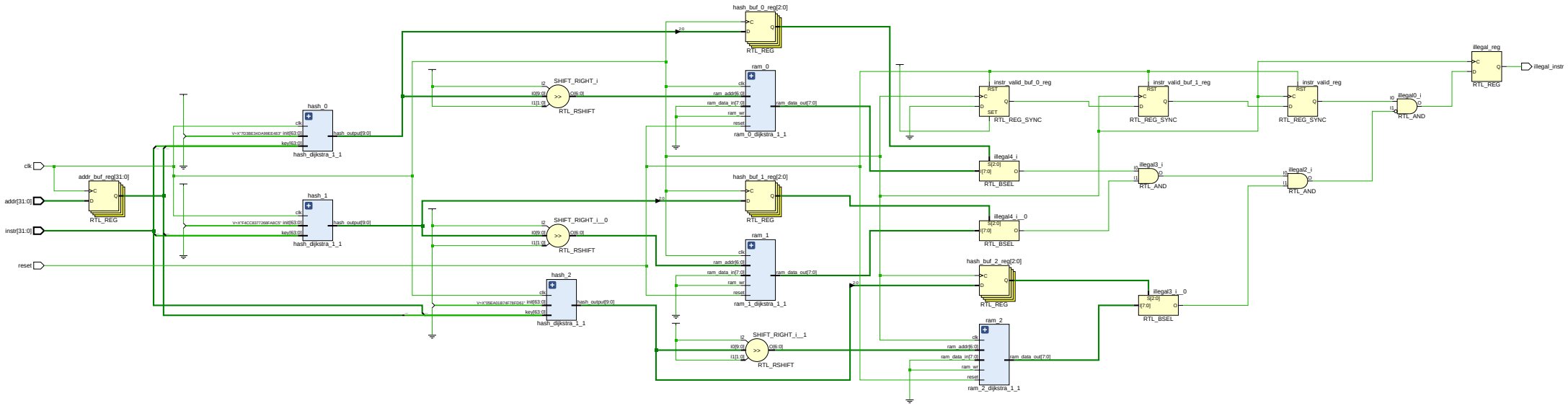
Diagrams

B.1 Negative slack diagram

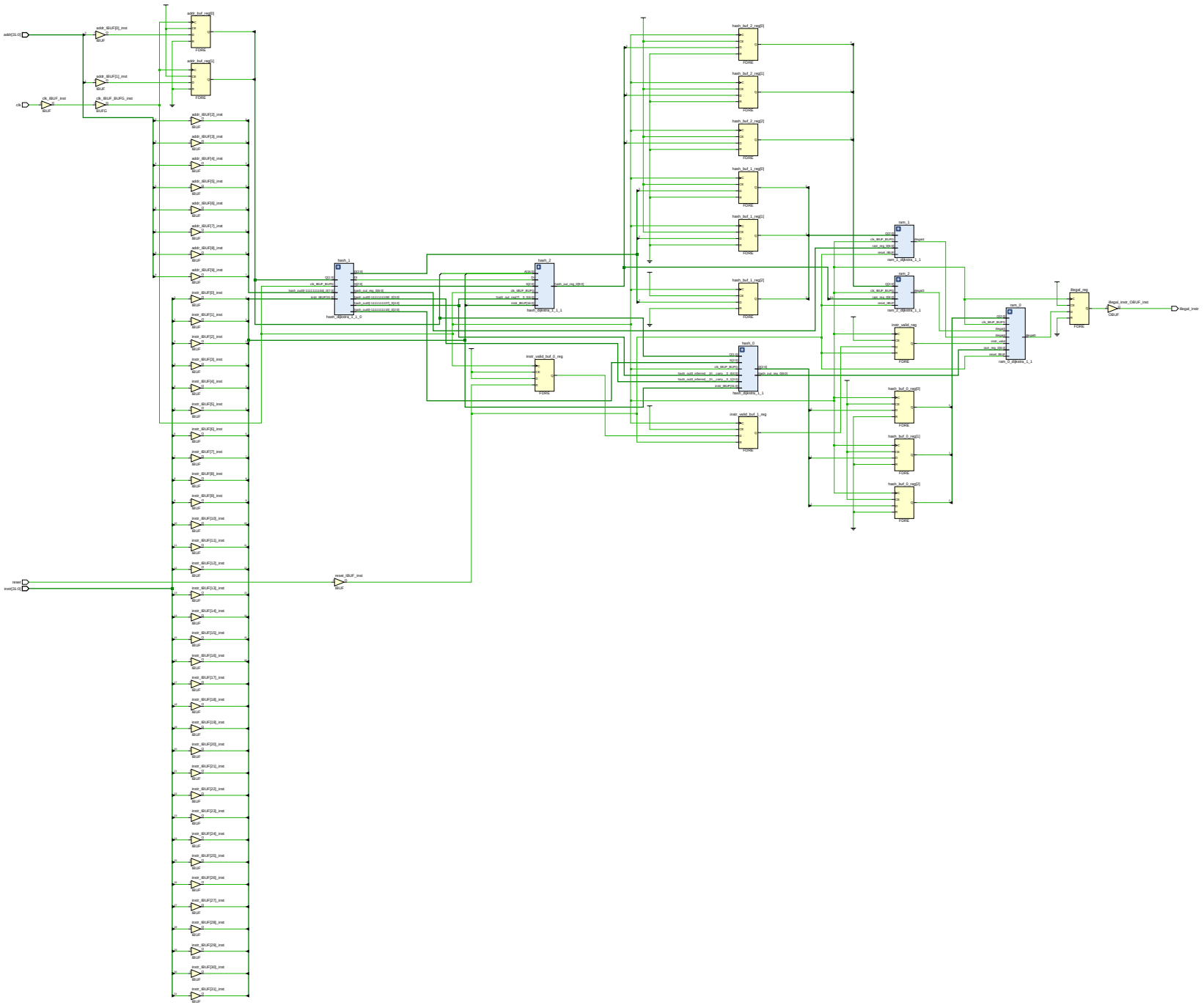


hash_1
hash_out_reg[2]
FDRE
hash_output__0_carry__0_i_3
LUT3
hash_output__0_carry__0
CARRY4
hash_output__0_carry__1
CARRY4
hash_output__0_carry__2
CARRY4
hash_output__0_carry__3
CARRY4
hash_output__0_carry__4
CARRY4
hash_output__483_carry__1
CARRY4
ram_reg_i_1
LUT3
ADDRARDADDR[9:0]
ADDRARDADDR[9:0]
ram_1
ram_reg
ADDRARDADDR[13:0]
ADDRBWRADDR[13:0]
CLKARDCLK
CLKBWRCLK
DIADI[15:0]
DIBDI[15:0]
DIPADIP[1:0]
DIPBDIP[1:0]
ENARDEN
ENBWREN
REGCEAREGCE
REGCEB
RSTRAMARSTRAM
RSTRAMB
RSTREGARSTREG
RSTREGB
WEA[1:0]
WEBWE[3:0]
DOADO[15:0]
DOBDO[15:0]
DOPADOP[1:0]
DOPBDOP[1:0]
RAMB18E1
ram_1_aes_0_0

B.2 Instruction validator schematic


clk
addr[31:0]
instr[31:0]
reset
addr_buf_reg[31:0]
RTL_REG
hash_0
hash_dijkstra_1_1_1
hash_output[9:0]
hash_1
hash_dijkstra_1_1_1
hash_output[9:0]
hash_2
hash_dijkstra_1_1_1
hash_output[9:0]
SHIFT_RIGHT_i
RTL_RSHIFT
SHIFT_RIGHT_i__0
RTL_RSHIFT
SHIFT_RIGHT_i__1
RTL_RSHIFT
hash_buf_0_reg[2:0]
RTL_REG
hash_buf_1_reg[2:0]
RTL_REG
hash_buf_2_reg[2:0]
RTL_REG
ram_0
ram_0_dijkstra_1_1_1
ram_1
ram_1_dijkstra_1_1_1
ram_2
ram_2_dijkstra_1_1_1
instr_valid_buf_0_reg
RTL_REG_SYNC
instr_valid_buf_1_reg
RTL_REG_SYNC
instr_valid_reg
RTL_REG_SYNC
illegal4_i
RTL_BSEL
illegal4_i__0
RTL_BSEL
illegal3_i__0
RTL_BSEL
illegal3_i
RTL_AND
illegal2_i
RTL_AND
illegal0_i
RTL_AND
illegal_reg
RTL_REG
illegal_instr

B.3 Instruction validator synthesized



B.4 Instruction validator debug