THÈSE POUR OBTENIR LE GRADE DE DOCTEUR DE L'UNIVERSITÉ DE MONTPELLIER

En Électronique

École doctorale : Information, Structures et Système (I2S)

Unité de recherche : Institut d'Electronique et des Systèmes (IES)

Conception et évaluation de la fiabilité d'une unité vectorielle pour l'inférence d'intelligence artificielle sur un système sur puce RISC-V tolérant aux fautes

(Development and Reliability Assessment of a Vector Extension Unit for Artificial Intelligence Inference on Fault-Tolerant RISC-V System-on-Chip)

Présentée par Carolina IMIANOSKY

Le 19 décembre 2025

Sous la direction de Luigi DILILLO

Devant le jury composé de

Luigi DILILLO, Chargé de Recherche, Université de Montpellier	Directeur de thèse
Luis Alfonso ENTRENA ARRONTES, Full Professor, University Carlos III de Madrid	Rapporteur
Elena-Ioana VATAJELU, Chargée de Recherche, Université Grenoble Alpes	Rapporteur
Alberto BOSIO, Full Professor, École Centrale de Lyon	Examinateur
Davide BERTOZZI, Full Professor, University of Manchester	Examinateur
Frédéric SAIGNÉ, Professeur, Université de Montpellier	Président du jury
Douglas Rossi de MELO, Professeur, University of Vale do Itajaí	Invité
Douglas Almeida dos SANTOS, Chargé de Recherche, Université de Montpellier	Invité

UNIVERSITÉ DE MONTPELLIER

"If you don't like something, change it. If you can't change it, change your attitude."

- "Maya Angelou."

Résumé

L'intelligence artificielle (IA) est de plus en plus intégrée aux systèmes spatiaux afin de permettre le traitement embarqué des données et la prise de décision autonome. Cette évolution s'inscrit dans le cadre plus large de l'Edge AI, où l'inférence est effectuée au plus près de la source de données afin de réduire la latence de communication et la consommation d'énergie. Les réseaux de neurones convolutifs (Convolutional Neural Network – CNN) sont particulièrement adaptés à des tâches telles que la classification d'images et la reconnaissance de formes. Cependant, leur processus d'inférence est très exigeant en calcul, ce qui rend son exécution difficile dans les contraintes strictes de puissance et de mémoire des équipements spatiaux. En outre, dans les missions spatiales, les dispositifs électroniques sont exposés aux radiations ionisantes, pouvant compromettre leur fiabilité. Par conséquent, l'exécution de l'inférence CNN à bord des plateformes spatiales nécessite des architectures capables d'assurer à la fois une computation efficace et un fonctionnement fiable sous irradiation.

Le RISC-V est une architecture de jeu d'instructions (Instruction Set Architecture – ISA) ouverte et modulaire, qui permet de concevoir des processeurs adaptés à des besoins spécifiques. Le HArdened RISC-V System-on-Chip (HARV-SoC) a été développé comme un système sur puce (System-on-Chip – SoC) tolérant aux fautes et à faible consommation. Il intègre plusieurs techniques de tolérance aux fautes destinées à détecter et à atténuer les effets des radiations. Le HARV-SoC a été validé au moyen de plusieurs campagnes d'irradiation et est actuellement testé en orbite basse terrestre.

Parmi les différentes extensions du RISC-V, l'extension vectorielle (RISC-V Vector Extension – RVV) permet d'exploiter le parallélisme au niveau des données (Data-Level Parallelism – DLP). Cette approche de vectorisation est particulièrement avantageuse pour accélérer l'inférence des CNN, car les opérations de convolution présentent des schémas réguliers et hautement parallélisables.

Cette thèse étudie si l'utilisation de la vectorisation pour accélérer l'inférence des CNN influence la fiabilité des processeurs tolérants aux fautes dans des environnements radiatifs. Pour cela, une unité d'extension vectorielle (Vector Extension Unit – VEU) a été conçue et intégrée dans le HARV-SoC. Elle implémente un sous-ensemble de l'extension vectorielle RISC-V pour les systèmes embarqués (Zve32x). Afin d'accélérer l'inférence des CNN, un

noyau de convolution vectorisé (vectorized convolution kernel) a été développé pour exploiter les instructions prises en charge par la VEU.

La VEU développée a été évaluée au moyen de la synthèse et de bancs d'essai de performance. De plus, sa fiabilité a été caractérisée par des campagnes d'injection de fautes et des expérimentations d'irradiation. Les résultats ont montré que la vectorisation améliore l'efficacité de calcul pour l'inférence des CNN, atteignant une accélération allant jusqu'à 2,54 fois par rapport à la version scalaire. En termes de fiabilité, bien que la VEU augmente la surface sensible, elle a permis d'améliorer la métrique Mean Work To Failure (MWTF) sur l'ensemble des bancs d'essai analysés.

Ces résultats montrent que la vectorisation peut être intégrée efficacement dans les processeurs RISC-V tolérants aux radiations afin d'accroître les performances sans dégrader la fiabilité. Cette thèse présente la première analyse expérimentale de l'impact de la vectorisation sur la fiabilité de l'inférence IA pour les systèmes embarqués, apportant de nouvelles perspectives sur l'Edge AI fiable et tolérant aux fautes.

Keywords: RISC-V; Réseau Neuronal; SIMD; Vectorisation; Effets des radiations; Systèmes embarqués.

Abstract

Artificial Intelligence (AI) is increasingly being integrated into space systems to enable onboard data processing and autonomous decision-making. This trend follows the broader concept of Edge AI, in which inference is executed close to the data source to reduce communication latency and energy consumption. Convolutional Neural Networks (CNNs) are particularly relevant for tasks such as image classification and pattern recognition. However, the inference process of CNNs is computationally intensive, making it challenging to implement within the strict power and memory limitations of space-grade hardware. Furthermore, in space missions, electronic devices are susceptible to ionizing radiation, which may compromise their reliability. Therefore, deploying CNN inference on board space platforms requires architectures capable of efficient computation and reliable operation under radiation exposure.

The RISC-V is an open and modular instruction set architecture that enables processor implementations to be designed to meet specific needs. The HArdened RISC-V System-on-Chip (HARV-SoC) was developed as a low-power, fault-tolerant soft-core System-on-Chip. It implements multiple fault tolerance techniques to detect and mitigate radiation-induced faults. The HARV-SoC has been validated through various irradiation campaigns and is currently being tested in low Earth orbit.

Within the several RISC-V extensions, there is the RISC-V Vector Extension, which allows vectorization to explore data-level parallelism. The vectorization technique is particularly advantageous for accelerating CNN inference since convolution operations have regular and parallelizable patterns.

This thesis explores whether using vectorization to accelerate CNN inference affects its reliability while operating in radiation environments. To investigate this, a Vector Extension Unit (VEU) was designed and integrated into the HARV-SoC, and implements a subset of the RISC-V Vector Extension for embedded systems (Zve32x). To accelerate CNN inference, a vectorized convolution kernel was implemented to leverage the supported instructions from the VEU.

The developed VEU was evaluated through synthesis and performance benchmarks. Furthermore, it was characterized by reliability analysis through fault-injection simulations

and experimental irradiation. The results showed that vectorization enhanced computational efficiency for CNN inference, achieving a speedup of up to 2.54 times compared to the scalar version. In terms of reliability, although the VEU increases the sensitive area, it was capable of improving the Mean Work To Failure (MWTF) metric across the analyzed benchmarks.

These findings show that vectorization can be effectively employed to enhance performance in radiation-tolerant RISC-V processors without degrading reliability. This thesis provides the first analysis of how vectorization affects the reliability of AI inference for embedded systems, giving new insights into dependable Edge AI computing.

Keywords: RISC-V; Neural Networks; SIMD; Vectorization; Radiation effects; Embedded Systems;

Résumé Substantiel en Français

Dans les missions spatiales, la bande passante de descente est limitée et la latence élevée, ce qui réduit la quantité de données pouvant être transmises en temps réel vers les stations au sol. Dans ces conditions, le traitement embarqué est nécessaire pour minimiser les transmissions de données et favoriser l'autonomie des systèmes [1]. Cette exigence est particulièrement critique pour les nano- et pico-satellites, qui présentent des contraintes strictes en termes de puissance et de mémoire. Les réseaux de neurones (Neural Networks – NN) ont été étudiés pour leur potentiel à réaliser ce traitement local, car ils peuvent exécuter des tâches complexes telles que la reconnaissance de formes et la classification directement à bord des satellites [2, 3]. Cependant, leur déploiement pratique reste limité par des problèmes liés à la consommation d'énergie, à la validation et à la robustesse face aux radiations [1].

Les réseaux de neurones convolutifs (Convolutional Neural Networks – CNN) sont couramment utilisés pour la reconnaissance d'images, la détection d'objets et la segmentation d'images, et plusieurs travaux ont exploré leur application dans le domaine spatial [2, 4, 3]. Toutefois, l'inférence des CNN est coûteuse en calcul, ce qui limite leur utilisation sur des plateformes à ressources restreintes. La majeure partie de ce coût provient des couches de convolution [5], qui présentent une forte réutilisation spatiale des données et des schémas d'accès mémoire réguliers, les rendant hautement parallélisables tant au niveau des données que des opérations. La technique de vectorisation exploite le parallélisme au niveau des données (Data-Level Parallelism – DLP) selon le principe Single Instruction Multiple Data (SIMD), dans lequel une seule instruction agit sur plusieurs éléments. Cette approche permet de remplacer plusieurs itérations scalaires par une seule instruction, réduisant la surcharge des boucles et améliorant l'efficacité des accès mémoire [6].

Outre les limitations computationnelles des dispositifs embarqués, la fiabilité constitue une préoccupation majeure dans les systèmes spatiaux en raison de la présence de radiations ionisantes. L'exposition à ces particules peut induire des fautes transitoires ou permanentes dans les circuits électroniques, susceptibles d'entraîner des défaillances critiques [7, 8]. Ces systèmes doivent donc intégrer des techniques de tolérance aux fautes capables de détecter, masquer ou corriger les erreurs [9].

Plusieurs travaux ont adopté la vectorisation dans la conception de processeurs afin d'accélérer les calculs [10–12]. Cependant, ces architectures sont généralement optimisées pour la performance et l'efficacité énergétique, sans viser un fonctionnement fiable en environnement radiatif. Dans ce contexte, le RISC-V est une architecture de jeu d'instructions (Instruction Set Architecture – ISA) qui a gagné en popularité grâce à sa simplicité, son ouverture et sa modularité. Elle se compose d'un ensemble d'instructions de base et d'extensions optionnelles [13]. Cette ouverture permet d'intégrer des techniques de tolérance aux fautes à différents niveaux architecturaux, renforçant ainsi la fiabilité du processeur. Parmi les extensions disponibles, l'extension vectorielle du RISC-V (RISC-V Vector Extension – RVV) introduit des instructions vectorielles qui permettent d'exploiter efficacement le parallélisme au niveau des données.

Le HArdened RISC-V System-on-Chip (HARV-SoC) intègre le processeur HARV avec plusieurs périphériques et mécanismes de protection au niveau système. Il comprend un code de correction d'erreurs (Error Correcting Code – ECC) dans les contrôleurs mémoire, un chien de garde (Watchdog Timer), des temporisations d'accès au bus, un contrôleur de réinitialisation et une vérification CRC-32 pour les applications. Le processeur HARV est un cœur RISC-V renforcé basé sur une implémentation multicycle de l'architecture RV32I, incluant les extensions de multiplication et de compression. Il renforce la fiabilité grâce à la redondance triple modulaire (Triple Modular Redundancy – TMR) dans l'unité arithmétique et logique (Arithmetic Logic Unit – ALU) et la logique de contrôle, ainsi qu'à la correction simple et détection double d'erreurs (Single Error Correction and Double Error Detection – SECDED) dans le fichier de registres, le compteur de programme et le registre d'instruction. De plus, le processeur HARV dispose d'un mécanisme de gestion des erreurs offrant une observabilité complète via des exceptions RISC-V personnalisées et des registres mappés en mémoire, rapportant les informations de fautes en détail. Le HARV-SoC a été caractérisé dans différentes installations d'irradiation et est actuellement testé en orbite basse (Low Earth Orbit – LEO) à bord du cubesat ROBUSTA-3A [14].

Dans ce contexte, cette thèse présente le développement et la caractérisation d'une unité d'extension vectorielle (Vector Extension Unit – VEU) intégrée au HARV-SoC. L'objectif principal est d'examiner si l'utilisation de la vectorisation pour accélérer l'inférence CNN influence la fiabilité du processeur lorsqu'il fonctionne dans un environnement soumis aux radiations. Le design proposé préserve les mécanismes de tolérance aux fautes du processeur tout en exploitant le parallélisme au niveau des données pour améliorer l'efficacité du calcul dans les noyaux fortement arithmétiques.

La mise en œuvre de la VEU a commencé par une version préliminaire visant à démontrer la faisabilité de l'intégration de capacités vectorielles dans le processeur HARV et à évaluer

leur impact sur les performances, la surface et la fiabilité à travers des simulations d'injection de fautes. L'architecture était basée sur une version simplifiée de l'extension vectorielle du RISC-V, axée sur la validation fonctionnelle et l'exploration matérielle plutôt que sur la conformité complète à la spécification. Certains paramètres architecturaux ont été fixés lors de la synthèse afin de simplifier la conception.

Par exemple, la spécification RVV adopte une approche vector-length agnostic, dans laquelle la largeur vectorielle n'est pas fixée matériellement. Un paramètre permettant de la modifier est le multiplicateur de longueur (Length Multiplier – LMUL), qui définit combien de registres vectoriels consécutifs sont regroupés pour former un seul opérande vectoriel. Augmenter la valeur de LMUL permet à une instruction vectorielle d'opérer sur un vecteur plus large réparti sur plusieurs registres, tandis que des valeurs fractionnaires (par exemple 1/2 ou 1/4) permettent une utilisation plus efficace du fichier de registres avec des types de données étroits. Dans la version préliminaire, au lieu d'utiliser un LMUL configurable dynamiquement, le design employait un paramètre statique appelé largeur de la VEU, définissant combien de registres étaient regroupés pour former un vecteur plus large. Cette configuration statique n'altérait pas le degré de parallélisme mais déterminait combien d'éléments pouvaient être traités par instruction, permettant ainsi d'analyser les compromis de la vectorisation dans un contexte de ressources limitées.

Ensuite, une version améliorée de la VEU a été développée. Cette version a été restructurée pour être conforme au sous-ensemble Zve32x du RISC-V, l'un des cinq sous-ensembles réduits introduits pour répondre aux contraintes des systèmes embarqués. Ces sous-ensembles conservent la compatibilité avec l'extension complète tout en limitant la taille des éléments et les catégories d'instructions prises en charge. Le Zve32x fixe la longueur maximale des éléments (Element Length – ELEN) à 32 bits, et la largeur des éléments sélectionnés (Selected Element Width – SEW) peut être configurée à 8, 16 ou 32 bits.

De plus, l'ensemble d'instructions pris en charge par la VEU a été étendu pour inclure la multiplication vectorielle, facilitant les opérations de convolution. Le support du paramètre LMUL a été implémenté, permettant un ajustement dynamique de la largeur vectorielle, contrairement à la configuration fixe utilisée dans la version préliminaire. Enfin, l'organisation de la VEU a été repensée, et une unité de contrôle vectorielle a été ajoutée afin d'améliorer la modularité et de simplifier la synchronisation des signaux entre la VEU et l'unité de contrôle principale.

La VEU se compose d'un fichier de registres vectoriels (Vector Register File – VRF), d'une unité d'exécution, d'une unité de contrôle, d'un registre compteur d'éléments et d'une interface de sortie. Le VRF contient 32 registres exclusifs de 32 bits chacun et prend en charge deux lectures et une écriture par cycle. Pour augmenter le nombre d'éléments traités

par instruction, la spécification RISC-V définit le paramètre LMUL. L'unité d'exécution comprend une ALU de 32 bits, une de 16 bits et deux de 8 bits, qui opèrent exclusivement sur les éléments vectoriels. Le degré de parallélisme dépend de la configuration SEW : avec SEW fixé à 8, quatre opérations peuvent être exécutées simultanément.

Pour permettre l'inférence CNN, le framework TensorFlow Lite for Microcontrollers (TFLM) a été porté sur le HARV-SoC. Ce framework léger exécute des inférences quantifiées sur des plateformes à ressources limitées. Pour exploiter la VEU, le noyau de convolution a été adapté pour utiliser les instructions vectorielles. La structure des boucles a été réorganisée, et les opérations scalaires internes ont été remplacées par des instructions vectorielles. De plus, la VEU a été étendue pour inclure des instructions d'élargissement vectoriel (vector widening), car les valeurs d'entrée et de filtre sont stockées en mémoire sur 8 bits mais doivent être traitées sur 16 bits pour éviter le dépassement de capacité. L'utilisation de l'instruction d'addition élargie (*vaddw.vx*) a permis de supprimer une boucle de conversion et d'améliorer l'efficacité d'exécution.

La VEU a été analysée en termes de synthèse, de performance et de fiabilité. La synthèse sur circuit logique programmable (Field-Programmable Gate Array – FPGA) visait à évaluer le coût matériel de l'intégration de la VEU dans le processeur HARV. Les résultats ont montré une augmentation d'environ 14% de l'utilisation logique et de 9% du nombre de bascules, principalement due à l'ajout du fichier de registres vectoriels et de la logique de contrôle, tandis que la fréquence maximale a diminué de moins de 3%, restant compatible avec les contraintes du HARV-SoC.

Pour l'évaluation des performances, des inférences CNN ont été exécutées avec le framework *TensorFlow Lite for Microcontrollers* (TFLM), en utilisant à la fois les versions scalaire et vectorisée. L'impact de la vectorisation sur les performances a été évalué en mesurant le nombre de cycles pour chaque convolution et pour l'inférence complète, à l'aide du registre *mcycle* lu avant et après l'exécution. En conséquence, les couches de convolution s'exécutent beaucoup plus rapidement et l'inférence complète est 2,54 fois plus rapide que l'implémentation originale.

L'évaluation de la fiabilité a été divisée en deux parties complémentaires : des simulations d'injection de fautes et des expérimentations d'irradiation. Afin d'estimer le compromis entre performance et fiabilité, la *Mean Work to Failure* (MWTF) et la section efficace de défaillance ont été analysées. Une défaillance a été définie comme toute exécution non classée comme *correcte*. La MWTF a été calculée comme le rapport entre le travail total effectué et le nombre de défaillances, une exécution complète de l'algorithme étant considérée comme une unité de travail. La section efficace (XS) représente le taux de défaillance, calculé en divisant le nombre de défaillances par la fluence simulée.

Les expériences d'injection de fautes ont été réalisées au niveau RTL afin d'émuler des fautes transitoires dans les bascules et registres de la VEU. Des inversions de bits uniques ont été injectées aléatoirement pendant l'exécution des noyaux de convolution vectorisés. Les simulations ont montré que les noyaux vectorisés sont plus résistants que la version originale. Le temps d'exécution réduit diminue l'exposition aux fautes, rendant le système plus robuste face aux erreurs induites par les radiations. De plus, l'introduction des instructions d'élargissement a amélioré l'efficacité des accès mémoire tout en maintenant la précision nécessaire aux calculs exacts. Même dans des configurations non durcies, le noyau vectorisé a surpassé la version scalaire, démontrant que la vectorisation améliore à la fois les performances et la fiabilité dans des environnements sévères. Les résultats de MWTF ont également montré que les configurations vectorisées pouvaient effectuer davantage de travail avant de rencontrer des erreurs. Ces résultats indiquent que la vectorisation, qu'elle soit associée ou non à des techniques de durcissement, améliore de manière significative la fiabilité.

L'expérience d'irradiation a été réalisée au *Proton Irradiation Facility* (PIF) de l'*Paul Scherrer Institute* (PSI) en Suisse. L'irradiation a été effectuée avec des protons d'une énergie de 200 MeV, avec des flux compris approximativement entre 1.3×10^8 et 2.71×10^8 p/cm^2/s. Le faisceau présentait une homogénéité de 10 % sur la zone d'irradiation sélectionnée (20 mm $\times$ 20 mm) et pour les énergies du faisceau. La fluence cumulée sur l'ensemble des *Systems Under Test* (SUT) était de 2.4×10^{12} p/cm^2.

Une comparaison directe avec les résultats de [15] est possible, puisque le même banc d'essai de réseau de neurones a été utilisé et que les expérimentations d'irradiation ont été effectuées dans la même installation et à la même énergie de protons. Dans des conditions de faisceau similaires, le processeur RISC-V durci avec la VEU a présenté des sections efficaces de l'ordre de 10^{-12} cm^2/dispositif, soit une amélioration d'un à deux ordres de grandeur par rapport aux dispositifs testés dans [15].

En résumé, les résultats obtenus démontrent que la vectorisation accélère considérablement l'inférence CNN tout en maintenant, voire en améliorant, la fiabilité globale du processeur. La VEU proposée a atteint des gains de performance allant jusqu'à 2,54 fois, sans dégradation mesurable sous irradiation protonique. Ces résultats montrent que la vectorisation peut être utilisée efficacement pour améliorer les performances des processeurs RISC-V tolérants aux radiations sans compromettre leur fiabilité. Cette thèse présente la première analyse de l'impact de la vectorisation sur la fiabilité de l'inférence IA pour les systèmes embarqués, apportant de nouvelles perspectives sur un *Edge AI* fiable et tolérant aux fautes.

In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of University of Montpellier's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to `http://www.ieee.org/publications_standards/publications/rights/rights_link.html` to learn how to obtain a License from RightsLink. If applicable, University Microfilms and/or ProQuest Library or the Archives of Canada may supply single copies of the dissertation.

Acknowledgements

I would like to thank Prof. Luigi Dilillo for his guidance, patience, and the opportunity to work under his supervision. I am also deeply grateful to Douglas A. Santos for all his help, for every coffee break, and for everything he taught me along the way. My sincere thanks to my professors from Brazil, who built the foundation of what I am today and whose lessons and encouragement will stay with me forever. I thank my research group for the shared lunches, ideas, and the good moments that made work lighter.

My warmest thanks go to my family, who have always been present even from far away, for their constant love, support, and willingness to help whenever I needed. I also thank my friends in Brazil, who never let me think they had forgotten about me, and my friends and my loved one whom I met during this PhD, for their laughter, kindness, and for reminding me why this journey was worth it.

This research has been achieved thanks to the financial support of the Region d'Occitanie and the École Doctorale I2S from the University of Montpellier (contract number 00137932/ 22009671), the EU Horizon Europe Twinning project TWIN-RELECT (Grant 101160314), the EU project RADNEXT – Horizon 2020 (Grant 101008126), and Project HARV (project PE24PR01) in the framework of the action "Accelerateur d'innovation" of the University of Montpellier.

Contents

List of Author's Publications **xxiii**

List of Figures **xxv**

List of Tables **xxvii**

Acronyms **xxix**

1 Introduction **1**

1.1 Edge Artificial Intelligence 2
1.2 Convolutional Neural Networks 5
1.3 Parallelism and Vector Architectures 7
1.4 The RISC-V Instruction Set and Vector Extension 8
1.4.1 Vector Extension 9
1.5 Radiation environments 13
1.5.1 Space Environment 13
1.5.2 Athmospheric Environment 15
1.5.3 Artificial Radiation Environment 16
1.6 Radiation Effects in Electronic Devices and Reliable System Design 16
1.6.1 Cumulative Effects 17
1.6.2 Single-Event Effects 18
1.6.3 Reliable Systems 20
1.7 HARV-SoC 21
1.8 Related Work 24
1.9 Thesis Outline 26

2 Reliability Evaluation of a Commercial Edge AI Microcontroller **29**

2.1 Experimental Setup 29
2.1.1 Device Under Test 30

2.1.2 Layer-by-Layer Implementation . . . 31
2.1.3 Neural Networks and Datasets . . . 34
2.1.4 Test Setup . . . 35
2.1.5 Metrics . . . 36
2.2 Results . . . 37
2.2.1 Major, Medium, and Minor Output Disturbance . . . 38
2.2.2 System Failure . . . 41
2.2.3 Reliability Analysis . . . 41
2.3 Discussion . . . 42

3 Evaluation Methodology for the HARV-SoC and Vector Extension Unit 45
3.1 Synthesis and Performance Evaluation . . . 45
3.2 Reliability Evaluation . . . 46
3.2.1 Fault Injection Simulation Methodology . . . 46
3.2.2 Classification of Executions . . . 47
3.2.3 Reliability Metrics . . . 47

4 Preliminary Implementation 49
4.1 Vector Extension Unit . . . 50
4.2 Experimental Setup . . . 52
4.3 Results . . . 53
4.3.1 Synthesis . . . 53
4.3.2 Performance . . . 55
4.3.3 Fault Injection Simulation . . . 56
4.3.4 Discussion . . . 57

5 Enhanced Implementation of the Vector Extension Unit 59
5.1 Enhanced Vector Extension Unit . . . 60
5.2 Experimental Setup . . . 62
5.3 Results . . . 63
5.3.1 Synthesis . . . 64
5.3.2 Performance . . . 64
5.3.3 Fault Injection Simulation . . . 66
5.3.4 Discussion . . . 68

6 Vectorized Neural Network Inference 69
6.1 TensorFlow Lite Micro Support and Vector Extension Unit Modifications . 70

6.1.1 Vectorized Convolution Kernel 70
6.1.2 Vector Extension Unit Modifications 73
6.2 Experimental Setup 73
6.2.1 Application Benchmark 74
6.2.2 Fault injection simulations 74
6.2.3 Irradiation experiment 74
6.3 Results 76
6.3.1 Synthesis 77
6.3.2 Performance 78
6.3.3 Fault injection simulations 79
6.3.4 Irradiation experiment 80
6.4 Discussion 82

7 Summary and Conclusions **85**

Bibliography **89**

List of Author's Publications

This doctoral thesis is based in part on the following peer-reviewed publications. The number used in this reference list is self-contained and is not related to the thesis's main matter.

Journals

[JO-1] **C. Imianosky**, A. M. P. Mattos, D. A. Santos, D. R. Melo, M. Kastriotou, C. Cazzaniga, and L. Dilillo, "Characterization of Single-Event Effects in a Microcontroller with an Artificial Neural Network Accelerator," in *Electronics*, vol. 13, no. 22, art. no. 4461, Nov. 2024, DOI: 10.3390/electronics13224461. © 2024 by the authors (MDPI) CC BY 4.0

[JO-2] A. Coronetti, R. García-Alía, L. Dilillo, **C. Imianosky**, D. A. Santos, L. M. Luza, A. Bosser, K. Bilko, A. Waets, K. Klimek, F. Saigné, "SRAM-based heavy ion beam flux and LET dosimetry," in IEEE Transactions on Nuclear Science, DOI: 10.1109/TNS.2024.3487647.

International Events

[IE-1] **C. Imianosky**, D. A. Santos, W. Grignani, and L. Dilillo, "Reliability Analysis of Neural Network Inference on a Fault-Tolerant RISC-V SoC with Vector Extension". IEEE European Conference on Radiation and Its Effects on Components and Systems (RADECS), 2025. © 2025 IEEE

[IE-2] **C. Imianosky**, D. A. Santos, and L. Dilillo, "Efficient TinyML Inference on a Fault-Tolerant RISC-V SoC with Vector Extension". 10th IEEE International Workshop on Advances in Sensors and Interfaces, July. 2025, Manfredonia, Italy. © 2025 IEEE

[IE-3] W. Grignani, D. A. Santos, **C. Imianosky**, M. Kastriotou, C. Cazzaniga, and L. Dilillo, "Characterization of HARV-SoC for Reliable Avionics Applications". 1st Workshop on

REliable and SeCUrE RISC-V architectures - RESCUER, May. 2025, Tallinn, Estonia. © 2025 IEEE

[IE-4] D. A. Santos, **C. Imianosky**, and L. Dilillo, "HARV Hardened RISC-V System on Chip," 2nd RISC-V in Space Workshop, Apr. 2025, Gothenburg, Sweden.

[IE-5] N. Chatzivangelis, N. Zazatis, W. Grignani, G. Paliaroutis, D. A. Santos, **C. Imianosky**, M. Kastriotou, C. Cazzaniga, F. Wrobel, A. Verones, C. Sotiriou, M. Andjelkovic, D. Bertozzi, and L. Dilillo, "Simulation Methodologies and Experiments for Reliability Analysis of Devices in Radiation Harsh Environments," 2025 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Barcelona, Spain, 2025. © 2025 IEEE

[IE-6] **C. Imianosky**, D. A. Santos, D. R. Melo, F. Viel and L. Dilillo, "Special Session: Reliability and Performance Evaluation of a RISC-V Vector Extension Unit for Vector Multiplication," in *2024 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Didcot, United Kingdom, 2024, pp. 1–6, DOI: 10.1109/DFT63277.2024.10753524. © 2024 IEEE

[IE-7] **C. Imianosky**, D. A. Santos, D. R. Melo, F. Viel and L. Dilillo, "Implementation and Reliability Evaluation of a RISC-V Vector Extension Unit," 2023 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Juan-Les-Pins, France, 2023, pp. 1-6, DOI: 10.1109/DFT59622.2023.10313569. © 2023 IEEE

[IE-8] A. M. P. Mattos, D. A. Santos, **C. Imianosky**, D. R. Melo, and L. Dilillo, "Using HARV-SoC for Reliable Sensing Applications in Radiation Harsh Environments," in *9th International Workshop on Advances in Sensors and Interfaces (IWASI)*, Monopoli (Bari), Italy, June 2023, pp. 227-232, DOI: 10.1109/IWASI58316.2023.10164578. © 2023 IEEE

List of Figures

1.1 Representation of the convolution operation, where the kernel slides across the input and computes a weighted sum at each spatial position. 6
1.2 Representation of scalar and vector executions. 8
1.3 Instruction formats of the integer base set. 9
1.4 Example of vector register grouping for different Vector Register Group Multiplier (LMUL) configurations, considering SEW = 8b and VLEN = 32b. 10
1.5 Vector instruction opcodes used by the RISC-V Vector extension (RVV). . . 11
1.6 Simplified representation of the trapped radiation and its motion in the Van Allen belts. Figure from: [16]. 15
1.7 HArdened RISC-V System-on-Chip (HARV-SoC) architecture. 23
1.8 HARV-SoC Error Handler. 24

2.1 MAX78000 platform architecture. 30
2.2 Comparison of Convolutional Neural Network (CNN) execution runtime flow. The left diagram shows the standard approach. Comparison of CNN execution runtime flow. (a) Standard approach. (b) Combination of the Layer-by-Layer Transient Toolchain (LbLTT) [17] layer-by-layer approach with the implemented techniques for observability. 33
2.3 Experimental setups prepared for the irradiation campaign composed of four MAX78000FTHR boards. 36

4.1 Vector Extension Unit. 50
4.2 Vector Execution Unit. 52
4.3 Resources utilization for the baseline HArdened RISC-V (HARV)-System-on-Chip (SoC) and the HARV-SoC with different Vector Extension Unit (VEU) widths. 53
4.4 Resource usage for the hardened HARV-SoC and the HARV-SoC with different *VEU widths*. 54

4.5 Execution acceleration for different *VEU widths*. 55

5.1 Vector Extension Unit. 60
5.2 Vector Execution Unit. 61
5.3 State machine of the vector control unit. 62
5.4 Performance variation for different versions of the benchmark algorithm in comparison to the scalar C. 65

6.1 Memory layout and access pattern in single-channel and multi-channel convolution. 71
6.2 Simplified pseudo algorithm for original, single-channel, and multi-channel vectorized convolution kernels. 72
6.3 Experimental boards attached to the test frame inside the irradiation room at Paul Scherrer Institute (PSI). 75
6.4 SmartFusion2 System Design for the experiment Using HARV-SoC. 76
6.5 Performance variation of the inference with different convolution kernel versions. 78

List of Tables

1.1 Embedded vector extensions. 12

2.1 Architecture of the AI85Net5 model. 35
2.2 Architecture of the AI85NASCifarNet model. 35
2.3 Characteristics of the CNNs. 35
2.4 Classification of errors per board. 38
2.5 Errors reported in Minor Output Disturbance (MiOD) executions. 38
2.6 Errors reported in Medium Output Disturbance (MeOD) executions. 39
2.7 Errors reported in Major Output Disturbance (MaOD) executions. 39
2.8 Reported cases of system failure. 41
2.9 Summary of System Failure (SF) cross-sections per board. 43
2.10 Weight errors Failure In Time (FIT) and cross-section. 43

4.1 Maximum operating frequency and dynamic power dissipation results. . . . 55
4.2 Fault injection executions. 57
4.3 Injections analysis. 57

5.1 Synthesis results. 64
5.2 Fault injection executions. 67

6.1 Synthesis results for Baseline and Hardened versions. 77
6.2 Synthesis results for the SmartFusion2 Field-Programmable Gate Array (FPGA). 77
6.3 Fault injection executions. 79
6.4 Executions classification and cross-section. 81
6.5 Events classification and cross-section for *B3*. 82

Acronyms

AI Artificial Intelligence. 2, 3, 24–26, 29, 30, 34, 68, 69, 82, 85, 86

ALU Arithmetic Logic Unit. 23, 50, 51, 60, 61, 64, 66

ASIC Application-Specific Integrated Circuit. 29

ChipIr ISIS Chip Irradiation facility. 29, 30

CIFAR Canadian Institute For Advanced Research. 34, 35, 37, 39, 40, 42

CME Coronal Mass Ejection. 14

CMEM Configuration Memory. 22

CMOS Complementary Metal Oxide Semiconductor. 20

CNN Convolutional Neural Network. xxv, xxvii, 1, 2, 5–7, 25–27, 29–35, 37, 41, 42, 68, 69, 85, 86

CRC Cyclic Redundancy Check. 22

CSR Control and Status Register. 10–12, 62

DBU Double Bit Upset. 19

DD Displacement Damage. 17, 18

DLP Data-Level Parallelism. 1, 2, 7, 58, 70, 87

DMR Dual Modular Redundancy. 21

DNN Deep Neural Network. 25, 30

DRAM Dynamic Random Access Memory. 19

DSP Digital Signal Processing. 64, 77

DUT Device Under Test. 29, 30, 42

ECC Error Correcting Code. 22, 25

EDAC Error Detection and Correction. 21

ELEN Maximum Element Width. 9, 12, 60

eNVM Embedded Non-Volatile Memory. 75, 76, 83

FF Flip-Flop. 45–47, 53, 54, 57, 64, 77

FIT Failure In Time. xxvii, 36, 37, 41–43, 46

FPGA Field-Programmable Gate Array. xxvii, 2, 21, 22, 25, 45, 74, 76, 77, 82, 83

FSM Finite State Machine. 61, 62, 73

GCC GNU Compiler Collection. 45

GCR Galactic Cosmic Ray. 13, 14

GPIO General Purpose Input/Output. 22

HARV HArdened RISC-V. xxv, 21, 22, 45, 49–55, 57, 58, 61, 63

HARV-SoC HArdened RISC-V System-on-Chip. xxv, xxvi, 2, 22–24, 26, 45–47, 64, 66, 68–70, 74–77, 79, 80, 82, 85, 86

HPC High Performance Computing. 7, 24, 26

I2C Inter-Integrated Circuit. 22

IR Instruction Register. 23

ISA Instruction Set Architecture. 2, 8, 9, 50

ITER International Thermonuclear Experimental Reactor. 16

LbLTT Layer-by-Layer Transient Toolchain. xxv, 31–33

LHC Large Hadron Collider. 16

LMUL Vector Register Group Multiplier. xxv, 10, 26, 49, 51, 58–60, 62, 63, 65, 66, 68, 71, 86

LOCOS LOCal Oxidation of Silicon. 20

LUT Look-Up Table. 45, 53, 54, 57, 64, 77

MAC Multiply–Accumulate. 25, 68, 70, 72

MaOD Major Output Disturbance. xxvii, 37–40, 42

MBU Multiple Bits Upset. 19

MCU Multiple-Cell Upset. 19

MeOD Medium Output Disturbance. xxvii, 37–39

MiOD Minor Output Disturbance. xxvii, 37–39

ML Machine Learning. 24, 26

MNIST Modified National Institute of Standards and Technology. 25, 34, 35, 37–40, 42, 74

MOS Metal-Oxide-Semiconductor. 17, 18

MSS Microcontroller Subsystem. 75, 83

MWTF Mean Work to Failure. 47, 66, 67, 79, 86

NAS Neural Architecture Search. 34

NI No Impact. 37, 38

NN Neural Network. 1, 3–5, 25, 26, 29, 31, 32, 34, 40, 42, 82, 87

PC Program Counter. 8, 23

PIF Proton Irradiation Facility. 16, 74

PSI Paul Scherrer Institute. xxvi, 16, 74, 75

ReLU Rectified Linear Unit. 7, 34, 35, 74

RV32I RISC-V 32-bit integer instruction set. 8, 21

RVV RISC-V Vector extension. xxv, 2, 9–12, 24–26, 45, 49–51, 58, 59, 68, 70, 86

SBU Single Bit Upset. 19, 81, 82

SDK Software development Kit. 31, 34

SDRAM Synchronous Dynamic Random-Access Memory. 22, 74, 76

SECDED Single Error Correction and Double Error Detection. 21, 22

SEE Single-Event Effect. 17, 18

SEFI Single-Event Functional Interrupt. 20

SEL Single-Event Latch-up. 18, 20

SEP Solar Energetic Particle. 13, 14

SET Single-Event Transient. 19, 21

SEU Single-Event Upset. 19, 41, 42

SEW Selected Element Width. xxv, 9, 10, 60, 62, 63, 65, 66, 68, 71

SF System Failure. xxvii, 37, 38, 40, 41, 43

SIMD Single Instruction, Multiple Data. 1, 7

SoC System-on-Chip. xxv, 2, 22, 23, 45, 47, 53–55, 57, 64, 69, 74, 79, 83, 85

SOI Silicon On Insulator. 20

SPI Serial Peripheral Interface. 22

SRAM Static Random-Access Memory. 22

SUT System Under Test. 76, 80, 83

TFLM TensorFlow Lite for Microcontrollers. 2–4, 69, 70, 72, 74, 78, 79, 87

TID Total Ionizing Dose. 17, 81, 83

TinyML Tiny Machine Learning. 3, 25

TMR Triple-Modular Redundancy. 21, 23, 64

UART Universal Asynchronous Receiver Transmitter. 22, 36, 41, 46, 47, 66, 76

VEU Vector Extension Unit. xxv, xxvi, 2, 26, 45, 46, 49–60, 63–66, 68–70, 73, 74, 78, 82, 85–87

VHDL VHSIC Hardware Description Language. 21, 45, 75

VL Active Vector Length. 10–12, 59, 62, 63, 65, 68, 86

VLA Vector-Length Agnostic. 9

VLEN Vector Register Length. xxv, 9, 10, 12, 60

VLENB Vector Register Length in Bytes. 10

VLMAX Maximum Vector Length. 10, 63

VRF Vector Register File. 50, 51, 57, 60, 62, 64, 66

VSTART Vector Start Index. 10

VTYPE Vector Type. 10, 11

VXRM Fixed-Point Rounding Mode. 10

VXSAT Fixed-Point Saturation Flag. 10

WDT WatchDog Timer. 22, 41, 43

XIP Execute In Place. 75, 76

Chapter 1

Introduction

In space missions, downlink bandwidth is limited and latency is high, which reduces the amount of data that can be sent to ground stations in real-time. Under these conditions, onboard processing is needed to minimize data transmission and facilitate autonomous operation [1]. This requirement is particularly critical for nano- and pico-satellites, which have strict power and memory limitations. Neural Networks (NNs) have been studied for their potential to support onboard processing, as they can perform complex tasks such as pattern recognition and classification directly on satellites [2, 3]. However, its practical deployment is still limited by issues related to power consumption, validation, and robustness under radiation [1].

Convolutional Neural Networks (CNNs) are commonly used for image recognition, object detection, and image segmentation, and several works have investigated their use in space applications [2, 4, 3]. However, the inference of CNNs is computationally demanding, which limits their deployment on platforms with restricted resources. Most of this cost is due to the convolution layers [5], which exhibit high spatial data reuse and regular memory access patterns, making them highly parallelizable across both data and operations. The vectorization technique exploits Data-Level Parallelism (DLP) based on the Single Instruction, Multiple Data (SIMD) concept, in which a single instruction operates on multiple elements. With this approach, a single instruction replaces several scalar iterations, reducing loop overhead and improving memory access efficiency [6].

In addition to the computational limitations of embedded devices, reliability is a primary concern in space systems due to the presence of ionizing radiation. The exposure to radiation particles can induce transient or permanent faults in electronics, which may lead to critical failures [7, 8]. Thus, these systems must incorporate fault-tolerance techniques, capable of detecting, masking, or correcting errors [9].

Several works adopt the vectorization technique in processor design to accelerate computations [10–12]. However, these designs are typically optimized for performance and energy efficiency, and are not intended to ensure reliable operation under ionizing radiation. In this context, RISC-V is an Instruction Set Architecture (ISA) that has gained popularity due to its simplicity, openness, and modularity, i.e., it consists of a basic ISA and optional extensions [13]. The openness of RISC-V makes it possible to integrate fault-tolerance techniques at different architectural levels, increasing the processor's reliability. Besides that, among the available extensions, the RISC-V Vector extension (RVV) introduces vector instructions that enable efficient exploitation of DLP.

The HArdened RISC-V System-on-Chip (HARV-SoC)[18] is a low-cost soft-core fault-tolerant RISC-V System-on-Chip (SoC) designed for Field-Programmable Gate Arrays (FPGAs). The SoC has been characterized in different irradiation facilities and is currently being tested in Low Earth Orbit (LEO) aboard the ROBUSTA-3A cubesat [14].

This thesis presents the development and characterization of a Vector Extension Unit (VEU) integrated into the HARV-SoC. The main goal is to explore whether using vectorization to accelerate CNN inference affects its reliability while operating in radiation environments. The proposed design preserves the fault-tolerant mechanisms of the processor while exploiting DLP to improve computational efficiency in arithmetic-intensive kernels.

The system was evaluated through architectural synthesis, performance measurements, and reliability analyses combining fault-injection simulations and experimental irradiation, providing a detailed assessment of the trade-offs between vector acceleration and dependable operation in a radiation-tolerant RISC-V platform.

The following sections of this chapter present the background that supports this thesis. Section 1.1 discusses edge Artificial Intelligence (AI) in the context of space systems, followed by Section 1.2 that introduces CNNs and their computational characteristics. Section 1.3 describes the TensorFlow Lite for Microcontrollers (TFLM) framework used for on-board inference. Section 1.4 presents vector architectures and the RISC-V instruction set, emphasizing the RVV. Section 1.5 reviews radiation effects and fault-tolerance techniques relevant to space electronics, and Section 1.6 describes the HARV-SoC, which serves as the baseline for the implementation and validation of the proposed vector extension.

1.1 Edge Artificial Intelligence

Edge computing is a computing paradigm that brings data processing and storage closer to the data source, instead of relying exclusively on centralized cloud servers. By performing

computation at the edge, it reduces latency, saves bandwidth, and enhances privacy for time-critical or data-sensitive applications [19, 20].

Edge AI represents the convergence of AI and edge computing, enabling intelligent processing and decision-making directly on embedded and low-power devices [21]. Notably, Tiny Machine Learning (TinyML) applies the principles of Edge AI to microcontrollers, enabling on-device inference under extreme power and memory constraints [22]. These approaches mitigate the limitations of cloud-based systems and support autonomous operation in scenarios with restricted connectivity.

However, the computational demands of AI inferences often exceed the capabilities of traditional microcontrollers and low-power processors. These constraints are even more severe in space-grade systems, where the need for fault-tolerant components with strict power and area budgets limits performance. To cope with these limitations, AI models for Edge AI applications typically rely on techniques to reduce the memory footprint and the number of arithmetic operations [22, 23]. Thus, efficient algorithms, hardware-aware optimizations, and deployment frameworks are used to address the strict constraints on computational resources, processing speed, and power consumption [21, 23].

A core technique in Edge AI is quantization, which enables the compression of NN models. Essentially, quantization converts model parameters, such as weights and activations, from high-precision formats, like 32-bit floating-point, into lower-precision representations, typically 8-bit integers. This process significantly reduces the model's memory footprint and computational requirements. Another common optimization technique is pruning, which removes weights or neurons that do not substantially contribute to the model's prediction, thereby further reducing its computational complexity and energy consumption [23, 24].

In addition to these model-level transformations, performance can also be improved through hardware acceleration. Many embedded processors and low-power systems provide specialized instructions or dedicated accelerators that reduce latency and energy consumption during inference [21, 23, 24]. Examples of microcontrollers include the STM32N6, the Renesas RA8D1, and the MAX78000, each combining a microcontroller core with dedicated AI acceleration hardware optimized for power-efficient Edge AI applications.

During deployment, frameworks are used to implement the optimizations and create the runtime environment for executing models on target devices. These frameworks manage hardware-specific integration, implement optimized operators, and manage memory resources. Some examples of these frameworks are TFLM, uTensor, Edge Impulse, and STM32Cube.AI [21].

Tensorflow Lite Micro

TensorFlow Lite for Microcontrollers (TFLM) is built on TensorFlow Lite, a simplified version of TensorFlow designed explicitly for deploying machine learning models on mobile and embedded devices. TensorFlow Lite optimizes model size and enhances inference speed by converting standard TensorFlow models into a compact format (.tflite) that is tailored for devices with limited computational resources, such as smartphones, single-board computers, and various embedded platforms [25].

TFLM employs a specialized approach developed for low-power and resource-constrained embedded systems, designed initially for microcontrollers. To optimize performance within these constraints, TFLM avoids dynamic memory allocation and does not rely on operating system services. Instead, it allocates a fixed amount of memory, ensuring consistent and predictable performance. Furthermore, TFLM offers a simplified collection of operators optimized for embedded inference, supporting tasks such as image classification, keyword spotting, and sensor data analysis.

The deployment process begins by training a NN in the standard TensorFlow environment using Python. During this phase, the model is trained and validated with all available tools in TensorFlow, typically employing floating-point arithmetic to maximize accuracy. Once the desired performance is achieved, the model is converted into the TensorFlow Lite format (.tflite) using the TensorFlow Lite Converter. This conversion produces a compact representation optimized for inference, applying graph-level optimizations. Additionally, this step enables the integration of post-training techniques, such as quantization and pruning, to further reduce the model's memory footprint and computational cost.

The resulting `.tflite` model is then translated into a C array that can be compiled as part of the embedded firmware. On the device side, the TensorFlow Lite for Microcontrollers library provides the runtime environment needed for inference. The library is written in C/C++ and includes an interpreter that schedules and runs operations sequentially, a set of kernel implementations for supported operators, and memory management utilities that allocate all tensors within the statically defined arena.

TFLM supports a wide range of microcontrollers, including ARM Cortex-M, ESP32, nRF52, and GAP8. It also provides support for RISC-V platforms, with an official implementation available in the TensorFlow repository. While the core of TFLM is portable in C++ and can run on any architecture with sufficient resources, deploying it on RISC-V requires additional porting efforts to adapt the build system and hardware-specific layers.

1.2 Convolutional Neural Networks

As mentioned above, CNNs are a type of NN that are particularly effective in image-based tasks, and some works studied their use for space applications [2, 4, 3]. In general, a deep neural network consists of multiple layers (including input, output, and hidden layers) and has several trainable parameters [26]. The original information (e.g., an image) is fed into the CNN through the input layer, and the prediction (e.g., a classification result) is given through the output layer. All the information is transferred and processed via the hidden layers inside the network.

The core building block of a CNN architecture is the convolutional layer. Convolution is a classical signal-processing operation defined for continuous signals by Equation 1.1, where $x(a)$ and $w(t-a)$ are the input and the kernel functions, respectively. The variable t in Equation 1.1 is the time parameter that specifies the point in time at which the convolution is evaluated. However, in digital image processing, the independent variable becomes an integer index, so Equation 1.1 is replaced by the discrete convolution in Equation 1.2, where $t, a \in \mathbb{Z}$ denote pixel positions [27].

$$y(t) \;=\; (x*w)(t) \;=\; \int x(a)w(t-a)da \tag{1.1}$$

$$y(t) \;=\; (x*w)(t) \;=\; \sum_{a=-\infty}^{\infty} x(a)\,w\big(t-a\big) \tag{1.2}$$

In CNN, the input and kernel for the convolution are usually multidimensional arrays of data and parameters, respectively. Additionally, the convolution is often calculated over more than one axis simultaneously. Considering I as a two-dimensional input image and K as a two-dimensional kernel, the 2D convolution is given by Equation 1.3. The indices i and j depend on the image dimensions , while m and n relate to the kernel .

$$Y(i,j) \;=\; (K*I)(i,j) \;=\; \sum_{m=-\infty}^{\infty}\sum_{n=-\infty}^{\infty} I(i-m,\, j-n)\,K(m,n) \tag{1.3}$$

Convolutional layers consist of one or more kernels (also called filters), and each kernel is a small matrix of trainable parameters that represents a specific pattern, such as edges or textures. The output values generated as the kernel moves across the input form a two-dimensional array known as a feature map, which indicates how strongly the local region of the input matches the pattern encoded by the kernel.

In practical implementations, the convolution is performed by sliding the kernel over the input image and computing the sum of element-wise multiplications at each position, as

represented in Figure 1.1. For each output pixel (i, j), it accumulates the products between the overlapping region of the input image and the kernel weights. In a typical implementation, this process is realized with nested loops iterating over the image height and width, and for each position, an inner loop iterating over the kernel dimensions. Notably, convolutional layers concentrate most of the computational load in a CNN [28]. However, since the same computation is applied to multiple regions of the input and across different kernels, the operations are independent and follow a regular pattern. As a result, the convolutional layer is highly parallelizable.

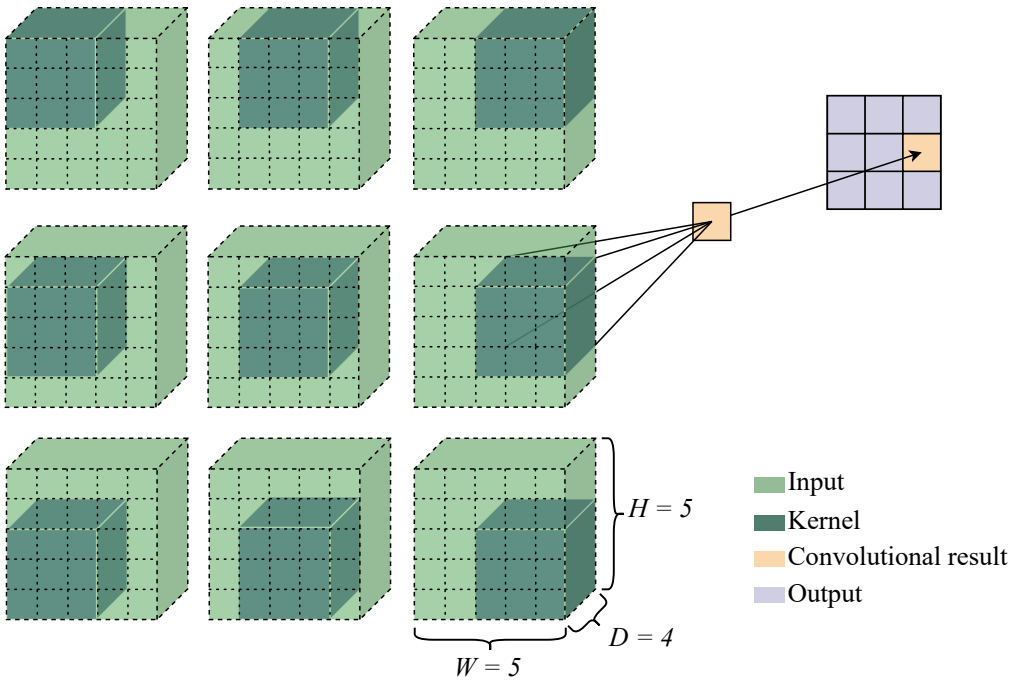


Figure 1.1 – Representation of the convolution operation, where the kernel slides across the input and computes a weighted sum at each spatial position.

Convolutional layers focus on feature extraction. However, a complete CNN architecture also includes other layers that make learning more stable and efficient while improving generalization. Each type of layer has a specific role, such as transforming feature maps, regulating activations, or preventing overfitting. The most common and relevant layers for this work are listed below.

- **Pooling layers:** Downsample the feature maps by selecting the maximum or average value within small regions, reducing spatial dimensions and computational cost.
- **Activation layers:** introduce non-linearity into the network, allowing the model to learn complex patterns that cannot be represented by linear operations alone. The

Rectified Linear Unit (ReLU), defined as $f(x) = \max(0, x)$, is the most used activation function in CNNs. It truncates negative values to zero, activating neurons only for features with positive responses, which helps the network focus on relevant patterns.

- **Batch normalization layers:** Normalize intermediate feature distributions, accelerating training and increasing stability.
- **Dropout layers:** Randomly deactivate neurons during training to reduce overfitting and improve generalization.
- **Flatten layers:** Convert the multidimensional feature maps into a single feature vector for classification.
- **Fully connected layers:** Use all extracted features to compute the final class scores.
- **Softmax layers:** Convert these scores into normalized probabilities associated with each output class.

1.3 Parallelism and Vector Architectures

One effective strategy for accelerating computational processing is to exploit DLP, which enables the simultaneous processing of multiple data elements. Architectures following the SIMD paradigm [29] exploit DLP by applying the same instruction to multiple data elements simultaneously. This parallel execution model increases computational throughput while minimizing instruction fetch and decode overhead, leading to more efficient hardware utilization [30]. Furthermore, to increase throughput, SIMD designs have progressively expanded their register widths, allowing a greater number of elements to be processed concurrently. However, the vector width in SIMD architectures is fixed and defined by the instruction set, which restricts scalability and requires additional instruction encoding whenever wider registers are introduced [31, 30].

Vector architectures emerged from this limitation as a more flexible evolution of the SIMD concept. They also exploit DLP, but the vector width is defined at the hardware implementation level rather than being explicitly encoded in the instruction set. This design choice reduces instruction overhead and provides scalability, since the same binary code can execute efficiently on processors with different vector lengths without altering the programming model [32].

Initially introduced for High Performance Computing (HPC), vector architectures continue to be an effective means of exploiting DLP in modern computing systems, particularly

in applications dominated by regular and repetitive computations [33]. Figure 1.2 shows a representation of scalar and vector executions.

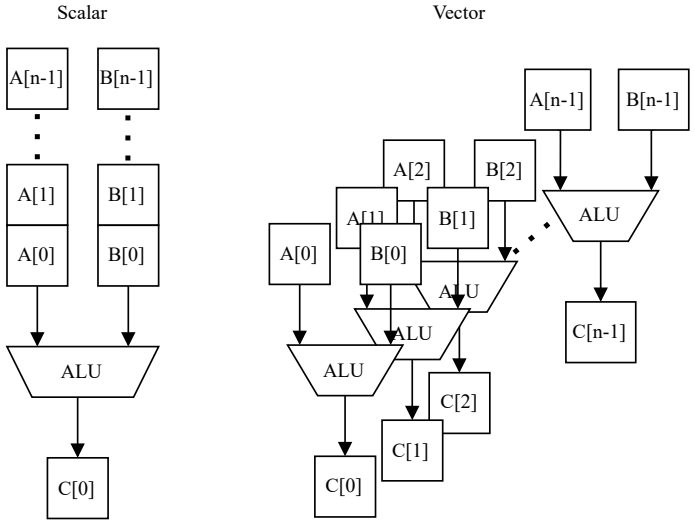


Figure 1.2 – Representation of scalar and vector executions.

1.4 The RISC-V Instruction Set and Vector Extension

RISC-V is an open ISA developed by Waterman et al. [34], that follows a modular design composed of a base integer instruction set and multiple optional extensions that can be implemented according to specific application needs. The RISC-V specifies three basic instruction sets: (*i*) RISC-V 32-bit integer instruction set (RV32I), a set of 40 instructions complete enough to satisfy the basic requirements of modern operating systems; (*ii*) RV32E, resembles the previous set, but has only 16 registers; and (*iii*) RV64I, similar to the RV32I set, differs in the width of the data registers and Program Counter (PC) [30]. The modularization enables minimal processor implementations since the processor is not required to support the entire instruction set.

The integer instruction set uses a fixed 32-bit encoding organized into several instruction formats. The R-type format is used for register-to-register operations, the I-type format for short immediates and memory load instructions, the S-type format for memory store instructions, the B-type format for conditional branches, the U-type format for long immediates, and the J-type format for unconditional jumps [30]. These formats are illustrated in Figure 1.3. In this specification, the fields *Opcode*, *funct3*, and *funct7* define the operation to be executed, while the *rs* and *rd* fields indicate the source and destination registers, respectively [30, 35].

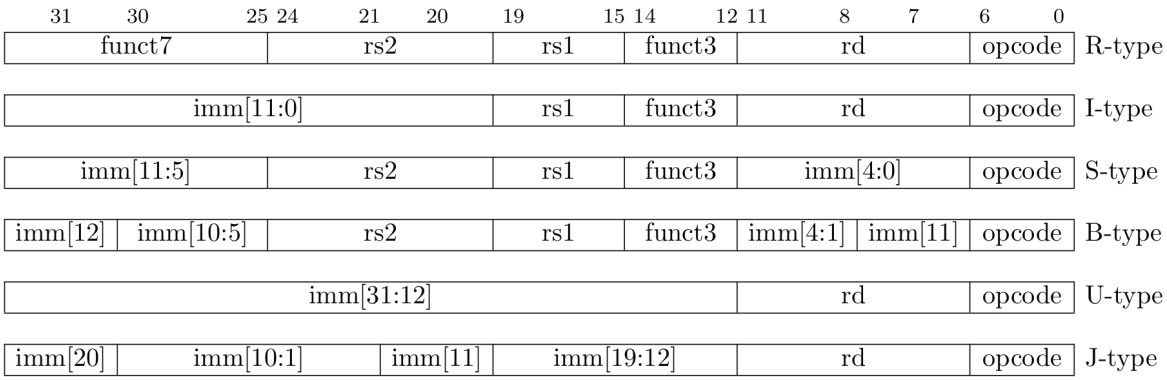


Figure 1.3 – Instruction formats of the integer base set.

Most of the RISC-V base ISA and its standard extensions have now been ratified by RISC-V International, ensuring long-term stability and compatibility across implementations [35]. Although several extensions continue to evolve, the ratified modules are considered final and guaranteed to remain supported in future revisions. Among the available extensions, the vector extension was defined to enable the implementation of vector architectures based on the RISC-V ISA.

1.4.1 Vector Extension

The RVV follows a Vector-Length Agnostic (VLA) approach and does not fix the vector width in hardware. This model allows the same binary code to execute correctly on processors with different vector register lengths, dynamically adapting the number of processed elements. Such flexibility makes the architecture applicable to many systems, from small embedded processors to high-performance computing platforms [36].

At the hardware level, each RVV-enabled implementation defines a set of structural parameters that determine its architectural capabilities. These include the Vector Register Length (VLEN), which specifies the number of bits available in each of the 32 vector registers (v_0–v_{31}), and the Maximum Element Width (ELEN), which indicates the largest element size supported by the hardware. All element widths smaller than or equal to ELEN must be implemented to ensure compatibility across devices. Together, these parameters establish the physical boundaries of the vector register file and the level of data parallelism achievable in a given design.

On top of the hardware-defined parameters, the RVV allows software to configure additional parameters dynamically to match the data type and workload characteristics. The Selected Element Width (SEW) determines the size of each element stored in the vector registers, and it can be 8, 16, 32, or 64 bits depending on the application. Another

configuration parameter, the Vector Register Group Multiplier (LMUL), defines how many consecutive vector registers are grouped to form a single vector operand. Increasing the LMUL value allows a vector instruction to operate on a wider vector across multiple registers. In contrast, fractional values (e.g., 1/2 or 1/4) enable more efficient use of the register file when working with narrow data types.

The configurations of software parameters are stored in a dedicated set of Control and Status Registers (CSRs) defined by the RVV specification. The Vector Type (VTYPE) encodes the active settings for SEW and LMUL, as well as flags that specify mask- and tail-agnostic behaviors. The control registers Vector Start Index (VSTART), Fixed-Point Rounding Mode (VXRM), and Fixed-Point Saturation Flag (VXSAT) manage the starting index for partial operations, the fixed-point rounding mode, and the saturation status, respectively. The Active Vector Length (VL) register stores the number of elements that will be effectively updated in the following vector instruction, and the Maximum Vector Length (VLMAX) is given by $\text{VLMAX} = \text{LMUL} \times \text{VLEN}/\text{SEW}$. Additionally, the Vector Register Length in Bytes (VLENB) register provides the VLEN in bytes.

Figure 1.4 shows how the vector registers are grouped according to the LMUL configuration. In this example, the SEW is 8b and VLEN is 32b, and as the LMUL value increases, more registers are combined to form a wider vector, allowing a single instruction to operate on a larger number of elements.

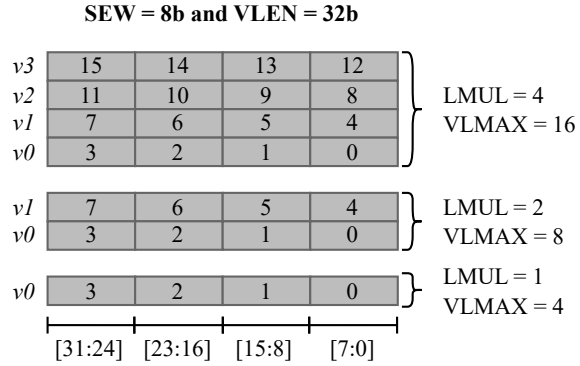


Figure 1.4 – Example of vector register grouping for different LMUL configurations, considering SEW = 8b and VLEN = 32b.

At the instruction-set level, the RVV reuses the base RISC-V instruction formats within three major opcodes to encode vector operations. Specifically, vector load and store instructions are mapped onto the `LOAD-FP` and `STORE-FP` major opcodes. Furthermore, a new major opcode, `OP-V`, encodes arithmetic, logical, and configuration operations. Within these opcodes, the RVV defines several vector instruction formats, such as vector-vector (`VV`), vector-scalar (`VX`), and vector-immediate (`VI`), which specify the bitfield organization

for each operation type. Each instruction defines source and destination vector registers along with control fields such as the `funct6` field, which identifies the operation, and the `vm` bit, which enables or disables masking. When masking is active, the mask is stored in the vector register v_0, and only elements whose corresponding mask bits are set to one are updated. Figure 1.5 illustrates these vector instructions encoding as defined in the RVV v1.0 specification.

The configuration of the vector CSRs is done through vector configuration instructions that initialize the VL and VTYPE registers according to the selected VL and element type. The `vsetvl` instruction calculates the new VL using register operands for both the source and configuration values. The `vsetvli` variant uses a register for the source and an immediate field for the configuration, and `vsetivli` uses immediates for both.

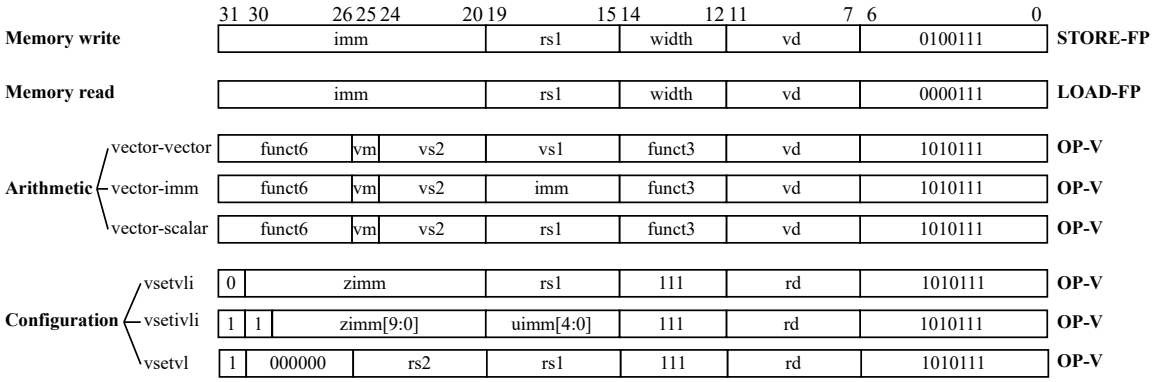


Figure 1.5 – Vector instruction opcodes used by the RVV.

The RVV defines three memory-access modes. The unit-stride mode transfers sequential elements from memory to registers. Meanwhile, the strided and indexed modes enable more flexible patterns for accessing non-sequential data. Segment load and store instructions further allow multiple vectors to be transferred simultaneously, supporting interleaved or structured data such as complex numbers or multi-dimensional coordinates.

Arithmetic and logical operations in the RVV are organized into functional groups according to operand width and operation type. The architecture provides single-width, widening, and narrowing versions of basic arithmetic operations such as addition, subtraction, and multiplication. Widening operations are generally given a `vw*` prefix on the opcode, or `vfw*` for vector floating-point instructions, and produce results with twice the element width of their inputs. Contrarily, narrowing instructions convert operands into smaller element widths, facilitating mixed-precision computations and efficient data reuse within the vector register file.

The execution model is consistent across configurations, and each element of the VL executes the same operation in parallel. Implementations may divide the execution into multiple segments depending on the hardware datapath width. A design with narrow execution units can process a single vector across several cycles, whereas a wider datapath may complete the operation in a single step. This decoupling between architectural definition and physical datapath ensures functional portability across implementations with different performance and area constraints.

Overall, the RVV provides a modular and scalable mechanism for vector computation. Its configurable parameters and instruction set allow designers to balance performance and hardware complexity according to system requirements. This flexibility has also enabled the definition of specialized subsets designed for embedded processors, collectively referred to as the *Zve** extensions.

*Zve**: Vector Extensions for Embedded Processors

Within the RVV specification, a family of five reduced vector extensions, denoted as *Zve**, was introduced to address the requirements of embedded systems. These subsets preserve compatibility with the full vector extension while limiting the supported element widths and instruction categories. The vector operations are limited to 32-bit or 64-bit integers, fixed-point, and single- or double-precision floating-point.

All *Zve** extensions share the same fundamental vector architecture, including the 32 vector registers and the associated CSRs. The main distinctions between them are the ELEN and the set of supported data types depending on the selected variant. Table 1.1 lists the minimum VLEN and supported element widths for each *Zve* extension and what floating-point types are supported.

Table 1.1 – Embedded vector extensions.

Extension	Minimum VLEN	Supported Element Widths	FP32	FP64
Zve32x	32	8, 16, 32	–	–
Zve32f	32	8, 16, 32	Supported	–
Zve64x	64	8, 16, 32, 64	–	–
Zve64f	64	8, 16, 32, 64	Supported	–
Zve64d	64	8, 16, 32, 64	Supported	Supported

Each subset provides a core set of vector arithmetic, logical, comparison, load/store, and reduction operations suitable for embedded workloads. In contrast to the full RVV, the *Zve** extensions exclude features such as advanced permutation and floating-point fused operations,

segment loads and stores, and other high-throughput instructions for large-scale computing platforms. Among these subsets, the *Zve32x* represents the minimal configuration and serves as the foundation for most embedded implementations of the vector extension. It provides an efficient balance between computational capability and implementation simplicity, making it suitable for low-power processors and microcontrollers that integrate vector acceleration within constrained hardware resources.

1.5 Radiation environments

Radiation is defined as the emission or transmission of energy in the form of electromagnetic waves or energetic particles. It is naturally present in the environment with diverse spectra and energy ranges, classified as ionizing or non-ionizing based on the ability to alter atoms and molecular structures through several physical processes (e.g., Compton scattering, nuclear interactions, Coulombic interactions, fission reaction) [37]. In electronic devices, the interaction with ionizing radiation can affect their correct functioning, causing errors and failures [37].

Ionizing radiation exists in both natural and artificial environments, each characterized by distinct particle spectra and energy distributions [38]. The radiation environments can be categorized into three groups [39]. The first category is space, which includes areas near Earth as well as outer space. The second category is the atmosphere, which refers to the radiation found within the Earth's atmosphere. The third category is the ground, including natural terrestrial sources such as radioactive elements found in the air, soil, and rocks, as well as artificially generated sources like particle accelerators. Therefore, the following subsections will present the different radiation environments mentioned.

1.5.1 Space Environment

The space radiation environment is mainly composed of Galactic Cosmic Rays (GCRs), Solar Energetic Particles (SEPs), and the trapped radiation inside the Earth's magnetic field, known as the Van Allen belts [40–42]. GCR are high-energy particles that originate from beyond the solar system. While their exact origins remain a topic of ongoing research, supernova explosions are widely considered to be the primary sources [43]. These cosmic rays are predominantly composed of protons (approximately 87%), followed by alpha particles (around 12%) and a smaller fraction of heavy ions (about 1%) [44]. Their energy spectrum reaches up to 10^{11} GeV, with fluxes generally ranging between 1 and 10 cm^{-2}/s [40].

Most GCR are filtered by the heliosphere and its interactions with the interstellar medium before reaching Earth. However, some of this flux still penetrates to near-Earth space. The intensity of GCRs is modulated by the cyclical nature of solar activity, which follows an approximately eleven-year cycle characterized by fluctuations in activity levels by a factor of two. This cycle can be divided into a minimum phase lasting about four years and a maximum phase lasting around seven years [45, 7]. As a result, higher fluxes of GCRs are observed during solar minimum, while during solar maximum, the stronger heliospheric magnetic field reduces their intensity [46].

SEPs are energetic particles emitted during solar bursts, usually occurring as solar wind, solar flares, and Coronal Mass Ejections (CMEs) [40]. Their frequency and intensity are strongly influenced by the eleven-year solar cycle previously mentioned [45]. The solar wind is a continuous emission of a stream of protons, electrons, helium ions, and small amounts of other heavy ions by the Sun's outer atmosphere (corona) [45]. These particles stream radially outward from the Sun at a velocity of between 300 and 900 km/s, with a temperature on the order of 10^4 to 10^6 K, a density of 1 to 30 $particles/cm^3$, and energies that range from around 0.5 to 2.0 keV/nuc [45].

Solar superstorms can produce intense X-rays and radio bursts, accelerate solar particles, and disturb the solar wind [47]. These superstorms are responsible for triggering solar flares and CMEs, which are considered more unpredictable events. The solar flare is a sudden burst of energy caused by an increase in the localized energy storage in the coronal magnetic fields that exceeds a critical instability threshold [45, 7, 48]. CMEs are eruptions that expel accelerated particles such as protons, electrons, alpha particles, and heavy ions, with speeds ranging from 50 to $2,500 km/s$ depending on the event [45, 7, 40].

The Earth's magnetic field can deflect and trap particles coming mainly from the solar particles and GCRs. It then creates a cavity known as the magnetosphere, a dynamic structure that suffers from the influence of the solar activity variations [40, 49]. The created trapped fields of radiation are known as Van Allen belts [42], which are presented in Figure 1.6. These regions are mainly composed of protons and electrons. Alpha particles and heavy ions are also present but usually negligible due to spacecraft shielding [41].

The inner belt extends from about 1.2 to 2 R_E, where R_E corresponds to Earth's radius, which is approximately 6371 km. It contains protons with fluxes from 10^4 to 10^5 $cm^{-2}s^{-1}$ and energies from a few MeV up to hundreds of MeV, together with electrons with fluxes around 10^9 $cm^{-2}s^{-1}$ and energies above 10 MeV [40]. The outer belt begins at $\gtrsim 3$ R_E and is dominated by electrons with variable fluxes and energies up to 7 MeV [40]. The region between the inner and outer belts is denominated the slot region [41, 50].

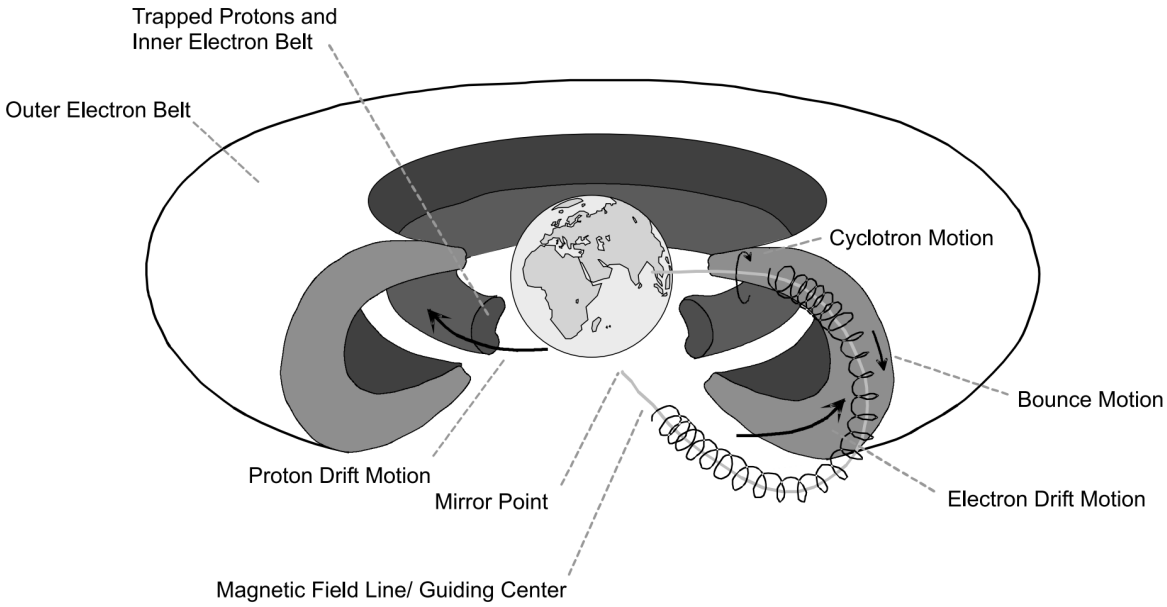


Figure 1.6 – Simplified representation of the trapped radiation and its motion in the Van Allen belts. Figure from: [16].

Satellites in Low Earth Orbit (LEO) are particularly vulnerable to trapped radiation when passing through two portions of the orbit: the polar horns and the South Atlantic Anomaly (SAA) [40]. The polar horns expose spacecraft to electron fluxes below 1,000 *km* altitude and to both electrons and protons at higher altitudes. The SAA is located over South America and has a weak magnetic field. This magnetic field depression is caused by the 11° angle between the magnetic and geographic axes, resulting in an increase in particle flux at lower altitudes [51, 45]. Because of this higher flux, some spacecraft designers choose to turn off their systems when passing over this region, to avoid failures, as is the case for the Hubble Space Telescope [51].

1.5.2 Athmospheric Environment

While the Earth's atmosphere offers substantial protection against radiation, it is not capable of blocking all incoming particles. Galactic cosmic rays and solar particles, depending on their energy levels and angles of incidence, can penetrate the atmosphere and trigger interactions that generate secondary radiation [41]. These interactions mainly occur with oxygen and nitrogen nuclei, resulting in cascades of secondary particles, also known as air showers, which include neutrons, protons, mesons, muons, and electrons [47]. The flux of these secondary particles increases with altitude, peaking at approximately 18 *km*, a

phenomenon referred to as the Pfotzer maximum, and subsequently declines towards ground level as the atmosphere absorbs the majority of the cascades [47, 52, 53].

At sea level, the dominant particles are neutrons, followed by a minor contribution from protons and a few pions. Neutrons are considered the most critical component because they are neutral, highly penetrating, and the leading cause of soft errors in electronic systems [41, 54]. According to JEDEC specifications, the integral flux of neutrons in New York at sea level is about 13 n/cm^{-2}/h for energies above 10 MeV and 6.5 n/cm^{-2}/h for thermal neutrons with energies below 400 meV [55]. It is estimated that at altitudes for subsonic flight levels (around 12 *km*), the particle flux is 300 times greater than at sea level, increasing to 500 times at an altitude of 18 *km* [47]. For this reason, neutron-induced effects are a significant concern for both avionics and electronic systems located in high-altitude regions, as they significantly impact system reliability [56].

1.5.3 Artificial Radiation Environment

The radiation environments created by human activities are known as artificial radiation environments. These are found in biomedical devices, nuclear power plants, particle accelerators, and other high-energy physics experiments [7]. A well-known example is the Large Hadron Collider (LHC) at CERN, a 27 *km* particle accelerator where high radiation levels necessitate the careful qualification of all electronic units to ensure the correct functioning of controller systems [57]. Ionizing radiation is also present in nuclear fission power plants. In the International Thermonuclear Experimental Reactor (ITER) facility, a large flux of neutrons, X-rays, and gamma rays impacts the electronics located near the reactor vessel and the bioshield, making the reliability of these systems a critical issue [58, 7].

To characterize electronic devices for harsh radiation environments, experiments are commonly conducted in artificial facilities that replicate the particle spectra encountered in space or atmospheric conditions. Several irradiation infrastructures are used worldwide, including ChipIr at the Rutherford Appleton Laboratory in the United Kingdom [59], Proton Irradiation Facility (PIF) at the Paul Scherrer Institute (PSI) in Switzerland [60], PARTREC in the Netherlands [61], RADEF in Finland [62], and CHARM at CERN in Switzerland [63].

1.6 Radiation Effects in Electronic Devices and Reliable System Design

The type and severity of the radiation effects on electronic devices depend on several factors, including the type of particle, its energy, the accumulated dose, and the characteristics of the

device technology. In general, radiation effects in electronics are divided into Total Ionizing Dose (TID), Displacement Damage (DD), and Single-Event Effects (SEEs) [7]. TID and DD are cumulative, caused by the accumulation of ionizing radiation resulting in a long-term degradation of the device parameters, such as the increase of the leakage current and shift of the transistor threshold voltage [7, 64]. SEEs occur due to the interaction of a single particle that deposits charge in a sensitive region of the device [7, 45].

In systems, the radiation-induced effects are seen as faults at higher levels. In this context, ensuring reliability is essential to maintain correct operation, and this is done by applying fault-tolerant architectures and protection mechanisms. The following subsections describe the different types of ionizing radiation effects in electronics and the mitigation approaches commonly adopted to reduce their impact on system behavior.

1.6.1 Cumulative Effects

As mentioned before, Total Ionizing Dose (TID) and Displacement Damage (DD) are types of cumulative radiation effects in digital electronics, observed after a certain amount of exposure, usually leading to critical damage in the device that becomes no longer operational. These effects are widely known and studied since the 1960s [65, 66].

The TID is defined as the amount of energy deposited in a material through an ionization process, typically measured in rad or gray (*Gy*) [64]. In this context, 1 rad is equivalent to 100 ergs of absorbed energy per gram of material, while 1 *Gy* is defined as 100 rads. In Metal-Oxide-Semiconductor (MOS) devices, the energy deposited in the oxide generates electron-hole pairs that may recombine or be trapped. When these pairs become trapped in the silicon oxide (SiO_2) or its interfaces (Si/SiO_2), they can lead to significant changes in the device's characteristics[67], depending on the specific device structure, materials, and operating principle [8, 67–69]. For example, this accumulation of charge can induce a shift in the threshold voltage (ΔV_{th}) and may lead to an increase in the device's leakage current. With continuous technology scaling, the gate oxide thickness has been reduced, limiting the amount of charge that can accumulate in this region.

The generation of electron-hole pairs depends on the electric field applied during irradiation. When an electric field is present, the higher mobility of electrons enables them to move away from the oxide regions, while the holes are drawn toward the oxide and the interface. Although some of these pairs recombine, a portion becomes trapped, resulting in fixed charges. These trapped charges can lead to shifts in threshold voltage and an increase in sub-threshold leakage current. With continuous technology scaling, the gate oxide thickness has been reduced, limiting the amount of charge that can accumulate in this region. As a result, other structures, such as thick isolation oxides and spacers, now account for a

significant portion of TID-related degradation [7]. Furthermore, the presence of trapped charges at the interface degrades channel mobility due to increased carrier scattering. Overall, the intensity of TID effects depends on the specific device structure, materials, and operating principle.

DD results from coulombic interactions or nuclear reactions involving the nuclei [70]. When the energy transferred during the interaction between radiation and matter exceeds a certain threshold, it can eject a nucleus from its lattice position and create a lattice defect. These lattice defects are vacancies, the absence of atoms, and interstitials, which are displaced atoms residing on non-lattice positions [7, 70]. The formation of defects can either occur as isolated entities or in clusters, depending on the incident particle mass and energy [70]. The DD effects in semiconductors primarily relate to the energy levels introduced by the bandgap, and the accumulation of this effect can have significant impacts on the electrical and optical properties of semiconductors. At the device level, this leads to higher leakage current, shorter carrier lifetime, and reduced gain, particularly in bipolar transistors. In general, DD effects are a significant concern for electronic devices and minority carriers, such as a bipolar transistor. In the context of MOS transistors, these effects are particularly impactful, especially under high particle fluences [70].

1.6.2 Single-Event Effects

A single ionizing particle passing through a semiconductor dissipates energy by coulombic interactions and leaves behind a dense track of electron–hole pairs. The excess charge carriers may reach sensitive regions of a device and induce a state change, which is known as a Single-Event Effect (SEE) [7, 45]. The most vulnerable regions are often reverse-biased *p/n* junctions, since the electric field in these areas enhances charge collection efficiency [71].

In contrast to dose effects that accumulate over time, SEEs arise from a single particle strike, causing a transient state on a device that may lead to recoverable (soft) or permanent (hard) damage. Soft errors are those that can be mitigated, such as a memory bit flip that is corrected when the cell is rewritten. In contrast, hard errors involve physical damage, like a ruptured gate dielectric. Some events, such as Single-Event Latch-up (SEL), may be either soft or hard depending on the system's reaction. If the supply is not cut quickly after the trigger, the error can escalate into permanent damage [7, 45]. The following subsections describe some SEEs observed in electronic devices.

Single-Event Upset

A Single-Event Upset (SEU) occurs when a particle strike corrupts a stored value in memory. The effects of SEUs can vary based on the number of bits affected. If only one bit is flipped, it is referred to as a Single Bit Upset (SBU). If two bits are flipped, it is referred to as a Double Bit Upset (DBU). When more than two bits are affected, it is known as a Multiple Bits Upset (MBU). Additionally, when the bit flips occur across multiple cells within the same memory array and are not limited to a single word, this condition is referred to as a Multiple-Cell Upset (MCU).

The mechanisms to generate an SEU have a strong dependency on memory technology. In Dynamic Random Access Memory (DRAM), the technology is based on a one-transistor, one-capacitor cell structure, and the charge stored by a capacitor represents the information, accessed through a transistor. The SEU mainly occurs due to charge collection within the memory binary cell, caused by a particle strike that occurs in or near the storage capacitor or the access transistor [71, 72]. Cell upsets occur through charge collection at the biased junction of the cell-access transistor or through charge transfer due to a low-resistive path created by an ionizing particle that moves electrons from a low-voltage node to a high-voltage node [71]. Also, upsets can be caused by a charge collection in the pre-charged bit line during a memory access, introducing an imbalance in the sensing signal during or before the sensing operation.

In Flash memories, the effect appears as a threshold voltage shift of the floating-gate transistor. Radiation can discharge or redistribute trapped charges, reducing the threshold below the reference used to define the logical state, which leads to an upset. The precise physical mechanisms remain debated in the literature, and different hypotheses have been proposed for various technologies [73].

Single-Event Transient

The passage of an ionizing particle through a semiconductor can create a path of free carriers that is collected by a sensitive node, producing unexpected behavior. In this case, the phenomenon is defined as a Single-Event Transient (SET), which differs from the SEU because it is primarily associated with combinational logic in digital circuits and with analog circuits. The charge collected during the strike generates a voltage or current transient that can propagate through the circuit. If the transient maintains a sufficiently high amplitude and duration, it may be latched by a storage element, resulting in a soft error. This mechanism is particularly critical in high-frequency circuits, since the short clock period increases the probability that the transient coincides with the sampling window of a memory element [7].

In analog circuits, this effect can manifest as unintended voltage pulses at the output of operational amplifiers or as deviations from the expected system behavior.

Single-Event Functional Interruption

A Single-Event Functional Interrupt (SEFI) is the temporary or persistent loss of functionality of a device caused by a single ion strike. It may be triggered by upsets in configuration registers or other critical control elements, leading the system into undefined states [74]. SEFIs may persist until a reset or power cycling, but in some cases, the interruption can resolve itself after a long time. The precise sensitivity depends on the design choices and the technology used [75].

Single-Event Latch-Up

The Single-Event Latch-up (SEL) is produced by a parasitic bipolar structure present in bulk Complementary Metal Oxide Semiconductor (CMOS) technology, which behaves as a thyristor. This parasitic structure is commonly found in CMOS circuits, such as in a typical inverter, and it usually remains in a high-impedance state. However, the passage of an ionizing particle and the subsequent charge collection may provide the excitation that forces one of the transistors to enter an on-state, creating a loop condition that defines a low-impedance path between the supply voltage and ground. As a consequence, the circuit reaches a high-current state, which causes the device to malfunction [75, 76].

The latch-up effect can usually be interrupted by performing a power cycle operation, after which the device returns to its normal behavior. If the power is not removed quickly, the current may persist and cause thermal runaway, producing irreversible damage [77]. Different techniques, such as those described in [77], can protect the circuits against latch-up. Moreover, the Silicon On Insulator (SOI) circuit's dielectric isolation makes it immune to SEL, since it eliminates the parasitic thyristor [78]. However, as demonstrated in [79], some of the SOI processes, like the use of LOCal Oxidation of Silicon (LOCOS) as lateral dielectric, can generate the *pnpn* path and cause SELs.

1.6.3 Reliable Systems

Reliable systems are designed to maintain correct operation even in the presence of faults that may occur in their components. A fault-tolerant system is characterized by its ability to continue operating correctly when one or more faults arise in its structure [9]. Faults can result from physical defects, such as a damaged transistor or wire, and manifest as errors at the logical level, for instance, when a bit that should be '0' is read as '1'. These errors

may propagate through the system and eventually cause failures if they are not detected or corrected [80].

To achieve fault tolerance, several techniques can be applied at different abstraction levels, depending on the criticality of each component. The most common strategy is redundancy, which can be classified into spatial, temporal, and informational forms [80].

Spatial redundancy is one of the most used methods for error detection and mitigation. Dual Modular Redundancy (DMR) is the simplest implementation, in which the module is duplicated and the two outputs are compared [80]. This technique allows for the detection of errors but is not capable of correcting them. To tolerate faults, the Triple-Modular Redundancy (TMR) technique adds a third module and a voter that selects the majority result, allowing the system to mitigate the errors.

Temporal redundancy uses time as a resource to detect and mitigate transient faults, especially SETs. In this approach, an operation is executed multiple times in the same hardware unit, and the results are compared. When the outputs differ, the operation can be repeated or corrected based on the majority result. This technique improves reliability without requiring extra hardware resources, though it increases execution time [80].

Information redundancy enhances data reliability by adding redundant bits that enable error detection and correction. The simplest method is the use of parity bits for error detection [80]. More advanced techniques apply Error Detection and Correction (EDAC) codes, also known as Single Error Correction and Double Error Detection (SECDED), which allow both single-bit correction and double-bit detection. The most widely used EDAC algorithm is the Hamming code [9].

1.7 HARV-SoC

HArdened RISC-V (HARV) is a hardened RISC-V processor designed with specific fault tolerance strategies based on the base integer instruction set from RISC-V (RV32I). The processor is implemented as a soft-core described in VHSIC Hardware Description Language (VHDL), which allows its synthesis on different FPGA technologies and ensures portability across platforms. It employs a hardened register file with SECDED in critical memory elements. HARV uses a combination of TMR and redundancy-aware control logic to mitigate transient faults. Its instruction execution path is designed to detect and correct faults dynamically, ensuring high reliability in critical applications. HARV integrates these mechanisms within a lightweight and resource-efficient framework to ensure adaptability to different platforms and constraints.

The architecture of HARV is multi-cycle, divided into five stages: instruction fetch, execution, memory access, write-back, and trap handling. This architecture enables faster execution for non-memory access instructions, which take at least one cycle less than memory operations. Moreover, the dedicated stage for handling traps simplifies the logic for managing interruptions and exceptions.

The HARV-SoC comprises the HARV processor core and several peripherals integrated into an architecture targeting reliability with flexibility and low hardening overheads [81]. It has standard features of modern microcontrollers, such as General Purpose Input/Output (GPIO), Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C), Universal Asynchronous Receiver Transmitter (UART), WatchDog Timer (WDT), and controllers that enable access to external memories, such as Synchronous Dynamic Random-Access Memory (SDRAM) and Flash devices.

Regarding reliability, additional mechanisms of the HARV-SoC enhance robustness. The external data memory is protected by an Error Correcting Code (ECC) controller implementing SECDED for each 32-bit word. The master bus controller includes a timeout flag that raises an exception when a peripheral fails to respond within the expected time, allowing the application to identify faulty peripherals. A reset controller enables selective recovery of peripherals, while the WDT resets the system when a global deadlock or timeout occurs. During startup, a Cyclic Redundancy Check (CRC)-32 checksum verification checks the integrity of the program loaded from Flash memory.

This SoC was developed to be flexible and can be used in various FPGAs, which may use different technologies that suffer different effects when exposed to radiation. While flash-based FPGAs have a more resilient configuration memory, Static Random-Access Memory (SRAM)-based FPGAs' most common errors are in their Configuration Memory (CMEM). Therefore, the HARV-SoC has an interface with a primitive structure of Xilinx devices that reports errors in the configuration memory through a built-in scrubbing. The reports provided by this structure are integrated with a specific error handler, which also interfaces with HARV to report these errors to the application.

Figure 1.7 presents an overview of the SoC architecture, including the processor and the error handler modules, which are responsible for error detection and reporting. The external memory can be implemented by interfacing with a memory controller or internal FPGA memories. The dashed square highlights the structures specific to reporting upsets in the configuration memory of Xilinx FPGAs.

Concerning observability and monitoring capabilities, HARV includes an advanced observability and monitoring structure that enables real-time error tracking, logging, and analysis. The system comprises multiple fault detection mechanisms that allow the processor

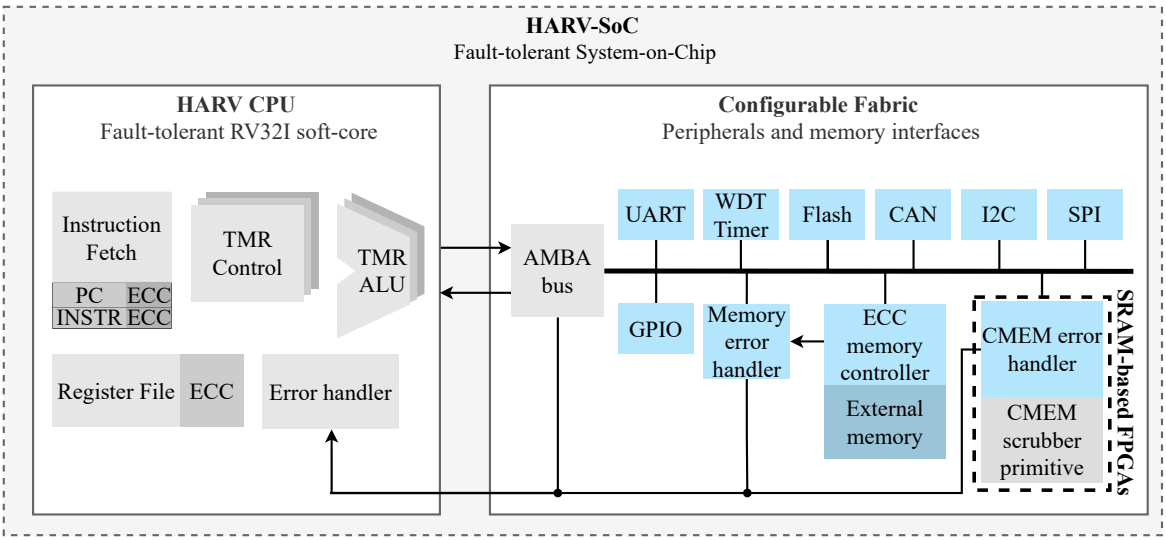


Figure 1.7 – HARV-SoC architecture.

to classify and respond to radiation-induced errors autonomously. By leveraging error monitors and detailed registers, the SoC can efficiently detect transient faults, track error propagation, and assess the impact of single-event effects on performance [82].

Fig. 1.8 presents the error handler structure, which has the hardened structure flags as inputs, stores detailed error information in internal registers, and requests an exception to the processor whenever there is an error that could affect the applications' reliability. The detectable errors in the error handler include the following errors of the processor: PC single- and double-bit upsets, Instruction Register (IR) single- and double-bit upsets, register file single- and double-bit upsets, control TMR error, and Arithmetic Logic Unit (ALU) TMR error. Besides reporting errors in the processor core, the error handler design also includes the external error input, which is available as a processor core input signal. The purpose of this signal is to enable the SoC to report errors in its structures, such as interconnections and peripherals.

The fault observability system is designed to correlate error events with operational conditions, providing insights into environmental influences on processor behavior. This allows for adaptive fault tolerance strategies at the application level, wherein the system dynamically adjusts redundancy levels, modifies error correction parameters, and optimizes workload distribution based on observed error trends. Such adaptability ensures that HARV-SoC maintains high reliability over extended mission durations.

The fault tolerance techniques implemented in HARV-SoC were validated through irradiation experiments with neutrons, protons, and mixed-field radiation at facilities such as ChipIr, CHARM, and PARTREC [83, 82, 84, 85]. Moreover, HARV-SoC achieved in-orbit

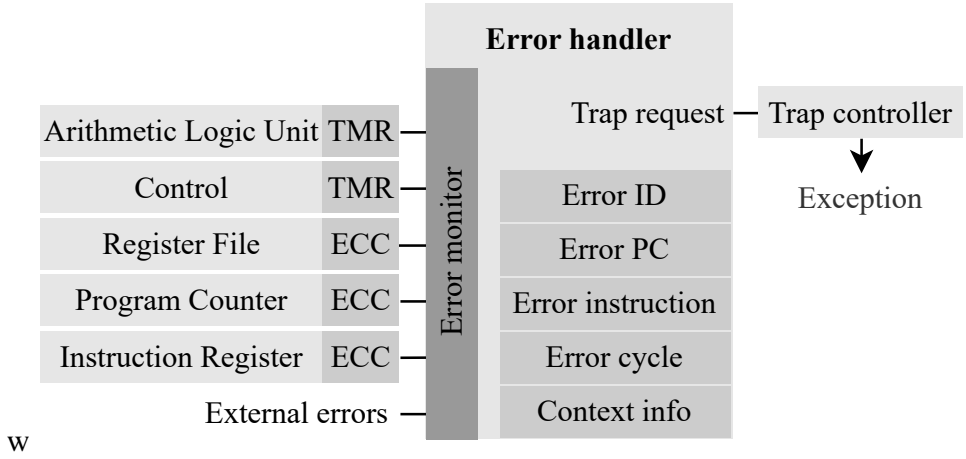


Figure 1.8 – HARV-SoC Error Handler.

validation as part of the *ROBUSTA-3A Méditerranée* CubeSat mission launched in June 2024 [86].

1.8 Related Work

Several implementations of the RVV have been proposed to explore different aspects of vector computing, from high-performance systems to embedded processors. These studies demonstrate the flexibility of the RVV model to accelerate data-parallel workloads across different domains, including AI inference. The following paragraphs summarize the main developments that shaped this field.

The Ara processor [87], developed at ETH Zurich, was the first high-performance implementation of the RVV. Designed for computationally demanding workloads such as dense linear algebra, Ara integrates a decoupled vector unit into a superscalar RISC-V pipeline and sustains high utilization on scientific kernels. Its evolution, Ara2 [88], is the first fully open-source design compliant with RVV 1.0 and extends the evaluation beyond traditional HPC kernels to include Machine Learning (ML)-oriented workloads such as *conv2d* and *softmax*. *Ara2* added the reduction operations that were absent in Ara and are widely used in ML, improves the microarchitecture and memory coherence with the scalar core, and explores single- and multi-core scaling. Across compute-bound kernels, Ara2 reports near-peak functional-unit utilization and analyzes bottlenecks tied to vector length and scalar issue rate [88].

The Vitruvius+ [89] is an area-efficient decoupled vector coprocessor design for HPC, focusing on floating-point computation and energy balance. While primarily targeting scientific applications, its architecture also provides support for matrix-oriented operations,

commonly used for NN inference. Building on Vitruvius+, Barbirotta *et al.* [90] extended its design to investigate reliability mechanisms in RVV-based architectures. The proposed approach integrated instruction duplication, ECC protection in the vector register file, and rollback recovery to mitigate transient faults. Fault-injection campaigns demonstrated up to 85 % reduction in silent data corruption with moderate area overhead.

RVVe [91] introduces a minimalist vector extension co-designed for AI workloads in embedded systems. Instead of implementing a full RVV profile, the work defines a reduced instruction set optimized for Deep Neural Network (DNN) inference. It integrates it as a multi-cycle vector unit within the NEORV32 SoC [92]. The unit is controlled directly from the execution stage without a dedicated pipeline and includes configurable vector length and lightweight control logic. The processor was evaluated on convolutional layers inspired by ResNet and a CNN for Modified National Institute of Standards and Technology (MNIST) classification, achieving up to 11.9 times speedup compared to the scalar NeoRV32 baseline with significantly lower hardware overhead than prior RVV-based designs.

The processor proposed in [93] implements a soft RVV core based on the RVV 0.7.1 specification, targeting Edge AI applications. The architecture integrates a vector unit with Multiply–Accumulate (MAC) units to accelerate convolutions and matrix multiplications. The design was evaluated on convolutional layers, perceptrons, and an image classification network using the MNIST dataset, achieving execution-time reductions from 2 to 4 times compared to the scalar baseline.

Vicuna 2.0 [94] extends the earlier Vicuna [95] processor by adding support to the Zve32f and Zvfh subsets. The Zvfh is an extension that provides support for half-precision floating-point operations. The vector unit is integrated with a RISC-V scalar core and was evaluated using a set of lightweight NN benchmarks representative of TinyML applications, such as keyword spotting and image classification. The results show performance improvements of approximately 3 to 5 times compared to the scalar implementation, while reducing the logic resources by around 25%.

Di Mascio *et al.* [96] extended the NOEL-V processor [97], a space-qualified RISC-V implementation, by integrating a subset of integer vector instructions. The prototype, validated on FPGA, achieved up to 23 times acceleration in compute-bound kernels and noticeable improvement in CNNs, demonstrating the potential of vector processing for on-board AI computing in radiation environments.

Discussion

The reviewed works illustrate the evolution of the RVV from performance-oriented implementations to embedded processors designed for AI acceleration and, more recently, for

space applications. Early developments [87, 88] focused on maximizing throughput and implementing the full RVV specification. These designs targeted scientific and HPC applications and validated the vectorization for dense linear algebra and ML kernels. However, their resource and power requirements limit their use for embedded systems.

More recent efforts redirected the use of the RVV to support AI inference in constrained systems. The works from [91, 93, 94] demonstrated that reduced subsets of the vector extension can effectively accelerate NN workloads while maintaining low resource usage. These works highlight the adaptability of the standard to small embedded designs by trading architectural complexity for energy efficiency and integration simplicity.

Reliability and dependability have more recently become relevant aspects of vector-processor design. Works such as [90] explored protection mechanisms for the Vitruvius+, focused on HPC. Also, Di Mascio *et al.* [96] integrated vector acceleration into a radiation-tolerant RISC-V processor, showing the potential of the RVV for onboard AI computing in space.

However, none of the reviewed implementations analyzed the impact of vectorization on the reliability of CNN inference. This thesis addresses this gap by evaluating both the performance and reliability of a vector extension integrated into the fault-tolerant HARV-SoC, enabling reliable vector processing for CNN inference in embedded systems.

1.9 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 investigates the reliability of a commercial Edge AI microcontroller executing CNNs inference under irradiation. This initial study establishes a reference for understanding how radiation affects AI devices that are not hardened for space applications.

Chapter 3 describes the evaluation methodology adopted for the HARV-SoC and the VEU. It details the synthesis and performance evaluation, as well as the fault injection simulation used to assess reliability.

Chapter 4 introduces the preliminary implementation of a VEU integrated into the HARV-SoC. It describes the architectural concept, synthesis results, and the first performance and reliability evaluations.

Chapter 5 presents an enhanced version of the VEU, which aligns with the RISC-V Zve32x subset and adds support for vector multiplication and parameterized LMUL configuration. The chapter reports refined synthesis, benchmarking, and fault-injection results.

Chapter 6 focuses on the application-level evaluation through a vectorized CNN inference. It details the implementation of the vectorized convolution kernel, the experimental setup, and the results obtained from synthesis, performance, and reliability analysis.

Finally, Chapter 7 summarizes the main contributions and findings of this thesis, demonstrating that vectorization can improve CNN inference performance on radiation-tolerant RISC-V processors without compromising reliability.

Chapter 2

Reliability Evaluation of a Commercial Edge AI Microcontroller

In this chapter, I present a reliability analysis of the MAX78000 microcontroller, an ultra-low-power, low-cost Application-Specific Integrated Circuit (ASIC) designed for Edge AI applications with a hardware-based CNN accelerator. The primary focus of this research was to investigate the effects of radiation exposure on the performance and fault behavior of the MAX78000. A layer-by-layer analysis technique was implemented to increase system observability during NN inference, facilitating more effective monitoring and assessment of radiation-induced faults. The contents of this chapter are based on the author's publication [JO-1].

[JO-1] **C. Imianosky**, A. M. P. Mattos, D. A. Santos, D. R. Melo, M. Kastriotou, C. Cazzaniga, and L. Dilillo, "Characterization of Single-Event Effects in a Microcontroller with an Artificial Neural Network Accelerator," in *Electronics*, vol. 13, no. 22, art. no. 4461, Nov. 2024, DOI: 10.3390/electronics13224461. © 2024 by the authors (MDPI) CC BY 4.0

The remainder of this chapter is structured as follows: Section 2.1 presents the devices used for the experiment, the test facility, and the applied test methodology; Section 2.2 presents the results obtained during the experiment; and Section 2.3 presents the conclusion.

2.1 Experimental Setup

To evaluate the Device Under Test (DUT) reliability, an experiment was performed at the ISIS Chip Irradiation facility (ChipIr) beamline [59], part of the ISIS Neutron and Muon Source,

at the Rutherford Appleton Laboratory, United Kingdom. ChipIr generates a neutron beam with a spectrum representative of atmospheric environments, which causes SEE on electronic devices. The facility can generate neutron irradiation with fluxes that are several orders of magnitude higher than those found at ground level on Earth, enabling the accelerated characterization of devices and systems. The beam flux is 5×10^6 cm^{-2}/s for energies higher than 10 MeV, and during the experiment, the total accumulated fluence was 1.11×10^{12} n/cm^2.

2.1.1 Device Under Test

MAX78000 is an ultra-low-power Edge AI microcontroller. It has a heterogeneous architecture that combines two main processing hardware modules: a microprocessor cluster and a CNN accelerator. The microprocessor cluster comprises an ARM Cortex-M4 processor with a floating-point unit, running at up to 100 MHz, and a 32-bit RISC-V co-processor, running at up to 60 MHz, both interfaced to the bus. This bus is connected to several peripherals and the CNN accelerator.

Figure 2.1 illustrates the DUT architecture, highlighting the internal structures of the CNN accelerator. The hardware-based CNN accelerator in MAX78000 features 64 parallel convolutional processors and supports a wide variety of CNN and DNN models. It has dedicated memory for corresponding weights (442 KB), biases (2 KB), and data (512 KB), and it operates at a frequency of up to 50 MHz. Each convolutional processor consists of a pooling engine, input cache, weight memory, and convolution engine, and is responsible for running a convolutional operation of a single input channel.

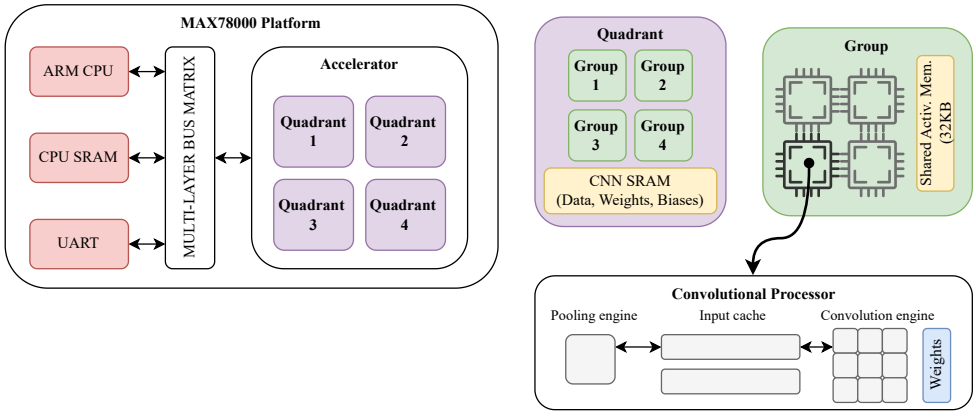


Figure 2.1 – MAX78000 platform architecture.

The convolutional processors are divided into four groups, and processors within a group share a common activation memory for data input and activation data. These groups are then clustered into four sets, forming quadrants, each containing programmable registers to configure the execution of individual layers. This hierarchical structure, dividing the accelerator into quadrants, groups, and processors, allows for precise control over which accelerator sections are active during operation.

MAX78000's Software development Kit (SDK) [98] handles the complete training, synthesis, and deployment of NN models. The training is performed with the help of the PyTorch library for machine learning, data loader, and corresponding datasets. The trained NN model is converted into C code during the synthesis and deployment stages. Once deployed on the device, the accelerator is initialized by configuring the SRAM control bits and specifying the number of CNN layers. Subsequently, weights and biases are moved from general-purpose SRAM into their dedicated memory locations. The configuration then continues by defining the rows and columns for each quadrant, which consists of 16 processors, and mapping the necessary memory pointers. Input data are then loaded into the memory of the accelerator, triggering the inference process. MAX78000 performs NN inference as a single, indivisible task, which inherently limits the ability to monitor intermediate computational results and restricts system observability.

2.1.2 Layer-by-Layer Implementation

In this study, I address a challenge posed by the MAX78000 CNN accelerator, which performs NN inference as a single, indivisible task. This design feature limits the capacity to observe intermediate computational results, making it challenging to verify the accuracy and reliability of the inference process. To overcome this limitation, I used the Layer-by-Layer Transient Toolchain (LbLTT) [17], an external tool designed to enable layer-by-layer processing on MAX78000. LbLTT converts NN inference into several single-layer NNs, allowing each layer to be individually processed. This method allows to observe the output of intermediate layers and verify that the parameters are correctly stored, improving observability without altering the NN structure.

LbLTT is an open-source toolchain developed specifically for NNs compatible with the MAX78000, optimized to allow for intermittent execution by shrinking the atomic execution size to a single layer. This tool achieves this segmentation without modifying network weights or configurations, ensuring that inference accuracy remains consistent with the original network design. It is important to note that LbLTT is applied solely to improve observability, and it introduces some latency to the inference speed [17].

Initially, the toolchain uses the *ai8xize.py* Python script to translate the PyTorch model into C code. This process generates two essential files: *cnn.c*, which contains the functions necessary for loading, starting, and obtaining outputs from the NN accelerator, and *weights.c*, which stores the values of the NN weights. Following this, LbLTT segments the NN into individual layers. The LbLTT then segments the neural network into separate layers, creating a standalone network for each layer, with an input, a layer, and an output. This segmentation enables the creation of configuration files to set up the CNN accelerator at the beginning of the computation of each layer. In the final stage, the LbLTT toolchain merges the segmented layers into a unified execution framework for the MAX78000 CNN accelerator. This framework includes functionality that allows the intermediate output to be extracted and verified as needed. The resulting C code is then ready to be deployed on the MAX78000 CNN accelerator, supporting intermittent execution.

The execution flow, as illustrated in Figure 2.2, contrasts the standard implementation of the entire NN, presented in Figure 2.2(a), with the layer-by-layer approach enabled by LbLTT, shown in Figure 2.2(b). The green dashed rectangle indicates the loop performed for each layer of the original NN model, and the purple one indicates the verification steps implemented to increase the observability of the accelerator. It is important to note that while network weights are loaded at the start of execution, biases are loaded individually for each layer. This is done by modifying the *ai8xize.py* script to include memory offsets for the biases, a necessary step since the Maxim toolchain treats each layer as a separate NN.

The default script maps all layers from a fixed address (e.g., 0×0000) and does not allow for specifying different memory locations for weights, requiring their preloading. After the CNN initialization, the input data for the first layer are loaded into the memory of the accelerator. Subsequently, the biases for this layer are loaded, and the accelerator is configured and started. This process is repeated for each layer of the NN model, allowing for fine-grained control and observation of intermediate results.

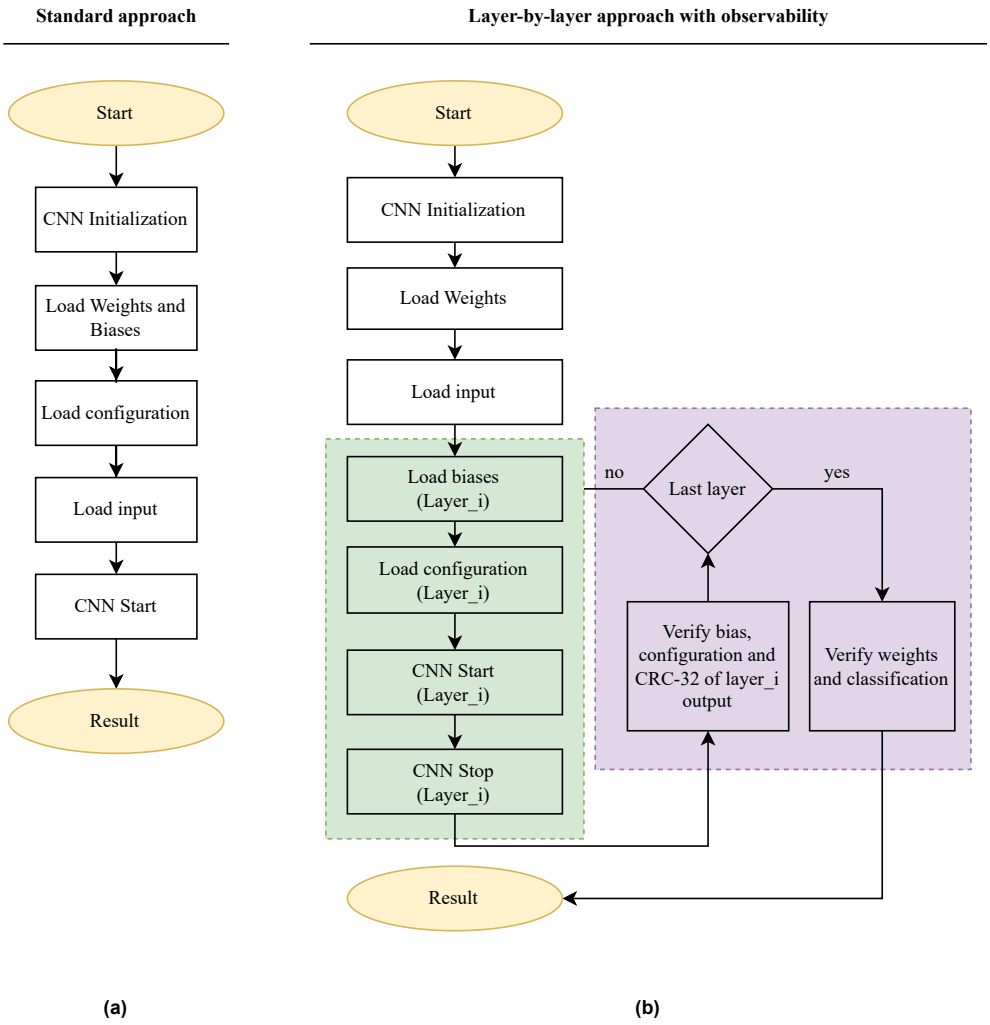


Figure 2.2 – Comparison of CNN execution runtime flow. The left diagram shows the standard approach. Comparison of CNN execution runtime flow. (a) Standard approach. (b) Combination of the LbLTT [17] layer-by-layer approach with the implemented techniques for observability.

I performed golden runs, i.e., error-free runs, on all selected test images from the dataset, obtaining the expected output values for each layer and each image. I then calculated the Cyclic Redundancy Check-32 (CRC-32) for these outputs. CRC-32 is a widely used error-detection method that produces a unique checksum for data [99, 100], enabling the detection of discrepancies that may indicate errors in any output of a layer. With this approach, it was possible to verify the outputs of each layer without storing all of the intermediate results. Additionally, at the end of each layer, I verified the values of the bias and configuration registers. The weights were checked only at the final layer, as verifying them after each layer would significantly extend the inference time.

2.1.3 Neural Networks and Datasets

The datasets used for this study were the MNIST and the Canadian Institute For Advanced Research (CIFAR)-10. The MNIST is a collection of 70,000 28×28 grayscale images of handwritten digits from 0 to 9 [101]. The CIFAR-10 consists of 60,000 32×32 color images divided into 10 classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck [102].

The datasets are normally split into a training set and a testing set. MNIST is divided into 60,000 training images and 10,000 testing images, and CIFAR-10 is split into 50,000 training images and 10,000 test images. For training, I used the complete training sets of both datasets. For testing the applications, I selected a subset of 20 images from each dataset testing set due to the processor's internal memory constraints. The goal was to investigate the accelerator's performance by comparing the results of each image with the ones from the golden execution rather than evaluating the overall accuracy of the AI on a larger dataset. The subset of images was chosen randomly while guaranteeing an equal number of images for each category.

I evaluated and characterized MAX78000 using two different CNN models: the AI85Net5 and AI85NASCifarNet. Both NN models are available in the Maxim Integrated AI8X SDK [98] and were selected since they fit the device's hardware limits while covering different complexity levels. The AI85Net5 is a compact 5-layer model that was trained using the MNIST dataset, whereas the AI85NASCifarNet is a deeper architecture obtained through Neural Architecture Search (NAS) and was trained with the CIFAR-10 dataset.

Tables 2.1 and 2.2 describe the architectures of the two networks used in the MAX78000 experiments, detailing the sequence of layers, output dimensions, and the main configuration parameters such as kernel, padding, and pooling. In both models, each convolutional layer is followed by a ReLU activation, and some of them include pooling to progressively reduce the feature-map dimensions. The AI85Net5 combines max and average pooling, while the AI85NASCifarNet uses only max pooling and includes batch normalization for numerical stability. Both networks end with a fully connected layer that produces ten output classes corresponding to the CIFAR-10 or MNIST datasets.

Table 2.3 presents some additional characteristics of the CNNs used fo the evaluations, including inference times for both the standard and layer-by-layer approaches. The layer-by-layer method increases inference time by 13.7% for CIFAR-10 and 12.8% for MNIST. While this approach adds some latency overhead, it enables detailed fault monitoring across layers, increasing the observability of the system.

Table 2.1 – Architecture of the AI85Net5 model.

	Layer	Output	Kernel	Pad	Pool
1	Conv2D + ReLU	60×28×28	3×3	1	–
2	MaxPool + Conv2D + ReLU	60×14×14	3×3	2	2×2
3	MaxPool + Conv2D + ReLU	56×7×7	3×3	1	2×2
4	AvgPool + Conv2D + ReLU	12×4×4	3×3	1	2×2
5	Fully Connected	10	–	–	–

Table 2.2 – Architecture of the AI85NASCifarNet model.

	Layer	Output	Kernel	Pad	Pool
1	Conv2D + BatchNorm + ReLU	64×32×32	3×3	1	–
2	Conv2D + BatchNorm + ReLU	32×32×32	1×1	0	–
3	Conv2D + BN + ReLU	64×32×32	3×3	1	–
4	MaxPool + Conv2D + BatchNorm + ReLU	32×16×16	3×3	1	2×2
5	Conv2D + BatchNorm + ReLU	64×16×16	1×1	0	–
6	MaxPool + Conv2D + BatchNorm + ReLU	128×8×8	3×3	1	2×2
7	Conv2D + BatchNorm + ReLU	128×8×8	1×1	0	–
8	MaxPool + Conv2D + BatchNorm + ReLU	64×4×4	3×3	1	2×2
9	Conv2D + BatchNorm + ReLU	128×4×4	3×3	1	–
10	MaxPool + Conv2D + BatchNorm + ReLU	128×2×2	1×1	0	2×2
11	Fully Connected	10	–	–	–

Table 2.3 – Characteristics of the CNNs.

Dataset	Weights [Bytes]	Bias [Bytes]	Num. of Operations	Inference Time (µs)	
				Standard	Layer-by-Layer
CIFAR-10	301,760	842	36.5×10^6	4872	5541
MNIST	71,148	10	10.9×10^6	1486	1676

2.1.4 Test Setup

The MAX78000FTHR was used as an evaluation board for the experiment, which provides a platform for programming and debugging the MAX78000 microcontroller and several peripherals. The test setup, shown in Figure 2.3, consists of four MAX78000FTHR boards, where *B1* and *B2* were performing inferences with the CIFAR-10 dataset, and *B3* and *B4* performed inferences with the MNIST one.

For logging the experimental data, serial connections were used to transmit the logs generated to a host computer outside the irradiation room. The setup is fixed in a frame containing the mentioned test boards and other boards utilized for other experiments to optimize the beam time utilization. The supply current for each board was individually monitored and limited in case of a single-event latch-up. The UART output was also being monitored to trigger a power cycle in case of a hang, i.e., if no output was detected for more than one minute.

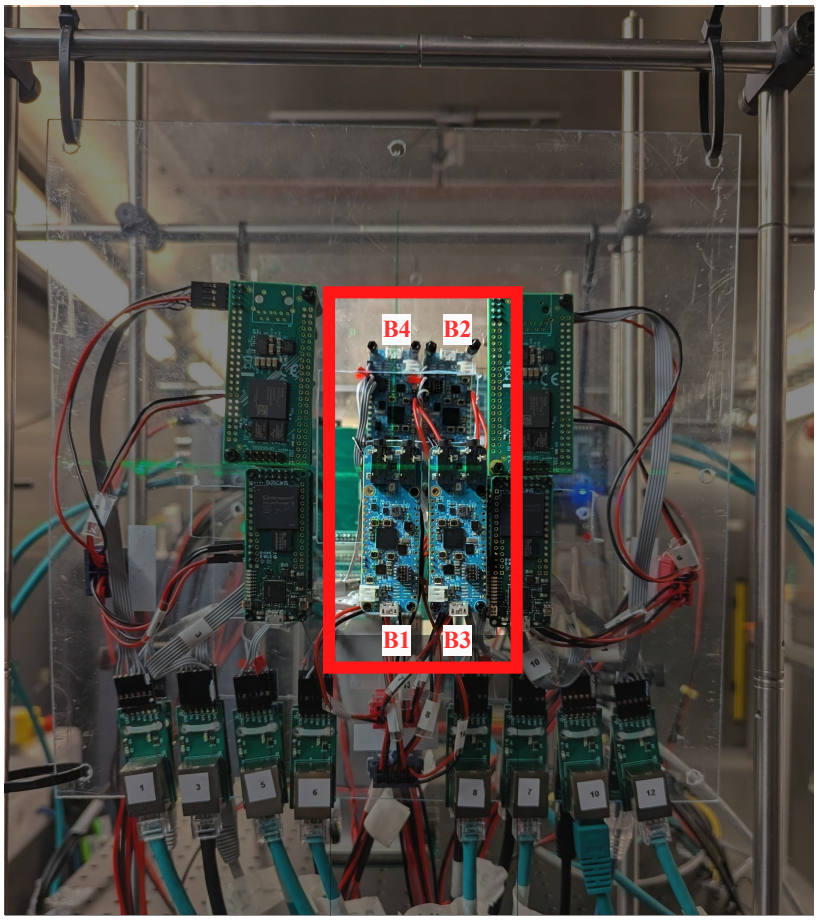


Figure 2.3 – Experimental setups prepared for the irradiation campaign composed of four MAX78000FTHR boards.

2.1.5 Metrics

I estimated the system failure and weight error cross-section to assess the reliability. The cross-section calculates the device's failure rate based on the fluence, given in cm^2/device. Furthermore, I calculated the Failure In Time (FIT) metric based on New York City's sea-level neutron flux (Φ_{NYC}). The FIT metric shows how many failures occur in a billion hours,

depending on the sensitivity of the device and the particle flux to which it will be exposed. The FIT is given by Equation (2.1), where σ represents the cross-section and 10^9 is a billion hours.

$$\text{FIT}_{\text{NYC}} = \sigma \times \Phi_{\text{NYC}} \times 10^9 \tag{2.1}$$

2.2 Results

During the experiment, various types of radiation-induced errors were observed. These errors had different levels of impact on the system, ranging from no noticeable effects to complete system failure. The executions in which errors were reported were classified into five categories:

- **No Impact (NI):** Executions where the corrupted data were either unused or did not affect the circuit's functionality, resulting in no observable change in the program output.
- **Major Output Disturbance (MaOD):** The execution finished, but the classification result was entirely incorrect.
- **Medium Output Disturbance (MeOD):** The execution finished, but the confidence level or percentage of the classification was incorrect, even though the predicted class itself was still correct.
- **Minor Output Disturbance (MiOD):** The execution finished, but the error caused a slight disturbance in the output of the last layer. Despite this, the final classification remained unaffected.
- **System Failure (SF):** When the error led to the program stopping prematurely or caused the entire system to crash.

Table 2.4 presents the number of images that were processed, the number of images that showed any error, and the classification of errors per board, detailing the number of errors reported for each category. It is important to note that the CNN used for the MNIST dataset, processed in *B3* and *B4*, was less complex, with fewer layers and parameters, and it had a lower latency than the one used for the CIFAR-10 in *B1* and *B2*.

The error rates observed for the MNIST dataset were notably lower compared to CIFAR-10, likely due to a combination of factors. The simpler CNN architecture used for MNIST

had lower resource demands, which reduced the critical area exposed to faults and limited fault propagation within the system. Also, the inherent characteristics of the MNIST images, such as their lower complexity and grayscale format, reduced the computational load, making the system less vulnerable to errors. These findings align with existing literature suggesting that simpler models can offer improved reliability [103–105].

Table 2.4 – Classification of errors per board.

Board	Num. of Images	Images w/ Errors	MaOD	MeOD	MiOD	NI	SF
B1	134,783	130,140	775	44,161	82,698	2506	167
B2	131,416	127,031	686	44,134	79,839	2372	220
B3	169,927	124,357	9	27	14,764	109,557	171
B4	203,574	136,881	86	9	18,123	118,663	218

2.2.1 Major, Medium, and Minor Output Disturbance

Tables 2.5 and 2.7 show the errors reported during MiOD, MeOD, and MaOD executions, respectively. Of all the images processed that reported an error (third column of Table 2.4), 97.08% of them reported only weight errors. Both configuration and bias values were updated at each layer; therefore, they were updated frequently, leading to a reduced probability of a bit upset occurrence.

Table 2.5 – Errors reported in MiOD executions.

Board	Weight	Weight and Config.	Weight and Bias	All	Unknown
B1	80,391	10	0	1	9
B2	78,088	5	1	0	6
B3	14,752	0	0	0	12
B4	18,088	3	0	0	32

Table 2.6 – Errors reported in MeOD executions.

Board	Weight	Config.	Weight and Config.	Weight and Bias	All	Unknown
B1	42,365	1	6	1	1	4
B2	42,952	0	3	0	1	4
B3	27	0	0	0	0	0
B4	9	0	0	0	0	0

Table 2.7 – Errors reported in MaOD executions.

Board	Weight	Config.	Weight and Config.
B1	763	7	3
B2	573	0	1
B3	8	0	1
B4	85	0	1

Of the few executions that reported errors in the configuration or bias values, most of these errors were not attributable to bit upset. For the configuration cases, most of the observed values were significantly different, rather than just altered by a single bit. In all the cases where a bias error was reported, it indicated that all the values were completely wrong. Therefore, most configuration and bias errors were caused by a failure in the writing process. For MeOD and MiOD executions, the unknown cases refer to instances where no errors besides the values of the output of the layers were reported, yet the final result was still impacted. These were likely caused by an error in the control of the accelerator.

In the analysis of MaOD executions, I observed that, although these executions showed weight errors, the total number was low compared to the overall weights, reaching a maximum of 490 errors for CIFAR-10 and 110 for MNIST. Given this limited number, I conducted fault injection simulations on the weights to determine if these errors alone could explain the observed misclassifications. Faults were injected exclusively into the weights, as bias errors were minimal, and most configuration errors appeared unrelated to actual upsets.

For the fault injection, I implemented a function that randomly selects a bit within a random address in the weights storage and flips it. To closely match the original experimental conditions, I ran the same flow used in the experiment, processing the same set of images from both the MNIST and CIFAR-10 datasets. The fault injection function was called once before processing each image, ensuring one injected fault per inference. Whenever a misclassification occurred, I restarted the entire execution, resetting the injected faults. I

repeated this procedure 20 times to observe the average number of weight errors required to trigger a MaOD event.

The results indicated that a large number of weight errors was needed to induce misclassification, with an average of 1316 errors for CIFAR-10 and 2628 for MNIST. The lowest number of weight errors leading to misclassification was 570 for CIFAR-10 and 824 for MNIST, though these were exceptions, with most cases closer to the average thresholds.

Given that the weight error counts observed in the experiment were significantly below the thresholds needed to induce misclassifications in the fault injection simulations, these misclassifications were likely influenced by factors other than the weight errors. During the analysis, I noticed recurring patterns in the MaOD executions, where specific types of misclassifications were repeated multiple times. For instance, 10, 12, 5, and 5 distinct patterns accounted for all misclassifications on devices *B1*, *B2*, *B3*, and *B4*, respectively. It is worth noting that these patterned occurrences ceased only after a processor reset was triggered by one of the SF occurrences. The identified patterns and their potential causes are detailed below:

- **Stuck output values:** There were cases of stuck output values where the values from the last layer of the NN unloaded from the accelerator remained the same over a sequence of images, despite the images being different. The intermediate layer values varied in some instances, indicating that the error occurred only at the final output. In other cases, the intermediate and final output values were identical, likely indicating that an error occurred during image loading. In a few cases, no errors were detected in the intermediate layers, probably caused by an event on the controller of the accelerator.

- **Sequential misclassifications:** One type of pattern involves sequential misclassifications of the same image with similar output values. The same image was misclassified multiple times in sequence in a few instances. While the classification was wrong each time, the output values varied slightly but remained similar. Since, in these cases, the error occurred for only one image (and not for the others from the same class), it is most likely that the issue lies in the variable that stores the memory pointer to the image, which results in loading the wrong values for the input memory and thus an incorrect classification.

- **Multiple misclassifications in sequence:** Finally, there was one scenario where several images were misclassified in sequence. Each one had different output values and class predictions. However, there was no report of wrong output between the layers, and only a few weight errors were reported. This was also probably caused by an event on the controller of the accelerator.

2.2.2 System Failure

During the testing phase of the microcontroller, several types of system failures were identified. Table 2.8 summarizes the frequency of these failures across different boards during testing. Hard faults occur when the microcontroller encounters a critical error, such as attempting to access invalid memory or executing an illegal instruction. Another failure type involved the WDT triggering a reset when the system failed to respond within a set time limit, typically due to software hangs or deadlocks. Additionally, there were instances where the system reset itself due to an unknown cause, categorized as an unrecognized reset, likely caused by an SEU in the control of the system. In all these cases, the system performed a soft reset.

As mentioned in Section 2.1.4, the current and the UART output were monitored during the experiment. Some failures were attributed to system hangs, identified when no output was received from the UART for over one minute, indicating that the system became unresponsive. Also, some overcurrent cases were detected. In both cases, the monitoring system triggered a power cycle on the board.

Table 2.8 – Reported cases of system failure.

Board	Hard Fault	WDT Reset	Unrecognized Reset	Overcurrent	Hang
B1	55	15	44	5	30
B2	55	28	47	17	36
B3	41	21	56	3	28
B4	63	32	49	3	51

2.2.3 Reliability Analysis

Table 2.9 summarizes the cross-sections for each type of SF for each board. I summed all the errors from each category across all boards to determine the total number of SFs. The overall cross-section (XS) was calculated to be 1.72×10^{-10} cm^2/device. Based on this, I derived a total SF FIT value of 2.24.

Table 2.10 shows the accumulated fluence for weight errors, the number of errors reported in the weight memory, the weight errors cross-section, the number of weights, the FIT, and the normalized FIT. The cross-section and FIT values for boards *B1* and *B2* were significantly higher than those for boards *B3* and *B4*, This is because the memory used by the weights in the CNNs on boards *B1* and *B2* is much larger than that of boards *B3* and *B4*, resulting in a substantially larger sensitive memory area and, consequently, a higher probability of

soft errors. Therefore, the sixth column shows the normalized weight error cross-section by dividing the cross-section by the number of weights. Since the cross-section value is used to calculate the FIT, I also estimated a normalized FIT, represented in the seventh column, using the normalized cross-section value. After the normalization of the cross-section and the FIT, it became evident that the values across all the boards were similar, which means that these events were independent of the CNN architecture and dataset.

2.3 Discussion

This chapter provides an analysis of the MAX78000, a microcontroller with a hardware CNN accelerator, offering insights into the behavior and reliability of this device. I increased system observability through layer-by-layer monitoring, enabling detailed tracking of faults at each NN layer. Additionally, I examined the types of errors caused by radiation and evaluated the reliability of the DUT.

The varying performance across the different boards tested, particularly between those processing the MNIST dataset and those handling CIFAR-10, underscores the influence of the CNN architecture on error rates. The CNN used in boards *B3* and *B4* exhibited lower error rates, likely due to their simpler architecture requiring less memory usage, memory pointers, and variables. Thus, this reduced resource usage makes them less vulnerable to errors. This finding aligns with existing literature suggesting that simpler models can offer improved reliability [103–105].

Many observed errors originated from the processor, with configuration and bias errors mostly related to memory access issues rather than direct SEUs. Additionally, the number of MaOD executions remained relatively low, which indicates that the inference process with the CNN accelerator is reasonably robust under radiation.

Table 2.9 – Summary of SF cross-sections per board.

Board	Fluence	SF XS [cm^2/Device]				
		Hard Fault	**WDT Reset**	**Unrecognized Reset**	**Overcurrent**	**Hang**
B1	1.03×10^{12}	5.33×10^{-11}	1.45×10^{-11}	4.27×10^{-11}	4.85×10^{-12}	2.91×10^{-11}
B2	9.01×10^{11}	6.11×10^{-11}	3.11×10^{-11}	5.22×10^{-11}	1.89×10^{-11}	4.00×10^{-11}
B3	9.53×10^{11}	4.30×10^{-11}	2.20×10^{-11}	5.88×10^{-11}	3.15×10^{-12}	2.94×10^{-11}
B4	1.05×10^{12}	5.97×10^{-11}	3.03×10^{-11}	4.65×10^{-11}	2.84×10^{-12}	4.84×10^{-11}

Table 2.10 – Weight errors FIT and cross-section.

Board	Fluence	Weight Errors	Weight Error XS [cm^2/Device]	Num. of Weights	Normalized XS	FIT	Normalized FIT
B1	9.43×10^{11}	14,318	1.52×10^{-8}	301,760	5.03×10^{-14}	197.30	6.54×10^{-4}
B2	7.93×10^{11}	14,258	1.80×10^{-8}	301,760	5.96×10^{-14}	233.73	7.75×10^{-4}
B3	8.77×10^{11}	3368	3.84×10^{-9}	71,148	5.40×10^{-14}	49.94	7.02×10^{-4}
B4	9.76×10^{11}	4232	4.33×10^{-9}	71,148	6.09×10^{-14}	56.35	7.92×10^{-4}

Chapter 3

Evaluation Methodology for the HARV-SoC and Vector Extension Unit

The support for the RVV on the HARV processor was initially explored during the author's undergraduate manuscript and later extended in [106]. However, that first attempt was tightly integrated within the HARV core rather than implemented as an independent unit, and only a small subset of vector instructions was supported.

In this thesis, the implementation approach was reconsidered to improve modularity, flexibility, and scalability. Thus, a Vector Extension Unit (VEU) was integrated into the HARV-SoC as an exclusive unit. The following chapters describe the design and progressive development of this unit, from the preliminary implementation to the last version, and analyze how its integration affects system performance and reliability.

This chapter describes the methodology used to evaluate both the HARV-SoC and the VEU across all versions presented in this thesis. The following Sections detail the procedures for synthesis, performance measurement, and reliability analysis.

3.1 Synthesis and Performance Evaluation

The VEU was implemented in VHDL, consistent with the rest of the HARV-SoC design, and all benchmarks were written in C, C++, or Assembly language. For vector instructions in C and C++, the RISC-V vector C intrinsics were used [107]. The applications were compiled using the GNU Compiler Collection (GCC) for RISC-V [108].

Synthesis was performed using Xilinx Vivado 2020.2 targeting the Zynq ZC7020 FPGA SoC. The results provided the number of Look-Up Tables (LUTs), Flip-Flops (FFs), the

maximum operating frequency (F_{max}), and the dynamic power (P_{dyn}) for each version of the HARV-SoC.

To evaluate the performance, the RISC-V *mcycle* register was read at the start and end of each benchmark execution. The difference between the two values represents the total cycle count, and the acceleration metric was obtained as the ratio between the cycles of the baseline HARV and those of the versions including the VEU.

3.2 Reliability Evaluation

The reliability of the HARV-SoC and the VEU was primarily analyzed through fault-injection simulations. All versions were evaluated using the same procedure, consisting of a reference execution, fault injection at the FF level, and classification of the executions. In addition, the final implementation was also tested under proton irradiation to validate the simulated results under real radiation exposure. The setup and results of this experiment are presented later in Chapter 6. The following subsections describe the methodology used in the simulations, the classification of executions, and the metrics adopted for reliability analysis.

3.2.1 Fault Injection Simulation Methodology

The goal of the fault injection simulations was to estimate the fluence required to obtain a representative number of events during an irradiation experiment. It starts by executing a golden run, i.e., executing the application without fault injections. This run provides reference parameters for comparison with the subsequent simulations, such as execution time, UART output, and all the FFs of the implementation. These indicators are also used as parameters for the fault injections. When the golden run is finished, the fault injection simulations start.

For each simulation, the script randomly selects the time and location of the injected bit flips. The injections follow the neutron-characterized FF FIT_{NYC} of 248 and a neutron flux value that varies across the evaluations.

At first, the script fetches all register signals and sets the simulation time to zero. Then, it increments the simulation time by a random value and calculates the neutron fluence from the previous simulation time until the incremented time. The script uses the cross-section and calculates the FF error rate for a given fluence. After this, it iterates through all FFs and randomly decides whether it should inject an error based on the error rate. Finally, it calculates the random injection time between the previous simulation and the incremented

time and stores this in a file with the corresponding FF signal. This flow runs in a loop, incrementing the simulation time until it reaches the set maximum simulation time.

For all the fault injection experiments presented in this thesis, four scenarios were considered. These scenarios are baseline HARV-SoC, baseline HARV-SoC with the VEU, hardened HARV-SoC, and hardened HARV-SoC with the VEU. This way, it was possible to assess the impact of the VEU on fault tolerance for each SoC version.

3.2.2 Classification of Executions

The reliability of the SoC was evaluated by analyzing the internal registers of the processor and the outputs. The executions were classified into up to six categories, when applicable. The categories are: (*i*) *correct*, when the application was executed correctly; (*ii*) *benchmark error*, when the application is executed but has incorrect results; (*iii*) *hang*, when the application does not end the execution nor produces any result; (*iv*) *uncorrectable*, when it detected an uncorrectable error; (*v*) *TFLM halted*, when one of the TFLM internal verifications failed; and (*vi*) *exception*, when an unscheduled event disrupts the program execution [109]. The considered exceptions were instruction address misaligned, instruction access fault, illegal instructions, load access fault, and store access fault.

The classification of the executions was performed using a Python script that analyzes the files containing the UART outputs of each execution. Thus, I could determine the number of executions for each category. This script also reports the average number of injected faults and the average execution time.

3.2.3 Reliability Metrics

To estimate the trade-off between performance and reliability, I analyzed the metrics Mean Work to Failure (MWTF) and the event and failure cross-sections. Based on [110], I calculated the MWTF by Equation 3.1, where a unit of work corresponds to a single benchmark execution. For the cross-sections, I considered as an event all the events reported by the HARV-SoC's event handler and a failure all the cases in which the execution was not classified as correct.

$$\text{MWTF} = \frac{\text{amount of work}}{\text{number of errors encountered}} \tag{3.1}$$

Chapter 4

Preliminary Implementation

This chapter presents the contents based on the author's publication [IE-7], which describes the first implementation of the VEU as a separate unit inside the HARV core. This preliminary design aimed to demonstrate the feasibility of integrating vector capabilities into the HARV processor and to evaluate their impact on performance, area, and reliability through fault injection simulations.

[IE-7] **C. Imianosky**, D. A. Santos, D. R. Melo, F. Viel and L. Dilillo, "Implementation and Reliability Evaluation of a RISC-V Vector Extension Unit," 2023 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Juan-Les-Pins, France, 2023, pp. 1-6, DOI: 10.1109/DFT59622.2023.10313569. © 2023 IEEE

The architecture was based on a simplified version of the RVV, focused on functional validation and hardware exploration instead of following the complete specification. Some architectural parameters were fixed during synthesis to simplify the design. For instance, instead of using a dynamically configurable LMUL, the design used a static parameter called *VEU width*, which defined how many vector registers were grouped to form wider vectors. While the static configuration did not alter the level of parallelism, it specified how many elements each instruction could handle. This design enabled the analysis of vectorization trade-offs in a design with limited resources.

The remainder of this Chapter is organized as follows. Section 4.1 details the preliminary VEU design. Section 4.2 presents the experimental setup, and Section 4.3 details the obtained results. A discussion is presented in Section 4.3.4 to summarize the Chapter.

4.1 Vector Extension Unit

The first VEU implementation is represented in 4.1 and comprises the Vector Register File (VRF), the interface, vector elements counter, and vector execution unit. To enable HARV's support for the VEU, I first modified two core components: the instruction decoder and the execution unit.

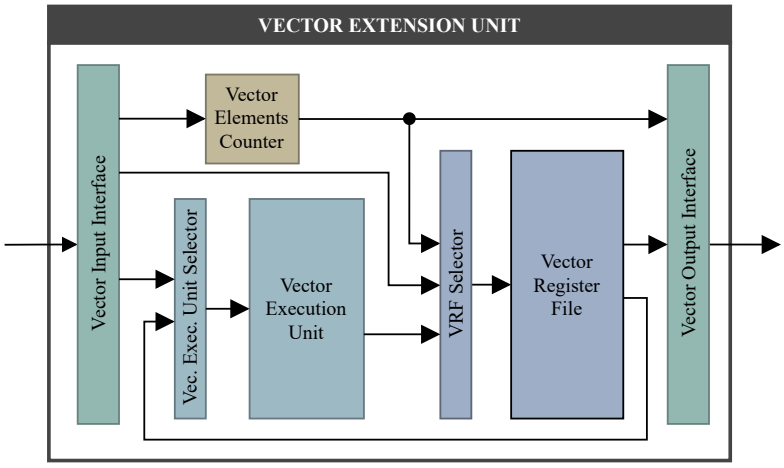


Figure 4.1 – Vector Extension Unit.

The HARV control unit integrates the instruction decoder. The instruction bits 6 to 0 are the input to a combinational logic component that determines the instruction format, according to the RISC-V ISA specification [30]. Afterward, the outputs of the control component are either enabled (1) or disabled (0) according to the instruction's operation. Despite the VEU being implemented separately from the HARV core, in this first implementation, the instruction decode of vector instructions was still performed by the HARV's control unit. To support a reduced set of RVV instructions, I extended the decoder to recognize the three vector instruction formats defined in the RVV specification: LOAD-FP, STORE-FP, and OP-V.

In the HARV microarchitecture [18], the main and ALU control units were merged as a single component to simplify the implementation. I also extended it to control the vector execution unit. Additionally, the HARV processor reads the source register during the decoding stage and performs the same operation for the VRF.

The HARV's execution unit performs the arithmetic and logic operations. These operations include add, subtraction, logical shift (left or right), arithmetic shift (right only), set to 1 (on less than), AND, OR, and XOR. In this first implementation, I decided to use HARV's execution unit as a complement to the VEU. To support the vector instructions, this unit was extended to operate on 64-bit operands, eventually processing some vector elements.

The Interface Units are responsible for directing the inputs and outputs of the VEU. The inputs are signals from the controller, the addresses to read and write from the VRF, the result from the external ALU, and the values from the vector CSRs. The outputs are one signal indicating whether all the necessary elements were processed, according to the value from the elements counter unit, and two arrays of data that serve either for memory writing or as input for the external ALU, if required.

The VRF contains 32 registers similar to those of the scalar register file, but dedicated exclusively to the vector unit, and it supports two reads and one write per cycle. In this first implementation, each vector register had a physical width of 32 bits. To enable operations on wider elements, the hardware allowed adjacent registers to be grouped, forming logical vectors whose width was determined by a static synthesis parameter called *VEU width*. This parameter had a similar purpose to the LMUL parameter in the RVV, but its configuration was fixed at design time rather than dynamically set by software.

The output width of the VRF is defined according to the *VEU width* specified at synthesis time, allowing all vector elements to be read simultaneously. The data input path of the VRF was set to 64 bits, corresponding to the maximum element size supported. When operating on 64-bit elements, each element was internally distributed across two 32-bit registers, allowing wider operations without increasing the physical register width.

The vector execution unit was implemented with four 8-bit simplified ALUs that operate on vector elements and are designed explicitly for arithmetic vector instructions. The simplified ALUs had carry-in and carry-out signals, which were multiplexed to enable simultaneous operations for two 16-bit or one 32-bit element. A diagram of this unit is shown in Figure 4.2. In the case of 32- and 64-bit vector elements, the external HARV ALU was also used to operate on one element. With this architecture, it is possible to operate on one 64-bit element, two 32-bit elements, four 16-bit elements, or eight 8-bit elements simultaneously.

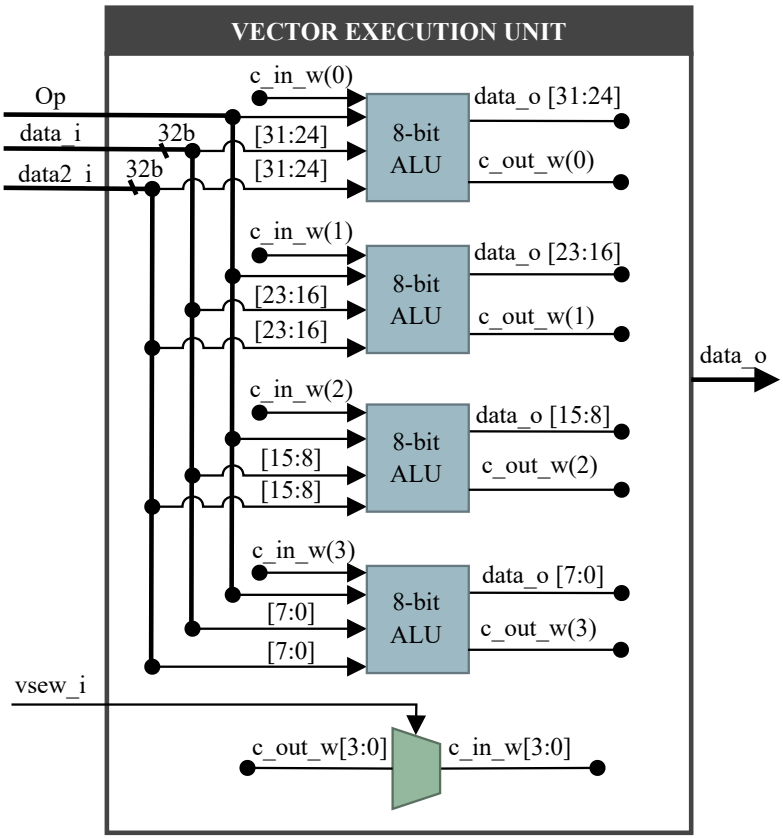


Figure 4.2 – Vector Execution Unit.

The primary goal of this implementation was to support simple vector instructions to optimize application processing and assess the behavior of the HARV with the VEU in fault injection simulations. Therefore, the subset of instructions was limited to unit-stride memory access (word, half-word, and byte) and a limited subset of integer arithmetic operations.

4.2 Experimental Setup

To estimate the performance gain, I implemented an Assembly algorithm that reads two vectors from memory, adds each of their elements, and stores the result in memory. The maximum synthesized vector width was set to 256 bits. In the experiments, the *VEU width* was varied between 32 and 256 bits, while the element width ranged from 8 to 64 bits. The algorithm was executed using the baseline HARV and the HARV with VEU.

For the reliability evaluation, I implemented a benchmark algorithm in both C and Assembly languages for the vector instructions. This algorithm adds two vectors with 8192 16-bit elements each. I executed the algorithms with the baseline processor (RV32I) and with the vector extension (RV32IV). Without the VEU, each vector element is read and

operated sequentially in a loop. For the execution with the VEU, the vectors are subdivided into 256-bit vectors of 16-bit elements and operated using OP-V instructions. Thus, the VEU reads the two 256-bit vectors from the data memory, adds the elements, and writes the result to the memory. I ran 1024 fault injection experiments with the described benchmark for each of the four scenarios described in Chapter 3.2.1.

4.3 Results

The following subsections present the results of the VEU implementation. First, I describe the synthesis results, such as the number of FFs and LUTs, maximum operating frequency, and dynamic power dissipation. After, I show the acceleration obtained in comparison to the baseline processor. Finally, I show the results of the fault injection simulations.

4.3.1 Synthesis

The plot in Figure 4.3 reports the number of LUTs and FFs for each implementation. The plot shows the resource usage of the baseline HARV-SoC and the versions synthesized with different *VEU widths* (32, 64, 128, and 256 bits).

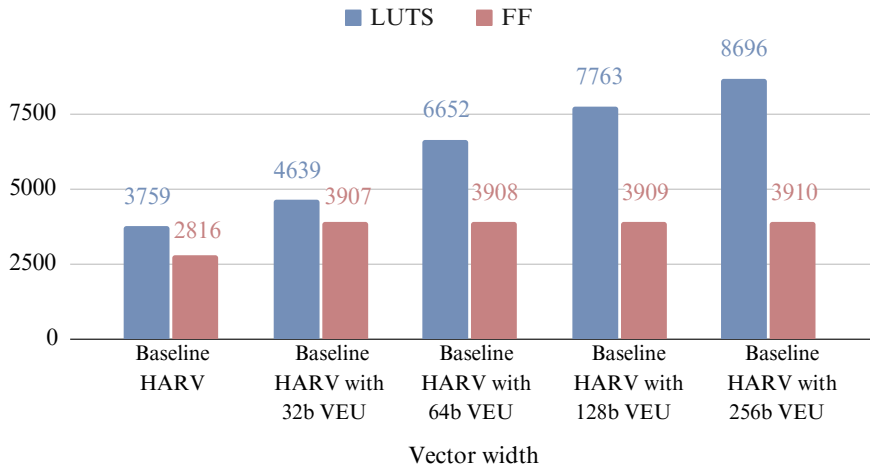


Figure 4.3 – Resources utilization for the baseline HARV-SoC and the HARV-SoC with different VEU widths.

The number of the LUTs and FFs increases according to the *VEU width*. The baseline processor needs 3759 LUTs, whereas the version with the 32-, 64-, 128-, and 256-bit VEU requires 1.23, 1.77, 2.07, and 2.31 times the amount of LUTs, respectively. For FFs, it

requires 2816, and the versions with the 32-, 64-, 128-, and 256-bit VEU require 1.39 times more LUTs than baseline HARV-SoC.

For the hardened versions of the implementations, the number of LUTs and FFs is represented in Figure 4.4. In the hardened versions, the baseline processor needs 6172 LUTs, whereas the version with the 32-, 64-, 128-, and 256-bit VEU requires 1.16, 1.67, 1.85, and 2.01 times more LUTs, respectively. The number of necessary FFs is 3864 and increases 1.28 times for all the versions with different sizes of VEU.

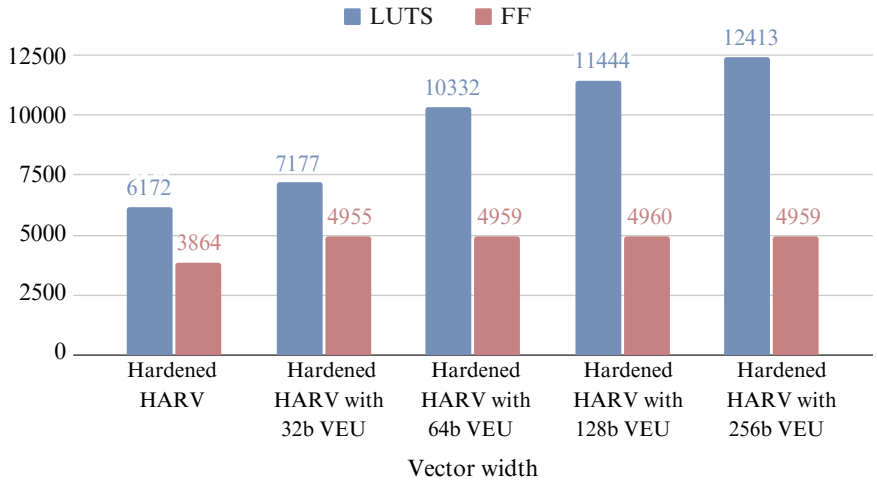


Figure 4.4 – Resource usage for the hardened HARV-SoC and the HARV-SoC with different *VEU widths*.

Table 4.1 provides an overview of the values for the (F_{max}) and (P_{dyn}). The HARV-SoC achieves a F_{max} of 57.9 MHz and consumes 129 mW of dynamic power for the baseline scenario. The addition of the 32-bit VEU exhibits a slightly lower F_{max} of 56.8 MHz while consuming 131 mW of power. The 64-bit VEU reduces the F_{max} to 49.2 MHz and increases the power consumption to 146 mW. Similarly, the 128-bit VEU shows a decrease in F_{max} to 47.8 MHz while maintaining the same power consumption of 146 mW. Lastly, the 256-bit VEU causes a slight improvement in F_{max} to 50.3 MHz with a power consumption of 140 mW.

Considering the hardened scenarios, the HARV-SoC achieves a F_{max} of 38.8 MHz and dissipates 158 mW. With 32-bit VEU width, the F_{max} decreases to 34.1 MHz while dissipating 188 mW. The 64-bit *VEU width* reduces the F_{max} to 34.3 MHz and increases the P_{dyn} to 188 mW. Similarly, the 128-bit *VEU width* decreases the F_{max} to 33.7 MHz with a P_{dyn} of 183 mW. Lastly, the 256-bit VEU has a F_{max} of 34.8 MHz with a P_{dyn} of 187 mW.

Table 4.1 – Maximum operating frequency and dynamic power dissipation results.

HARV-SoC	VEU Width	F_{max} (MHz)	P_{dyn} (mW)
Baseline	-	57.9	129
	32 bits	56.8	131
	64 bits	49.2	146
	128 bits	47.8	146
	256 bits	50.3	140
Hardened	-	35.8	158
	32 bits	34.1	165
	64 bits	34.3	188
	128 bits	33.7	183
	256 bits	34.8	187

4.3.2 Performance

The plot in Figure 4.5 presents the acceleration values achieved through the execution of vector operations compared to scalar operations. It shows the performance improvements obtained through different vector configurations, varying between 32-, 64-, 128-, and 256-bit vectors, along with 8-, 16-, 32-, and 64-bit elements.

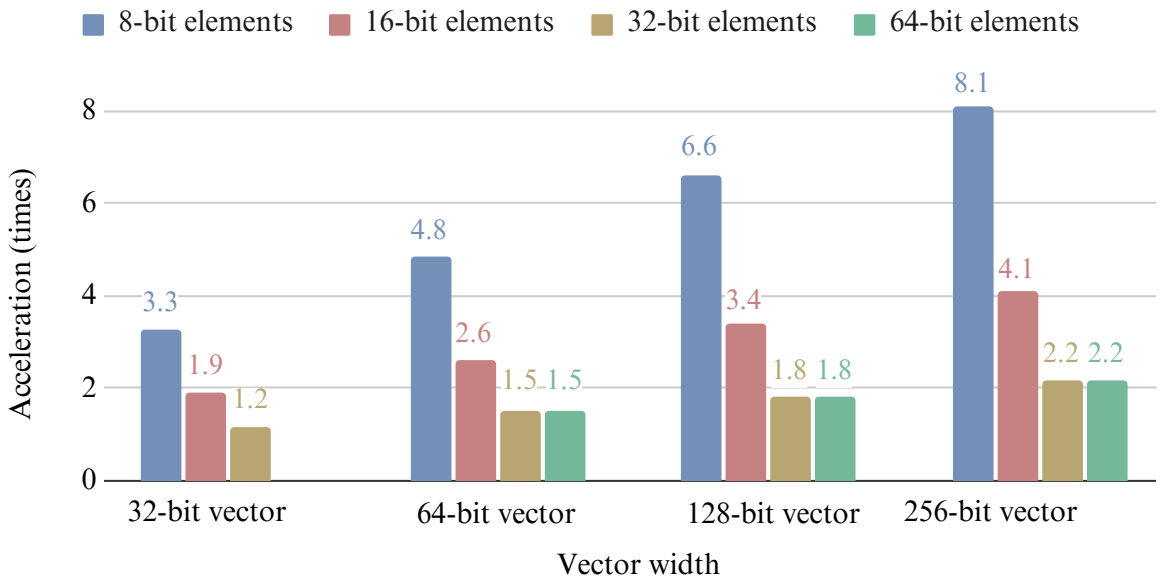


Figure 4.5 – Execution acceleration for different *VEU widths*.

For the scalar execution, the 64-bit elements are treated as pairs of 32-bit operations, due to the core architecture and its register width, making the execution time in this case similar to the two 32-bit elements. With the VEU, it is possible to operate in just one 64-bit element per instruction, and for 32-bit elements, it is possible to operate in two elements simultaneously. Thus, due to the architecture of the implementation, the acceleration values from scalar to vector executions for 32-bit and 64-bit elements are similar.

The acceleration for all synthesized *VEU widths* was more significant for 8-bit elements and decreased as the element width increased. For 32-bit vectors, an element width of 8 bits results in a speedup of 3.3 times and increases 1.5 times for 64-bit vectors. The 128-bit vector shows an increase of 1.8 times compared to the 64-bit vector. For the 256-bit, it has an acceleration increase of 1.5 times compared to the 128-bit vector.

The 16-bit elements show an increase of approximately 1.7 times as the size of the vector increases. For 32- and 64-bit elements, the acceleration is similar in all cases. It is worth noting that 32-bit vectors have no executions with 64-bit elements. The acceleration increase is 0.3 timeThe acceleration increase is 0.3 times from the 64-bit to the 32-bit vectors and the 128-bit to 64-bit.ce in acceleration is 0.4 times..4 times.

4.3.3 Fault Injection Simulation

The fault injection results are presented in Table 4.2, which compares the results of the baseline scalar and baseline vector approaches and the hardened scalar and hardened vector approaches. Thus, it presents the number of correct executions, benchmark errors, hangs, and exception executions achieved by each approach for the 1024 executions.

In the baseline processor, the vector approach showed an increase of 40 correct executions. Within the hardened category, the vector approach improved with 15 more correct executions. The scalar approach had one fewer execution with benchmark errors than the vector in the baseline processor. Meanwhile, the hardened scenario presented one more benchmark error than the vector. For the hang executions, the vector approach showed an improvement of 25 executions in the baseline scenario. In the hardened design, the vector approach has a difference of 14 executions, where it did not show any case of hang executions. Regarding the exceptions, the baseline processor with the vector approach had 16 instances fewer than the scalar. The hardened scalar scenario had two more exceptions than the vector scenario.

Table 4.2 – Fault injection executions.

Execution classification	Baseline		Hardened	
	Scalar	Vector	Scalar	Vector
Correct	962	1002	1002	1017
Benchmark error	6	7	3	4
Hang	28	3	14	0
Exception	28	12	5	3

Table 4.3 compares the average number of fault injections and execution times of the baseline and hardened scenarios for both scalar and vector operations to assess the impact of the execution time on the number of injected faults. In both baseline and hardened systems, the average execution time of the vector approach was approximately half of the scalar one. The difference in the execution time directly impacts the number of injections since the longer it takes to execute, the more faults are likely to be injected. In the baseline scenarios, the scalar execution had an average of 2.1 fault injections, while the vector had just one. In the hardened scenarios, the scalar operation had an average of 4.44 injected faults, and the vector operation had 2.05.

Table 4.3 – Injections analysis.

Metric	Baseline		Hardened	
	Scalar	Vector	Scalar	Vector
Average execution time (ms)	99	48	147	70
Average number of injections	2.1	1.0	4.44	2.05

4.3.4 Discussion

The synthesis and simulation results demonstrate that the inclusion of the VEU increases hardware resource utilization compared to the baseline HARV-SoC, with LUTs and FFs growing proportionally to the configured *VEU width*. This parameter directly controls the maximum vector size processed by the unit, since wider configurations require additional multiplexing logic in the VRF and wider data paths. As expected, increasing the *VEU width* slightly reduced the maximum operating frequency and raised dynamic power consumption. Despite these overheads, the static parameterization of the *VEU width* provided an effective way to explore design scalability without adding control complexity. Nevertheless, fixing

this parameter at synthesis limits flexibility, as the hardware cannot adapt the vector width at runtime. Thus, a dynamic configuration, with the LMUL parameter defined in the RVV specification, would offer greater versatility by allowing runtime adjustment of vector grouping according to application needs.

Regarding performance, the implementation achieved significant acceleration compared to scalar execution. By exploiting DLP, the VEU reduced execution latency by up to 8.1 times, particularly for narrow data widths (8-bit elements), where the benefits of parallel computation are more evident. Also, as the *VEU width* increased, the acceleration improved, since larger *VEU widths* allow more elements to be processed per instruction. These results confirm that even a simplified vector unit can provide substantial gains for computationally intensive applications executed on low-power processors.

In terms of reliability, fault injection simulations showed that the use of the VEU decreases the average number of injected faults and improves the rate of correct executions. This improvement is mainly attributed to the shorter execution times, which reduce the system's exposure window to transient faults. Consequently, increasing the *VEU width* not only enhances performance but also indirectly contributes to dependability, as faster execution lessens the probability of faults affecting critical computations.

Overall, this preliminary implementation validated the feasibility of integrating a vector unit into the HARV processor. The static parameterization of the *VEU width* enabled exploring the trade-offs between performance, area, and reliability, providing essential insights for subsequent refinements and for the implementation of a standard-compliant vector architecture described in the next chapter.

Chapter 5

Enhanced Implementation of the Vector Extension Unit

Building on the preliminary implementation presented in Chapter 4, an enhanced version of the VEU was implemented. The improved design was restructured to align with the RISC-V Zve32x subset specification, and the supported subset of instructions was extended to include vector multiplication. Thus, the support for the LMUL parameter was implemented, enabling runtime adjustment of the VL, instead of using a fixed synthesis parameter as in the preliminary version. Lastly, the VEU's organization was redesigned, and a vector control unit was included, improving modularity and simplifying signal synchronization between the VEU and main control unit. This chapter details the enhanced VEU and its evaluation, based on the author's publication [IE-6].

[IE-6] **C. Imianosky**, D. A. Santos, D. R. Melo, F. Viel and L. Dilillo, "Special Session: Reliability and Performance Evaluation of a RISC-V Vector Extension Unit for Vector Multiplication," 2024 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Didcot, United Kingdom, 2024, pp. 1-6, DOI: 10.1109/DFT63277.2024.10753524. © 2024 IEEE

The modifications on the VEU aimed to increase flexibility and offer a more realistic representation of the RVV for embedded processors. To evaluate the impact of these improvements, the VEU was analyzed in terms of synthesis, performance, and reliability through fault injection simulations.

The following sections are organized as follows. Section 5.1 details the enhanced VEU architecture, followed by the experimental setup in Section 5.2 and the obtained results in Section 5.3. Finally, Section 5.3.4 concludes with a discussion of the main findings.

5.1 Enhanced Vector Extension Unit

The enhanced VEU focused on the Zve32x extension, so the ELEN and VLEN are set to 32 bits. Thus, the SEW can be defined to 8, 16, and 32 bits, and the registers in the VRF are 32-bit wide. Furthermore, the supported subset of vector extensions was extended. Thus, the VEU supports: (i) all vector configuration instructions, (ii) unit-stride load and store instructions, (iii) integer arithmetic operations (*addition*, *subtraction*, *multiplication*, and bitwise operations).

Another modification in the VEU was the support for setting the LMUL parameter, allowing up to 256-bit vectors, when LMUL is set to 8. Therefore, it is possible to operate on vectors up to the same width as the preliminary implementation, but with reduced overhead of logic resources.

The internal structure of the VEU was redesigned, and its organization is illustrated in Figure 5.1. The VRF contains 32 registers of 32 bits each, consistent with the Zve32x specification. When LMUL is greater than 1, adjacent registers are dynamically grouped to form longer logical vectors, up to 256 bits (when LMUL is set to 8). This mechanism replaces the static *VEU width* parameter from the previous version, making the design more flexible during execution.

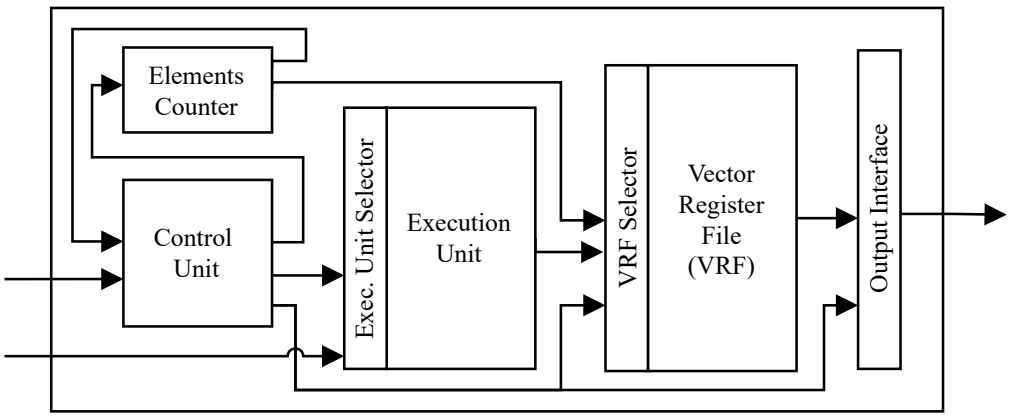


Figure 5.1 – Vector Extension Unit.

The execution unit, represented in Figure 5.2, was also adjusted and is composed of one 32-bit, one 16-bit, and two 8-bit simplified ALUs. This way, it can perform one 32-bit, two 16-bit, or four 8-bit operations simultaneously, depending on the SEW configuration. Therefore, the level of parallelization of operations depends on the SEW value. The inputs and outputs are mapped to their respective ALUs according to the SEW value. The elements that are being operated on by wider ALUs are zero-extended. The simplified ALUs support

only vector operations, and are limited to vector integer addition, subtraction, multiplication, and bitwise operations.

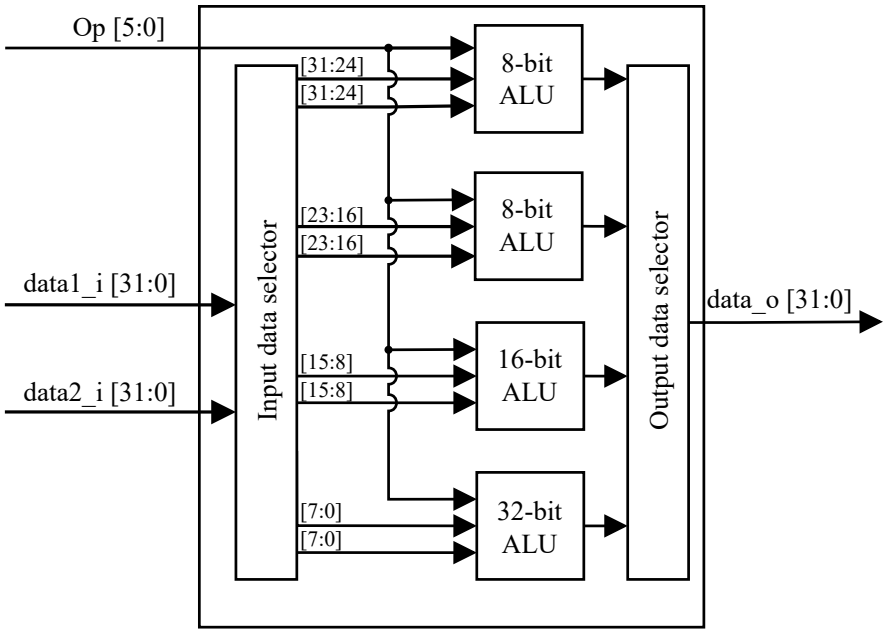


Figure 5.2 – Vector Execution Unit.

Another modification is the introduction of a dedicated vector control unit that replaces the dependence on the main control unit for decoding and managing vector operations. This separation simplifies synchronization between the main core and the vector unit and allows the vector unit to operate more autonomously. Following the same design logic as the HARV microarchitecture, the vector ALUs are controlled directly by the vector control unit to simplify the implementation.

The vector control unit was implemented using the Finite State Machine (FSM) illustrated in Figure 5.3. The states are Idle, Decode, Run (instruction execution), Register, and DMemStall (data memory stall). As mentioned in Chapter 4, the main control unit of the core integrates the instruction decoder and was extended with an output signal, *v_inst*, to indicate when the fetched instruction is a vector operation. The vector control unit starts in the Idle state, waiting for the *v_inst* to be asserted. When this occurs, the main control remains in its run state, and the vector control advances to the Decode state. Once the vector operation is complete, the vector control asserts `v_done`, allowing the main control to resume the execution.

In case of a configuration instruction (`vsetvl`, `vsetvli`, or `vsetivli`), it goes directly from Decode to Register, enabling signals that indicate the writing of new values into the

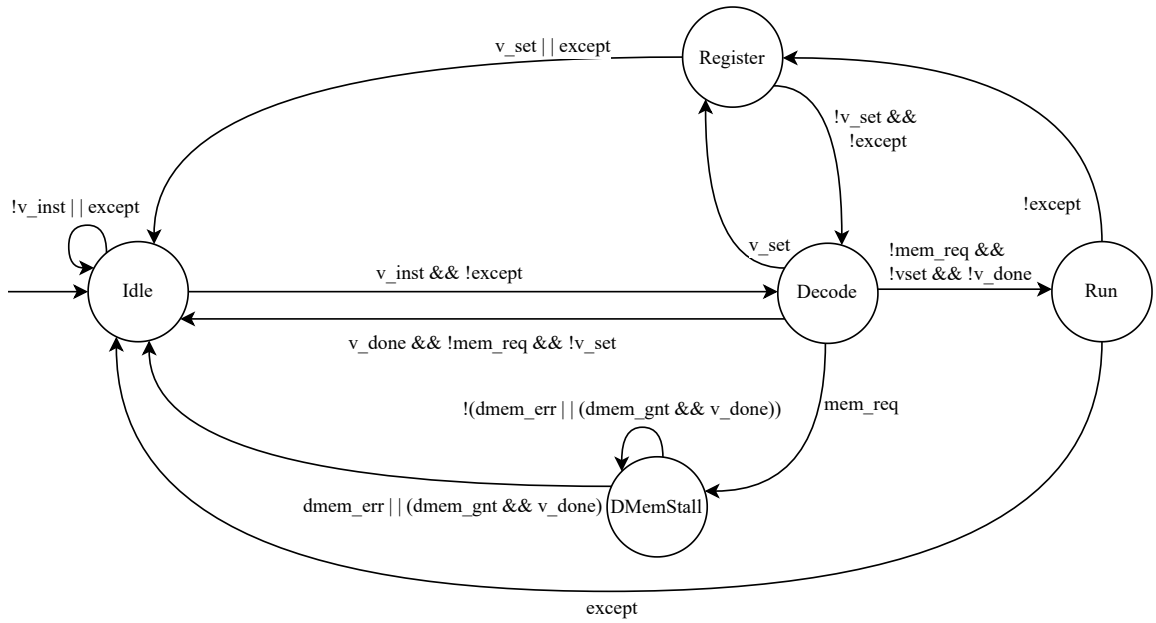


Figure 5.3 – State machine of the vector control unit.

CSRs. After this step, it returns to the Idle state. The SEW and LMUL values are extracted from the `vtype` field and updated accordingly.

If the instruction corresponds to a vector load or store, the FSM advances to DMemStall, asserting the request signal for the data memory and waiting for a grant. It repeats this stage until *v_done* is asserted or an error occurs in the memory access, after which it returns to the Idle state. For arithmetic instructions, the FSM goes from Decode to Run to execute the operation, then advances to Register to store the results into the VRF. After Register, it returns to Decode, where it verifies if *v_done* is asserted or if another set of vector elements should be processed. If an exception occurs during Run or Register, the FSM immediately returns to Idle.

The *v_done* signal is asserted by the vector control unit using the value stored in the elements counter that holds the number of vector elements already processed and comparing it to the VL. The control unit also defines when and by how much the elements counter should be incremented.

5.2 Experimental Setup

As a benchmark, an algorithm was implemented to read elements from two vectors, multiply them, and store the results in memory. The benchmark algorithm was implemented in both scalar and vector versions using the C language. Additionally, scalar and vector

implementations were developed in assembly language to maximize performance through low-level optimizations.

The same benchmark algorithms were used to evaluate the performance and reliability. However, the number and size of elements in the vectors varied. The following subsections provide details on each of the scenarios.

Performance evaluation

For the performance evaluation, I executed the benchmark algorithms to evaluate the performance variation using the baseline HARV and the HARV with VEU, comparing both the C and assembly versions of the benchmark algorithm. This comparison allowed an assessment of the impact of C and intrinsic functions on performance across different configurations. The performance variation was calculated relative to the scalar C version of the benchmark algorithm to evaluate the impact of vector instructions and Assembly-level optimizations on performance.

In all scenarios of the performance evaluation, the number of elements in each vector was set to 256. In the baseline HARV scenario, I varied the elements' width between 8, 16, and 32 bits. For the executions with the VEU, I varied the LMUL parameter between 1, 2, 4, and 8 and the SEW between 8, 16, and 32 bits, with the VL parameter always being set to VLMAX.

Reliability evaluation

For the fault injection, I ran 1024 experiments for each of the four scenarios described in Chapter 3.2.1. I used the C language version of the benchmark algorithm for reliability evaluation, as it is more commonly used in practical applications. The number of elements in each vector was set to 16384. In the executions with the VEU, I set the LMUL parameter to 8 and the VL to VLMAX, aiming to exploit maximum parallelism. I varied the SEW between the best and worst parallelization levels, i.e., 8 and 32 bits, to investigate how different SEWs affect reliability. Thus, I also used 8- and 32-bit elements in the scalar scenarios.

5.3 Results

The following subsections introduce the results of the VEU implementation. First, I present the synthesis results, followed by an analysis of the performance achieved compared to the baseline processor. Finally, I show the results of the fault injection simulations.

5.3.1 Synthesis

Table 5.1 shows the resource usage for the HARV-SoC and the HARV-SoC with the VEU in both baseline and hardened scenarios. The table presents the number of FFs, LUTs, Digital Signal Processings (DSPs), maximum operating frequency, and dynamic power dissipation for each implementation.

Table 5.1 – Synthesis results.

HARV-SoC	LUTs	FFs	DSP Blocks	F_{max} (MHz)	P_{dyn} (mW)
Baseline	3,748	3,583	4	65.40	138
Baseline with VEU	5,399	4,170	9	64.40	140
Hardened	5,802	4,381	12	41.91	164
Hardened with VEU	7,555	4,968	17	40.77	175

In the baseline scenario, adding the VEU increases the number of LUTs by 44%. The number of FFs increases by 16%, mainly because of the VRF, and the number of DSP blocks increases due to the vector execution unit. Using fault tolerance techniques increases the number of logical resources the processor uses. The number of DSP blocks in the hardened scenario increased 3 times compared to the baseline because of the TMR implemented in the ALU. Adding the VEU to the hardened system increases LUT usage by 30% and FFs by 13%.

The maximum operating frequency of the processor decreases as the critical path increases. In the baseline implementation, the SoC can operate at a maximum of 65.40 MHz, and it drops 35% when applying the hardening techniques. Adding the VEU slightly decreases F_{max} by 1.5% in both baseline and hardened scenarios. The addition of the VEU also results in a moderate increase in P_{dyn} by 1.4% and 6.7% in the baseline and hardened scenarios, respectively.

5.3.2 Performance

The plot in Figure 5.4 presents the variation in the performance achieved by different configurations of the benchmark algorithm relative to the scalar C version. In cases where the performance factor is less than 1, the performance is worse than the scalar C.

The scalar Assembly implementation shows an acceleration between 2.46 and 2.81 times faster than scalar C across different configurations due to the manual optimization in Assembly, simplifying code execution. The vector Assembly achieves the highest speedup

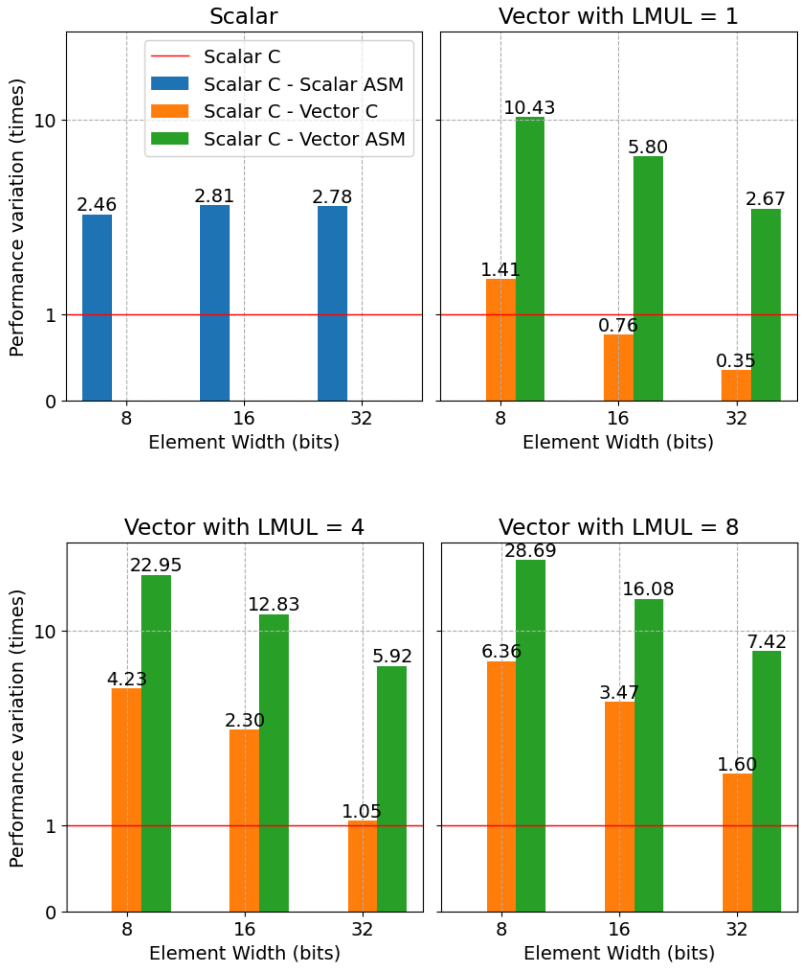


Figure 5.4 – Performance variation for different versions of the benchmark algorithm in comparison to the scalar C.

among all configurations, varying from 10.43 to 28.69 times faster than scalar C, since it benefits from the parallelization and optimized assembly coding. The LMUL parameter significantly influences the performance since it affects the number of elements that a single instruction will operate on. Also, smaller SEW values explore greater parallelism, resulting in higher performance.

The scenario of vector C, using intrinsic functions, shows performance variations ranging from 0.35 to 6.36 times compared to scalar C. The cases where the VEU performed worse than the scalar C occur because, unlike in the vector Assembly, in which it executes the configuration instructions (setting the VL, LMUL, and SEW) only once at the beginning of the execution, the intrinsic functions run them every time before each operation, introducing repetitive overhead. Thus, vector C can reduce the performance compared to scalar C

when the parallelization provided by vector operations is not enough to offset the repeated configuration overhead.

5.3.3 Fault Injection Simulation

I evaluated the output of the UART of each execution by using the Python script and the five categories of results mentioned previously in Chapter 3. The experimental results are presented in Table 5.2, which compares the results for different configurations of the HARV-SoC and HARV-SoC with the VEU. Thus, it presents the number of executions of each category achieved by each approach for the 1024 executions. Also, it presents the average number of fault injections, the total simulated fluence, the failure cross-section, and the MWTF for each scenario.

The failure cross-section gives an indication of the susceptibility of devices to radiation-induced faults since lower cross-section values mean lower vulnerability to faults. The results show an increase in the failure cross-section values in the scenarios with the VEU since it increases the sensitive area. However, in the cases for the execution with the VEU and SEW = 8, the MWTF was higher. This is because the executions with higher LMUL and smaller SEW values apply more parallelism and, therefore, have a shorter execution time. The difference in the execution time directly impacts the number of fault injections since the longer the execution lasts, the more faults are likely to be injected.

In the case of SEW = 32, the amount of provided parallelism does not compensate for the increase in the sensitive area, resulting in fewer correct executions. In the hardened scenario, the difference in correct executions with and without the VEU is larger due to the application of hardening techniques, especially in the ALU and the register file, which are not applied for the vector ALUs and the VRF. Thus, even though fewer faults were injected, the MWTF is smaller.

Most of the benchmark errors in the scenarios with the VEU were caused by fault injections in the VRF. Unlike the main register file of the core, the VRF does not apply fault tolerance techniques due to the hardware overhead that would be required. Thus, the difference in the number of benchmark error executions between the vector and scalar executions is greater in the hardened scenario than in the baseline.

Table 5.2 – Fault injection executions.

Execution classification	Baseline				Hardened			
	Scalar 8b	**Vector 8b**	**Scalar 32b**	**Vector 32b**	**Scalar 8b**	**Vector 8b**	**Scalar 32b**	**Vector 32b**
Correct	722	849	699	694	913	934	911	839
Benchmark error	32	27	54	101	15	31	15	84
Hang	7	2	5	1	1	0	0	0
Uncorrectable	198	109	196	145	89	49	92	60
Exception	65	37	70	83	6	10	6	41
Reliability analysis								
Average # of injections	18.70	11.11	18.73	19.36	21.98	13.63	21.52	19.44
Simulated fluence*	$1.84 \cdot 10^{14}$	$8.92 \cdot 10^{13}$	$1.78 \cdot 10^{14}$	$1.49 \cdot 10^{14}$	$1.74 \cdot 10^{14}$	$9.20 \cdot 10^{13}$	$1.72 \cdot 10^{14}$	$1.31 \cdot 10^{14}$
Failure XS†	$1.61 \cdot 10^{-12}$	$1.94 \cdot 10^{-12}$	$1.80 \cdot 10^{-12}$	$2.22 \cdot 10^{-12}$	$6.31 \cdot 10^{-13}$	$9.79 \cdot 10^{-13}$	$6.59 \cdot 10^{-13}$	$1.41 \cdot 10^{-12}$
MWTF	3.47	5.92	3.20	3.11	9.31	11.38	9.06	5.54

*[n/cm^2]; †[cm^2/device]

5.3.4 Discussion

The enhanced VEU improved both architectural flexibility and execution efficiency compared to the preliminary version. The introduction of a dedicated vector control unit simplified synchronization with the main core, and the implementation of the LMUL parameter replaced the fixed synthesis parameter of the previous design, enabling runtime configuration of the VL. These changes aligned the architecture with the *Zve32x* subset of the RVV.

Although adding the VEU to the HARV-SoC introduces hardware, frequency, and power consumption overhead, it can significantly increase performance and reliability. The vector Assembly application achieved an execution up to 28.69 times faster than scalar C. Also, for the C vectorized application, the maximum acceleration was 6.36 times in comparison to the scalar C.

Smaller SEWs and larger LMULs increase parallelism, while combinations with reduced parallelism (e.g., low LMUL and high SEW) lead to limited acceleration or even slower executions. Therefore, I noticed that for VEU configurations where less parallelism is being applied, the use of the C version of the benchmark was not worth it since the parallelization provided by the VEU does not compensate for the repeated instruction overhead introduced by the compiler. The impact of this overhead is also present in the reliability evaluation, where for an SEW of 8 bits, the VEU enhances the reliability, and for 32 bits, it worsens it.

However, achieving these advantages depends on selecting the appropriate parameters, such as LMUL and SEW. Incorrect parameter choices can lead to degraded performance and reliability rather than improvement. For instance, in the hardened scenario with the VEU, using 8-bit elements increased the number of correct executions by 2.3%, but using 32-bit elements decreased it by 7.9% compared to the hardened scenario without the VEU.

Overall, the enhanced implementation combined flexibility and efficiency while keeping a moderate hardware cost. It also made the design consistent with the RVV specification and provided a better foundation for future improvements. These results showed that vector acceleration was a practical approach for low-power fault-tolerant processors. Moreover, the addition of vector multiplication support made the VEU particularly relevant for AI applications, especially for CNNs that rely on MAC operations with 8-bit data that could benefit from higher parallelism.

Chapter 6

Vectorized Neural Network Inference

The previous chapter demonstrated that the vector extension improved both performance and reliability for arithmetic operations, suggesting its potential for embedded AI applications. Building on those results, this chapter explores the use of the VEU to accelerate CNN inference on the HARV-SoC. The presented contents are based on the author's publications [IE-1] and [IE-2], but also contain unpublished work and additional details.

[IE-1] **C. Imianosky**, D. A. Santos, W. Grignani, and L. Dilillo, "Reliability Analysis of Neural Network Inference on a Fault-Tolerant RISC-V SoC with Vector Extension". IEEE European Conference on Radiation and Its Effects on Components and Systems (RADECS), 2025.

[IE-2] **C. Imianosky**, D. A. Santos, and L. Dilillo, "Efficient TinyML Inference on a Fault-Tolerant RISC-V SoC with Vector Extension". 10th IEEE International Workshop on Advances in Sensors and Interfaces, July. 2025, Manfredonia, Italy.

I ported the TFLM to the HARV-SoC to support AI inference. As mentioned in Chapter 1, several works focused on the use of CNNs for space application, and the convolution operation, the most computationally demanding operation in CNNs, is highly parallelizable. Therefore, a vectorized version of the TFLM's convolution kernel was implemented.

To further improve efficiency, I extended the VEU with support for widening instructions, allowing direct operations on 8-bit data with 16-bit accumulation to avoid overflow. These modifications enabled the execution of quantized CNNs with increased parallelism and reduced latency.

The proposed implementation was evaluated in terms of synthesis, performance, and reliability. The reliability was analyzed by fault injection simulations and validated by proton irradiation experiments.

The following sections are organized as follows. Section 6.1 describes the integration of TFLM and the vectorized convolution kernel. Section 6.2 presents the experimental setup, including the CNN benchmark, the fault injection simulations, and the irradiation campaign. Section 6.3 presents the synthesis, performance, and reliability results. Section 6.4 discusses the main findings.

6.1 TensorFlow Lite Micro Support and Vector Extension Unit Modifications

I ported the TFLM to the HARV-SoC as a static library, following the standard porting steps. The source tree was generated using the official *create_tflm_tree.py* script, which extracts the required files from the TFLM repository and places them in a minimal directory (*tflm-tree/*) containing only the sources and third-party dependencies necessary for embedded use.

During the build of any HARV-SoC application, the Makefile compiles the *tflm-tree/* into a static library (*libtflm.a*) using the same RISC-V cross-compilation flags as the rest of the system. The library is then linked with the application, providing a consistent TFLM integration and enabling kernel-level customization.

The TFLM's convolution kernel is fully scalar and relies on nested loops over the batch size, output spatial dimensions, output channels, filter dimensions, and input channels. The kernel performs scalar MAC operations for each output element, over all contributing input values and filter weights. After the accumulation, it applies bias, quantization, and activation to generate the final output. While functional, this structure does not exploit hardware-level parallelism. Thus, I redesigned this function using RVV instructions to leverage the VEU in HARV-SoC, enabling multiple elements to be processed simultaneously through DLP.

6.1.1 Vectorized Convolution Kernel

I reorganized the loop structure of the convolution kernel and replaced the innermost scalar operations with vector instructions. I implemented two vectorization strategies, which are applied according to the number of input channels, due to the data memory layout. Figure 6.1 shows the difference in the memory layout, where in the single-channel case, adjacent pixels in a row are stored sequentially in memory, and in the multi-channel case, the channels of each pixel are stored contiguously.

Since the data is 8-bit quantized, both vectorization strategies rely on 8-bit memory access. However, arithmetic operations are performed with 16-bit precision to prevent overflow. For this, the input and filter values must be widened to 16-bit, i.e., converted from 8-bit to 16-bit

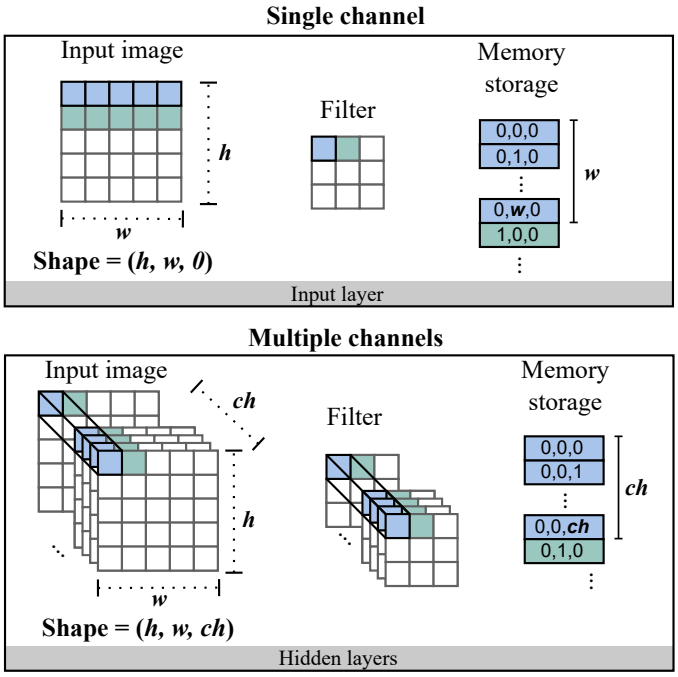


Figure 6.1 – Memory layout and access pattern in single-channel and multi-channel convolution.

representation before multiplication, through sign or zero extension depending on the data type. Thus, inside the vectorized convolution function, the SEW is set to 8 bits and LMUL to 4 for memory access, while for arithmetic operations the SEW is set to 16 bits and LMUL to 8. This way, up to 16 elements can be processed by a single instruction.

For single-channel inputs, the input image is stored row by row, so adjacent pixels in a row are placed sequentially in memory. To take advantage of this layout, I apply spatial vectorization, so a row of 8-bit input values is loaded at once with a vector load instruction (*vle8*), widened to 16-bit, offset-adjusted, and then multiplied in parallel by the corresponding filter coefficient (*vmul.vx*). As the convolution window slides across the image, each coefficient of the filter is reused for the entire row segment currently under that position. The partial sums are accumulated into the output buffer if the computed coordinates are within valid bounds.

For multi-channel data, channels are stored sequentially for each pixel. Thus, in this case, vectorization is applied across channels. At each pixel position, a vector of 8-bit input values from different channels of the same pixel is loaded. The same is done for the corresponding filter weights, as represented in Figure 6.1. Both input and filter weight vectors are widened to 16-bit and multiplied element-wise (*vmul.vv*), with the results accumulated in an intermediate buffer. The process is repeated across the filter dimensions until all operations for the output pixel are completed. Since either input data and filter weights are stored sequentially in the

channel dimension and processed in parallel, memory access and computation efficiency are further improved compared to the single-channel case.

Figure 6.2 shows a simplified pseudo-algorithm of the original version of the TFLM convolution function and the two vectorized ones. In both adapted convolution versions, bias addition, quantization, and activation are calculated as the final step of the convolution, after all MAC operations are completed and were not accelerated using vector instructions. Performing this step after the convolution is required to preserve accuracy, as the output has an 8-bit width.

Original

```
for batch = 0 to batches - 1:
  for out_y = 0 to output_height - 1:
    for out_x = 0 to output_width - 1:
      for out_channel = 0 to output_depth - 1:
        acc = 0
        for filter_y = 0 to filter_height - 1:
          for filter_x = 0 to filter_width - 1:
            for in_channel = 0 to filter_input_depth - 1:
              Load input value
              Load filter value
              Accumulate product
        if bias exists:
          Add bias
        Apply quantization
        Store output value
```

Single-channel vector version

```
for filter_batch = 0 to output_depth - 1:
  for filter_y = 0 to filter_height - 1:
    for filter_x = 0 to filter_width - 1:
      for in_batch = 0 to batches - 1:
        for in_y = 0 to input_height - 1:
          for in_x = 0 to input_width - 1, step = vl:
            Load vl 8-bit input elements.
            Widen the input vector to a 16-bit vector.
            Add the input offset to the input vector.
            vmul.vx between the input vector and filter value
            Calculate the respective out_x and out_y.
            if it is a valid output position:
              Accumulate the result in a temporary buffer.
```

Multi-channel vector version

```
for filter_batch = 0 to output_depth - 1:
  for filter_y = 0 to filter_height - 1:
    for filter_x = 0 to filter_width - 1:
      for in_batch = 0 to batches - 1:
        for in_y = 0 to input_height - 1:
          for in_x = 0 to input_width - 1:
            for in_channel = 0 to input_depth - 1, step = vl:
              Load vl 8-bit input elements.
              Load vl 8-bit filter elements.
              Widen input and filter vector to 16-bit.
              Add input and filter offsets.
              vmul.vv between the input and filter vectors.
              Calculate the respective out_x and out_y.
              if it is a valid output position:
                Accumulate the result in a temporary buffer.
```

Figure 6.2 – Simplified pseudo algorithm for original, single-channel, and multi-channel vectorized convolution kernels.

6.1.2 Vector Extension Unit Modifications

For our first version of the vectorized kernel, the widening of input and filter weight values from 8-bit to 16-bit was performed via a loop before loading them into vector registers. Although this method enabled parallelism and reduced computation compared to the scalar kernel, it could be further improved by utilizing vector widening instructions.

Therefore, I added support for the vector widening instructions and updated the convolution to optimize this step. This allows input and filter values to be read as 8-bit vectors and widened to 16-bit using a widening addition with zero (*vaddw.vx*), eliminating the conversion loop and improving execution efficiency.

To support the widening instruction, the control of the VEU was extended with an additional state in the FSM. The new state, RegisterWide, handles the second writeback required when the destination element size is larger than the source. The advance to RegisterWide occurs after the standard Register state whenever a widening instruction is detected, and the control returns to Decode once the second writeback is completed. This modification required minor adjustments to the writeback and counter control signals to ensure synchronization with the new state, while the overall operation of the vector unit and its interface with the main core remained unchanged.

Compared to the previous implementation, the updated FSM also introduces a condition to detect when a second writeback is necessary. This decision is based on the element counter and the next element value (next_counter_i) that is used to detect register boundaries within the 32-bit vector register file. For widening operations, this check ensures that both halves of a larger destination element are written correctly. When the next element index reaches the end of a physical register, the control transitions from Register to RegisterWide, performing the additional write before returning to Decode.

6.2 Experimental Setup

In this section, I describe the setup used for the experiment. For this version of the VEU, besides the fault injection campaigns, a proton irradiation experiment was conducted to validate the results in a real radiation environment. The following subsections detail the application benchmark I used for both tests, as well as the fault injection simulations and the irradiation experiment setup.

6.2.1 Application Benchmark

The CNN architecture used for the experiments follows the radiation testing benchmark proposed by [15] for MNIST digit classification with minimal cost. Designed to exceed 95% accuracy, it relies on a public dataset and is feasible for low-power microcontroller deployment. It includes three 3×3 convolution layers (8, 24, and 24 filters), 2×2 max-pooling, two dense layers (24 and 8 neurons), and a softmax output, with all layers using ReLU activation. The network contains 12,500 parameters and achieves 98% accuracy after quantization.

The CNN benchmark was deployed following the deployment flow, which includes model training, post-training quantization, and conversion to the *.tflite* format, and integration as a C array within the application. At runtime, TFLM handled memory allocation and operator scheduling on the HARV-SoC.

The input images were converted into a C-byte array and compiled within the application. For the fault injection simulation, I used a fixed image from the MNIST dataset to ensure consistent evaluations and facilitate debugging across different kernel versions. For the irradiation experiments, I randomly selected one image from each class of the dataset.

For the irradiation experiment, I added checksum verifications for the convolution input, bias, filter, and output. Since the VEU is not protected by fault-tolerance mechanisms, these verifications provide software-level observability, enabling the detection of faults during convolution. This allows distinguishing whether errors originate from corrupted parameters, intermediate computations, or final outputs.

6.2.2 Fault injection simulations

A total of 128 fault injection experiments for each case were executed for each of the scenarios described previously in Chapter 3.2.1. The neutron flux defined for the experiments was set to 2.5×10^{10}n/cm^2/s.

6.2.3 Irradiation experiment

The irradiation experiment was conducted in the PIF at the PSI. The experimental setup consisted of four boards (*B0*, *B1*, *B2*, *B3*) hosting the HARV-SoC. These boards comprise a Flash-based Microchip SmartFusion2 M2S025-VFG400I FPGA and a W9864G6JT SDRAM, used as system memory for the SoC. Similar to the experiment presented in Chapter 2, the experimental data were logged and transmitted to a host computer outside the irradiation room by serial connections. The setup is fixed in a frame and is shown in Figure 6.3, containing

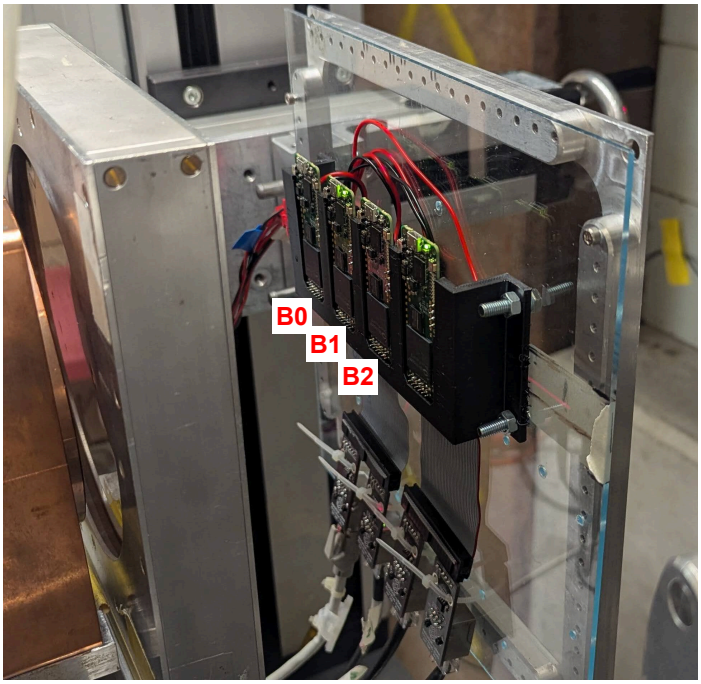


Figure 6.3 – Experimental boards attached to the test frame inside the irradiation room at PSI.

the mentioned test boards and the logging transceivers. Each board has an independent and monitored power supply line.

In this experiment, two versions of the HARV-SoC were used. For boards *B0*, *B1*, and *B2*, I used a version of the HARV-SoC where the reports from the error handler were disabled. This configuration allowed the system to detect and correct errors without reporting them, simulating a real application environment while minimizing execution time, which is crucial given the limited beam time. Thus, for boards *B0*, *B1*, and *B2*, I analyzed the failure cross-section, with a failure being any execution not classified as *correct*. In contrast, for board *B3*, I performed a more detailed analysis based on the events reported by the processor, as the reports from the error handler were enabled in this case.

The irradiation was carried out with protons at an energy of 200 MeV, with target proton fluxes from approximately 1.1×10^8 to 2.71×10^8 p/cm^2/s. The beam provided 10% homogeneity for the selected irradiation area (20 mm $\times$ 20 mm for *B0*, *B1*, *B2*, and 30 mm $\times$ 30 mm for *B3*) and beam energies.

For this experiment, the application was executed from an external Flash memory, since the Embedded Non-Volatile Memory (eNVM) from the Microcontroller Subsystem (MSS) was not large enough to store the complete binary of the benchmark application. To enable this, I implemented the SPI Flash controller in VHDL to manage the communication between the HARV-SoC and the external memory, enabling read, write, and Execute In Place (XIP)

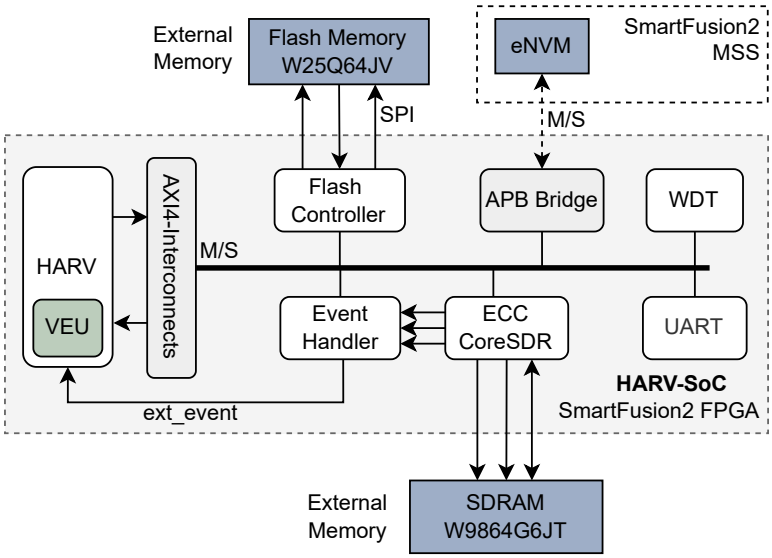


Figure 6.4 – SmartFusion2 System Design for the experiment Using HARV-SoC.

operations. The internal eNVM stores a bootloader application that waits about five seconds for a possible upload via UART. If a new binary is received, it is written to the external Flash. After this step or once the timeout expires, the bootloader enables XIP and jumps to the application start address, allowing the HARV processor to fetch and execute instructions directly from the external Flash. Figure 6.4 shows an overview of the system using the SmartFusion2 FPGA and HARV-SoC. To optimize beam utilization, the Systems Under Test (SUTs) were only irradiated once the program was executing from the external Flash.

Each of the four boards was irradiated and tested individually due to the size of the irradiation area. For the *B0* and *B3*, the beam was aligned to irradiate the FPGA and the SDRAM, which is located just above the FPGA. For the *B1* and *B2*, only the FPGA was irradiated. All the SUTs were running the same benchmark, but with some differences. The boards *B0* and *B1* were cycling between hardened and unhardened configurations, as well as between vector and scalar kernels. The boards *B2* and *B3* were running the benchmark with the hardening configurations and with the vector kernel. However, in this thesis, I analyze only the executions performed within the hardened vector scenarios.

6.3 Results

The following subsections present the synthesis results, the performance evaluation of the vectorized convolution kernel, and the reliability analysis under fault injection simulation. A final discussion presents the main observations and trade-offs.

6.3.1 Synthesis

To provide a preliminary evaluation of the impact of adding support for widening vector operations, I compared the resource utilization of the previous design previously presented in Chapter 5. For consistency with the earlier implementations, a preliminary synthesis was carried out on the same platform used in Chapters 4 and 5, the Zynq ZC7020 FPGA. The analysis considered LUTs, FFs, DSP blocks, maximum frequency (F_{max}), and dynamic power (P_{dyn}) for each HARV-SoC variant, as summarized in Table 6.1.

The results show that adding widening support introduces minimal hardware overhead in baseline and hardened configurations. The LUT usage increases by less than 3%, and changes in register usage and power are minimal. In the baseline version, widening leads to a moderate reduction in frequency, while in the hardened version, it results in a slight improvement in F_{max}, likely due to synthesis optimizations that restructure the datapath and reduce critical path delays.

Table 6.1 – Synthesis results for Baseline and Hardened versions.

HARV-SoC	Slice LUTs	Slice registers	DSP Blocks	F_{max} [MHz]	P_{dyn} [mW]
Baseline [111]	5,399	4,170	9	64.40	140
Baseline widening	5,487	4,173	9	54.07	141
Hardened [111]	7,555	4,968	17	40.77	175
Hardened widening	7,787	4,975	17	44.24	183

Since the irradiation experiments presented later in this chapter were performed on the SmartFusion2 FPGA, I also report synthesis results for this device. Table 6.2 summarizes the synthesis results for the SmartFusion2 FPGA. The HARV-SoC implementation uses approximately half of the available LUTs and less than one quarter of the FFs, while requiring 19 out of 34 DSP blocks.

Table 6.2 – Synthesis results for the SmartFusion2 FPGA.

HARV-SoC	LUTs	FFs	DSP Blocks	uSRAM	F_{max} [MHz]
Used	14,174	6,804	19	3	37.9
Total	27,696	27,696	34	34	-

6.3.2 Performance

The performance results are shown in Figure 6.5, which presents the results for the original implementation, the first vectorized version without widening support, and the final optimized version using vector widening instructions. Within all the cases, the three convolutions together represent more than 96% of the total runtime.

The original TFLM implementation executes convolution operations using scalar nested loops that iterate individually through each dimension. This approach results in many memory accesses and repeated arithmetic instructions, representing most of the total inference time. As expected, this version presents the longest execution time.

In the first vectorized version, I introduced vector instructions to replace the scalar computation without widening support. Since the input and filter values are stored as 8-bit integers, they must be manually converted to 16-bit integers inside the accumulation loop. Although this step adds an overhead, vectorizing the computation over spatial and channel dimensions reduces the number of loop iterations and improves data locality. As a result, the total runtime had a speedup of 1.89 times compared to the original kernel.

The final implementation fully utilizes the VEU by using widening instructions that read 8-bit values from memory and operate directly in 16-bit precision. Thus, the casting loop is eliminated, which reduces the instruction count. As a result, the convolution layers run significantly faster, and the total inference was 2.54 times faster compared to the original implementation.

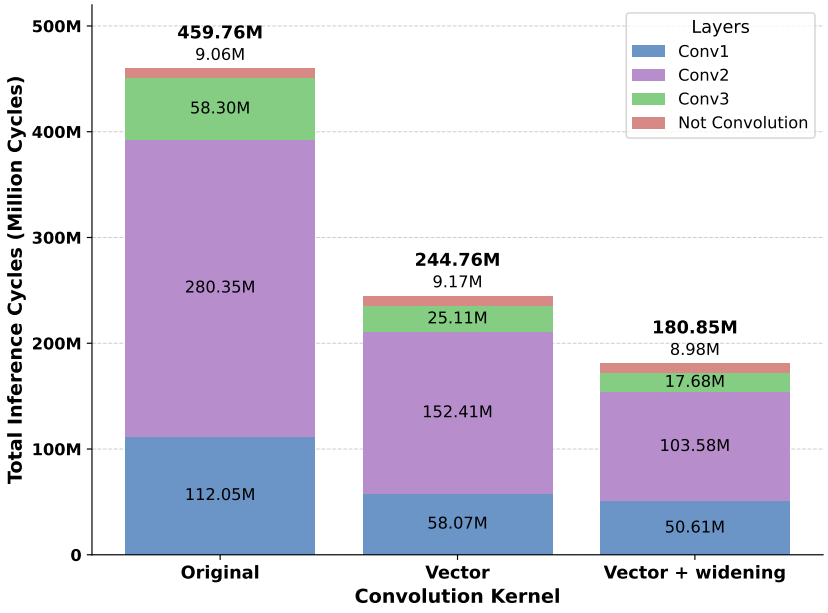


Figure 6.5 – Performance variation of the inference with different convolution kernel versions.

6.3.3 Fault injection simulations

Through the fault injection simulations, I evaluated the SoC's reliability by analyzing the processor's internal registers and outputs. Table 6.3 presents the results for the four evaluated scenarios. The vectorized convolution implementation achieved a higher number of correct executions compared to the original version in both hardened and unhardened scenarios. This improvement is due to the reduced execution time enabled by vectorization and widening, which limits the exposure window for faults to impact critical operations. Notably, no benchmark errors occurred in the vectorized case, while other failure types, such as hang and exception, were also reduced.

Exceptions were more frequent in the original kernel than in the vectorized one, likely due to its greater number of memory accesses, which increased exposure to access faults. The vectorized kernels, on the other hand, optimized memory access and reduced these issues, contributing to better reliability. Regarding TFLM failures, these were observed in the baseline scenario but not in the hardened one, which points to the effectiveness of the fault mitigation mechanisms in the hardened HARV-SoC design.

Finally, reliability metrics such as failure cross-section and simulated fluence showed that the hardened and unhardened vectorized kernels had fewer errors than the original kernel. Also, MWTF results demonstrated that the vectorized configurations could handle more work before encountering errors. These results indicate that vectorization, whether coupled with hardening techniques or not, significantly improves reliability.

Table 6.3 – Fault injection executions.

Execution classification	**Baseline**		**Hardened**	
	Original	**Vector**	**Original**	**Vector**
Correct	25	49	116	124
Benchmark error	2	0	0	0
Hang	6	5	0	0
TFLM halted	57	56	0	0
Exception	38	18	12	4
Reliability analysis				
Simulated fluence*	1.2×10^{13}	9.6×10^{12}	2×10^{13}	1.3×10^{13}
Failure XS†	9×10^{-12}	8.3×10^{-12}	6×10^{-13}	3.1×10^{-13}
MWTF	1.25	1.62	10.67	32

*[n/cm^2]; †[cm^2/device]

6.3.4 Irradiation experiment

As mentioned before, the irradiation experiment was performed with two different versions of the HARV-SoC, with the reports from the error handler disabled or enabled. The following subsections present the analysis of the classification of the executions and the events reported by the HARV-SoC.

Classification of executions

The classification of the executions is the same as that defined for the fault injection simulations, with the addition of the checksum verifications. Table 6.4 shows the classification of the executions and failure cross-section for *B0*, *B1*, and *B2*.

Multiple irradiation runs were performed for each SUT. For *B0*, I performed four irradiation runs. The first one reached a fluence of 1.93×10^{10} p/cm^2 with a flux of 5.24×10^{7} p/cm^2/sec, and since no errors were being observed, I incremented the flux and targeted fluence in each run, where the last one was incremented to a flux of 2.7×10^{8} p/cm^2/sec, reaching a fluence of 5×10^{11} p/cm^2. In the *B0*, only one failure was observed, which was a checksum error on the output of the second convolution, and it did not affect the inference result. The other seven failures occurred in *B1* and *B2*, which reached an accumulated dose of 47.28 and 50.64 krad, respectively.

For *B1* and *B2*, I performed two irradiation runs for each SUT. As mentioned earlier, hang executions occur when the processor stops responding. Two hangs were observed in *B1*, each occurring in a different run. The first run reached a fluence of 5×10^{11} p/cm^2 and accumulated a dose of 29.1 krad, and the hang observed happened at the end of the run, and after a power cycle, the SUT was still functional. However, in the second run, which had a fluence of 3.1×10^{11} p/cm^2, after the hang, which occurred when the accumulated dose reached 47.28 krad, a power cycle was ineffective in restoring operation, resulting in a non-recoverable device failure, possibly due to a dose effect.

For *B2*, the first run reached a fluence of 5×10^{11} p/cm^2 and an accumulated dose of 29.1 krad, while the second run obtained a fluence of 3.7×10^{11} p/cm^2 and an accumulated dose of 21.54 krad. Regarding the hangs for the *B2*, the first one occurred during the first irradiation run, when the accumulated dose was approximately 18 krad, and a power cycle was required to restore operation. The second hang happened during the second and final run of the experiment, when the accumulated dose reached 50.64 krad.

Of the five failures observed in *B2*, four of them happened on the second run. The exception that occurred was an illegal instruction, and after a soft reset, the SUT restarted the execution normally. The benchmark error and checksum error are likely caused by upsets

in the VRF or the vector execution unit, as these components do not have fault tolerance techniques. The benchmark error execution had a checksum error on the output of the first convolution layer.

Table 6.4 – Executions classification and cross-section.

Failure	***B0***	***B1***	***B2***	**Total**
Checksum error	1	0	1	2
Benchmark error	0	0	1	1
Hang	0	2	2	4
Exception	0	0	1	1
Reliability analysis				
Total failures	1	2	5	8
Fluence*	7.2×10^{11}	8.13×10^{11}	8.7×10^{11}	2.4×10^{12}
TID†	41.88	47.28	50.64	139.8
Failure XS‡	1.39×10^{-12}	2.46×10^{-12}	5.75×10^{-12}	3.33×10^{-12}

*[p/cm^2]; †[krad]; ‡[cm^2/device];

Classification of events

For the *B3*, nine irradiation runs were performed with a total accumulated fluence and dose of 8.95×10^{11} p/cm^2 and 52.09 krad, respectively. Among all the runs, a total of 17 events were observed. Table 6.5 presents the events' classification and cross-section (XS) metrics. Most events corresponded to SBUs affecting either the memory or the processor register file outputs (`rs1` and `rs2`). All reported SBUs were corrected, and the application continued to produce the expected output. A checksum error was also detected in the output of the second convolution, but it did not alter the final inference result.

Four events were classified as illegal instruction exceptions, and two hangs were observed, in which the processor became unresponsive and required a power cycle to resume operation. These events led to application failures and resulted in a failure cross-section of 7.82×10^{-12} cm^2.

During the final irradiation run, an illegal-instruction exception followed by a soft reset left the processor permanently unresponsive, even after a power cycle. This probably happened due to a dose effect, and the accumulated dose ate the moment of the event was approximately 52 krad.

Table 6.5 – Events classification and cross-section for *B3*.

Type of event	Number of events	Events XS [cm^2/device]
Memory SBU	3	3.35×10^{-12}
Register file SBU	6	6.70×10^{-12}
Exception	4	4.46×10^{-12}
Hang	3	3.35×10^{-12}
Checksum error	1	1.11×10^{-12}
Total	17	1.90×10^{-11}

6.4 Discussion

The results show that vectorization significantly improves the performance of the convolution operation. The vectorized kernel enables parallel processing of multiple input elements, reducing the overall execution time. These performance improvements are crucial for embedded AI applications that require real-time processing, particularly in systems with limited resources, such as those used in space and other radiation-prone environments.

Regarding reliability, the fault injection simulations showed that the vectorized kernels are more resilient than the original version. The reduced execution time minimizes the exposure to faults, making the system more resilient to radiation-induced errors. Moreover, introducing widening instructions improved memory access efficiency while maintaining the precision for accurate calculations. Even in unhardened configurations, the vectorized kernel outperformed the scalar baseline, demonstrating that vectorization improves performance and enhances reliability in harsh environments.

A direct comparison of the irradiation experiment results with [15] is possible, as I used the NN benchmark proposed by the authors, and the irradiation experiments were performed in the same facility under the same proton energy. In their experiments, Cortex-M0+ and Cortex-M4 devices were tested at fluxes on the order of 10^7 to 10^8 p/cm^2/sec and fluences up to 10^{11} p/cm^2, reporting cross-sections for CNN inference failures reaching 9×10^{-11} cm^2/device.

In the results presented in this thesis, the HARV-SoC with VEU exhibited cross-sections in the order of 10^{-12} cm^2/device, corresponding to an improvement of one to two orders of magnitude in comparison to the results from [15] under similar beam conditions (i.e., 200 MeV, fluxes up to 2.7×10^8 p/cm^2/sec, and fluences up to 5×10^{11} p/cm^2).

In a previous work [112] from our group, the HARV-SoC on a Microchip SmartFusion2 M2S010 FPGA was characterized under proton irradiation. The authors reported a sharp

increase in error indicators for accumulated doses over 25 krad. Furthermore, they observed unrecoverable device failures for accumulated doses higher than 35 krad. However, in analyses, even with 50.64 krad in *B2*, I observed only a few failures reported by the processor. Possible reasons are that the FPGA models used in the two experiments are different, as in [112] it was the M2S010-VFG400, whereas for this experiment, the M2S025-VFG400I, which is an industrial model, and has a more recent fabrication according to the SUT's date code (2023 vs 2016). Besides that, contrary to [112], in this experiment, the program was being executed directly from the external Flash, which can handle a higher TID than the eNVM of the MSS. The mentioned unrecoverable error was likely due to the eNVM's TID failure, since the bootloader was required to start up the application, and even a power cycle was ineffective.

Given the discussion above shows that the vectorized convolution, when combined with fault tolerance techniques, improves both performance and reliability in a fault-tolerant RISC-V SoC, making it suitable for radiation-prone environments. By integrating vector widening instructions, I optimized memory access while maintaining numerical accuracy, resulting in significant speedup in inference tasks.

Chapter 7

Summary and Conclusions

This thesis investigated how vectorization can be applied to improve the computational efficiency of CNN inference on radiation-tolerant processors, while preserving reliability in radiation environments. Thus, I implemented a Vector Extension Unit (VEU) for the HARV-SoC, a fault-tolerant and low-power RISC-V SoC. The study aimed to determine whether the introduction of a VEU could accelerate CNN inference without compromising the reliability of the processor under radiation exposure.

In Chapter 1, I introduced the motivation and objectives of this thesis. The increasing adoption of AI in embedded systems, together with the need for reliability in radiation environments, motivated the design of a VEU for a fault-tolerant processor capable of accelerating CNN inference. The chapter presented the scientific context and background, the challenges of combining performance and dependability, and the goals pursued throughout the thesis.

The first step was to understand how radiation affects existing AI hardware. In Chapter 2, I analyzed the behavior of a commercial Edge AI microcontroller under neutron irradiation. The tests showed that several faults originated from the processor core or the data memory and propagated to the application level, causing misclassifications and program interruptions. Thus, it confirmed the importance of a fault-tolerant processor for supporting reliable AI inference.

The remaining Chapters focus on the design, integration, and enhancement of the VEU within the HARV-SoC. In Chapter 3, I established the adopted methodology for assessing both the HARV-SoC and the HARV-SoC with VEU across all the versions. The same evaluation methodology was applied to all processor versions, combining synthesis, performance benchmarking, and fault injection. This process ensured consistent comparison across designs and enabled a combined interpretation of performance and reliability metrics. The

chapter also defined the MWTF metric, which relates execution time and fault rate, providing a single indicator to assess the trade-off between performance and reliability.

With the evaluation methodology defined, I introduced the preliminary implementation of the VEU integrated into the HARV-SoC in Chapter 4. The implementation was based on a minimalist version of the RVV, with the aim of functional validation and hardware exploration. This version validated the concept of adding vector processing to a fault-tolerant RISC-V processor. Benchmark tests using simple vector-arithmetic operations showed that the vectorized implementation reduced the execution time compared to the scalar version, validating the expected performance gain. Besides that, the MWTF metric improved, since the shorter execution time compensated for the increase in sensitive area. Overall, this preliminary implementation confirmed the functional integration of the VEU into the HARV-SoC. However, this first version had limited instruction coverage and relied on a simplified control logic, which constrained performance scalability.

In Chapter 5, I presented an enhanced version of the VEU developed to overcome the limitations of the preliminary design. The control unit was redesigned to support additional vector instructions and improve the microarchitecture of the VEU, introducing support for different VLs and the parameterization of LMUL. The synthesis results showed a moderate rise in logic utilization but a similar operating frequency. Performance benchmarks confirmed faster execution for all tested configurations, with higher scalability across data widths. The fault injection analysis revealed the expected trade-off between area and sensitivity, but the reduction in execution time resulted in a higher MWTF, indicating a reliability improvement.

The focus of Chapter 6 was to validate the processor in a realistic AI inference scenario and to demonstrate that the developed VEU can efficiently perform CNN inference. The convolution kernels, which dominate the overall inference time, were adapted to use vector instructions. The implementation achieved up to 2.54$\times$ speedup compared to the scalar baseline. Reliability experiments with proton irradiation demonstrated that the HARV-SoC with the VEU remained operational throughout the tests. The cross-section values obtained experimentally matched the fault injection results, confirming the consistency of the adopted evaluation methodology.

Overall, this thesis established and experimentally validated the first integration of a RISC-V vector extension within a fault-tolerant processor, demonstrating that data-level parallelism can enhance CNN inference performance while maintaining dependable behavior under radiation. The presented methodology and results provide a reference for future designs of reliable and efficient vectorized processors for Edge AI applications in space environments.

Regarding future work, several directions can be explored to further improve the VEU in terms of performance and capabilities. An important next step is to extend the VEU to fully support the RISC-V Zve32x subset, allowing the use of auto-vectorization. Furthermore, additional improvements may focus on extending the TFLM vectorization to other NN layers and better exploiting DLP during inference.

Architectural optimizations can also be investigated, such as increasing the number of vector lanes and introducing internal pipelining to overlap memory access and computation, aiming to enhance throughput. Finally, future radiation experiments with different particle types and energy ranges could provide a more realistic assessment of the system behavior and complement the proton-based results presented in this thesis.

Bibliography

[1] G. Furano, A. Tavoularis, and M. Rovatti, "Ai in space: applications examples and challenges," in *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 2020, pp. 1–6.

[2] J. Manning, D. Langerman, B. Ramesh, E. Gretok, C. Wilson, A. George, J. MacKinnon, and G. Crum, "Machine-learning space applications on smallsat platforms with tensorflow," in *32nd Annual AIAA/USU Conference on Small Satellites*, no. SSC18-WKVII-03. American Institute of Aeronautics and Astronautics (AIAA), 2018.

[3] S. Silvestrini and M. Lavagna, "Deep learning and artificial neural networks for spacecraft dynamics, navigation and control," *Drones*, vol. 6, no. 10, p. 270, 2022.

[4] M. Biasielli, C. Bolchini, L. Cassano, E. Koyuncu, and A. Miele, "A neural network based fault management scheme for reliable image processing," *IEEE Transactions on Computers*, vol. 69, no. 5, pp. 764–776, 2020.

[5] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *International conference on artificial neural networks*. Springer, 2014, pp. 281–290.

[6] G. Agosta, A. Galimberti, and D. Zoni, "Deep learning on risc-v platforms at the edge: A perspective on the hardware and software support," *ACM Computing Surveys*, 2025.

[7] M. Bagatin and S. Gerardin, *Ionizing Radiation Effects in Electronics: From Memories to Imagers*. CRC Press, 2016.

[8] J. R. Schwank, "Basic mechanisms of radiation effects in the natural space radiation environment," in *Presented at the 31st Annual International Nuclear and Space Radiation Effects Conference (NSREC)*, United States, Jun. 1994, [Online]. Available: https://www.osti.gov/biblio/10158182.

[9] M. Yang, G. Hua, Y. Feng, and J. Gong, *Fault-tolerance techniques for spacecraft control computers*. Singapore: John Wiley & Sons, 2017.

[10] S. Larsen, R. Rabbah, and S. Amarasinghe, "Exploiting vector parallelism in software pipelined loops," in *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. IEEE, 2005, pp. 11–pp.

[11] J. M. Cebrián, L. Natvig, and J. C. Meyer, "Performance and energy impact of parallelization and vectorization techniques in modern microprocessors," *Computing*, vol. 96, no. 12, pp. 1179–1193, 2014.

[12] J. Ren and L. Xu, "On vectorization of deep convolutional neural networks for vision tasks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, no. 1, 2015.

[13] S. D. 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.

[14] D. A. Santos, C. Imianosky, and L. Dilillo, "HARV - Hardened RISC-V System-on-Chip," in *2nd RISC-V in Space Workshop*, Mar. 2025.

[15] M. Giordano, R. Ferraro, M. Magno, and S. Danzeca, "General purpose and neural network approach for benchmarking microcontrollers under radiation," in *2021 21th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*. IEEE, 2021, pp. 1–8.

[16] E. R. Benton and E. V. Benton, "Space radiation dosimetry in low-earth orbit and beyond," *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms*, vol. 184, no. 1, pp. 255–294, 2001, DOI: 10.1016/S0168-583X(01)00748-0.

[17] L. Caronti, K. Akhunov, M. Nardello, K. S. Yıldırım, and D. Brunelli, "Fine-grained hardware acceleration for efficient batteryless intermittent inference on the edge," *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 5, pp. 1–19, 2023.

[18] 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)*, Marrakesh, 2020, pp. 1–5.

[19] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE internet of things journal*, vol. 3, no. 5, pp. 637–646, 2016.

[20] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos, "Challenges and opportunities in edge computing," in *2016 IEEE international conference on smart cloud (SmartCloud)*. IEEE, 2016, pp. 20–26.

[21] X. Wang, Y. Han, V. C. Leung, D. Niyato, X. Yan, and X. Chen, *Edge AI: Convergence of edge computing and artificial intelligence*. Springer Nature, 2020.

[22] P. Warden and D. Situnayake, *Tinyml: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers*. O'Reilly Media, 2019.

[23] E. Li, L. Zeng, Z. Zhou, and X. Chen, "Edge ai: On-demand accelerating deep neural network inference via edge computing," *IEEE transactions on wireless communications*, vol. 19, no. 1, pp. 447–457, 2019.

[24] G. Furano, G. Meoni, A. Dunne, D. Moloney, V. Ferlet-Cavrois, A. Tavoularis, J. Byrne, L. Buckley, M. Psarakis, K.-O. Voss, and L. Fanucci, "Towards the use of artificial intelligence on the edge in space systems: Challenges and opportunities," *IEEE Aerospace and Electronic Systems Magazine*, vol. 35, no. 12, pp. 44–56, 2020.

[25] P. W. David *et al.*, "Tensorflow lite for microcontrollers," https://www.tensorflow.org/lite/microcontrollers, 2020.

[26] S. Cai, S. Zhou, C. Xu, and Q. Gao, "Dense motion estimation of particle images via a convolutional neural network," *Experiments in Fluids*, vol. 60, no. 4, p. 73, 2019.

[27] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[28] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[29] M. J. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.

[30] D. Patterson and A. Waterman, *The RISC-V Reader: an open architecture Atlas*. San Francisco: Strawberry Canyon, 2017.

[31] J. Gebis and D. Patterson, "Embracing and extending 20th-century instruction set architectures," *Computer*, vol. 40, no. 4, pp. 68–75, 2007.

[32] SiFive, "Sifive vectors. optimized for the modern workload," Disponível em: https://www.sifive.com/technology/vectors. Acesso em 06 de junho de 2022., Jun. 2022.

[33] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[34] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The RISC-V instruction set manual, volume I: Base user-level ISA," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, vol. 116, 2011.

[35] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*, December 2019.

[36] RISC-V International, "The RISC-V "V" Vector Extension, Version 1.0," https://github.com/riscv/riscv-v-spec/releases/tag/v1.0, 2021.

[37] M. F. L'Annunziata, "Nuclear radiation, its interaction with matter and radioisotope decay," in *Handbook of Radioactivity Analysis*, 2nd ed. San Diego: Academic Press, 2003, ch. 1, pp. 1–121, DOI: 10.1016/B978-012436603-9/50006-5.

[38] M. Bagatin and S. Gerardin, "Introduction to the effects of radiation on electronic devices," in *Ionizing Radiation Effects in Electronics: From Memories to Imagers*, M. Bagatin and S. Gerardin, Eds. CRC Press, 2016, ch. 1, pp. 1–22.

[39] J. Barth, C. Dyer, and E. Stassinopoulos, "Space, atmospheric, and terrestrial radiation environments," *IEEE Transactions on Nuclear Science*, vol. 50, no. 3, pp. 466–482, Jul. 2003, DOI: 10.1109/TNS.2003.813131.

[40] S. Bourdarie and M. Xapsos, "The near-earth space radiation environment," *IEEE Transactions on Nuclear Science*, vol. 55, no. 4, pp. 1810–1832, Sept. 2008, DOI: 10.1109/TNS.2008.2001409.

[41] J. L. Barth, C. Dyer, and E. Stassinopoulos, "Space, atmospheric, and terrestrial radiation environments," *IEEE Transactions on nuclear science*, vol. 50, no. 3, pp. 466–482, 2003.

[42] J. A. VAN ALLEN, G. H. LUDWIG, E. C. RAY, and C. E. McILWAIN, "Observation of high intensity radiation by satellites 1958 alpha and gamma," *Journal of Jet Propulsion*, vol. 28, no. 9, pp. 558–592, Jun 1958, DOI: 10.2514/8.7396.

[43] M. Ackermann, M. Ajello, A. Allafort, L. Baldini, J. Ballet, G. Barbiellini, M. G. Baring, D. Bastieri, K. Bechtol, R. Bellazzini, R. D. Blandford, E. D. Bloom, E. Bonamente, A. W. Borgland, E. Bottacini, T. J. Brandt, J. Bregeon, M. Brigida, P. Bruel, R. Buehler, G. Busetto, S. Buson, G. A. Caliandro, R. A. Cameron, P. A. Caraveo, J. M. Casandjian, C. Cecchi, Ö. Çelik, E. Charles, S. Chaty, R. C. G. Chaves, A. Chekhtman, C. C. Cheung, J. Chiang, G. Chiaro, A. N. Cillis, S. Ciprini, R. Claus, J. Cohen-Tanugi, L. R. Cominsky, J. Conrad, S. Corbel, S. Cutini, F. D'Ammando, A. de Angelis, F. de Palma, C. D. Dermer, E. do Couto e Silva, P. S. Drell, A. Drlica-Wagner, L. Falletti, C. Favuzzi, E. C. Ferrara, A. Franckowiak, Y. Fukazawa, S. Funk, P. Fusco, F. Gargano, S. Germani, N. Giglietto, P. Giommi, F. Giordano, M. Giroletti, T. Glanzman, G. Godfrey, I. A. Grenier, M.-H. Grondin, J. E. Grove, S. Guiriec, D. Hadasch, Y. Hanabata, A. K. Harding, M. Hayashida, K. Hayashi, E. Hays, J. W. Hewitt, A. B. Hill, R. E. Hughes, M. S. Jackson, T. Jogler, G. Jóhannesson, A. S. Johnson, T. Kamae, J. Kataoka, J. Katsuta, J. Knödlseder, M. Kuss, J. Lande, S. Larsson, L. Latronico, M. Lemoine-Goumard, F. Longo, F. Loparco, M. N. Lovellette, P. Lubrano, G. M. Madejski, F. Massaro, M. Mayer, M. N. Mazziotta, J. E. McEnery, J. Mehault, P. F. Michelson, R. P. Mignani, W. Mitthumsiri, T. Mizuno, A. A. Moiseev, M. E. Monzani, A. Morselli, I. V. Moskalenko, S. Murgia, T. Nakamori, R. Nemmen, E. Nuss, M. Ohno, T. Ohsugi, N. Omodei, M. Orienti, E. Orlando, J. F. Ormes, D. Paneque, J. S. Perkins, M. Pesce-Rollins, F. Piron, G. Pivato, S. Rainò, R. Rando, M. Razzano, S. Razzaque, A. Reimer, O. Reimer, S. Ritz, C. Romoli, M. Sánchez-Conde, A. Schulz, C. Sgrò, P. E. Simeon, E. J. Siskind, D. A. Smith, G. Spandre, P. Spinelli, F. W. Stecker, A. W. Strong, D. J. Suson, H. Tajima, H. Takahashi, T. Takahashi, T. Tanaka, J. G. Thayer, J. B. Thayer, D. J. Thompson, S. E. Thorsett, L. Tibaldo, O. Tibolla, M. Tinivella, E. Troja, Y. Uchiyama, T. L. Usher, J. Vandenbroucke, V. Vasileiou, G. Vianello, V. Vitale, A. P. Waite, M. Werner, B. L. Winer, K. S. Wood, M. Wood, R. Yamazaki, Z. Yang, and S. Zimmer, "Detection of the characteristic pion-decay signature in supernova remnants," *Science*, vol. 339, no. 6121, pp. 807–811, 2013.

[44] J. Becker Tjus and L. Merten, "Closing in on the origin of galactic cosmic rays using multimessenger information," *Physics Reports*, vol. 872, pp. 1–98, Aug. 2020, DOI: 10.1016/j.physrep.2020.05.002.

[45] E. Petersen, *Single Event Effects in Aerospace*. Wiley-IEEE Press, Sept. 2011, DOI: 10.1002/9781118084328.

[46] F. Rahmanifard, W. C. de Wet, N. A. Schwadron, M. J. Owens, A. P. Jordan, J. K. Wilson, C. J. Joyce, H. E. Spence, C. W. Smith, and L. W. Townsend, "Galactic cosmic radiation in the interplanetary space through a modern secular minimum," *Space Weather*, vol. 18, no. 9, p. e2019SW002428, Jul. 2020, DOI: 10.1029/2019SW002428.

[47] P. Cannon, M. Angling, L. Barclay *et al.*, "Chapter 7 and 9—radiation impacts on satellites and ionising radiation impacts on avionics and ground systems," *Extreme Space Weather: Impacts on Engineered Systems and Infrastructure; Royal Academy of Engineering: London, UK*, 2013.

[48] C. Verbeeck, E. Kraaikamp, D. F. Ryan, and O. Podladchikova, "Solar flare distributions: Lognormal instead of power law?" *The Astrophysical Journal*, vol. 884, no. 1, p. 50, Oct. 2019, DOI: 10.3847/1538-4357/ab3425.

[49] M. A. Xapsos, P. M. O'Neill, and T. P. O'Brien, "Near-earth space radiation models," *IEEE Transactions on Nuclear Science*, vol. 60, no. 3, pp. 1691–1705, Dec. 2013, DOI: 10.1109/TNS.2012.2225846.

[50] Y. Y. Shprits, D. Subbotin, A. Drozdov, M. E. Usanova, A. Kellerman, K. Orlova, D. N. Baker, D. L. Turner, and K.-C. Kim, "Unusual stable trapping of the ultrarelativistic electrons in the van allen radiation belts," *Nature Physics*, vol. 9, no. 11, p. 699–703, Sept. 2013, DOI: 10.1038/nphys2760.

[51] P. Anderson, F. Rich, and S. Borisov, "Mapping the south atlantic anomaly continuously over 27 years," *Journal of Atmospheric and Solar-Terrestrial Physics*, vol. 177, pp. 237–246, Oct. 2018, DOI: 10.1016/j.jastp.2018.03.015.

[52] W. K. Tobiska, W. Atwell, P. Beck, E. Benton, K. Copeland, C. Dyer, B. Gersey, I. Getley, A. Hands, M. Holland *et al.*, "Advances in atmospheric radiation measurements and modeling needed to improve air safety," *Space Weather*, vol. 13, no. 4, pp. 202–210, 2015.

[53] P. Carlson and A. A. Watson, "Erich regener and the ionisation maximum of the atmosphere," *History of Geo-and Space Sciences*, vol. 5, no. 2, pp. 175–182, 2014.

[54] E. Normand, "Single event upset at ground level," *IEEE transactions on Nuclear Science*, vol. 43, no. 6, pp. 2742–2750, 1996.

[55] J. JEDEC, "Measurement and reporting of alpha particles and terrestrial cosmic ray-induced soft errors in semiconductor devices: Jesd89a," *JEDEC STANDARD, JEDEC Sold State Technology Association*, vol. 89, pp. 1–85, 2006.

[56] J. Barak and N. M. Yitzhak, "SEU rate in avionics: From sea level to high altitudes," *IEEE Transactions on Nuclear Science*, vol. 62, no. 6, pp. 3369–3380, 2015, doi: 10.1109/TNS.2015.2495324.

[57] R. García Alía, M. Brugger, F. Cerutti, S. Danzeca, A. Ferrari, S. Gilardoni, Y. Kadi, M. Kastriotou, A. Lechner, C. Martinella, O. Stein, Y. Thurel, A. Tsinganis, and S. Uznanski, "LHC and HL-LHC: Present and future radiation environment in the high-luminosity collision points and RHA implications," *IEEE Transactions on Nuclear Science*, vol. 65, no. 1, pp. 448–456, Jan. 2018, DOI: 10.1109/TNS.2017.2776107.

[58] M. Bagatin, A. Coniglio, M. D'Arienzo, A. De Lorenzi, S. Gerardin, A. Paccagnella, R. Pasqualotto, S. Peruzzo, and S. Sandri, "Radiation environment in the ITER neutral beam injector prototype," *IEEE Transactions on Nuclear Science*, vol. 59, no. 4, pp. 1099–1044, Mar. 2012, DOI: 10.1109/TNS.2012.2187461.

[59] C. Cazzaniga and C. D. Frost, "Progress of the scientific commissioning of a fast neutron beamline for chip irradiation," in *Journal of Physics: Conference Series*, vol. 1021. IOP Publishing, 2018, p. 012037.

[60] W. Hajdas, L. Adams, B. Nickson, and A. Zehnder, "The Proton Irradiation Facility at the Paul Scherrer Institute," *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms*, vol. 113, no. 1, pp. 54–58, 1996, DOI: 10.1016/0168-583X(95)01327-X.

[61] A. Gerbershagen, L. Barazzuol, S. Both, S. Brandenburg, R. Coppes, P. Dendooven, B. Jones, J. Schippers, E. V. D. Graaf, P. V. Luijk, and M.-J. van Goethem, "Establishment of the new particle therapy research center (PARTREC) at UMCG groningen," in *Proc. 15th International Conference on Heavy Ion Accelerator Technology (HIAT'22)*, no. 15, Darmstadt, Germany, Aug. 2022, pp. 20–23, DOI: 10.18429/JACoW-HIAT2022-MO3C2.

[62] A. Virtanen, "Radiation effects facility RADEF," in *Proceedings of the Eighth IEEE International On-Line Testing Workshop (IOLTW 2002)*, Bendor, France, July 2002, p. 188, DOI: 10.1109/OLT.2002.1030212.

[63] J. Mekki, M. Brugger, R. G. Alia, A. Thornton, N. C. Dos Santos Mota, and S. Danzeca, "A mixed field facility at CERN for radiation test: CHARM," in *15th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, Moscow, Russia, 2015, pp. 284–287, DOI: 10.1109/RADECS.2015.7365580.

[64] D. M. Fleetwood, "Total ionizing dose effects in MOS and low-dose-rate-sensitive linear-bipolar devices," *IEEE Transactions on Nuclear Science*, vol. 60, no. 3, pp. 1706–1730, Jun. 2013, DOI: 10.1109/TNS.2013.2259260.

[65] H. L. Hughes and R. R. Giroux, "Space radiation affects MOS FET's," *Electronics*, vol. 37, pp. 58–60, Dec. 1964.

[66] H. Hughes and J. Benedetto, "Radiation effects and hardening of MOS technology: devices and circuits," *IEEE Transactions on Nuclear Science*, vol. 50, no. 3, pp. 500–521, 2003, DOI: 10.1109/TNS.2003.812928.

[67] E. Chatzikyriakou, K. Morgan, and C. H. K. de Groot, "Total ionizing dose hardened and mitigation strategies in deep submicrometer CMOS and beyond," *IEEE Transactions on Electron Devices*, vol. 65, no. 3, pp. 808–819, Mar. 2018, DOI: 10.1109/TED.2018.2792305.

[68] K. Galloway and R. Schrimpf, "Mos device degradation due to total dose ionizing radiation in the natural space environment: A review," *Microelectronics Journal*, vol. 21, no. 2, pp. 67–81, 1990, DOI: 10.1016/0026-2692(90)90027-Z.

[69] R. H. Maurer, M. E. Fraeman, M. N. Martin, and D. R. Roth, "Harsh environments: Space radiation environment, effects, and mitigation," *Johns Hopkins APL technical digest*, vol. 28, no. 1, pp. 17–29, 2008, [Online]. Available: https://www.jhuapl.edu/Content/techdigest/pdf/V28-N01/28-01-Maurer.pdf.

[70] J. R. Srour and J. W. Palko, "Displacement damage effects in irradiated semiconductor devices," *IEEE Transactions on Nuclear Science*, no. 3, pp. 1740–1766, Jun. 2013, DOI: 10.1109/TNS.2013.2261316.

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

[72] T. May and M. Woods, "Alpha-particle-induced soft errors in dynamic memories," *IEEE Transactions on Electron Devices*, vol. 26, no. 1, pp. 2–9, Jan. 1979, DOI: 10.1109/T-ED.1979.19370.

[73] M. Bagatin, S. Gerardin, and A. Paccagnella, "Space and terrestrial radiation effects in flash memories," *Semiconductor Science and Technology*, vol. 32, no. 3, p. 033003, Feb. 2017, DOI: 10.1088/1361-6641/32/3/033003.

[74] R. Koga, S. Penzin, K. Crawford, and W. Crain, "Single event functional interrupt (sefi) sensitivity in microcircuits," in *RADECS 97. Fourth European Conference on Radiation and its Effects on Components and Systems (Cat. No. 97TH8294)*. IEEE, 1997, pp. 311–318.

[75] R. Gaillard, "Single event effects: Mechanisms and classification," in *Soft Errors in Modern Electronic Systems*, 1st ed., ser. 41, M. Nicolaidis, Ed. New York, NY, USA: Springer US, 2011, ch. 2, pp. 27–54, DOI: 10.1007/978-1-4419-6993-4.

[76] G. Bruguier and J.-M. Palau, "Single particle-induced latchup," *IEEE transactions on nuclear science*, vol. 43, no. 2, pp. 522–532, 1996.

[77] H. Puchner, R. Kapre, S. Sharifzadeh, J. Majjiga, R. Chao, D. Radaelli, and S. Wong, "Elimination of single event latchup in 90nm sram technologies," in *2006 IEEE International Reliability Physics Symposium Proceedings*, San Jose, CA, USA, Mar. 2006, DOI: 10.1109/RELPHY.2006.251342.

[78] J. Schwank, V. Ferlet-Cavrois, M. Shaneyfelt, P. Paillet, and P. Dodd, "Radiation effects in soi technologies," *IEEE Transactions on Nuclear Science*, vol. 50, no. 3, pp. 522–538, Jun. 2003, DOI: 10.1109/TNS.2003.812930.

[79] N. A. Dodds, N. C. Hooten, R. A. Reed, R. D. Schrimpf, J. H. Warner, N. J.-H. Roche, D. McMorrow, S.-J. Wen, R. Wong, J. F. Salzman, S. Jordan, J. A. Pellish, C. J. Marshall, N. J. Gaspard, W. G. Bennett, E. X. Zhang, and B. L. Bhuva, "Effectiveness of SEL hardening strategies and the latchup domino effect," *IEEE Transactions on Nuclear Science*, vol. 59, no. 6, pp. 2642–2650, 2012, DOI: 10.1109/TNS.2012.2224374.

[80] D. Sorin, *Fault tolerant computer architecture*. Morgan & Claypool Publishers, 2009.

[81] D. A. Santos, L. M. Luza, L. Dilillo, C. A. Zeferino, and D. R. Melo, "Reliability analysis of a fault-tolerant risc-v system-on-chip," *Microelectronics Reliability*, vol. 125, p. 114346, 2021.

[82] D. A. Santos, L. M. Luza, M. Kastriotou, C. Cazzaniga, C. A. Zeferino, D. R. Melo, and L. Dilillo, "Characterization of a RISC-V system-on-chip under neutron radiation," in *Proc. of the 16th Int. Conference on Design & Technology of Integrated Systems (DTIS)*, 2021, pp. 1–6.

[83] D. A. Santos, A. M. P. Mattos, D. R. Melo, and L. Dilillo, "Enhancing fault awareness and reliability of a fault-tolerant RISC-V system-on-chip," *Electronics*, vol. 12, no. 12, 2023.

[84] D. A. Santos, A. M. P. de Mattos, L. M. Luza, C. Cazzaniga, M. Kastriotou, D. R. Melo, and L. Dilillo, "Neutron irradiation testing and analysis of a fault-tolerant RISC-V system-on-chip," in *Proc. of the IEEE Int. Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2022, pp. 1–6.

[85] D. A. Santos, P. M. Aviles, A. M. P. Mattos, M. García-Valderas, L. Entrena, A. Lindoso, and L. Dilillo, "Hybrid hardening approach for a fault-tolerant risc-v system-on-chip," *IEEE Transactions on Nuclear Science*, vol. 71, no. 8, pp. 1722–1730, 2024.

[86] D. Santos *et al.*, "Harv-soc: Hardened risc-v soc with vector extension for fault-tolerant applications," RADEC Conference submission, 2025, 2025.

[87] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, "Ara: A 1-ghz+ scalable and energy-efficient risc-v vector processor with multiprecision floating-point support in 22-nm fd-soi," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 530–543, 2019.

[88] M. Perotti, M. Cavalcante, R. Andri, L. Cavigelli, and L. Benini, "Ara2: Exploring single-and multi-core vector processing with an efficient rvv 1.0 compliant open-source processor," *IEEE Transactions on Computers*, vol. 73, no. 7, pp. 1822–1836, 2024.

[89] F. Minervini, O. Palomar, O. Unsal, E. Reggiani, J. Quiroga, J. Marimon, C. Rojas, R. Figueras, A. Ruiz, A. Gonzalez *et al.*, "Vitruvius+: An area-efficient risc-v decoupled vector coprocessor for high performance computing applications," *ACM Transactions on Architecture and Code Optimization*, vol. 20, no. 2, pp. 1–25, 2023.

[90] M. Barbirotta, F. Minervini, C. R. Morales, A. Cristal, O. Unsal, and M. Olivieri, "Enhancing fault tolerance in high-performance computing: A real hardware case study on a risc-v vector processing unit," *Authorea Preprints*, 2024.

[91] P. Schmidt, J. Pfau, T. Hotfilter, M. Stammler, T. Harbaum, and J. Becker, "Rvve: A minimal risc-v vector processor for embedded ai acceleration," in *2024 IEEE 37th International System-on-Chip Conference (SOCC)*. IEEE, 2024, pp. 1–6.

[92] S. Nolting and A. the Awesome Contributors, "The neorv32 risc-v processor," https://github.com/stnolting/neorv32, 2023, accessed: 2025-10-15.

[93] V. N. Chander and K. Varghese, "A soft risc-v vector processor for edge-ai," in *2022 35th International Conference on VLSI Design and 2022 21st International Conference on Embedded Systems (VLSID)*. IEEE, 2022, pp. 263–268.

[94] J. P. Jones, P. Van Kempen, and D. Mueller-Gritschneder, "Vicuna2. 0: Risc-v embedded vector unit with half-precision floating-point support for tinyml," in *2025 Austrochip Workshop on Microelectronics (Austrochip)*. IEEE, 2025, pp. 69–72.

[95] M. Platzer and P. Puschner, "Vicuna: A timing-predictable risc-v vector coprocessor for scalable parallel computation," in *33rd euromicro conference on real-time systems (ECRTS 2021)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021, pp. 1–1.

[96] S. Di Mascio, A. Menicucci, E. Gill, and C. Monteleone, "Extending the noel-v platform with a risc-v vector processor for space applications," *Journal of Aerospace Information Systems*, pp. 1–10, 2023.

[97] J. Andersson, "Development of a noel-v risc-v soc targeting space applications," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE Computer Society, 2020, pp. 66–67.

[98] M. Integrated, "Maxim integrated ai development," https://github.com/MaximIntegratedAI, 2023, (Accessed on 09/09/2024).

[99] W. W. Peterson and D. T. Brown, "Cyclic codes for error detection," *Proceedings of the IRE*, vol. 49, no. 1, pp. 228–235, 1961.

[100] G. Castagnoli, S. Brauer, and M. Herrmann, "Optimization of cyclic redundancy-check codes with 24 and 32 parity bits," *IEEE Transactions on Communications*, vol. 41, no. 6, pp. 883–892, 1993.

[101] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[102] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," University of Toronto, Toronto, Ontario, Tech. Rep., 2009.

[103] C. Amarnath, M. Mejri, K. Ma, and A. Chatterjee, "Error resilience in deep neural networks using neuron gradient statistics," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 4, pp. 1149–1162, 2024.

[104] L. M. Luza, A. Ruospo, D. Söderström, C. Cazzaniga, M. Kastriotou, E. Sanchez, A. Bosio, and L. Dilillo, "Emulating the effects of radiation-induced soft-errors for the reliability assessment of neural networks," *IEEE Transactions on Emerging Topics in Computing*, vol. 10, no. 4, pp. 1867–1882, 2021.

[105] F. Libano, P. Rech, B. Neuman, J. Leavitt, M. Wirthlin, and J. Brunhaver, "How reduced data precision and degree of parallelism impact the reliability of convolutional neural networks on fpgas," *IEEE Transactions on Nuclear Science*, vol. 68, no. 5, pp. 865–872, 2021.

[106] C. Imianosky, D. A. Santos, D. R. Melo, L. Dilillo, C. A. Zeferino, E. A. Bezerra, and F. Viel, "A Performance Evaluation of a Fault-tolerant RISC-V with Vector Instruction Support to Space Application," LASSS/LACW 2022 - Joint 3rd IAA Latin American Symposium on Small Satellites and 5th IAA Latin American CubeSat Workshop, Nov. 2022.

[107] RISC-V Organization, "Risc-v vector intrinsic document," https://github.com/riscv-non-isa/rvv-intrinsic-doc, 2024, (Accessed on 06/27/2024).

[108] RISC-V:Organization, "Gnu toolchain for risc-v, including gcc," https://github.com/riscv-collab/riscv-gnu-toolchain, 2023, (Accessed on 06/28/2023).

[109] J. L. Hennessy and D. A. Patterson, "Computer organization and design RISC-V edition: The hardware software interface," 2017.

[110] G. A. Reis, J. Chang, N. Vachharajani, S. S. Mukherjee, R. Rangan, and D. I. August, "Design and evaluation of hybrid fault-detection systems," in *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 2005, pp. 148–159.

[111] C. Imianosky, D. A. Santos, D. R. Melo, F. Viel, and L. Dilillo, "Special session: Reliability and performance evaluation of a risc-v vector extension unit for vector multiplication," in *2024 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2024, pp. 1–6.

[112] D. A. Santos, A. M. Mattos, D. R. Melo, and L. Dilillo, "Characterization of a fault-tolerant risc-v system-on-chip for space environments," in *2023 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 2023, pp. 1–6.