Développement d'une plate-forme multiprocesseur à base de cœurs RISC-V

Rédigé par :

Mahmoud EL HAJJ CHEHADE

Encadrants :

Sébastien LE NOURS
Antoine CANTIN

ETN 5 2025-2026

Remerciements

Je tiens tout d'abord à exprimer ma profonde gratitude à Monsieur **Sébastien Le Nours** pour son encadrement tout au long de ce projet. Ses conseils pertinents, sa disponibilité constante et son expertise technique ont été d'une aide précieuse. Grâce à son accompagnement rigoureux et à ses orientations méthodologiques, j'ai pu mener ce travail dans de bonnes conditions et atteindre les objectifs fixés.

Je souhaite également adresser mes sincères remerciements à Monsieur **Antoine Cantin** pour son soutien technique déterminant. Je le remercie tout particulièrement pour l'aide apportée au développement, notamment pour avoir guidé et fourni le code en langage C du réseau LeNet, aussi bien en version mono-cœur qu'en version dual-cœur. Ses explications claires et son accompagnement ont grandement facilité la compréhension et la mise en œuvre des solutions développées dans ce projet.

Résumé

Ce rapport présente la conception et l'évaluation d'une plateforme multiprocesseur basée sur des cœurs RISC-V, destinée à l'exécution d'algorithmes d'intelligence artificielle. Ce travail s'inscrit dans la continuité de la plateforme MicroBlaze existante au laboratoire IETR, avec pour objectif d'exploiter l'extensibilité de l'architecture RISC-V pour accélérer les calculs des réseaux de neurones convolutionnels.

Un état de l'art approfondi des cœurs RISC-V disponibles a permis de sélectionner deux candidats : le RI5CY, issu du projet PULP et le NEORV32. Ces deux cœurs ont été implémentés sur une carte ZedBoard (FPGA Zynq-7000) à l'aide de l'environnement Xilinx Vivado.

L'architecture mono-cœur a d'abord été validée en exécutant l'algorithme LeNet-5 sur chaque processeur, avec une méthode de chronométrage basée sur le registre CSR `mcycle`. Des optimisations ont ensuite été explorées, notamment l'utilisation des instructions PULP sur le RI5CY, permettant une réduction du nombre de cycles, et l'activation de l'unité de calcul en virgule flottante sur le NEORV32.

Le passage en configuration dual-cœur a été réalisé pour les deux processeurs, démontrant la faisabilité d'une exécution parallèle.

Mots-clés : RISC-V, FPGA, intelligence artificielle, multiprocesseur, LeNet-5, PULP, NEORV32, RI5CY, ZedBoard.

Abstract

This report presents the design and evaluation of a multiprocessor platform based on RISC-V cores, intended for the execution of artificial intelligence algorithms. This work builds upon the existing MicroBlaze platform at the IETR laboratory, aiming to leverage the extensibility of the RISC-V architecture to accelerate convolutional neural network computations.

A comprehensive survey of available RISC-V cores led to the selection of two candidates : RI5CY, derived from the PULP project and NEORV32. Both cores were deployed on a ZedBoard (Zynq-7000 FPGA) using the Xilinx Vivado design environment.

The single-core architecture was first validated by running the LeNet-5 algorithm on each processor, with performance measurement based on the `mcycle` CSR register. Optimizations were then explored, including the use of PULP instructions on the RI5CY, achieving reduction in cycle count, and the activation of the floating-point unit on the NEORV32.

The transition to a dual-core configuration was carried out for both processors, demonstrating the feasibility of parallel execution.

Keywords : RISC-V, FPGA, artificial intelligence, multiprocessor, LeNet-5, PULP, NEORV32, RI5CY, ZedBoard.

Table des matières

1 Introduction **5**
1.1 Contexte du Projet . . . 5
1.2 Objectifs du Projet . . . 5
1.3 Présentation de l'architecture RISC-V . . . 6
1.3.1 Qu'est-ce que RISC-V ? . . . 6
1.3.2 Formats d'instructions . . . 6
1.3.3 Extensions du RISC-V . . . 6
1.4 Spécifications Techniques . . . 7
1.5 Méthodologie de Sélection du Cœur RISC-V . . . 7
1.5.1 Critères de Sélection . . . 7
1.5.2 Processus de Sélection . . . 8

2 État de l'Art **9**
2.1 Phase 1 : Identification des Candidats . . . 9
2.2 Phase 2 : Filtrage par Compatibilité de Langage . . . 9
2.3 Phase 3 : Analyse Comparative Détaillée . . . 9
2.3.1 Grille d'Évaluation . . . 9
2.3.2 Matrice de Décision . . . 10
2.3.3 Analyse Détaillée des Candidats . . . 10
2.4 Conclusion et Sélection Finale . . . 12

3 Mise en Place de l'Architecture Mono-cœur **13**
3.1 Conception du Block Design . . . 13
3.1.1 Ressources utilisées . . . 13
3.1.2 Programmation du cœur . . . 15
3.2 Méthode de Chronométrage . . . 16
3.2.1 Test du chronométrage . . . 17
3.3 Validation Fonctionnelle . . . 17
3.4 Évaluation des Performances Mono-cœur . . . 18

4 Optimisation des Performances avec Instructions Spécialisées **19**
4.1 Instructions PULP . . . 19
4.2 Unité de Calcul en Virgule Flottante (FPU) . . . 21

5 Architecture Multi-cœurs **23**
5.1 NEORV32 en configuration dual-cœur . . . 23
5.1.1 Mise en place matérielle . . . 23
5.1.2 Programmation des deux cœurs . . . 24
5.2 RI5CY en configuration dual-cœur . . . 25
5.2.1 Mise en place matérielle . . . 25
5.2.2 Programmation des deux cœurs . . . 25
5.2.3 Comparaison des ressources mono-cœur vs dual-cœur . . . 26
5.3 Mémoire Partagée . . . 26
5.4 Évaluation des Performances Dual-cœur . . . 27
5.4.1 Stratégie de synchronisation par jeton mémoire . . . 27

5.4.2 Résultats RI5CY 28
5.4.3 Résultats NEORV32 28
5.5 Synthèse Comparative des Performances 30

6 Conclusion et axe de progression **32**
6.1 Axes de progression 32
6.1.1 Quantification du modèle en entier 8 bits 32
6.1.2 Migration vers le CV32E40P 32
6.1.3 Ajout d'instructions personnalisées sur le NEORV32 33
6.1.4 Parallélisme de données 33

Glossaire **34**

Table des figures

3.1.1 Block design du RI5CY avec architecture Harvard 13
3.1.2 Ressources matérielles utilisées par le RI5CY. 14
3.1.3 Ressources matérielles utilisées par le NEORV32 15
3.1.4 Block design du NEORV32 avec le Zynq-7000 16
3.1.5 Résultat de l'addition affiché sur le terminal du SDK 16
3.2.1 Comparaison des temps de calcul. 17
3.3.1 Architecture du réseau LeNet-5. 17
3.4.1 Comparaison des performances mono-cœur du LeNet-5 18

4.1.1 Extrait du désassemblage montrant les instructions PULP. 20
4.1.2 Comparaison des performances du RI5CY avec et sans instructions PULP 20
4.2.1 Extrait du désassemblage montrant les instructions FPU matérielles 21
4.2.2 Extrait du désassemblage montrant l'émulation logicielle du calcul flottant 22
4.2.3 Résultat de l'exécution du LeNet-5 sur le NEORV32 avec FPU activée 22

5.1.1 Ressources matérielles utilisées par le NEORV32 en configuration dual-cœur 23
5.1.2 Résultat du test dual-cœur sur le NEORV32 25
5.2.1 Block design du RI5CY en configuration dual-cœur 25
5.4.1 Résultat du LeNet-5 sur le RI5CY en configuration dual-cœur. 28
5.4.2 Résultat du NEORV32 en dual-cœur sans FPU 28
5.4.3 Résultat du NEORV32 en dual-cœur sans FPU (. 29
5.4.4 Architecture interne du NEORV32 en configuration dual-cœur 29
5.4.5 Résultats du NEORV32 en dual-cœur avec synchronisation par CLINT 30

Liste des tableaux

1.3.1 Extensions du jeu d'instructions RISC-V 7

2.3.1 Matrice de décision comparative des cœurs RISC-V 10

3.3.1 Comparaison de l'empreinte mémoire selon le format des poids 18

4.1.1 Impact des instructions PULP sur les performances du RI5CY 21
4.2.1 Impact de l'activation de la FPU sur les performances du NEORV32 22

5.2.1 Comparaison des ressources matérielles mono-cœur vs dual-cœur 26
5.4.1 Répartition des couches du LeNet-5 entre les deux cœurs 27
5.5.1 Synthèse comparative des performances sur le LeNet-5. 30

Chapitre 1

Introduction

L'intelligence artificielle transforme notre monde, mais elle demande des puissances de calcul énormes. Les processeurs classiques, bien qu'efficaces pour la plupart des tâches informatiques, peinent face aux besoins spécifiques des réseaux de neurones qui enchaînent sans fin des multiplications matricielles et des convolutions.

C'est là que RISC-V devient intéressant [1]. Contrairement aux architectures fermées d'Intel ou ARM, RISC-V permet d'ajouter des instructions personnalisées [2], parfaitement adaptées aux calculs d'IA. Au lieu d'enchaîner plusieurs instructions pour une multiplication matricielle, il devient possible de n'en utiliser qu'une seule, spécialement conçue pour cette opération.

1.1 Contexte du Projet

Le laboratoire IETR dispose déjà d'une plateforme multiprocesseur basée sur des cœurs MicroBlaze [3] pour l'exécution d'algorithmes d'intelligence artificielle. Cette plateforme a démontré l'efficacité du parallélisme pour accélérer les calculs de réseaux de neurones convolutionnels (CNN). Cependant, l'architecture MicroBlaze présente une limitation majeure : l'impossibilité d'ajouter des instructions personnalisées pour optimiser spécifiquement les opérations d'IA.

Opportunité RISC-V : L'architecture RISC-V, étant open-source, permet d'étendre le jeu d'instructions avec des opérations spécialisées pour l'IA, offrant un potentiel d'accélération significatif.

1.2 Objectifs du Projet

Ce projet vise à développer une plateforme multiprocesseur basée sur des cœurs RISC-V, constituant une évolution de la plateforme MicroBlaze existante au laboratoire IETR. L'objectif principal est d'évaluer et de comparer les performances d'exécution des algorithmes d'IA sur différentes configurations matérielles.

Démarche méthodologique : Réalisation d'un état de l'art des cœurs RISC-V disponibles, sélection de cœurs adaptés aux contraintes du projet, puis implémentation et benchmarking en configurations monoprocesseur et multiprocesseur.

Domaine d'application : Exécution d'algorithmes de réseaux de neurones convolutionnels (CNN) standards comme ResNet.

Architecture cible : Plateforme configurable en monoprocesseur ou multiprocesseur, permettant d'évaluer l'impact de la parallélisation sur les performances.

Innovation apportée : Exploration du potentiel d'intégration d'instructions personnalisées RISC-V pour accélérer les opérations critiques des CNN.

Métrique de performance : Mesure et analyse comparative des temps d'exécution entre les différentes configurations (monoprocesseur vs multiprocesseur, avec validation par rapport à la plateforme MicroBlaze).

1.3 Présentation de l'architecture RISC-V

1.3.1 Qu'est-ce que RISC-V ?

RISC-V est une architecture de processeur open-source, développée à l'Université de Californie à Berkeley en 2010. Contrairement aux architectures propriétaires comme ARM ou x86 qui nécessitent des licences coûteuses.

Le "V" dans RISC-V signifie "cinq" et fait référence à la cinquième génération de processeurs RISC développés à Berkeley. Mais c'est surtout une architecture pensée pour être simple, efficace et modulaire. L'idée est de proposer un jeu d'instructions minimal mais suffisant, que l'on peut ensuite enrichir selon les besoins spécifiques de chaque application [4].

1.3.2 Formats d'instructions

Il existe six formats de base :

Format R-type : pour les opérations entre registres. Par exemple, additionner le contenu de deux registres et mettre le résultat dans un troisième. C'est le format utilisé pour les instructions comme ADD, SUB, AND, OR, etc.

Format I-type : pour les opérations avec une valeur immédiate (une constante) ou pour charger des données depuis la mémoire.

Format S-type : pour stocker des données en mémoire.

Format B-type : pour les branchements conditionnels.

Format U-type : pour charger de grandes constantes (20 bits) dans un registre.

Format J-type : pour les sauts inconditionnels.

1.3.3 Extensions du RISC-V

La force de RISC-V réside dans sa modularité. Contrairement aux architectures monolithiques qui imposent un jeu d'instructions complet, RISC-V adopte une approche modulaire basée sur des extensions optionnelles. Cette conception permet aux concepteurs de processeurs de sélectionner précisément les fonctionnalités requises pour leur domaine d'application, évitant ainsi la complexité et le surcoût associés aux instructions superflues.

Extension	Nom complet	Description
I	Integer	Les instructions de base. Arithmétique simple, logique, mémoire, branchements.
E	Embedded	Version light de I avec seulement 16 registres au lieu de 32.
M	Multiplication	Ajoute la multiplication et la division matérielles.
A	Atomic	Instructions atomiques pour gérer plusieurs cœurs qui accèdent à la même mémoire.
F	Float	Calculs en virgule flottante simple précision (32 bits).
D	Double	Virgule flottante double précision (64 bits).
C	Compressed	Compresse les instructions de 32 à 16 bits quand c'est possible.
G	General	Juste un raccourci pour dire IMAFD. C'est le pack "complet" pour une utilisation générale.
V	Vector	Instructions vectorielles pour faire des calculs parallèles.

Table 1.3.1 – Extensions du jeu d'instructions RISC-V

1.4 Spécifications Techniques

Ce projet nécessite des cœurs dotés d'instructions spécialisées pour l'IA. Les opérations de convolution 2D et de multiplication matricielle, présentes dans les CNN, doivent être accélérées matériellement pour obtenir des performances acceptables.

Le choix se portera donc sur des cœurs offrant un équilibre entre performance, extensibilité et compatibilité avec l'environnement de développement Xilinx Vivado.

Jeu d'instructions

Les cœurs RISC-V sélectionnés doivent intégrer des instructions spécialisées pour l'IA, notamment des multiplications vectorielles accélérées matériellement, adaptées aux calculs parallèles.

Contraintes d'Implémentation

Plateforme de développement : Les cœurs doivent être compatibles avec Xilinx Vivado pour permettre l'intégration sur FPGA.

Toolchain : Un environnement de développement complet est requis, comprenant un compilateur C capable de générer du code pour les instructions spécialisées IA.

Le compilateur doit reconnaître et exploiter automatiquement les instructions IA spécifiques du processeur.

1.5 Méthodologie de Sélection du Cœur RISC-V

1.5.1 Critères de Sélection

La sélection du cœur RISC-V approprié repose sur trois critères principaux :

Présence d'Instructions Personnalisées pour IA (40%)

- Présence d'instructions spécialisées pour l'IA
- Support natif des opérations matricielles et vectorielles

Compatibilité du Multiprocesseur (30%)

- Facilité de réplication en configuration multi-cœur
- Support des mécanismes de communication inter-cœurs
- Gestion de la cohérence mémoire

Maturité de l'Écosystème (30%)

- Chaîne de compilation C complète et stable
- Outils de débogage disponibles
- Documentation et exemples d'utilisation

1.5.2 Processus de Sélection

Le processus de sélection s'effectue en trois phases progressives :
Phase 1 : Identification exhaustive des cœurs RISC-V open-source disponibles.
Phase 2 : Élimination des candidats ne respectant pas les contraintes techniques minimales.
Phase 3 : Analyse comparative approfondie selon les critères pondérés.

Cette méthodologie systématique garantit une sélection objective du cœur RISC-V le mieux adapté aux spécificités du projet. L'analyse détaillée est présentée dans le chapitre suivant.

Chapitre 2

État de l'Art

Ce chapitre présente une analyse comparative des cœurs RISC-V disponibles selon les critères établis dans l'introduction. L'objectif est d'identifier le cœur le mieux adapté pour remplacer les cœurs MicroBlaze dans notre plateforme multiprocesseur.

Note importante : Cette analyse se concentre exclusivement sur les **cœurs de processeur** (CPU cores), pas sur les SoCs complets. Un cœur est le composant central qui exécute les instructions, tandis qu'un SoC (System-on-Chip) intègre un cœur avec des périphériques, mémoires et bus système.

2.1 Phase 1 : Identification des Candidats

L'écosystème RISC-V [5], [6] offre une diversité importante de cœurs avec des caractéristiques variées. Nous avons listé les cœurs les plus pertinents pour notre application, en nous concentrant sur ceux qui respectent les contraintes fondamentales du projet.

2.2 Phase 2 : Filtrage par Compatibilité de Langage

Parmi les nombreux cœurs RISC-V identifiés, une première étape de filtrage s'impose basée sur la compatibilité avec notre environnement de développement Xilinx Vivado. Le langage de description matérielle utilisé pour implémenter un cœur détermine directement sa capacité à être synthétisé sur nos FPGAs Xilinx.

Les langages compatibles avec Vivado sont principalement le VHDL, le Verilog et le SystemVerilog. En revanche, des langages comme Chisel ,Bluespec, PyMTL, nMigen, SpinalHDL, CDL ou TL-Verilog nécessitent des chaînes de compilation spécifiques et ne garantissent pas une intégration directe dans Vivado.

En appliquant ce critère, on se retrouve à écarter plusieurs candidats pourtant très intéressants sur le plan technique. Les cœurs Rocket et Freedom de SiFive, par exemple, sont largement utilisés dans la communauté RISC-V, mais ils sont écrits en Chisel, ce qui les rend incompatibles avec notre environnement Vivado.

Cette première phase d'élimination réduit considérablement le nombre de candidats mais nous assure une compatibilité totale avec Vivado.

2.3 Phase 3 : Analyse Comparative Détaillée

Après ce filtrage initial, nous procédons à une analyse approfondie des cœurs restants selon les critères pondérés définis précédemment. Cette évaluation s'appuie sur la documentation disponible, les exemples d'utilisation et les retours de la communauté.

2.3.1 Grille d'Évaluation

Pour chaque critère, une note sur 10 est attribuée selon la méthodologie suivante :

Présence d'Instruction Personnalisées pour IA (40%) : Cette note reflète la disponibilité et l'utilisabilité des instructions personnalisées déjà présentes dans le cœur. Un cœur

obtient 10/10 s'il dispose d'instructions custom bien documentées, d'exemples concrets d'utilisation et d'une architecture permettant de les exploiter facilement. Un cœur obtient 0/10 s'il ne propose aucune instruction personnalisée ou si celles-ci sont présentes mais sans documentation ni exemples d'utilisation.

Compatibilité multiprocesseur (30%) : La note évalue la capacité du cœur à fonctionner en configuration multi-cœurs. Un 10/10 est attribué si le cœur supporte nativement les instructions atomiques (extension A), dispose d'exemples multiprocesseurs documentés et d'interfaces standardisées pour la cohérence mémoire. Un 0/10 correspond à un cœur sans support atomique ni documentation multiprocesseur.

Maturité de l'écosystème (30%) : Cette note traduit la disponibilité et la qualité des outils de développement. Un cœur obtient 10/10 s'il possède une chaîne de compilation C complète, des outils de débogage intégrés, une documentation exhaustive et une communauté active avec des mises à jour régulières. Un 0/10 indique l'absence de toolchain standard ou une documentation minimale.

2.3.2 Matrice de Décision

Cœur (lien)	Instr. Custom (/10)	Multiproc. (/10)	Écosystème (/10)	Note finale
CVA6	8	10	10	9.2
CV32E40P	9	7	10	8.7
RI5CY	10	6	9	8.5
SweRV EH2	7	9	9	8.1
NOEL-V	7	9	8	7.9
NEORV32	8	6	9	7.7
SweRV EH1	6	8	9	7.5
Taiga	7	8	7	7.3
SweRV EL2	6	7	9	7.2
SCR1	7	6	8	7.0
Klessydra-T13	7	7	7	7.0
Ibex	6	5	9	6.6
MicroBlaze V	2	7	10	5.9
RSD	6	5	6	5.7
Steel	6	5	6	5.7
DarkRISCV	6	4	6	5.4
biRISC-V	5	5	6	5.3
PicoRV32	5	3	7	5.0
Tinyriscv	5	4	5	4.7
ReonV	4	4	5	4.3
Maestro	4	3	4	3.7
SERV	3	2	6	3.5

TABLE 2.3.1 – Matrice de décision comparative des cœurs RISC-V

Comme le montre le tableau 2.3.1, trois cœurs se détachent nettement avec des notes supérieures à 8.5/10 : CVA6 (9.2/10), CV32E40P (8.8/10) et RI5CY (8.5/10), confirmant leur pertinence pour notre application.

2.3.3 Analyse Détaillée des Candidats

Le cœur CVA6 [7] de l'OpenHW Group [8] se distingue nettement avec une note finale de 9.2/10. Son architecture est particulièrement bien documentée pour l'ajout d'extensions personnalisées, avec plusieurs exemples concrets dans la littérature académique. La note de 8/10 en extensibilité reflète cette qualité de documentation, même si l'architecture 6-stages pipeline reste plus complexe à modifier qu'un design plus simple. En multiprocesseur, il a obtenu 10/10 grâce à son support complet des instructions atomiques, des exemples de configurations multi-cœurs déjà déployés et une interface AXI4 standardisée facilitant la cohérence mémoire. L'écosystème autour de CVA6 est exceptionnel avec un 10/10, bénéficiant du support de l'OpenHW Group qui

maintient une toolchain complète, des exemples FPGA Xilinx et une communauté très active. Ses performances de 2.1 CoreMark/MHz en font un processeur performant.

Le CV32E40P [9], également de l'OpenHW Group, obtient 8.8/10 et représente une alternative très solide. Sa note d'extensibilité de 9/10, légèrement supérieure à CVA6, s'explique par une architecture 4-stages plus simple à modifier et une documentation exceptionnelle sur l'ajout d'instructions. En multiprocesseur, il marque 7/10 car bien qu'il supporte les instructions atomiques de base, ses exemples multi-cœurs sont moins nombreux que CVA6. Son écosystème est au même niveau que CVA6 avec 10/10, bénéficiant des mêmes outils OpenHW Group.

RI5CY [10], le prédécesseur de CV32E40P développé initialement par ETH Zurich et l'Université de Bologna dans le cadre du projet PULP [11], [12], obtient une note remarquable de 8.5/10 qui le place en troisième position. Son extensibilité exceptionnelle mérite un 10/10 car il a été spécifiquement conçu dès l'origine pour le traitement du signal numérique et l'IA. L'architecture en SystemVerilog est particulièrement modulaire avec des interfaces clairement définies pour l'intégration de nouvelles unités d'exécution. En multiprocesseur, il obtient 6/10 car bien qu'il n'intègre pas les instructions atomiques, il a été utilisé dans des configurations multi-cœurs au sein du projet PULP. Toutefois, les exemples et la documentation restent moins nombreux que pour CVA6, le positionnant davantage comme un accélérateur que comme un processeur d'application multi-cœurs. Son écosystème marque 9/10 grâce au projet PULP qui fournit une toolchain complète avec des extensions GCC pour les instructions personnalisées, une documentation détaillée et une communauté académique active, même si celle-ci est moins large que celle d'OpenHW Group.

Le SweRV EH2 de Western Digital obtient 8.2/10 grâce à ses excellentes capacités multiprocesseurs. Sa note de 7/10 en extensibilité provient d'une architecture complexe à deux threads simultanés (SMT) qui rend les modifications plus délicates, mais compensée par une documentation technique détaillée de Western Digital. En multiprocesseur, il excelle avec 9/10 car il est spécifiquement conçu pour les systèmes multi-cœurs avec un support atomique robuste et des exemples industriels. L'écosystème à 9/10 bénéficie du soutien de Western Digital et d'une toolchain mature, bien que légèrement moins communautaire qu'OpenHW.

NOEL-V de Cobham Gaisler atteint 7.9/10 et se démarque par sa fiabilité industrielle. Son extensibilité de 7/10 reflète une architecture bien structurée mais avec moins d'exemples d'extensions personnalisées publiés. En multiprocesseur, il obtient 9/10 grâce à son héritage LEON et son utilisation dans des systèmes spatiaux multi-cœurs critiques. L'écosystème à 8/10 est professionnel avec des outils commerciaux de qualité, mais la communauté open-source est plus restreinte.

NEORV32 [13] obtient 7.7/10 avec une excellente extensibilité de 8/10. Son architecture modulaire en VHDL est particulièrement claire et bien documentée, facilitant grandement l'ajout d'instructions personnalisées. Cependant, son score multiprocesseur de 6/10 révèle une limitation : le NEORV32 est configurable uniquement en versions monocœur ou double cœur (dual-core), restreignant son évolutivité au-delà de deux unités de traitement. Son écosystème à 9/10 est remarquable pour un projet communautaire, avec une documentation exhaustive, des exemples pour Xilinx et une communauté GitHub très active.

Le MicroBlaze V [14] d'AMD/Xilinx obtient une note de 5.6/10, ce qui peut surprendre au premier abord pour une solution commerciale aussi bien intégrée. Le problème principal réside dans son extensibilité quasi nulle, d'où sa note de 2/10 sur ce critère pourtant crucial pour notre projet. Contrairement aux cœurs open-source, MicroBlaze V est une IP propriétaire dont le code source RTL reste inaccessible. Certes, AMD propose une ISA configurable qui permet d'activer ou désactiver à la carte plusieurs extensions RISC-V : le jeu d'instructions de base (RV32I/RV64I), la multiplication et division (extension M), les instructions atomiques (extension A), le calcul en virgule flottante (extension F), la compression de code (extension C), et même la manipulation de bits (extensions B, ZBa, ZBb, ZBc et ZBs). Malheureusement, impossible d'aller au-delà de ce

catalogue prédéfini. Sans accès au RTL, impossible de créer les extensions sur mesure pour l'IA. En multiprocesseur, MicroBlaze V s'en sort mieux avec 7/10. Les configurations multi-cœurs via AXI4 sont documentées et fonctionnent correctement dans Vivado, même si les exemples pratiques restent moins nombreux que ce qu'on trouve dans la communauté open-source autour de CVA6 par exemple. L'écosystème, lui, mérite largement son 10/10. C'est probablement le point fort de MicroBlaze V : l'intégration avec Vivado Design Suite et Vitis est parfaite. La toolchain GCC est optimisée, les bibliothèques pour les périphériques Xilinx sont prêtes à l'emploi, et le support technique commercial d'AMD est réactif quand on a un problème. Pour un projet classique nécessitant un processeur embarqué standard, ce serait un choix tout à fait pertinent.

2.4 Conclusion et Sélection Finale

Au terme de cette analyse comparative, le **CVA6** apparaît théoriquement comme le candidat optimal pour notre plateforme multiprocesseur d'IA avec sa note de 9.2/10. Cependant, la complexité inhérente à son pipeline à 6 étages s'est avérée être une contrainte trop importante pour l'intégration agile de nos instructions personnalisées. Le **CV32E40P** constituait une alternative séduisante (8.8/10), mais les tentatives préliminaires d'intégration dans notre environnement de développement spécifique n'ont pas abouti à un fonctionnement stable, nous contraignant à l'écarter.

Par conséquent, notre choix final se porte le **RI5CY** et le **NEORV32**. Avec une note de 8.5/10, le RI5CY représente un choix particulièrement pertinent pour notre projet d'accélération d'IA grâce à son héritage du projet PULP et ses capacités DSP natives. De son côté, le NEORV32 (7.7/10), bien que légèrement en retrait sur le score global, a démontré une robustesse et une facilité de mise en œuvre immédiate.

Chapitre 3

Mise en Place de l'Architecture Mono-cœur

Ce chapitre présente la conception et la validation de l'architecture matérielle mono-cœur pour les deux processeurs retenus : le RI5CY et le NEORV32. L'objectif est de mettre en place un environnement fonctionnel permettant d'exécuter un algorithme d'IA sur chaque cœur, puis de mesurer leurs performances.

3.1 Conception du Block Design

Pour fonctionner correctement, chaque cœur nécessite une mémoire BRAM pour stocker les données et les instructions. L'architecture retenue est de type Harvard, qui consiste à séparer la mémoire en deux BRAM distinctes : l'une dédiée aux instructions et l'autre aux données. Cette séparation offre une meilleure organisation et permet des accès simultanés aux instructions et aux données, améliorant ainsi les performances globales du système.

Chaque BRAM nécessite un contrôleur avec une interface AXI. Un bus AXI assure la communication entre le cœur et les mémoires, tout en permettant l'ajout de périphériques supplémentaires. Pour faciliter les tests et le débogage, un module GPIO relié aux LEDs de la ZedBoard a été intégré au design.

Le block design du RI5CY est illustré dans la figure 3.1.1 .

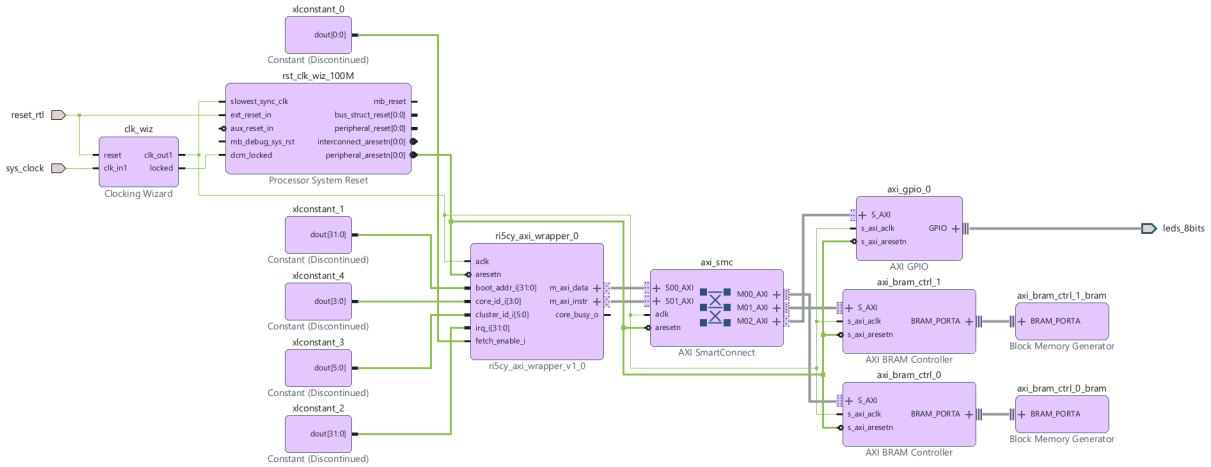

Figure 3.1.1 – Block design du RI5CY avec architecture Harvard

Le design du NEORV32 suit la même structure globale, seul le cœur de processeur change.

3.1.1 Ressources utilisées

Pour le RI5CY, après synthèse, le design utilise 23% des LUT de la ZedBoard. Concernant la mémoire, avec 64 Ko de BRAM pour les données et 256 Ko de BRAM pour les instructions,

l'utilisation atteint 57,14%, soit 80 blocs BRAM sur les 140 disponibles, comme le montre la figure 3.1.2 .

Resource	Utilization	Available	Utilization %
LUT	14054	53200	26.42
FF	11301	106400	10.62
BRAM	80	140	57.14
DSP	7	220	3.18
IO	9	200	4.50

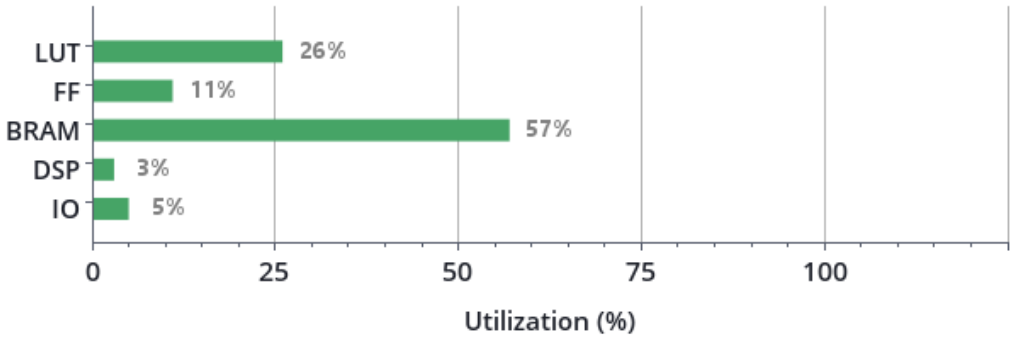

FIGURE 3.1.2 – Ressources matérielles utilisées par le RI5CY

Pour le NEORV32, le design utilise 19,19% des LUT. Avec la même configuration de BRAM, on obtient 85 blocs utilisés, soit 61% du total disponible, comme l'illustre la figure 3.1.3 .

Summary

Resource	Utilization	Available	Utilization %
LUT	10213	53200	19.20
FF	10556	106400	9.92
BRAM	85	140	60.71
DSP	6	220	2.73
IO	9	200	4.50

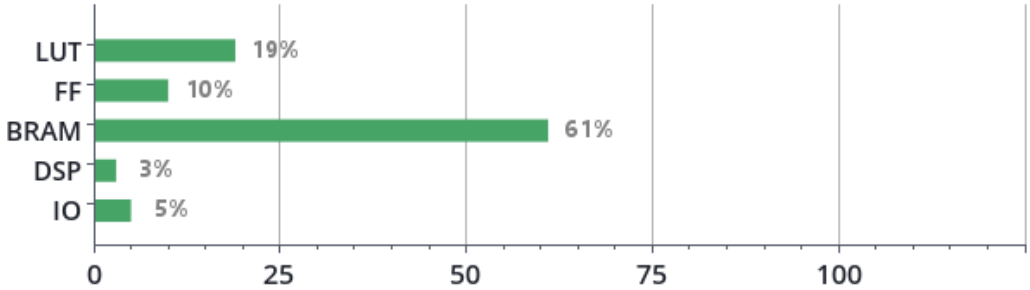

FIGURE 3.1.3 – Ressources matérielles utilisées par le NEORV32

3.1.2 Programmation du cœur

Après la génération du bitstream pour configurer le FPGA, il est nécessaire de charger le programme dans le cœur RISC-V. Parmi les différentes méthodes possibles, la solution retenue exploite le processeur Zynq-7000 présent sur la ZedBoard, qui écrit le programme directement dans la BRAM.

L'avantage principal de cette approche est qu'il n'est pas nécessaire de relancer la synthèse à chaque modification du programme, il suffit de reprogrammer le Zynq-7000 via le SDK. Le seul inconvénient est que, en cas de changement de carte, il faut adapter le design à la nouvelle plateforme et disposer d'un processeur programmable pour charger le programme.

Une fois le Zynq intégré, le design complet est présenté dans la figure 3.1.4 .

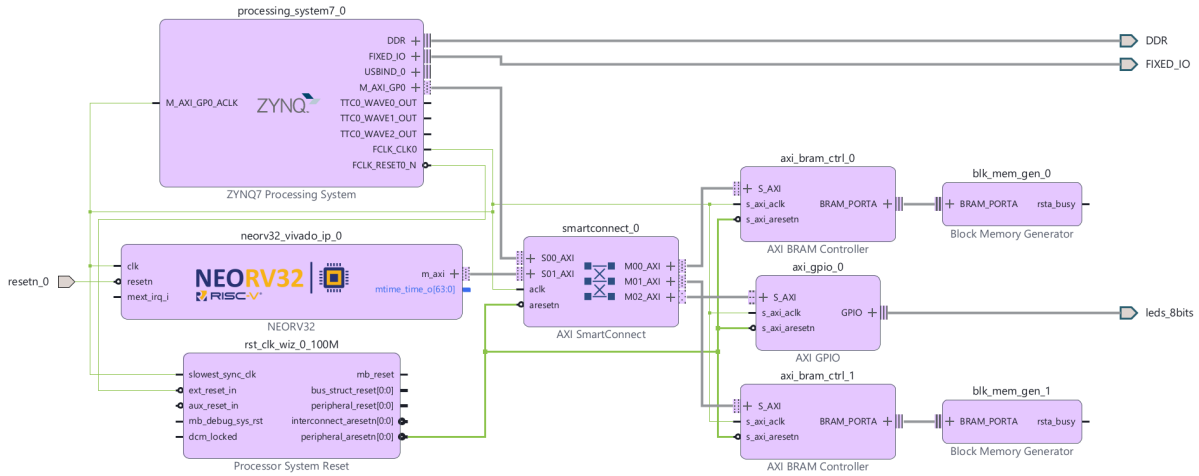

FIGURE 3.1.4 – Block design du NEORV32 avec le Zynq-7000

Chaîne de compilation : Le code source en C est compilé à l'aide du compilateur RISC-V GCC 7, qui supporte les instructions PULP pour le RI5CY. Pour le NEORV32, il suffit de sélectionner les extensions appropriées lors de la compilation. Le fichier ELF obtenu est ensuite converti au format HEX, permettant au Zynq de le charger en mémoire.

Pour valider le design, un programme simple effectuant une addition de deux nombres a été utilisé. Le résultat est écrit dans la BRAM de données, puis lu par le Zynq et affiché sur le terminal du SDK. La figure 3.1.5 confirme l'exactitude du résultat.

```
Demarrage du coeur
Operation: 10 + 20
Résultat attendu :30
Résultat lu : 30

Le calcul est correct.
```

FIGURE 3.1.5 – Résultat de l'addition affiché sur le terminal du SDK

Ce test a été validé avec succès sur les deux architectures : le RI5CY et le NEORV32.

3.2 Méthode de Chronométrage

Une fois l'architecture vérifiée et programmable, il est nécessaire de disposer d'une méthode fiable pour mesurer le temps de calcul du processeur. Cette mesure permet d'analyser et de comparer les performances des cœurs en fonction du nombre de cycles d'horloge.

Les processeurs RISC-V disposent d'un registre CSR interne, le registre `mcycle`, qui compte le nombre de cycles d'horloge écoulés depuis l'activation du cœur. En lisant ce registre avant et après un calcul, puis en effectuant la différence, on peut déterminer précisément le nombre de cycles nécessaires à l'exécution d'une portion de code. Cette méthode est universelle pour tous les cœurs RISC-V, car le registre `mcycle` fait partie des CSR obligatoires de l'architecture.

3.2.1 Test du chronométrage

Pour valider cette méthode, un programme effectuant une addition de deux nombres flottants a été exécuté sur chaque cœur. Le nombre de cycles et le temps d'exécution sont mesurés à l'aide de la formule suivante :

$$\text{Temps d'exécution (en secondes)} = \frac{\text{Nombre de cycles}}{\text{Fréquence du processeur (en Hz)}} \tag{3.1}$$

Les résultats sont présentés dans les figures 3.2.1 a et 3.2.1 b.

```
Demarrage du RISC-V...
Cycles mesurés pour RI5CY : 25 cycles
Résultat      : 10.3000
Temps mesuré  : 0.00065 ms
```

(a) Temps de calcul pour le RI5CY

```
Demarrage du NEORV32 ...
Cycles mesurés : 31 cycles
Résultat       : 10.3000
Temps mesuré   : 0.00031 ms
```

(b) Temps de calcul pour le NEORV32

FIGURE 3.2.1 – Comparaison des temps de calcul

Les résultats montrent que le RI5CY nécessite 25 cycles pour effectuer le calcul, ce qui correspond à 0.65 µs, tandis que le NEORV32 requiert 31 cycles, soit 0.31 µs. Bien que le RI5CY effectue le calcul en moins de cycles, il est plus lent en temps absolu en raison de sa fréquence d'horloge plus basse. Cette différence de fréquence s'explique par des violations de timing de type WHS (Worst Hold Slack) qui contraignent la fréquence maximale atteignable par le RI5CY sur la ZedBoard.

3.3 Validation Fonctionnelle

Une fois tous les éléments en place, l'étape suivante consiste à exécuter un algorithme d'IA complet. Le réseau LeNet-5 a été retenu pour cette validation. Il s'agit d'un CNN composé de 7 couches. Son architecture, illustrée dans la figure 3.3.1 , comprend trois couches de convolution,deux couches de sous-échantillonnage (max-pooling), puis deux couches fully connected.

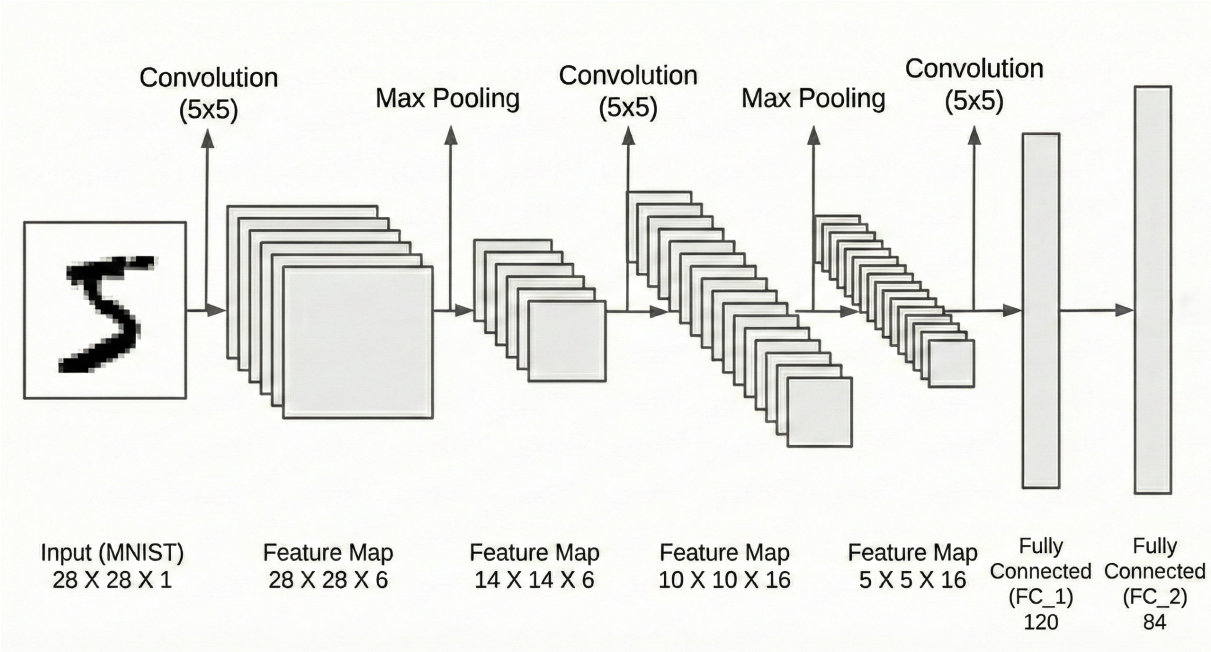

FIGURE 3.3.1 – Architecture du réseau LeNet-5

Lors de la compilation du code de l'algorithme, un problème de taille mémoire est apparu : la configuration de BRAM utilisée n'était pas suffisante pour contenir l'ensemble du programme avec les poids en `float32`. Les ressources BRAM disponibles sur la ZedBoard ne permettant pas d'augmenter la capacité mémoire, la solution retenue a été de passer les poids en `float16`. Le compilateur GCC 7 ne supportant pas ce format, un passage au compilateur xPack GCC 13 a été nécessaire.

Cette optimisation a permis de réduire significativement l'empreinte mémoire du programme :

Format des poids	Instructions (octets)	Données (octets)
`float32`	264 340	46 464
`float16`	150 768	46 464

Table 3.3.1 – Comparaison de l'empreinte mémoire selon le format des poids

Grâce à cette réduction, le programme peut désormais être chargé dans la configuration BRAM disponible, permettant ainsi l'exécution complète du LeNet-5 sur les deux cœurs.

3.4 Évaluation des Performances Mono-cœur

Avec l'architecture validée et la méthode de chronométrage en place, nous avons exécuté le réseau LeNet-5 complet sur chaque cœur en configuration mono-cœur. Les résultats obtenus sont présentés dans les figures 3.4.1 a et 3.4.1 b.

```
Demarrage RI5CY..
Prediction : 2
Reference  : 2
Nb de cycle : 1773968467
Temps de calcul : 46.123
```

(a) Résultat du LeNet-5 sur le RI5CY

```
Démarrage du Neorv32...
Prediction : 2
Reference  : 2
Nb de cycle : 2573513365
Temps de calcul : 25.735
```

(b) Résultat du LeNet-5 sur le NEORV32 (sans FPU)

Figure 3.4.1 – Comparaison des performances mono-cœur du LeNet-5

Les deux cœurs produisent une prédiction correcte, ce qui valide le bon fonctionnement de l'algorithme sur les deux architectures. Au niveau des performances, le NEORV32 (sans FPU activée) exécute le LeNet-5 en environ 2 573 513 365 cycles (25,7 secondes), tandis que le RI5CY réalise l'inférence en environ 46,1 secondes (1 773 968 467 cycles). Cela montre que le RI5CY est plus rapide en termes de cycles d'environ 31% par rapport au NEORV32, cela est dû à son architecture à 4 étages, comportant un étage de plus que le NEORV32. Ces résultats servent de référence pour les optimisations présentées dans les chapitres suivants.

Chapitre 4

Optimisation des Performances avec Instructions Spécialisées

Ce chapitre explore deux axes d'optimisation matérielle pour accélérer l'exécution des algorithmes de CNN : l'utilisation des instructions spécialisées PULP sur le RI5CY et l'activation de l'unité de calcul en virgule flottante (FPU) sur le NEORV32.

4.1 Instructions PULP

Le processeur RI5CY, issu du projet PULP, implémente un jeu d'instructions spécialisées conçu pour accélérer les calculs intensifs. Parmi ces instructions, on retrouve notamment l'opération MAC (*Multiply-Accumulate*), qui constitue l'opération fondamentale des réseaux de neurones convolutionnels.

L'intérêt de ces instructions réside dans leur capacité à regrouper plusieurs opérations en une seule instruction matérielle. Par exemple, une opération de multiplication suivie d'une accumulation, qui nécessiterait normalement deux instructions distinctes, est exécutée en un seul cycle grâce à l'instruction `p.mac`. De plus, les instructions de chargement (`p.lw`) et de stockage (`p.sw`) optimisées du jeu PULP réduisent la latence des accès mémoire grâce au post-incrément automatique de l'adresse, évitant ainsi une instruction d'addition supplémentaire.

Ces optimisations apportent un double avantage : d'une part, une réduction du nombre d'instructions à exécuter, ce qui diminue le temps de calcul et d'autre part, une réduction de la taille du code compilé.

En compilant le LeNet-5 avec le GCC 7 (supportant les instructions PULP), la taille du programme passe de 264 340 octets à 264 276 octets pour les instructions.

La présence des instructions PULP peut être vérifiée en désassemblant le fichier ELF, comme le montre la figure 4.1.1 .

```
44002882:       21b47503        p.lw    a0,s11(s0)
44002886:       c61a            sw      t1,12(sp)
44002888:       c846            sw      a7,16(sp)
4400288a:       c47a            sw      t5,8(sp)
4400288c:       ca76            sw      t4,20(sp)
4400288e:       3251            jal     44002212 <__mulsf3>
44002890:       4ed2            lw      t4,20(sp)
44002892:       85aa            mv      a1,a0
44002894:       0d91            addi    s11,s11,4
44002896:       8576            mv      a0,t4
44002898:       d36ff0ef        jal     ra,44001dce <__addsf3>
4400289c:       47d1            li      a5,20
4400289e:       8eaa            mv      t4,a0
440028a0:       4332            lw      t1,12(sp)
440028a2:       48c2            lw      a7,16(sp)
440028a4:       4f22            lw      t5,8(sp)
440028a6:       fcfd9ce3        bne     s11,a5,4400287e <L3_Conv_2+0x44>
440028aa:       0885            addi    a7,a7,1
440028ac:       03840413        addi    s0,s0,56 # 800038 <_sdata-0x417fffc8>
440028b0:       0351            addi    t1,t1,20
440028b2:       0058a463        p.beqimm        a7,5,440028ba <L3_Conv_2+0x80>
440028b6:       4d81            li      s11,0
440028b8:       b7d9            j       4400287e <L3_Conv_2+0x44>
440028ba:       0d05            addi    s10,s10,1
440028bc:       026d2163        p.beqimm        s10,6,440028de <L3_Conv_2+0xa4>
440028c0:       0c400793        li      a5,196
440028c4:       847a            mv      s0,t5
440028c6:       42fd0433        p.mac   s0,s10,a5
440028ca:       47e2            lw      a5,24(sp)
440028cc:       4372            lw      t1,28(sp)
440028ce:       4881            li      a7,0
440028d0:       040a            slli    s0,s0,0x2
440028d2:       943e            add     s0,s0,a5
440028d4:       06400793        li      a5,100
440028d8:       42fd0333        p.mac   t1,s10,a5
440028dc:       bfe9            j       440028b6 <L3_Conv_2+0x7c>
440028de:       408c            lw      a1,0(s1)
440028e0:       ceeff0ef        jal     ra,44001dce <__addsf3>
440028e4:       00000593        li      a1,0
440028e8:       842a            mv      s0,a0
440028ea:       89bff0ef        jal     ra,44002184 <__lesf2>
440028ee:       00055463        bgez    a0,440028f6 <L3_Conv_2+0xbc>
440028f2:       00000413        li      s0,0
440028f6:       008ca22b        p.sw    s0,4(s9!)
```

FIGURE 4.1.1 – Extrait du désassemblage montrant les instructions PULP

On distingue dans cette figure les instructions spécifiques PULP, identifiables par le préfixe `p.` :

1. `p.lw` : instruction de chargement avec post-incrément.
2. `p.beqimm` : instruction de branchement conditionnel avec immédiat.
3. `p.mac` : instruction de multiplication-accumulation.
4. `p.sw` : instruction de stockage avec post-incrément.

Pour évaluer l'impact des instructions PULP sur les performances, nous avons réalisé un test comparatif avec le compilateur GCC 7, qui permet d'activer ou de désactiver la génération des instructions PULP. Pour contourner les limitations de mémoire liées au format `float32` imposé par GCC 7, le benchmark a été limité aux deux premières couches du LeNet-5, la convolution 1 et le max-pooling 1 (L1 et L2). Cette restriction permet d'isoler l'influence des instructions PULP sans introduire le problème lié à la mémoire.

Les résultats obtenus sans et avec les instructions PULP sont présentés respectivement dans les figures 4.1.2 a et 4.1.2 b.

```
Demarrage RI5CY..
L1 et l2 sans instruction pulp
Nb de cycle : 461827379
Temps de calcul : 12.008
```

(a) RI5CY sans instructions PULP

```
Demarrage RI5CY..
L1 et l2 avec instruction pulp
Nb de cycle : 420668666
Temps de calcul : 10.937
```

(b) RI5CY avec instructions PULP

FIGURE 4.1.2 – Comparaison des performances du RI5CY avec et sans instructions PULP

Les résultats confirment l'apport significatif des instructions PULP. Sans ces instructions, l'exécution des deux premières couches (convolution 1 et max-pooling 1) nécessite 461 827 379 cycles. Avec les instructions PULP activées, ce nombre descend à 420 668 666 cycles, soit une réduction d'environ 8,9%. Ce gain provient principalement des opérations `p.mac` qui fusionnent la multiplication et l'accumulation en un seul cycle, et des instructions `p.lw`/`p.sw` avec post-incrément qui évitent les instructions d'addition d'adresse supplémentaires.

Bien que ce test soit réalisé sur un sous-ensemble du réseau (deux couches sur sept), il est représentatif du gain attendu sur l'ensemble du LeNet-5, car les couches de convolution concentrent la très grande majorité des opérations de multiplication-accumulation.

Configuration	**Cycles (L1+L2)**	**Taille code (octets)**	**Gain de cycles**
RI5CY sans PULP	46 827 379	264 340	–
RI5CY avec PULP	42 668 666	264 276	∼8,9%

TABLE 4.1.1 – Impact des instructions PULP sur les performances du RI5CY

4.2 Unité de Calcul en Virgule Flottante (FPU)

Le RI5CY mentionne dans sa documentation la possibilité d'intégrer une FPU. Cependant, l'implémentation de cette unité s'est avérée peu documentée et difficile à mettre en œuvre. Son successeur, le CV32E40P, dispose en revanche d'une documentation plus complète pour l'ajout ou le retrait de modules optionnels.

Le NEORV32, quant à lui, intègre nativement une FPU activable. En ajoutant l'extension `Zfinx` au compilateur, les calculs en virgule flottante sont exécutés directement par le matériel au lieu d'être émulés par logiciel. Cette extension utilise les registres entiers existants pour les opérations flottantes, évitant ainsi la nécessité d'un banc de registres flottants séparé.

L'activation de la FPU avec l'extension `Zfinx` permet de réduire la taille du programme : les instructions passent de 150 768 octets (avec émulation logicielle en `float16`) à 148 576 octets.

```
440001c0:       16f80363        beq     a6,a5,44000326 <L1_Conv_1+0x1a2>
440001c4:       4394            lw      a3,0(a5)
440001c6:       4310            lw      a2,0(a4)
440001c8:       10c6f4d3        fmul.s  s1,a3,a2
440001cc:       00a4f6d3        fadd.s  a3,s1,a0
440001d0:       43d0            lw      a2,4(a5)
440001d2:       4344            lw      s1,4(a4)
440001d4:       10967653        fmul.s  a2,a2,s1
440001d8:       00d67653        fadd.s  a2,a2,a3
440001dc:       4794            lw      a3,8(a5)
440001de:       4704            lw      s1,8(a4)
440001e0:       1096f6d3        fmul.s  a3,a3,s1
440001e4:       00c6f6d3        fadd.s  a3,a3,a2
```

FIGURE 4.2.1 – Extrait du désassemblage montrant les instructions FPU matérielles

La figure 4.2.1 montre les instructions `fmul.s` et `fadd.s` qui utilisent directement l'unité de calcul en virgule flottante. Ces instructions permettent d'effectuer les multiplications et additions flottantes au niveau matériel en un seul cycle d'horloge. Sans FPU, comme on peut le voir sur la figure 4.2.2 , les opérations flottantes passent par des fonctions d'émulation logicielle (`__mulsf3` pour la multiplication et `__addsf3` pour l'addition). Chaque opération nécessite alors un saut vers la fonction (Jump and Link `jal`), la sauvegarde et restauration des registres, puis l'exécution de plusieurs dizaines d'instructions pour simuler l'opération flottante.

```
440001e8:   1c8a0963   beq   s4,s0,440003ba <L1_Conv_1+0x226>
440001ec:   408c       lw    a1,0(s1)
440001ee:   4008       lw    a0,0(s0)
440001f0:   56c030ef   jal   4400375c <__mulsf3>
440001f4:   00000593   li    a1,0
440001f8:   08e030ef   jal   44003286 <__addsf3>
440001fc:   892a       mv    s2,a0
440001fe:   40cc       lw    a1,4(s1)
44000200:   4048       lw    a0,4(s0)
44000202:   55a030ef   jal   4400375c <__mulsf3>
44000206:   85ca       mv    a1,s2
44000208:   07e030ef   jal   44003286 <__addsf3>
4400020c:   892a       mv    s2,a0
4400020e:   448c       lw    a1,8(s1)
44000210:   4408       lw    a0,8(s0)
44000212:   54a030ef   jal   4400375c <__mulsf3>
44000216:   85ca       mv    a1,s2
44000218:   06e030ef   jal   44003286 <__addsf3>
```

FIGURE 4.2.2 – Extrait du désassemblage montrant l'émulation logicielle du calcul flottant

Pour mesurer l'impact concret de la FPU sur les performances, nous avons exécuté le LeNet-5 complet sur le NEORV32 avec l'extension `Zfinx` activée. Les résultats sont présentés dans la figure 4.2.3 .

```
Démarrage du Neorv32 avec FPU...
Prediction : 2
Reference  : 2
Nb de cycle : 417179748
Temps de calcul : 4.172
```

FIGURE 4.2.3 – Résultat de l'exécution du LeNet-5 sur le NEORV32 avec FPU activée

En comparant ces résultats avec ceux obtenus sans FPU (figure 3.4.1 b), l'activation de la FPU matérielle apporte une accélération considérable. Sans FPU, le NEORV32 exécute le LeNet-5 complet en environ 2 573 513 365 cycles (25,7 secondes). Avec la FPU `Zfinx` activée, le temps d'exécution chute à environ 417 179 748 cycles (4,2 secondes), soit une réduction d'environ 84%.

Cette accélération s'explique par le fait que chaque opération flottante, qui nécessitait auparavant plusieurs dizaines de cycles via l'émulation logicielle, s'exécute maintenant plus rapidement grâce aux instructions matérielles `fmul.s` et `fadd.s`. Ce gain est particulièrement marqué sur les couches de convolution, où les multiplications et additions flottantes constituent la grande majorité des opérations.

Configuration NEORV32	Temps d'exécution	Taille code (octets)	Gain de cycles
Sans FPU	∼25,7 s	150 768	–
Avec FPU	∼4,2 s	148 576	∼84%

TABLE 4.2.1 – Impact de l'activation de la FPU sur les performances du NEORV32

Chapitre 5

Architecture Multi-cœurs

Ce chapitre présente le passage de l'architecture mono-cœur à une configuration dual-cœur pour les deux processeurs retenus. L'objectif est d'évaluer la faisabilité et les performances d'une exécution parallèle de l'algorithme LeNet-5 sur deux cœurs.

5.1 NEORV32 en configuration dual-cœur

5.1.1 Mise en place matérielle

Le passage en dual-cœur du NEORV32 s'est avéré particulièrement simple. L'architecture du NEORV32 intègre nativement le support du mode SMP (*Symmetric Multiprocessing*). Il suffit d'activer l'option correspondante dans les paramètres du cœur au sein du block design Vivado, sans modification de la structure globale du design.

Les ressources matérielles utilisées par la configuration dual-cœur du NEORV32 sont présentées dans la figure 5.1.1 .

Resource	Utilization	Available	Utilization %
LUT	12858	53200	24.17
FF	12229	106400	11.49
BRAM	86	140	61.43
DSP	12	220	5.45
IO	9	200	4.50

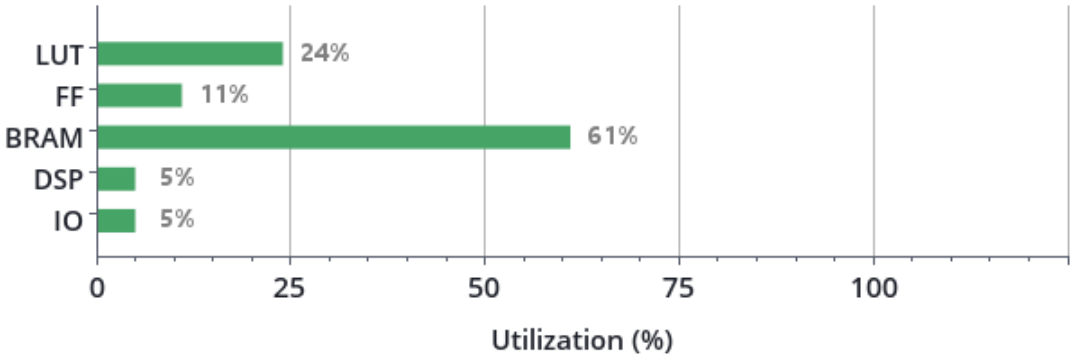

Figure 5.1.1 – Ressources matérielles utilisées par le NEORV32 en configuration dual-cœur

Le passage en configuration dual-cœur du NEORV32 a un impact relativement limité sur l'utilisation des ressources. La figure 5.1.1 montre que le design utilise maintenant 24,17% des LUT, contre 19,19% en mono-cœur.

La consommation de BRAM reste quasiment stable à 61,43% (86 blocs sur 140), soit un seul bloc de plus par rapport à la version mono-cœur.

L'utilisation des flip-flops (FF) est à 11,49%.

5.1.2 Programmation des deux cœurs

Du côté logiciel, la programmation des deux cœurs repose sur un mécanisme fourni par la bibliothèque NEORV32. Au démarrage, seul le cœur 0 s'exécute. La fonction suivante permet de lancer le cœur 1 sur une fonction spécifique :

Lancement du cœur 1 sur le NEORV32

```
neorv32_smp_launch(main_core1, (uint8_t*)core1_stack, sizeof(
    core1_stack));
```

Cette fonction prend trois paramètres : le point d'entrée de la fonction à exécuter sur le cœur 1, l'adresse de base de sa pile, et la taille de celle-ci. La spécification explicite de la pile permet de prévenir les débordements mémoire (*stack overflow*).

Pour valider le fonctionnement dual-cœur, un programme de test a été développé. Chaque cœur contrôle une LED distincte de la ZedBoard avec des fréquences de clignotement différentes, prouvant ainsi l'exécution indépendante et simultanée des deux cœurs. Un mécanisme de verrouillage (*lock*) basé sur les instructions atomiques protège les accès concurrents au registre GPIO.

Programme de test dual-cœur NEORV32

```
#include <neorv32.h>
#include <stdint.h>
#define BRAM_DATA_BASE 0x42000000
#define status_0 ((volatile uint32_t*)(BRAM_DATA_BASE + 0x14))
#define status_1 ((volatile uint32_t*)(BRAM_DATA_BASE + 0x18))
volatile uint8_t attribute ((aligned (16))) core1_stack[2048];
// Fonction executee par le coeur 1
int main_core1(void) {
neorv32_rte_setup();
*status_1 = 50 + 50;
// ... controle LED1 ...
return 0;
}
// Fonction principale executee par le coeur 0
int main(void) {
neorv32_rte_setup();
neorv32_smp_launch(main_core1, (uint8_t*)core1_stack, sizeof(
    core1_stack));

*status_0 = 10 + 20;
// ... controle LED0 ...
return 0;
}
```

Le résultat de l'exécution, présenté dans la figure 5.1.2 , confirme le bon fonctionnement des deux cœurs en parallèle : les valeurs écrites en BRAM par chaque cœur (30 et 100) sont correctement lues par le Zynq.

```
resultat:
core 0 resultat=  30
core 1 resultat= 100
```

FIGURE 5.1.2 – Résultat du test dual-cœur sur le NEORV32

5.2 RI5CY en configuration dual-cœur

5.2.1 Mise en place matérielle

Pour le RI5CY, le passage en dual-cœur nécessite l'ajout manuel d'un second cœur dans le block design Vivado, comme illustré dans la figure 5.2.1 .

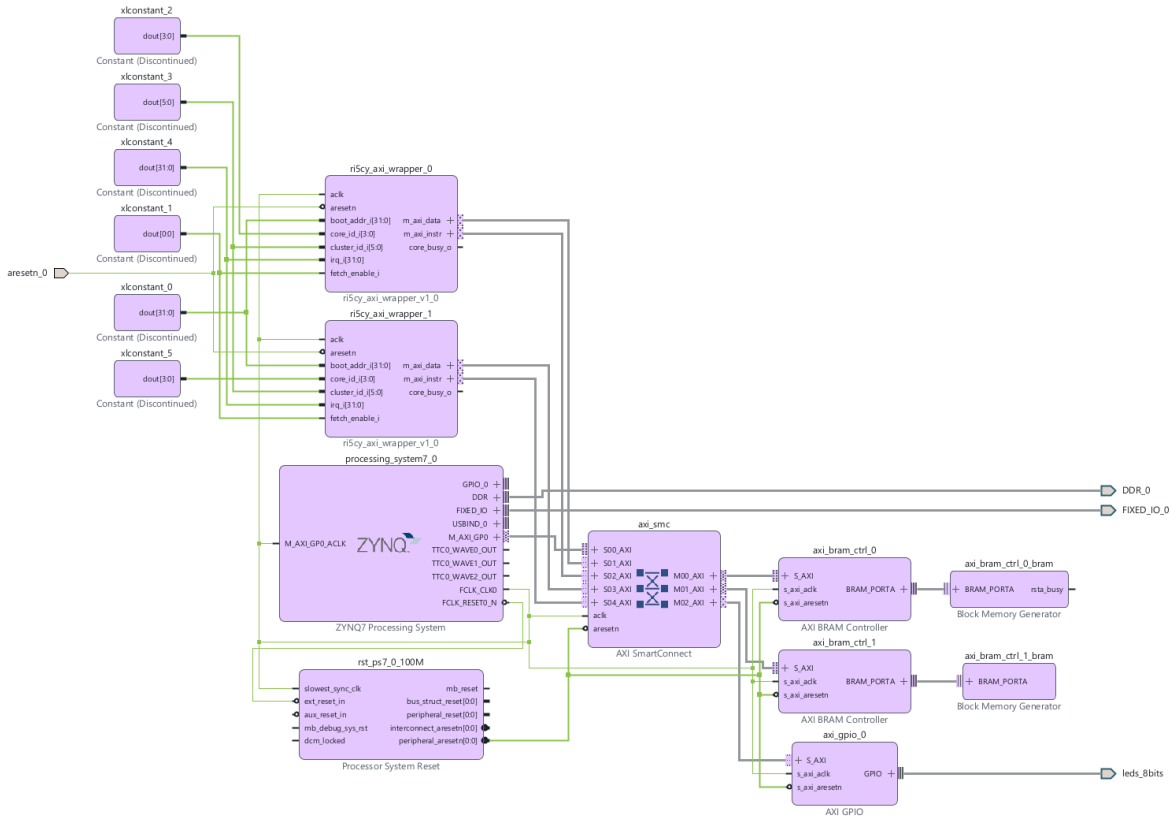

FIGURE 5.2.1 – Block design du RI5CY en configuration dual-cœur

Les deux cœurs partagent la même adresse de démarrage (*boot address*), car la différenciation s'effectue au niveau du fichier de démarrage assembleur `crt0.s`. Au démarrage, chaque cœur lit son identifiant unique depuis le registre CSR `MHARTID` (*Machine Hardware Thread ID*), qui contient l'identifiant du cœur et du cluster. Le code de démarrage utilise cet identifiant pour brancher chaque cœur vers sa routine d'initialisation respective. Pour ajouter un troisième cœur, il serait nécessaire de modifier le code assembleur du `crt0.s` afin d'ajouter un branchement supplémentaire correspondant au nouvel identifiant.

5.2.2 Programmation des deux cœurs

Contrairement au NEORV32 où chaque cœur dispose de sa propre fonction d'entrée, le RI5CY utilise un unique point d'entrée `main`. L'identifiant du cœur, passé en paramètre, permet de

différencier le comportement de chaque cœur à l'aide d'une structure conditionnelle :

Programme de test dual-cœur RI5CY

```
int main(int id) {
    volatile uint32_t* status_0 = (volatile uint32_t*)ADDR_STATUS_0
        ;
    volatile uint32_t* status_1 = (volatile uint32_t*)ADDR_STATUS_1
        ;

    if (id == 0) {
        GPIO_TRI = 0x00;
        GPIO_DATA = 0x00;
        *status_0 = 10 + 20;
        while(1) {
            safe_gpio_set(0);
            delay_slow();
            delay_slow();
            safe_gpio_clr(0);
            delay_slow();
            delay_slow();
        }
    } else if (id == 1) {
        *status_1 = 50 + 50;
        while(1) {
            safe_gpio_set(1);
            delay_slow();
            safe_gpio_clr(1);
            delay_slow();
        }
    }
    return 0;
}
```

5.2.3 Comparaison des ressources mono-cœur vs dual-cœur

Le tableau 5.2.1 récapitule la différence des ressources par le passage en dual-cœur pour les deux processeurs.

Ressource	NEORV32 mono	NEORV32 dual	RI5CY mono	RI5CY dual
LUT (%)	19,19	24,17	23,00	43,42
FF (%)	9,92	11,49	10,62	15,72
BRAM (%)	60,71	61,43	57,14	57,14
DSP (%)	2,73	5,45	3,18	6,36

TABLE 5.2.1 – Comparaison des ressources matérielles mono-cœur vs dual-cœur

5.3 Mémoire Partagée

Les deux architectures reposent sur une structure similaire : la BRAM de données est partagée entre les deux cœurs, ce qui permet l'échange de données et la communication inter-cœurs. Cependant, cette mémoire partagée introduit un défi majeur : la gestion des accès concurrents.

Lorsque deux cœurs tentent d'accéder simultanément à la même adresse mémoire, des incohérences peuvent survenir. Par exemple, si le cœur 0 lit une variable pendant que le cœur 1 est en train de la modifier, la valeur lue peut être incorrecte. Ce problème, est particulièrement critique dans le contexte d'un CNN où les résultats intermédiaires d'une couche sont utilisés par la couche suivante.

L'extension atomique (A) de RISC-V fournit des instructions dédiées à la résolution de ce problème. Les instructions atomiques, telles que `lr.w` (Load Reserved) et `sc.w` (Store Conditional), permettent d'implémenter des mécanismes de verrouillage (*mutex*) garantissant un accès exclusif à une zone mémoire critique. L'instruction `lr.w` lit une valeur en mémoire et réserve l'adresse, tandis que `sc.w` n'écrit la nouvelle valeur que si aucun autre cœur n'a modifié l'adresse entre-temps. Ce mécanisme assure la cohérence des données sans nécessiter de bus matériel dédié à la cohérence de cache.

5.4 Évaluation des Performances Dual-cœur

Pour évaluer les performances en configuration dual-cœur, nous avons réparti les couches du LeNet-5 entre les deux cœurs de manière séquentielle. Il faut noter que cette approche ne permet pas d'exploiter pleinement le parallélisme de données, qui consisterait à diviser le calcul des neurones d'une même couche entre les deux cœurs.

Cœur	**Couches exécutées**
Cœur 0	CONV_1, MAXPOOL_1, CONV_2, MAXPOOL_2
Cœur 1	CONV_3, FC_1, FC_2, OUTPUT

TABLE 5.4.1 – Répartition des couches du LeNet-5 entre les deux cœurs

Cette configuration est modulable et peut être ajustée selon la charge de calcul de chaque couche.

5.4.1 Stratégie de synchronisation par jeton mémoire

L'approche testée utilise un jeton de synchronisation stocké en BRAM partagée. Le cœur 0 positionne ce jeton à 1 lorsqu'il a fini ses couches, et le cœur 1 interroge ce jeton en boucle jusqu'à ce qu'il passe à 1.

Synchronisation par jeton mémoire

```
// Coeur 0 : execute les 4 premieres couches
#ifdef MAXPOOL_2
    L1_Conv_1(input, output_L1);
    L2_MaxPool_1(output_L1, output_L2);
    L3_Conv_2(output_L2, output_L3);
    token_l3 = EMPTY;
    L4_MaxPool_2(output_L3, output_L4);
    *token_l4 = 1;  // Signal pour le coeur 1
#endif

// Coeur 1 : attend le signal puis execute CONV_3
#ifdef CONV_3
    if (*token_l4 == 1) {
        *token_l4 = 0;
        L5_Conv_3(output_L4, output_L5);
```

```
16          token_l5 = FULL;
17      }
18  #endif
```

5.4.2 Résultats RI5CY

L'exécution du LeNet-5 en configuration dual-cœur sur le RI5CY donne les résultats présentés dans la figure 5.4.1 .

```
Demarrage du RI5CY...
Prediction : 2
Reference  : 2
Cycles Core 0 : 1563339298
Cycles Core 1 : 260100546
Temps de calcul : 47.409 s
```

FIGURE 5.4.1 – Résultat du LeNet-5 sur le RI5CY en configuration dual-cœur

La répartition des couches entre les deux cœurs permet une exécution correcte de l'algorithme, avec une prédiction identique à celle obtenue en mono-cœur. Le cœur 0 exécute les couches de convolution et de max-pooling (CONV_1, MAXPOOL_1, CONV_2, MAXPOOL_2) en 1 563 339 298 cycles, tandis que le cœur 1 prend en charge les couches restantes (CONV_3, FC_1, FC_2, OUTPUT) en 260 100 546 cycles, soit un total de 1 823 439 844 cycles, soit environ 2,7% qu'en mono-cœur.

5.4.3 Résultats NEORV32

En appliquant la même stratégie de répartition des couches sur le NEORV32, nous avons observé un résultat inattendu, le temps d'exécution en dual-cœur est environ deux fois plus long qu'en mono-cœur. Alors que le LeNet-5 s'exécutait en environ 25,7 secondes en mono-cœur (sans FPU), le passage en dual-cœur porte ce temps à environ 51,2 secondes (cœur 0 : 4 390 963 974 cycles, cœur 1 : 726 659 880 cycles), comme le montre la figure 5.4.2 soit au total 5 117 623 854 cycles.

```
Démarrage du dual core NEORV32 sans fpu...
Prediction : 2
Reference  : 2
Cycles Core 0 : 4390963974
Cycles Core 1 : 726659880
Total Cycles    : 5117623854
Temps de calcul : 51.176239 s
```

FIGURE 5.4.2 – Résultat du NEORV32 en dual-cœur sans FPU

Le même phénomène est observé avec la FPU activée (figure 5.4.3) : le temps d'exécution passe d'environ 4,2 secondes en mono-cœur à environ 8,2 secondes en dual-cœur (cœur 0 : 700 385 566 cycles, cœur 1 : 120 821 743 cycles), soit là encore un doublement du temps de calcul (total de 821 207 309 cycles).

```
Démarrage du dual core NEORV32 avec fpu...
Prediction : 2
Reference  : 2
Cycles Core 0 : 700385566
Cycles Core 1 : 120821743
Temps de calcul : 8.212 s
```

FIGURE 5.4.3 – Résultat du NEORV32 en dual-cœur sans FPU (

Analyse du problème : contention du bus

Après avoir examiné le code désassemblé sans y trouver le problème, la documentation du NEORV32 a révélé la cause de ce ralentissement. Comme l'illustre la figure 5.4.4 , les deux cœurs du NEORV32 partagent un bus interne unique pour accéder à la mémoire et aux périphériques.

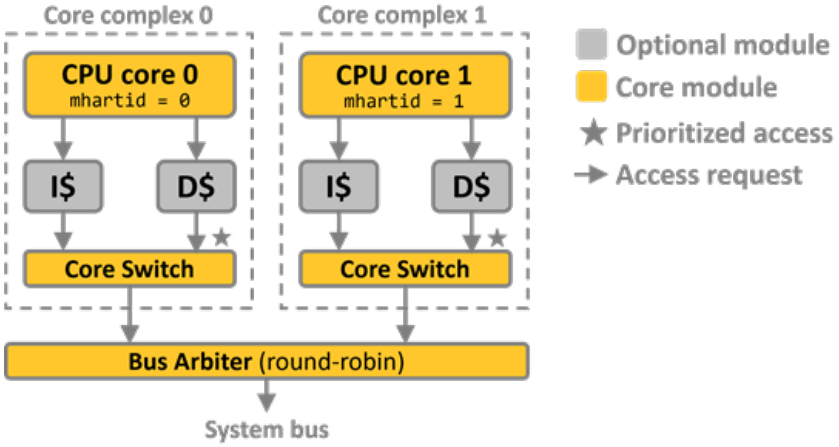

FIGURE 5.4.4 – Architecture interne du NEORV32 en configuration dual-cœur

Le ralentissement observé s'explique par un conflit de priorité entre les deux processeurs. Pendant que le cœur 0 sollicite énormément la mémoire pour traiter les premières couches du réseau, le cœur 1 vérifie sans interruption l'état du jeton de synchronisation. Cela force le cœur 0 à patienter régulièrement pour accéder aux données, ce qui finit par doubler son temps d'exécution total.

Ce problème ne s'est pas produit avec le RI5CY, car chaque cœur dispose de son propre bus AXI indépendant, ce qui élimine toute contention d'accès mémoire entre les cœurs.

Solution : interruption locale via le CLINT

Pour résoudre ce problème de contention, j'ai mis en place une approche par interruption en exploitant le CLINT (*Core Local Interrupt Controller*) intégré au NEORV32. Au lieu d'interroger en boucle un jeton en mémoire partagée, le cœur 1 se met en veille avec l'instruction `wfi` (*Wait For Interrupt*). Lorsque le cœur 0 a terminé l'exécution de ses couches, il envoie une interruption au cœur 1 en écrivant dans le registre `MSIP` du CLINT. Le cœur 1 se réveille alors et commence immédiatement son traitement.

Cette approche présente un double avantage, elle supprime totalement les accès mémoire de polling du cœur 1 pendant l'exécution du cœur 0, ce qui élimine la contention du bus. De plus, le cœur 1 consulte uniquement son propre registre interne `MSIP`, qui ne transite pas par le bus mémoire partagé.

Le code suivant illustre le mécanisme d'interruption. Le cœur 0 signale la fin de ses couches en écrivant dans le registre CLINT du cœur 1, et le cœur 1 attend ce signal via l'instruction `wfi` :

Synchronisation par CLINT sur le NEORV32

```
#define CLINT_MSIP_1  ((volatile int*)0xF0000004)

// Coeur 0 : signale la fin de ses couches
#ifdef MAXPOOL_2
    L4_MaxPool_2(output_L3, output_L4);
    // Mesure des cycles et envoi de l'interruption au coeur 1
    *CLINT_MSIP_1 = 1;
#endif

// Coeur 1 : attend le signal puis execute ses couches
#ifdef CONV_3
    while (*CLINT_MSIP_1 == 0) {
        asm volatile ("wfi");  // Mise en veille sans acces bus
    }
    *CLINT_MSIP_1 = 0;
    L5_Conv_3(output_L4, output_L5);
#endif
```

Avec cette approche par interruption, les temps d'exécution en dual-cœur retrouvent des valeurs cohérentes avec ceux obtenus en mono-cœur. Les résultats sont présentés dans les figures 5.4.5 a et 5.4.5 b.

```
Démarrage du dual core NEORV32 sans fpu...
Prediction : 2
Reference  : 2
Cycles Core 0 : 2237019019
Cycles Core 1 : 371051249
Total Cycles    : 2608070268
Temps de calcul : 26.080703 s
```

(a) NEORV32 dual-cœur avec CLINT (sans FPU)

```
Démarrage du dual core NEORV32 avec fpu...
Prediction : 2
Reference  : 2
Cycles Core 0 : 355570066
Cycles Core 1 : 61612092
Total Cycles    : 417182158
Temps de calcul : 4.171822 s
```

(b) NEORV32 dual-cœur avec CLINT (avec FPU)

FIGURE 5.4.5 – Résultats du NEORV32 en dual-cœur avec synchronisation par CLINT

5.5 Synthèse Comparative des Performances

Le tableau 5.5.1 récapitule l'ensemble des résultats obtenus au cours de ce projet. Afin de permettre une comparaison indépendante de la fréquence d'horloge, les performances sont exprimées en nombre de cycles d'horloge.

Processeur	Configuration	Cycles	Temps	Gain
RI5CY	Mono-cœur	1 779 684 670	46,1 s	réf.
	Dual-cœur	1 829 439 844	47,4 s	−2,7%
NEORV32	Mono-cœur sans FPU	2 573 513 365	25,7 s	réf.
	Mono-cœur avec FPU Zfinx	417 197 480	4,2 s	83,7%
	Dual-cœur sans FPU (CLINT)	2 608 070 268	26,1 s	−1,5%
	Dual-cœur avec FPU (CLINT)	417 182 158	4,2 s	83,7%

TABLE 5.5.1 – Synthèse comparative des performances sur le LeNet-5

L'impact des instructions PULP (gain de 8,9% sur les couches L1+L2, voir chapitre précédent) n'est pas inclus dans ce tableau car il a été mesuré uniquement sur un sous-ensemble du réseau

(2 couches sur 7) en `float32` avec GCC 7, alors que les autres résultats portent sur le LeNet-5 complet en `float16` avec GCC 13.

Chapitre 6

Conclusion et axe de progression

Ce projet a permis de valider la faisabilité d'une plateforme multiprocesseur RISC-V pour l'exécution d'algorithmes d'IA sur FPGA [1]. Deux architectures ont été déployées avec succès sur la ZedBoard.

L'un des points majeurs de ce travail réside dans l'importance des optimisations matérielles pour l'accélération des réseaux de neurones. Les instructions spécialisées et les unités de calcul dédiées transforment les performances, confirmant que l'extensibilité de RISC-V constitue un atout décisif face aux architectures fermées.

Le passage en configuration multiprocesseur a révélé que la simple réplication des cœurs ne garantit pas automatiquement un gain de performances. Les choix architecturaux, notamment en matière d'interconnexion et de gestion de la cohérence mémoire, influencent fortement l'efficacité du parallélisme. Cette observation souligne l'importance d'une approche globale, combinant architecture matérielle et stratégies logicielles adaptées.

Sur le plan méthodologique, ce projet confirme l'importance d'une analyse comparative rigoureuse avant la sélection d'un cœur RISC-V. La compatibilité avec la chaîne de développement, la maturité de l'écosystème et la qualité de la documentation se sont révélées aussi déterminantes que les performances brutes du processeur.

6.1 Axes de progression

Plusieurs pistes d'amélioration ont été identifiées pour prolonger et enrichir ce travail.

6.1.1 Quantification du modèle en entier 8 bits

L'un des axes les plus prometteurs consiste à quantifier le modèle LeNet-5 en format entier 8 bits (`int8`) au lieu d'utiliser des nombres flottants. Cette approche présente un double intérêt. D'une part, elle permettrait d'exécuter l'intégralité du modèle avec les instructions PULP sur le RI5CY, car ces instructions sont spécifiquement conçues pour manipuler des données 8 bits grâce aux opérations SIMD (*Single Instruction, Multiple Data*). Les instructions SIMD du RI5CY permettent d'effectuer quatre opérations sur des données 8 bits en un seul cycle, ce qui accélérerait considérablement les calculs de convolution. D'autre part, la réduction de la taille des données de 16 ou 32 bits à 8 bits diminuerait significativement l'empreinte mémoire, résolvant les contraintes de BRAM rencontrées lors de ce projet.

6.1.2 Migration vers le CV32E40P

Le CV32E40P, successeur direct du RI5CY au sein du projet OpenHW Group, constitue une évolution naturelle de ce travail. Ce cœur intègre à la fois les instructions PULP héritées du RI5CY et une FPU matérielle, combinant ainsi les deux axes d'optimisation explorés séparément dans ce projet. De plus, le CV32E40P est compatible avec GCC 12, un compilateur plus récent qui offre de meilleures optimisations et supporte nativement le format `float16`. Cette migration

1. L'ensemble du code source c et des configurations matérielles est disponible sur le dépôt Git : https://github.com/Mahmoud010103/Ptech

permettrait de réaliser des comparaisons plus complètes entre les différentes configurations : avec et sans instructions PULP, avec et sans FPU.

6.1.3 Ajout d'instructions personnalisées sur le NEORV32

Le NEORV32 offre une fonctionnalité particulièrement intéressante pour l'accélération des calculs d'IA : le CFU (*Custom Function Unit*). Ce module, bien documenté dans la datasheet du processeur, permet d'ajouter des instructions personnalisées directement dans le pipeline d'exécution. Le fichier source du CFU décrit précisément l'interface et la méthodologie pour intégrer de nouvelles instructions matérielles.

L'ajout d'une instruction MAC (*Multiply-Accumulate*) dédiée constituerait une optimisation majeure. Cette instruction permettrait d'effectuer une multiplication suivie d'une accumulation en un seul cycle d'horloge, au lieu des multiples instructions actuellement nécessaires. Étant donné que les couches de convolution du LeNet-5 reposent essentiellement sur des opérations de multiplication-accumulation, le gain potentiel serait significatif. Cette piste n'a pas pu être explorée par manque de temps durant ce projet.

6.1.4 Parallélisme de données

Enfin, l'approche dual-cœur pourrait être améliorée en passant d'une répartition séquentielle des couches à un véritable parallélisme de données. Au lieu d'attribuer des couches différentes à chaque cœur, il s'agirait de diviser le calcul des neurones d'une même couche entre les deux cœurs.

Glossaire

AXI Advanced eXtensible Interface (Bus d'interconnexion haute performance du protocole AMBA). 10, 12, 13

BRAM Block Random Access Memory (Mémoire vive intégrée dans les FPGA). 13–15, 18, 24, 26, 27, 32

CFU Custom Function Unit (Unité de fonction personnalisée du NEORV32). 33

CLINT Core Local Interrupt Controller (Contrôleur d'interruptions locales du cœur). 29

CNN Convolutional Neural Network (Réseau de neurones convolutionnel). 5, 7, 17, 19, 27

CSR Control and Status Registers (Registres de contrôle et d'état de l'architecture RISC-V). 16, 25

DSP Digital Signal Processing (Traitement numérique du signal). 12

ELF Executable and Linkable Format (Format de fichier exécutable standard). 16, 19

FPGA Field-Programmable Gate Array (Réseau de portes programmables). 7, 15

FPU Floating Point Unit (Unité de calcul en virgule flottante). 19, 21, 22, 28, 32, 33

GPIO General Purpose Input/Output (Entrées/sorties à usage général). 13

HEX Format hexadécimal utilisé pour le chargement de programmes en mémoire. 16

IA Intelligence artificielle. 5, 7–9, 11–13, 17, 33

IETR Institut d'Electronique et des Technologies du numéRique. 5

IP Propriété intellectuelle. 11

ISA Instruction set architecture (architecture du jeu d'instructions). 11

LUT Look-Up Table (Table de correspondance, élément logique de base des FPGA). 13, 14

MAC Multiply-Accumulate (Opération de multiplication-accumulation). 19, 33

PULP Parallel Ultra-Low-Power Processing Platform (Plateforme de traitement parallèle ultra-basse consommation). 4, 11, 12, 16, 19–21, 32, 33

SDK Software Development Kit (Kit de développement logiciel). 15

SIMD Single Instruction, Multiple Data (Instruction unique, données multiples). 32

SMP Symmetric Multiprocessing (Multitraitement symétrique). 23

SoC System-on-Chip (Système sur puce intégrant processeur, mémoire et périphériques). 9

WHS Worst Hold Slack (Marge temporelle de maintien, contrainte de timing). 17

Bibliographie

[1] « The RISC-V Instruction Set Manual, » RISC-V International, visité le 3 nov. 2025. adresse : https://riscv.org/specifications/

[2] « RISC-V Extensions, » RISC-V International, visité le 3 nov. 2025. adresse : https://riscv.org/technical/specifications/

[3] « MicroBlaze Soft Processor Core, » AMD Xilinx, visité le 3 nov. 2025. adresse : https://www.xilinx.com/products/design-tools/microblaze.html

[4] SiFive. « From Berkeley Lab to Global Standard : RISC-V's 15-Year Journey, » visité le 23 jan. 2025. adresse : https://www.sifive.com/blog/from-berkeley-lab-to-global-standard-risc-vs-15-ye

[5] « RISC-V Cores and SoC Overview, » RISC-V International, visité le 3 nov. 2025. adresse : https://github.com/riscvarchive/riscv-cores-list

[6] S. Mangaraj. « Awesome RISC-V Resources, » visité le 3 nov. 2025. adresse : https://github.com/suryakantamangaraj/AwesomeRISC-VResources

[7] « CVA6 RISC-V Application Processor, » OpenHW Group, visité le 3 nov. 2025. adresse : https://github.com/openhwgroup/cva6

[8] « OpenHW Group - Open Source Hardware for Tomorrow, » OpenHW Group, visité le 3 nov. 2025. adresse : https://www.openhwgroup.org/

[9] *CV32E40P User Manual*, OpenHW Group. visité le 3 nov. 2025. adresse : https://docs.openhwgroup.org/projects/cv32e40p-user-manual/en/latest/intro.html

[10] *RI5CY User Manual*, PULP Platform. visité le 3 nov. 2025. adresse : https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf

[11] L. Benini. « PULP : Parallel Ultra-Low Power Platform for IoT and AI at the Edge, » visité le 3 nov. 2025. adresse : https://ethz.ch/content/dam/ethz/special-interest/itet/efcl-dam/documents/EFCL_Summer_School_Benini.pdf

[12] « PULP Platform - Parallel Ultra Low Power, » ETH Zurich & Università di Bologna, visité le 3 nov. 2025. adresse : https://pulp-platform.org/

[13] S. Nolting. « NEORV32 Processor - ISA Extension, » visité le 3 nov. 2025. adresse : https://stnolting.github.io/neorv32/#_i_isa_extension

[14] « MicroBlaze V Processor Core, » AMD Xilinx, visité le 3 nov. 2025. adresse : https://www.amd.com/fr/products/software/adaptive-socs-and-fpgas/microblaze-v.html