Augmenter la fréquence du CVA6 sur FPGA

5e concours national d'étudiants RISC-V 2024-2025

Èquipe **OCTO-PULSE** – IMT Atlantique/Lab-STICC

Juliette Dumas
IMT Atlantique
Brest, France
juliette.dumas@imt-atlantique.net

Nicolas Barrera
IMT Atlantique
Brest, France
nicolas.barrera@imt-atlantique.net

Maria Florenza
IMT Atlantique
Brest, France
maria.florenza-lamberti@imt-atlantique.net

***Abstract*—Dans cet article, nous présentons la solution développée par notre équipe, OCTO-PULSE, composée de trois étudiants d'IMT Atlantique, campus de Brest, dans le cadre du 5ème Concours National d'Étudiants RISC-V 2024-2025 [1], sponsorisé par Thales, le GDR SOC2 et le CNFM. Le concours porte sur l'augmentation de la fréquence du CVA6 sur FPGA, mettant au défi les participants d'optimiser la performance et l'efficacité.**

Dans ce cadre, nous avons proposé et mis en œuvre trois optimisations majeures pour augmenter la fréquence de fonctionnement. La première optimisation porte sur l'amélioration de l'environnement d'implémentation, visant à maximiser l'efficacité des outils utilisés. La deuxième optimisation résout un problème d'inter-clock path, lié à la synchronisation entre la clock du cœur de processeur et celle du JTAG. Enfin, l'ajout d'étages de pipeline sur certains signaux critiques a permis d'atteindre une fréquence de fonctionnement de 60,56 MHz, soit une augmentation de 49% par rapport à la fréquence initiale, avec une perte en performance de 6,95% du score CoreMark/MHz, ce qui respecte la contrainte maximale de 10% imposée par le concours.

Ces optimisations ont permis une amélioration significative des performances globales tout en maintenant un compromis acceptable entre fréquence et perte en efficacité.

***Index Terms*—RISC-V, FPGA, CVA6, fréquence, optimisation, performance**

I. Introduction

L'accroissement des besoins en puissance de calcul et en efficacité énergétique, notamment dans les domaines à forte intensité algorithmique, tels que l'intelligence artificielle embarquée, la vision par ordinateur, la robotique ou le traitement du signal, impose une évolution profonde des architectures de processeurs embarqués. Dans ce contexte, RISC-V s'impose comme une architecture ouverte, libre de droits, modulaire et extensible, offrant aux concepteurs la possibilité de personnaliser finement le comportement des processeurs en fonction des exigences fonctionnelles et temporelles des applications ciblées.

Dans une démarche de valorisation du potentiel de RISC-V pour les systèmes embarqués de nouvelle génération, Thales, le GDR SoC² et le CNFM coorganisent chaque année un concours national dédié à cette architecture. La cinquième édition met l'accent sur l'optimisation de la fréquence de fonctionnement du cœur CVA6, un processeur 64 bits open source implémenté sur FPGA, dérivé du projet Ariane.

Dans ce cadre, nous présentons dans ce rapport la méthodologie mise en œuvre pour améliorer les performances du cœur CVA6, ainsi que les résultats obtenus. Ce projet s'inscrit dans une problématique plus large : concilier fréquence maximale, consommation et surface logique dans des environnements contraints. En tant que paramètre déterminant, la fréquence de fonctionnement est influencée à la fois par des facteurs architecturaux (chemins critiques, profondeur de pipeline, organisation mémoire) et technologiques (qualité de la synthèse, stratégie de placement-routage, contraintes temporelles). Notre approche repose sur une analyse fine du chemin critique, l'exploration des marges d'amélioration offertes par l'environnement Vivado, et des ajustements ciblés de la microarchitecture. Les performances sont évaluées selon plusieurs axes : gain de fréquence, score CoreMark/MHz, résultats de simulation, résultats sur carte FPGA, et occupation des ressources matérielles (LUTs, flip-flops, BRAM, DSP, etc.).

L'objectif est de proposer un compromis robuste entre performances et coût matériel, dans une logique de reproductibilité et de transférabilité à d'autres cœurs ou contextes applicatifs.

II. Recherche des paramètres optimaux de synthèse

La synthèse logique, le placement et le routage du cœur CVA6 ont été réalisés à l'aide de l'outil Vivado, développé par AMD-Xilinx. Dans une première phase exploratoire, notre objectif a été d'optimiser la configuration de Vivado, dans le but d'augmenter la fréquence de fonctionnement du processeur à son maximum atteignable sans modifier sa structure d'origine.

Par défaut, Vivado applique une stratégie d'implémentation généraliste qui cherche un compromis entre l'utilisation des ressources (LUTs, BRAM, flip-flops) et le temps d'exécution de l'outil, en maintenant des contraintes de timing raisonnables. [2] Cette approche générique ne permet toutefois pas d'exploiter pleinement les marges d'optimisation disponibles lorsque l'objectif principal est la performance temporelle. Dans ce contexte, nous avons cherché à modifier les directives de synthèse et d'implémentation via des scripts .tcl, en ciblant plus précisément les étapes critiques du flot de conception :

Parmi les différentes phases de l'implémentation listées dans l'environnement Vivado, notre analyse a mis en évidence que l'étape d'implémentation temporelle était particulièrement déterminante pour l'amélioration du chemin critique. Nous avons donc concentré nos efforts sur trois directives spécifiques, appliquées aux étapes clés du flot :

- `place_design -directive ExtraTimingOpt` :
  cette option active des algorithmes complémentaires visant à réduire les violations de timing à la fin du placement. Elle permet une réorganisation plus fine des instances logiques pour améliorer le WNS (Worst Negative Slack).]
- `route_design -directive AggressiveExplore` :
  elle étend significativement l'espace d'exploration durant le routage, en intensifiant l'analyse des chemins critiques et des options de connexion, au prix d'un temps de compilation accru.
- `route_design -directive NoTimingRelaxation` :
  cette directive empêche Vivado d'assouplir les contraintes temporelles durant le routage, forçant l'outil à respecter strictement les objectifs de fréquence définis, même si cela implique un allongement notable du temps d'exécution.

Si les directives associées à place_design et route_design, deux étapes indépendantes de l'implémentation, peuvent être combinées. Nous avons constaté que les stratégies AggressiveExplore et NoTimingRelaxation s'excluent mutuellement dans leur logique d'exécution. Il a donc été nécessaire de comparer leurs effets respectifs pour sélectionner l'option la plus efficace.

TABLE I
TABLEAU COMPARATIF DE L'AMÉLIORATION DU TIMING EN FONCTION DU CHOIX DE DIRECTIVE SUR L'ÉTAPE ROUTE_DESIGN POUR UNE SYNTHÈSE À 50MHZ

	Default	AggressiveExplore	NoTimingRelaxation
Worst Negative Slack (WNS)	-6.083 ns	-2.371 ns	-2.667 ns
Total Negative Slack (TNS)	-11496.617 ns	-2904.578 ns	-3315.488 ns
Number of Failing Endpoints	3536	2237	2238
Total Number of Endpoints	48444	48446	48446

L'analyse du tableau I montre une amélioration significative du WNS et du TNS lorsque les directives temporellement plus agressives sont activées, en particulier avec AggressiveExplore. Ces résultats ont guidé notre choix de stratégie dans les phases ultérieures d'optimisation. Il convient toutefois de souligner que ce gain en fréquence se traduit par une augmentation du temps de compilation, liée à l'intensification des algorithmes d'exploration des chemins critiques. Ce compromis, fréquent dans les processus d'optimisation temporelle, reste acceptable dans le contexte de ce concours, où la performance finale prime sur le temps de synthèse.

III. SYNCHRONISATION AU JTAG

Dans notre démarche d'augmentation de la fréquence du cœur de processeur, nous avons rapidement rencontré un problème lié à l'interface JTAG incluse dans le code. Ce problème survient lorsque la fréquence de fonctionnement du cœur (core clock) n'est pas un multiple de la fréquence de l'horloge TCK (la clock de l'interface JTAG). Ce phénomène entraîne un problème de clock domain crossing, en raison de la désynchronisation entre la clock du cœur et celle du JTAG.

L'interface JTAG, étant responsable de l'échange de données avec le cœur, doit être correctement synchronisée avec ce dernier pour éviter des erreurs de communication. Cependant, la fréquence TCK n'est pas dérivée de la clock principale de la carte, mais est plutôt imposée par le programmateur JTAG. Cela signifie qu'en l'absence d'une configuration correcte, la désynchronisation entre ces deux horloges entraîne des contraintes de timing difficiles à respecter.

Initialement, nous avons tenté de résoudre ce problème en forçant la fréquence de TCK à être égale à celle du cœur. Cependant, cette approche présente deux inconvénients majeurs. Premièrement, il n'est pas nécessaire que TCK soit aussi rapide. Augmenter cette fréquence représente une contrainte inutile, qui peut entraîner une perte d'espace sur le FPGA. Ensuite, toutes les fréquences ne sont pas nécessairement synthétisables. Le bloc en charge de la gestion des clocks sur notre système est le bloc clk_gen, qui est chargé de diviser et/ou de multiplier la clock de 125MHz présente sur la carte, pour créer les différentes clocks qui vont alimenter notre système. Cependant, il n'est pas possible de choisir n'importe quelle valeur de multiplicateur et de diviseur. Cela signifie que pour certaines fréquences, on peut s'approcher fortement de la fréquence souhaitée, mais sans pouvoir l'atteindre exactement. Ainsi, si on règle TCK par exemple à 60MHz, mais que le bloc clk_gen génère une clock pour le cœur à 59.9 MHz, un problème de clock domain crossing persistera.

La solution que nous avons adoptée pour résoudre ce problème consiste à ajuster la fréquence de TCK en fonction des capacités du bloc clk_gen. Pour ce faire, nous utilisons les valeurs exactes fournies par ce bloc, c'est-à-dire les multiplicateurs et diviseurs utilisés pour générer l'horloge du cœur. Cela nous permet de connaître la période exacte de l'horloge générée pour le cœur, et nous pouvons ensuite la diviser par 100 pour obtenir une fréquence plus réaliste pour l'interface JTAG.

Cette approche garantit que la fréquence de TCK sera exactement synchronisée avec celle du cœur, éliminant ainsi tout problème de clock domain crossing.

IV. RÉDUCTION DES CHEMINS CRITIQUES

Dans un troisième temps, notre démarche s'est portée sur l'analyse et la réduction des chemins critiques identifiés après

synthèse et routage. Cette phase visait à intervenir directement sur la microarchitecture du cœur CVA6, en ciblant les signaux contribuant de manière significative aux violations temporelles.

A. MMU

L'analyse des rapports de timing générés par Vivado a révélé qu'un des chemins critiques majeurs traversait le module MMU (Memory Management Unit), il s'agit plus précisément du signal `mmu_exception_valid`. Le MMU (Memory Management Unit) est un composant essentiel de l'architecture CVA6. C'est lui qui assure la traduction des adresses virtuelles en adresses physiques, la gestion des accès mémoire, ainsi que le contrôle des privilèges d'accès via les mécanismes de protection tels que les PMP (Physical Memory Protection). Le signal `mmu_exception_valid` est utilisé au sein du mmu pour propager, à travers le pipeline, les informations relatives aux fautes d'accès mémoire, telles que les erreurs de permissions ou les pages absentes. Il conditionne le déclenchement d'une séquence d'exception, qui interrompt l'exécution normale du flot d'instructions pour transférer le contrôle vers un gestionnaire d'exception.

Afin d'atténuer l'impact de ce chemin, nous avons choisi d'insérer un étage supplémentaire de pipeline sur ce signal pour réduire davantage la profondeur logique. Cette opération a été réalisée à l'aide d'un module `shift_register`, conçu pour retarder proprement le signal d'un cycle supplémentaire tout en maintenant la validité de l'information transmise. En introduisant un cycle de pipeline supplémentaire, la logique combinatoire initialement concentrée sur un seul cycle est répartie sur deux cycles successifs, ce qui allège le chemin critique et améliore le respect des contraintes de timing.

Cette modification a permis d'améliorer significativement la fréquence de fonctionnement, en réduisant le Worst Negative Slack associé à ce chemin. Toutefois, elle s'est accompagnée d'une légère dégradation de l'IPC (Instructions Per Cycle), liée au fait que la détection des exceptions intervient désormais avec un cycle de latence supplémentaire. Ce compromis reste néanmoins acceptable dans notre contexte.

B. LSU

En examinant plus en détail le module `shift_register` utilisé précédemment pour insérer un cycle de pipeline sur le signal `mmu_exception_valid`, nous avons constaté que ce composant était également présent dans le chemin de retour des opérations de type `load/store`, c'est-à-dire dans le *load-store return path* géré par le module LSU (Load Store Unit). Ce pipeline permet de réguler le délai de propagation des signaux issus de la mémoire, en lissant les pics de latence potentiels sur des chemins critiques sensibles. Un paramètre de profondeur configurable est exposé dans les fichiers de configuration locaux (fichier `cv32a6_im_contest_config_pkg.sv`), permettant de faire varier dynamiquement la longueur du pipeline. En modifiant ce paramètre, nous avons pu explorer différents compromis entre cycle time (fréquence maximale atteignable) et efficacité d'exécution (IPC). Nous avons mené une exploration paramétrique de cette profondeur pour identifier un point de fonctionnement optimal, maximisant la fréquence tout en limitant la dégradation de l'IPC.

Finalement, nous avons choisi les paramètres suivants pour configurer le système :

$$localparam \quad CVA6ConcfigNrLoadPiprRegs = 1;$$

$$localparam \quad CVA6ConcfigNrStorePiprRegs = 1;$$

$$localparam \quad CVA6ConcfigNrLoadBufEntries = 2;$$

Ces réglages se sont avérés déterminants pour améliorer les performances globales du processeur tout en maintenant une bonne stabilité d'exécution.

C. Perf counter dans le scoreboard

Le prochain chemin critique qui limite notre fréquence de fonctionnement passe en grande partie par le module scoreboard, avant de terminer dans le perf counter. Deux approches ont été utilisé pour réduire ce chemin critique.

La première repose sur le constat qu'il est relativement simple d'ajouter autant d'étages de pipeline que nécessaire à ce niveau, mais uniquement en amont de l'entrée dans le perf counter. En effet, ce module est destiné à l'observation des performances et du comportement du CVA6, sans interagir directement avec le fonctionnement logique du cœur. Par conséquent, retarder l'arrivée d'un signal critique dans le perf counter présente peu de risques d'altérer le comportement global du processeur. Nous avons donc inséré un étage de pipeline juste avant l'entrée de ce signal dans le perf counter. Grâce aux options de retiming proposées par les outils de synthèse, cet étage de pipeline peut être automatiquement déplacé plus tôt sur le chemin critique, permettant ainsi une réduction plus efficace de sa longueur que si le registre restait à sa position initiale.

Cette première modification permet d'augmenter la fréquence maximale atteignable, mais le chemin critique — bien que raccourci — reste inchangé et continue de limiter la montée en fréquence. Pour pouvoir aller plus loin, nous sommes allés directement dans le module scoreboard. Nous avons identifié trois signaux présents dans les chemins les plus lents : `vaddr_to_be_flushed`, `asid_to_be_flushed` et `current_instruction_is_sfence_vma`. Bien que ces signaux soient déjà registered, leur délai de propagation restait significatif. Nous avons donc rajouté un étage de pipeline de plus à tous ces signaux, offrant ainsi davantage de flexibilité aux outils de synthèse lors des opérations de retiming. Contrairement au cas du `perf counter`, ces signaux participent plus directement au fonctionnement du cœur. Une dégradation des performances fonctionnelles (en IPC) était donc attendue suite à cette modification. Toutefois, après analyse, nous avons constaté que ces signaux ne sont utilisés que dans le cadre des instructions de type

`sfence.vma`, principalement pour assurer la communication entre le scoreboard et le module `issue_read_operand`. Par conséquent, bien que l'ajout d'un pipeline dans cette partie du cœur puisse affecter légèrement l'efficacité par MHz, son impact reste limité et localisé à un ensemble restreint d'instructions.

Les essais réalisés sur carte ont confirmé que cette modification entraîne effectivement une baisse des performances. Toutefois, en limitant l'ajout à un seul étage de pipeline, la dégradation reste contenue, avec une perte inférieure à 10% par MHz, ce qui reste acceptable au regard des exigences du concours. L'ajout d'un second étage permettrait certes une légère amélioration supplémentaire de la fréquence de fonctionnement, mais au prix d'une chute de performance bien plus marquée, excédant les seuils tolérables. Nous avons donc retenu une configuration avec un seul étage de pipeline supplémentaire sur ces signaux.

V. Comparaison des performances avant et après optimisation

Afin d'évaluer l'impact des optimisations apportées à l'architecture du cœur CVA6, nous avons mené une analyse comparative selon trois axes principaux : la fréquence maximale atteinte, la performance fonctionnelle mesurée à l'aide du benchmark CoreMark, et l'utilisation des ressources FPGA. Ces indicateurs permettent de caractériser les gains obtenus, tout en mesurant les compromis induits en termes de surface logique.

A. Fréquence maximale

Les optimisations architecturales apportées, en particulier sur les chemins critiques, ont permis une amélioration significative de la fréquence de fonctionnement du cœur. Le tableau II présente les résultats de synthèse obtenus avant et après l'application de ces modifications.

TABLE II
Mesures du chemin critique et de la fréquence Maximal avant et après optimisation

	Instance	Valeur
Slack (MET)	CVA6(référence) pour 40Mhz	0.287 ns
	CVA6(optimisé) pour 60 Mhz	0.155 ns
f_{max}	CVA6(référence)	40.46 Mhz
	CVA6(optimisé)	60.56 Mhz
	Changement	+49,68 %

Ces résultats confirment que les techniques mises en œuvre ont permis de lever les limitations précédemment imposées par les chemins critiques, et d'atteindre une fréquence nettement plus élevée sans compromettre la stabilité fonctionnelle du système. L'augmentation de +49,68% de la fréquence maximale constitue une amélioration notable, témoignant de l'efficacité des optimisations apportées.

B. CoreMark/MHz score en simulation et sur la board FPGA

Les performances de la solution ont également été évaluées à l'aide des benchmarks CoreMark et MNIST, en simulation et sur FPGA. CoreMark est un benchmark utilisé pour évaluer les performances des microprocesseurs en mesurant la vitesse d'exécution d'un ensemble d'opérations de calcul générales telles que les boucles, les structures conditionnelles et les opérations sur des tableaux [3].

Les figures 1 et 2 présentent respectivement l'affichage du score CoreMark sur la console série (FPGA) et via simulation. Le tableau III résume les résultats de la comparaison des performances entre notre solution optimisée et la référence pour les applications CoreMark et MNIST, en termes de CoreMark par MHz et du nombre de cycles nécessaires pour exécuter MNIST.

Fig. 1. Affichage sur la console putty du CoreMark de notre solution

```
2K performance run parameters for coremark.
CoreMark Size    : 666
Total ticks      : 1237992
Total time (secs): 1.237992
Iterations/Sec   : 2.423279
Iterations       : 3
Compiler version : GCC13.1.0
Compiler flags   :
Memory location  : BRAM
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0x2e87
Correct operation validated. See README.md for run and reporting rules.
CoreMark 1.0 : 2.423279 / GCC13.1.0     / BRAM
```

Fig. 2. Affichage sur Questa du CoreMark de notre solution

```
# [UART]: 2K performance run parameters for coremark.
# [UART]: CoreMark Size    : 666
# [UART]: Total ticks      : 1190906
# [UART]: Total time (secs): 1.190906
# [UART]: Iterations/Sec   : 2.519091
# [UART]: Iterations       : 3
# [UART]: Compiler version : GCC13.1.0
# [UART]: Compiler flags   :
# [UART]: Memory location  : BRAM
# [UART]: seedcrc          : 0xe9f5
# [UART]: [0]crclist       : 0xe714
# [UART]: [0]crcmatrix     : 0x1fd7
# [UART]: [0]crcstate      : 0x8e3a
# [UART]: [0]crcfinal      : 0x2e87
# [UART]: Correct operation validated. See README.md for run and reporting rules.
# [UART]: CoreMark 1.0 : 2.519091 / GCC13.1.0     / BRAM
```

TABLE III
Comparaison des performences entre notre solution optimisé et la référence pour l'application CoreMark et MNIST

	Instance	Valeur
CoreMark Score	CVA6 (référence)	2,604 CM/MHz
	CVA6 (optimisé)	2,423 CM/MHz
	Changement	-6,951 %
Nombre de MNIST Cycles	CVA6 (référence)	2790524
	CVA6 (optimisé)	2823598
	Changement	+1,185 %

Les résultats montrent une légère dégradation du score CoreMark par MHz, de l'ordre de 6,95%, et une hausse marginale du nombre de cycles nécessaires pour exécuter MNIST (+1,18%). Cependant, ces pertes restent bien en deçà des seuils spécifiés par le concours, qui tolère une diminution de la performance CoreMark/MHz jusqu'à 10%. L'augmentation de la fréquence permet de compenser ces dégradations et d'améliorer le temps d'exécution global, tout en restant dans les limites imposées par le concours.

C. Ressource utilisé (LUTs, flip-flops, BRAM, DSP...)

Les deux tableaux suivants présentent l'utilisation des ressources pour la version de référence et la version optimisée de notre solution. Nous nous attendions à une augmentation des ressources utilisées, mais dans les limites de la capacité du FPGA. Le tableau IV montre les ressources utilisées pour une implémentation à 40 MHz, tandis que le tableau V présente l'utilisation des ressources de la solution finale optimisée.

TABLE IV
RÉFÉRENCE DE L'UTILISATION DES RESSOURCES POUR UNE IMPLÉMENTATION À 40MHZ

Resource	Utilization	Available	Utilization %
LUT	24475	53200	46.01
LUTRAM	902	17400	5.18
FF	14568	105400	13.69
BRAM	66	140	47.14
DSP	4	220	1.82
IO	9	125	7.2
BUFG	5	32	15.63
MMCM	1	4	25.00

TABLE V
UTILISATION DES RESSOURCES DE LA SOLUTION FINALE OPTIMISÉE

Resource	Utilization	Available	Utilization %
LUT	24135	53200	45.37
LUTRAM	902	17400	5.18
FF	14847	106400	13.95
BRAM	66	140	47.14
DSP	4	220	1.82
IO	9	125	7.20
BUFG	5	32	15.63
MMCM	1	4	25.00

Enfin, le tableau suivant VI propose une synthèse chiffrée des variations sur les principales ressources consommées entre la version de référence et notre solution optimisée.

Les optimisations effectuées n'ont pas entraîné de changements significatifs dans l'utilisation des ressources critiques, telles que les LUTs et les BRAMs, qui restent dans des proportions similaires. Cependant, une légère augmentation du nombre de flip-flops (FF) est observée (+5,1 %), passant de 5461 à 5740. Cela peut s'expliquer par l'ajout de nouveaux étages de pipeline pour réduire les chemins critiques. Ces changements restent néanmoins modérés et bien dans les limites des ressources disponibles sur le FPGA.

TABLE VI
COMPARAISON DES RESSOURCES UTILISÉES POUR CVA6 ENTRE NOTRE SOLUTION OPTIMISÉE ET LA RÉFÉRENCE

	Instance	Valeur
LUTS	CVA6 (référence)	13093
	CVA6 (optimisé)	13096
	Changement	+0,02 %
FFs	CVA6 (référence)	5461
	CVA6 (optimisé)	5740
	Changement	+5,1 %
RAM36	CVA6 (référence)	16
	CVA6 (optimisé)	16
	Changement	+0 %

VI. CONCLUSION

Dans le cadre de l'optimisation de notre solution basée sur le processeur CVA6, plusieurs améliorations architecturales ont été proposées et mises en œuvre pour augmenter la fréquence de fonctionnement et optimiser les performances globales tout en respectant les contraintes imposées par le concours. Les principales optimisations ont été appliquées sur les chemins critiques, notamment en ajustant les étages de pipeline et en reconfigurant certains modules pour réduire les délais de propagation des signaux.

Les résultats ont montré une amélioration significative de la fréquence maximale de fonctionnement, atteignant 60,56 MHz contre 40,46 MHz pour la version de référence, soit une augmentation de 49,68%. Cependant, cette optimisation a entraîné une légère dégradation des performances en score CoreMark/MHz, avec une baisse de 6,95%. Néanmoins, cette dégradation reste dans les limites tolérées par le concours, qui autorise une réduction de performance CoreMark/MHz jusqu'à 10%.

En termes d'utilisation des ressources FPGA, les optimisations ont conduit à une légère augmentation du nombre de flip-flops, mais les autres ressources, telles que les LUTs, BRAM et DSP, sont restées pratiquement inchangées, ce qui indique une gestion efficace des ressources disponibles sur le FPGA.

En conclusion, les modifications apportées ont permis d'atteindre un bon compromis entre fréquence, performance et utilisation des ressources, tout en respectant les exigences du concours. Ces résultats montrent qu'il est possible d'améliorer considérablement la fréquence de fonctionnement tout en minimisant l'impact sur l'IPC et l'utilisation des ressources, offrant ainsi une solution optimisée adaptée aux contraintes du projet.

REMERCIEMENTS

Nous tenons à remercier nos professeurs à IMT Atlantique, qui nous ont non seulement guidés tout au long de ce projet, mais ont également joué un rôle essentiel dans notre formation dans les domaines du développement sur FPGA, de RISC-V et des systèmes embarqués en général.

Nous exprimons tout particulièrement notre gratitude à Ali Al Ghouwayel, Amer Baghdadi, Stefan Weithoffer et

Yehya Nasser, tous professeurs sur le campus de Brest d'IMT Atlantique et membres de l'équipe Algorithm-Architecture Interaction (2AI) du laboratoire Lab-STICC.

Nous souhaitons également adresser nos remerciements aux organisateurs de la compétition, pour nous avoir donné l'opportunité de participer à ce projet : Thales, le CNFM et le GDR SOC2.

REFERENCES

[1] Thales. 5th National RISC-V student contest CV32A6, 2024-2025. https://github.com/ThalesGroup/cva6-softcore-contest.

[2] Advanced Micro Devices, Inc. *Vivado Design Suite User Guide: Implementation*. AMD Adaptive Computing, v2024.2 edition, November 2024. UG904.

[3] EEMBC. Coremark: A benchmark for embedded systems, n.d. Accessed: 2025-05-04.