Compas'2023 : Parallélisme/ Architecture / Système
LISTIC/USMB - Annecy France, 4-7 juillet 2023

Processeur RISC-V enrichi avec une unité de microdécodage dynamique

Thomas NIEDDU†, Bertrand LE GAL† et Sébastien PILLEMENT‡*

†Université de Bordeaux, UMR 5218 IMS, 33400 Talence, France
thomas.nieddu@u-bordeaux.fr, bertrand.le-gal@u-bordeaux.fr
‡Nantes Université, CNRS, IETR UMR 6164, F-44000 Nantes, France
sebastien.pillement@univ-nantes.fr

Résumé
Le jeu d'instructions *open-source* RISC-V stimule depuis des années le secteur de la conception de processeurs allant de cœurs *high-end* aux cœurs *low-cost* ou *low-power*. Après une décennie d'évolution, les architectures RISC sont maintenant aussi matures que les architectures CISC popularisées par le géant Intel. La course à la puissance de calcul se heurte actuellement aux lois de la physique nécessitant des modifications architecturales des processeurs. De plus, la sécurité et l'efficience énergétique rejoignent la vitesse d'exécution parmi les contraintes de conception. Dans cet article, les bénéfices et les coûts liés à l'intégration d'une unité de microdécodage inspirée des processeurs CISC ont été évalués.

Mots-clés : Architecture, RISC-V, ISA, CISC, Décodage dynamique

1. Introduction

Le domaine des objets connectés [5] nécessite de produire un large spectre d'architectures de processeurs répondant à différentes contraintes applicatives (puissance de calcul, coût, énergie, sécurité, etc). Les processeurs de types CISC (*Complex Instruction Set Architecture*) ou RISC (*Reduced Instruction Set Architecture*) procurent actuellement des niveaux de performance élevés mais de deux manières différentes.
L'efficacité des architectures CISC provient de leur capacité à implémenter des traitements complexes. Elles mettent à disposition du compilateur un nombre élevé de macro-instructions qui sont décodées et exécutées efficacement par l'architecture matérielle. Ces architectures sont principalement développées par de grands groupes industriels tels que Intel, ou AMD et sont donc fermées. Des efforts de la communauté scientifique ont permis d'extraire le microcode des AMD K8 et K10 et de comprendre le fonctionnement de l'unité de microdécodage [7] de ces processeurs. Tout en étant transparent pour l'utilisateur final, le microcode (sauvegardé dans une mémoire dédiée) peut être mis à jour par le fabricant pour résoudre d'éventuels bugs et/ou contrer certaines attaques post-fabrication. C'est notamment ce qui a permis à Intel de répondre aux failles de sécurité SPECTRE [6] et MELTDOWN [11].
Les architectures RISC intègrent quant à elles peu d'instructions et sont dépourvues de ce mécanisme de microdécodage. L'ISA (*Instruction Set Architecture*) *open-source* RISC-V [19], dans sa configuration RVI, offre au compilateur une cinquantaine d'instructions élémentaires. C'est

*. Le projet est soutenu par l'ANR sous le contrat ANR-21-CE39-0017.

alors au compilateur d'identifier les bonnes associations d'instructions afin de générer du code efficace. De par son implémentation matérielle le jeu d'instructions réduit ne supporte pas de modification post-fabrication. De nombreuses architectures aussi bien *low-cost* [2, 16, 20, 21, 1] que *high-end* [23, 24, 17] implémentent ce jeu d'instructions.
Les processeurs RISC-V *low-cost* bénéficient de diverses stratégies d'optimisation pour réduire au maximum leur coût matériel et leur consommation énergétique. A l'opposé, les processeurs RISC-V *high-end*, capables de faire tourner un système d'exploitation de type Linux intègrent des optimisations architecturales avancées [23, 24, 17].
L'introduction d'une unité de microcodage apporte de la flexibilité au processeur. Ce mécanisme permettrait le décodage d'instructions spécialisées vers des instructions *RISC-V* ou encore d'instructions *RISC-V* vers *RISC-V*. Il serait alors possible d'altérer dynamiquement le comportement de la microarchitecture sans modifier le binaire original, de compresser la taille des programmes, ou encore de modifier le déroulement d'une instruction. A notre connaissance, il n'existe pas de travaux dans la littérature traitant de l'intégration d'une telle unité dans un processeur RISC.
Le reste de l'article se décompose de la manière suivante. La section 2 présente le contexte de l'étude et l'architecture du cœur sélectionné, ainsi que les modifications apportées. L'environnement de test, la méthodologie d'expérimentation et les premiers résultats sont détaillés en section 3. La section 4, discute ces résultats, avant de conclure.

2. Proposition architecturale

2.1. Contexte de l'étude

Dans le cadre du projet ANR SEC-V, et en vue d'évaluer la pertinence du décodage dynamique d'instructions dynamique pour la sécurité matérielle, l'architecture RISC-V *high-end* CVA6 en version 64 bits développée par l'OpenHW Group a été choisie. Cette architecture nommée CV6A64 est un processeur à 6 étages de *pipeline*. Il est *out-of-order* à l'exécution et *in-order* à l'écriture. L'exécution dans le désordre permet d'exploiter les différentes unités fonctionnelles en parallèle lorsqu'une opération lente est exécutée, mais le flux d'instructions reste inchangé à la fin.
Pour rendre l'architecture réactive en fonction du contexte d'exécution, l'architecture doit pouvoir injecter des instructions voire des séquences d'instructions à la volée. Cette approche a été proposée dans certains travaux sur la sécurité matérielle. Dans [22], des instructions barrières sont insérées dans le *pipeline* d'architectures à exécution spéculative pour les protéger d'attaques de type SPECTRE. D'autres travaux [9], contrent les attaques par canaux cachés en injectant aléatoirement des instructions factices dans le flux d'origine.
L'objectif est donc de donner à l'architecture la capacité d'altérer le flux d'instructions venant du binaire exécuté de manière contrôlée. A l'instar des architecture CISC qui le font naturellement pour d'autres raisons, le processeur CVA6 sera doté d'une unité de microdécodage.

2.2. Étude des besoins fonctionnels

Une unité de microdécodage doit pouvoir générer des séquences d'instructions $\mathbf{N_P}$ pour chacune des **P** macro-instructions qu'elle doit gérer. Différentes stratégies sont possibles pour générer de telles séquences. Cependant, le plus générique se base sur l'utilisation d'une mémoire chargée de mémoriser les suites de micro-instructions nécessaires aux **P** macro-instructions tel que cela est schématisé dans la figure 1. Si on suppose que chaque macro-instruction est composée au maximum de **N** instructions, la plage d'adresses des différentes séquences à calculer est alors $[\mathbf{idx} \times \mathbf{N}, (\mathbf{idx} + 1) \times \mathbf{N} - 1]$, avec **idx** le numéro de la macro-instruction.

Chaque mot binaire représentant une instruction de la séquence $\mathbf{N_P}$ doit indiquer l'unité fonctionnelle ciblée (ex. *ALU* (*Arithmetic-Logic Unit*), *LSU* (*Load-Store Unit*), ...) et l'opération (ex. ADD, XORI, ...) qu'elle doit effectuer. L'étude se focalise sur des séquences d'opérations arithmétiques et logiques. Aussi, il est nécessaire de mémoriser les registres sources (`rs1` et `rs2`) et de destination (`rd`), ou une éventuelle valeur immédiate. Lors du microdécodage d'une macro-instructions, les registres attribués par le compilateur peuvent ne pas être suffisants pour mémoriser des données temporaires. Ces dernières doivent être mémorisées dans des registres annexes afin de ne pas corrompre l'exécution du programme. En conséquence, les adresses des registres `rd`, `rs1` et `rs2` ont été étendus sur 6 bits permettant l'ajout de 32 registres, uniquement accessible via le microdécodeur.
Sans optimisation de la structure de la mémoire du microdécodeur et en considérant que les données immédiates sont encodées sur 64 bits, la taille **M** de chaque micro-instruction est de 94 bits. Le coût global de la mémoire évoluerait de manière linéaire avec **P** : $\mathbf{P} \times \mathbf{N} \times \mathbf{M} = \mathbf{P} \times \mathbf{N} \times 94$ bits. Afin de piloter cette mémoire, une machine d'états (*FSM : Finite State Machine*) est nécessaire. Cette machine a deux modes principaux :

- un mode *bypass* qui transmet les instructions classiques à l'étage de *pipeline* suivant ;
- un mode d'injection des micro-instructions dans le *pipeline* qui se déclenche dès lors qu'une macro-instruction est détectée.

L'opération de microdécodage inclut le calcul d'adresse en fonction de séquence à injecter. La synchronisation avec les autres étages peut être faite à l'aide d'une barrière de registres voire d'une file de registres de type *FIFO* (*First In First Out*).

2.3. Proposition architecturale

Une vue simplifiée de l'architecture du cœur CV6A64 est présentée en figure 4. L'architecture ciblée comporte 6 étages de *pipeline*. Les 2 premiers étages (*Frontend*) regroupent la génération du PC, le cache d'instructions, la prédiction de branchement et une file d'attente d'instructions. L'étage suivant (*ID*) réaligne les instructions (décalage induit par la présence d'instructions compressées) et traduit les instructions en signaux de contrôle.
L'étage (*Issue*) intègre le *scoreboard* du CV6A64. Ce dernier contrôle les 2 étages suivants en ordonnançant les opérations en fonction des ressources disponibles. C'est dans cet étage que se réalisent les accès aux files de registres (GPR et FPR). L'étage d'exécution (*EX*) se charge de la réalisation des opérations par les unités fonctionnelles. Enfin, le dernier étage (*Commit*), retire les instructions du *pipeline* et récupère les résultats en attentes.
Le choix a été fait d'ajouter à l'architecture une tranche de *pipeline* dans laquelle se trouvera le microdécodeur. Cette stratégie permet d'apporter de la flexibilité et évite d'impacter le chemin critique de l'architecture à synthétiser. Comme illustré dans la figure 5, la tranche de *pipeline* supplémentaire intègre 3 éléments :

1. une *FIFO* en charge de la synchronisation (signaux d'acquittements) avec l'étage *Issue*. Elle permet de temporiser les instructions car le microdécodeur peut générer une instruction par cycle et donc saturer le *scoreboard*. Sa profondeur a été fixée à 2 tandis que sa largeur est liée aux signaux de contrôle véhiculés dans le *pipeline* soit 364 bits. C'est une *FIFO* de type *FWFT* (*First-Word-Fall-Through*), réduisant la pénalité d'accès aux données.
2. une FSM associée à un compteur d'adresses. Dans son état initial, elle se comporte comme une barrière de registres : elle enregistre les informations pertinentes (PC, valeur immédiate, `rd`, `rs1`, `rs2`, ...) des instructions décodées. Lorsqu'une macro-instruction décodée est présentée, elle déclenche une séquence de microdécodage. En fonction de son **idx** $\in$ **P**, la FSM

calcule l'adresse correspondant à la bonne séquence en mémoire et passe dans l'état de microdécodage. Lorsque le parcours de la séquence est terminé, la FSM repasse dans l'état (*bypass*).

3. la mémoire contenant les séquences de micro-instructions. Afin de minimiser sa complexité, la taille des champs `rd`, `rs1` et `rs2`, a été redimensionnée sur 3 bits. Cela laisse la possibilité de cibler `rd`, `rs1` et `rs2` de la macro-instruction, enregistrés par la FSM, ou bien 5 registres ajoutés au processeur pour les données temporaires. Ces 3 bits sont ensuite décodés pour former les adresses sur 6 bits, pour les étages suivants. Le champ contenant la valeur immédiate a été réduit à 10 bits. Enfin, un bit (*skip*) signale la fin de la séquence d'instructions. À l'aide de ces changements, la taille des mots est de 32 bits.

Le contenu de la mémoire est fortement lié au contexte applicatif. Un exemple inspiré d'une des applications de test (*benchmark*) utilisées dans la partie résultats a été développé. Il s'agit du calcul d'une table de substitution. La S-Box (table de substitution), utilisée dans l'algorithme de chiffrement AES, est généralement pré-calculée et stockée en mémoire afin d'accélérer le chiffrement des données. L'équation 1, qui génère le contenu de la S-Box, nécessite des opérations logiques : SHIFT, AND, OR, XOR afin d'effectuer de rotation sur 8 bits ($\circlearrowleft$).

$$s = b \oplus (b \circlearrowleft 1) \oplus (b \circlearrowleft 2) \oplus (b \circlearrowleft 3) \oplus (b \circlearrowleft 4) \oplus 0x63 \quad (1)$$

Cette opération, se décompose en 18 instructions RISC-V, visibles dans le Listing 1. La génération de la séquence d'instructions et son ordonnancement ont été obtenus à l'aide de GCC de manière à optimiser le temps d'exécution. Cependant, comme toutes les instructions ne sont pas interdépendantes, il est possible de modifier l'ordre et la nature de certaines des instructions. Deux séquences différentes effectuant le même calcul ont donc été encodées dans la ROM (cf. figure 6). La première version est un réarrangement de la séquence d'origine tandis que la seconde remplace les opérations OR par des XOR. Afin d'avoir des séquences de tailles **N**, multiples de 2, des instructions NOP ont été ajoutées. Le nombre moyen de micro-instructions par séquence dans la ROM permettra de trouver un compromis sur le paramètre **N**, limitant les pertes mémoires tout en bénéficiant de séquences d'instructions à tailles variables. Dans le cadre de cet exemple, le coût mémoire est de $P \times N \times M = 2 \times 32 \times 32 = 2048$ bits.

3. Résultats expérimentaux

Dans cette section, l'impact de l'insertion de la tranche de *pipeline* sur les temps d'exécution d'un ensemble d'applications de test a été évalué et l'augmentation de la complexité matérielle de l'architecture a été mesurée.

3.1. Environnement de test

Les travaux ont été évalués sur une version du cœur CV6A64 datant du 13 janvier 2023 [23]. L'architecture a été synthétisée et implémentée sur un circuit *FPGA* Kintex-7 via l'outil Vivado 2020.2 de Xilinx. La *toolchain* pour générer les programmes de test se base sur le compilateur GCC en version 12.1.0.

Dans un premier temps l'architecture a été simulée au niveau cycle à l'aide de l'outil Verilator (version 4.110). Cela a permis d'évaluer finement les temps d'exécution des divers *benchmarks*.

Dans un second temps, les *benchmarks* ont été exécutés sur *FPGA* grâce à l'utilisation d'un noyau Linux 5.10.7 fourni par OpenHW Group [15].

Les *benchmarks* issus de différents contextes applicatifs proviennent de la littérature [8, 18, 4, 3, 10]. Ces applications n'exploitent donc pas les capacités du microdécodeur implémenté. Elles

permettent de quantifier les sur-coûts temporels induits par les modifications architecturales apportées. Le nom, la provenance ainsi que les caractéristiques des applications de test sont résumés dans le tableau 1. Chacun des *benchmarks* intègre une procédure d'auto-validation afin de confirmer la validité du comportement de l'architecture enrichie.
Les *benchmarks* ont été déclinés en deux versions. La première, simulée par Verilator, a été adaptée pour une exécution *baremetal* réduisant les temps de simulations. La deuxième version a été exécutée sur la carte. Afin de simplifier les expérimentations et l'échange d'information, le format binaire `.elf` a été exploité pour nos exécutables.
La mesure du temps d'exécution se base sur l'utilisation des compteurs présents dans le CSR. Cette solution permet d'avoir une mesure précise des durées exprimées en cycles d'horloge.

3.2. Impact sur les performances temporelles

Les temps d'exécutions obtenus par simulation de l'architecture CV6A6 initiale sont résumés dans le tableau 1. Ces chiffres démontrent la diversité des applications sélectionnées, avec des *benchmarks* nécessitant de quelques milliers à plusieurs millions de cycles d'horloges.
La figure 2 présente les écarts relatifs entre les temps d'exécutions des *benchmarks* sur l'architecture vierge et ceux sur l'architecture enrichie. Les écarts minimum et maximum sur les temps d'exécutions sont respectivement de -1,2% et +0,3% au maximum tandis que la moyenne est de -0,1%. Ces résultats démontrent que la nouvelle tranche de *pipeline* n'induit quasiment pas de latence supplémentaire. La *FIFO FWFT* est même bénéfique sur certains *benchmarks* (cf. #3, #12, #15 et #17).
Les deux architectures simulées ont ensuite été synthétisées, placées et routées sur le *FPGA* présent sur une carte Genesys II. Cela nous a permis de mesurer le temps d'exécution en mode réel sur un *OS* Linux léger. La figure 3 montre les écarts relatifs de temps d'exécution dans cette configuration.
Contrairement aux résultats de simulation *cycle-accurate*, ceux obtenus sur circuit *FPGA* sont plus difficiles à interpréter. En effet, si les écarts de performance moyens sont de ≈ +3%, les différences varient de -47,5% à +64,4%. Cette grande disparité est dûe à l'ordonnanceur de l'*OS* qui préempte régulièrement les *benchmarks* pendant leur exécution. Ces priorisation des ressources bruitent fortement les mesures. On peut observer sur la figure 3) que l'*OS* a fortement pénalisé le CV6A64 vierge (cf. #3, #7, #10 et #16) et la version enrichie (cf. #8 et #17), lorsqu'il a monopolisé les ressources du cœur. Ce bruit de mesure ne pouvant pas être éliminé, un *boot loader* dédié permettant l'exécution de binaires sans *OS* est en cours de développement.
Le même protocole a été suivi pour les deux exemples pédagogiques (S-Box de l'algorithme AES). En simulation, sans exploiter le microdécodeur, la S-Box est remplie par le CV6A64 vierge en 55 564 cycles d'horloges. Les *timings* sur l'architecture proposée, avec et sans instructions *custom*, sont respectivement de 47 145 et 55 562 cycles d'horloges. Il est à noter que les deux séquences microcodées ont la même durée, contenant le même nombre d'instructions exécutées en un cycle d'horloge seulement. Les deux architectures sont donc équivalentes quand le microdécodeur n'est pas exploité. En revanche, les performances sont meilleures (-15,2% cycles d'horloges) lorsqu'il est utilisé. Cela est dû au fait que la ROM injecte dans le *pipeline* une instruction par cycle d'horloge tandis que l'architecture vierge doit les *fetch* depuis son cache d'instructions voire depuis sa mémoire principale, subissant de lourdes pénalités temporelles.
Sur *FPGA*, sans microdécodage et en neutralisant les mesures bruitées, la S-Box est remplie par le l'architecture vierge en 68 859 cycles et par celle enrichie en 68 048 cycles (-1,2%). En utilisant le microdécodage, l'architecture proposée requière approximativement 47 554 cycles d'horloges pour les deux versions (-30,9%), confirmant les écarts observés en simulation.

3.3. Impact sur la complexité matérielle

Les répercussions sur la complexité matérielle du processeur ont aussi été évaluées. Cette évaluation se base sur les rapports fournis par le logiciel de synthèse Xilinx Vivado après placement et routage. Une synthèse des données récoltées est fournie dans le tableau 2. Il est à noter que la fréquence maximale de fonctionnement reste inchangée, démontrant l'absence de chemin critique dans notre étage de microdécodage.
On note une augmentation de la complexité matérielle de 3,9% concernant les LUT et de 4,2% pour les FF. Ces augmentations sont principalement dues à l'ajout de 5 registres dans la file GPR et de l'insertion d'une *FIFO* de profondeur 2 et de largeur 364 bits entre l'étage de microdécodage et l'étage d'exécution. De plus, dans sa version avec 2 macro-instructions, la mémoire de notre étage est implantée par de la mémoire distribuée.
L'augmentation de la complexité n'est pas linéaire avec le nombre d'instructions comme le montre une seconde expérience dont les résultats sont fournis dans le tableau 2. En effet, le passage de **P** = 2 à **P** = 64 provoque la réservation de 2 blocs RAM supplémentaires alors que le nombre de LUT et de FF reste proche.

4. Discussion

Les travaux présentés dans cet article traitent du décodage de macro-instructions composées uniquement d'instructions arithmétiques et logiques. Cette unité de microdécodage permet à l'architecture de changer la manière dont les macro-instructions sont réalisées matériellement, lui permettant de s'adapter aux applications et aux contextes d'exécution. Afin d'augmenter sa versatilité, l'étude et l'intégration des instructions de branchement et d'accès mémoire sont en cours. La difficulté majeure étant de s'assurer de la cohérence des accès mémoire. En supplément de ces travaux, l'utilité de l'unité de microdécodage sera étudiée :

- pour réduire de la taille des binaires. En effet, l'identification de *patterns* d'instructions récurrents [12] et leurs ajouts dans la mémoire de microdécodage diminueraient la taille des programmes et limiteraient les défauts de cache d'instructions.
- pour faire de l'obfuscation statique. L'ajout d'instructions *spécialisées*, comparables à des "boîtes noires", permettrait de masquer au niveau du binaire le déroulement du programme et donc de complexifier des tentatives de *reverse-engineering*.
- pour résoudre des problématiques de sécurité. Il devient en effet possible d'injecter dans le *pipeline* du processeur des instructions fantômes [9, 22] et/ou de changer la manière dont les calculs sont réalisés pour complexifier les attaques par canaux cachés.

5. Conclusion

Dans cet article, une unité de microdécodage de type CISC a été implémentée dans un processeur *high-end* de l'écosystème *open source* RISC-V. Cette solution technologique n'impute que de faibles sur-coûts en terme de latence pour les applications n'exploitant pas cette fonctionnalité. De plus, le sur-coût en terme de complexité matérielle est faible au regard de la complexité du coeur CVA6. Ces travaux démontrent que l'hybridation d'une architecture RISC-V par l'ajout d'un microdécodeur est envisageable. Cet enrichissement ouvre la voie à de nouvelles perspectives, notamment dans le domaine de la sécurité.

Bibliographie

1. Albartus (N.), Nasenberg (C.) et al. – On the design and misuse of microcoded (embedded) processors — a cautionary note. – In *Proceedings of the 30th USENIX Security Symposium (USENIX Security 21)*, pp. 267–284, 2021.
2. Cheng (Y.-H.), Huang (L.-B.) et al. – RV16 : An Ultra-Low-Cost Embedded RISC-V Processor Core. *Journal of Computer Science and Technology*, vol. 37, n6, 2022, pp. 1307–1319.
3. Fossati, Luca. – TRAP benchmark suite for SystemC and TLM based Instruction Set Simulators (ISS). – `http://code.google.com/p/trap-gen`.
4. Guthaus (M.), Ringenberg (J.) et al. – MiBench : A free, commercially representative embedded benchmark suite. – In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pp. 3–14, 2001.
5. Hasan (M.). – State of IOT 2022 : Number of connected IOT devices growing 18% to 14.4 billion globally. – https ://iot-analytics.com/number-connected-iot-devices, 2022. Consulté : 29 Mars 2023.
6. Kocher (P.), Horn (J.) et al. – Spectre Attacks : Exploiting Speculative Execution. – In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.
7. Koppe (P.), Kollenda (B.) et al. – Reverse Engineering x86 Processor Microcode. *CoRR*, vol. abs/1910.00948, 2019.
8. Le Gal (B.) et Jego (C.). – Softcore Processor Optimization According to Real-Applicaion Requirements. *IEEE Embedded Systems Letters*, vol. 5, n1, 2013, pp. 4–7.
9. Leplus (G.), Savry (O.) et Bossuet (L.). – Insertion of random delay with context-aware dummy instructions generator in a RISC-V processor. – In *Proceedings of the 2022 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 81–84, 2022.
10. Lin (S.) et Costello (D. J.). – *Error control coding : fundamentals and applications*. – Upper Saddle River, NJ, Pearson/Prentice Hall, 2004.
11. Lipp (M.), Schwarz (M.) et al. – Meltdown : Reading Kernel Memory from User Space. – In *Proceedings of the 27th USENIX Security Symposium (USENIX Security 18)*, 2018.
12. Martin (K.). – *Génération automatique d'extensions de jeux d'instructions de processeurs*. – Thèse de PhD, Université Rennes 1, 2010.
13. OpenHW Group. – CVA6 Documentation. – `https://cva6.readthedocs.io`.
14. OpenHW Group. – CVA6 RISC-V CPU. – `https://github.com/openhwgroup/cva6`.
15. OpenHW Group. – CVA6 SDK. – `https://github.com/openhwgroup/cva6-sdk`.
16. Patsidis (K.), Konstantinou (D.) et al. – A low-cost synthesizable RISC-V dual-issue processor core leveraging the compressed Instruction Set Extension. *Microprocess. Microsystems*, vol. 61, 2018, pp. 1–10.
17. Petrisko (D.), Gilani (F.), Wyse (M.) et al. – BlackParrot : An Agile Open-Source RISC-V Multicore for Accelerator SoCs. *IEEE Micro*, vol. 40, n4, 2020, pp. 93–102.
18. RealNetworks, Inc. – Helix MP3 Decoder, Real Networks Inc. – `http://www.helixcommunity.com.cn`.
19. RISC-V Foundation. – The RISC-V Instruction Set Manual, Volume I : User-Level ISA, Document Version 20191213, 2019. Consulté : 30 Mars 2023.
20. Santos (D. A.), Luza (L. M.) et al. – A Low-Cost Fault-Tolerant RISC-V Processor for Space Systems. – In *Proceedings of the 2020 15th Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, pp. 1–5, 2020.
21. Serrano (R.), Sarmiento (M.) et al. – A Low-Power Low-Area SoC based in RISC-V Processor for IoT Applications. – In *Proceedings of the 2021 18th International SoC Design Conference (ISOCC)*, pp. 375–376, 2021.

22. Taram (M.), Venkat (A.) et Tullsen (D.). – Mitigating Speculative Execution Attacks via Context-Sensitive Fencing. *IEEE Design & Test*, vol. 39, n4, 2022, pp. 49–57.
23. Zaruba (F.) et Benini (L.). – The Cost of Application-Class Processing : Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, n11, 2019, pp. 2629–2640.
24. Zhao (J.), Korpan (B.) et al. – SonicBOOM : The 3rd Generation Berkeley Out-of-Order Machine. – In *Proceedings of the 4th Workshop on Computer Architecture Research with RISC-V*, 2020.

Type	#	Nom	Temps exec. (cycle)	Source
Communication numérique	1	ADPCM enc./dec.	208k	MiBench[4]
	2	BCH(31,21) enc./dec.	192k	[10]
	3	GOLAY enc./dec.	2k	[10]
	4	LDPC dec.	729k	[8]
	5	MP3 dec.	2,6M	Helix [18]
Cryptographie	6	CRC 32b enc./dec.	97k	TRAP [3]
	7	MD5	93k	Fast DES kit
	8	SHA-1	62k	MiBench[4]
	9	SHA-2	100k	OpenSSL lib.
	10	ARC4 enc./dec	118k	OpenSSL lib.
	11	DES/3-DES enc./dec	2,8M	Fast DES kit
	12	AES (128/512) enc./dec.	18k	OpenSSL lib.
Traitement divers	13	Engine control	1,5M	TRAP [3]
	14	Data sorting (bubble)	2,7M	TRAP [3]
	15	Queens	4k	TRAP [3]
	16	Pattern matching (text)	3,8M	TRAP [3]
Traitement du signal	17	LMS filter processing	12k	[8]
	18	FIR filter processing	87k	TRAP [3]
	19	FFT & iFFT (fixed p.)	1,1M	FFTW src
	20	Echo cancellation	212k	LibGSM
Traitement vidéo	21	JPEG dec.	883k	TRAP [3]
	22	Motion detection	368k	[8]
	23	Contrast egalization	388k	[8]

TABLE 1 – *Benchmarks* utilisés pour les mesures

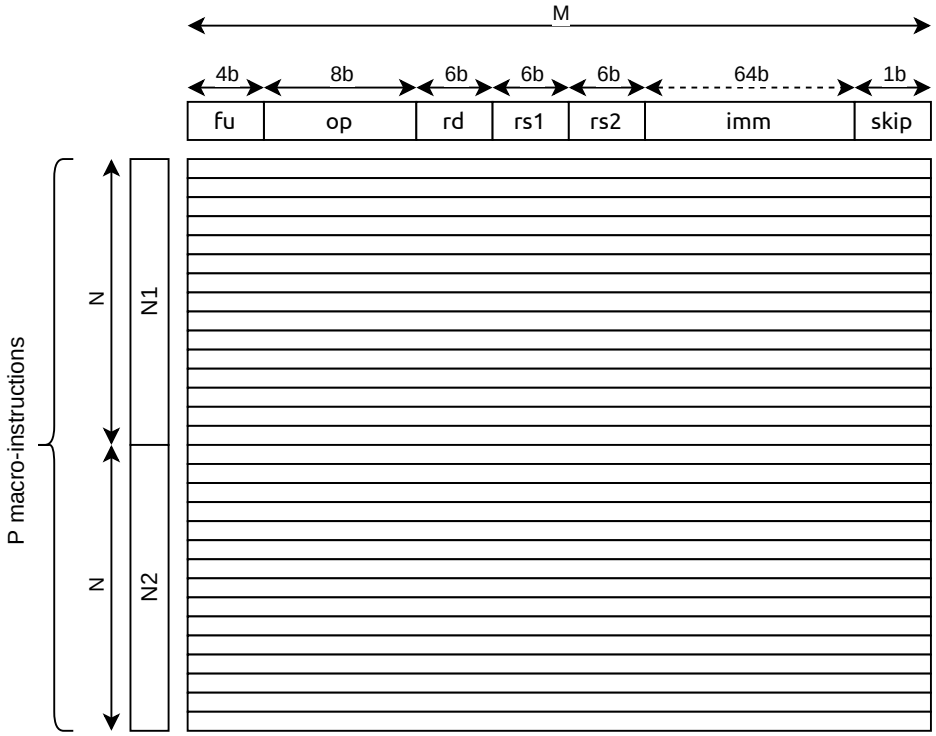

FIGURE 1 – Formalisation des propriétés de la ROM.

```
1  srliw   a2,a0,7
2  srliw   a3,a0,4
3  slliw   a5,a0,1
4  slliw   a4,a0,4
5  or      a4,a4,a3
6  or      a5,a5,a2
7  xor     a5,a5,a4
8  slliw   a2,a0,2
9  srliw   a4,a0,6
10 or      a2,a2,a4
11 slliw   a3,a0,3
12 srliw   a4,a0,5
13 xor     a5,a0,a5
14 xor     a0,a5,a2
15 or      a5,a3,a4
16 xor     a0,a0,a5
17 xori    a0,a0,99
18 andi    a0,a0,0xff
```

Listing 1 – Suite d'instructions assembleur pour le calcul d'un élément de la S-Box

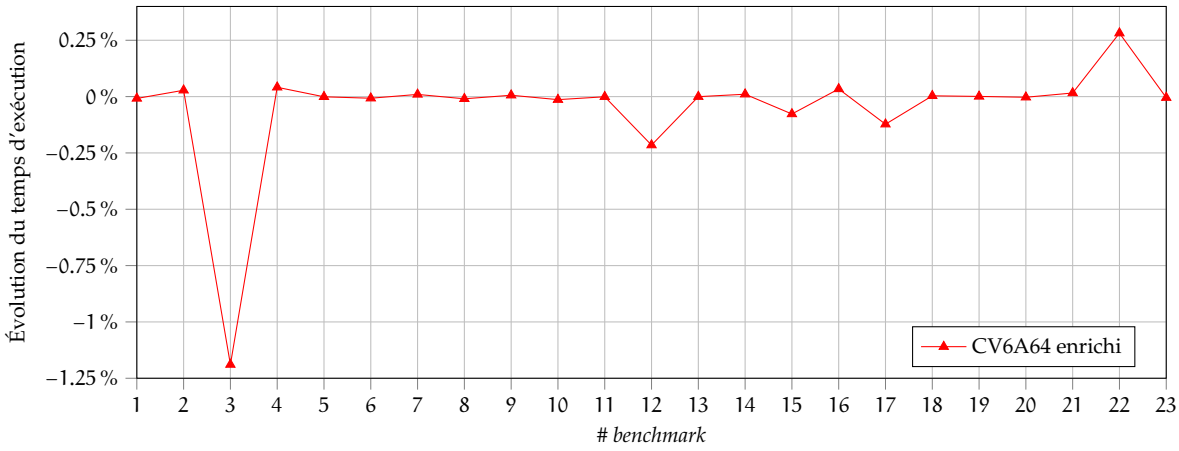

FIGURE 2 – Évolution du temps d'exécution des *benchmarks* simulés via Verilator entre les versions CV6A4 enrichie et d'origine

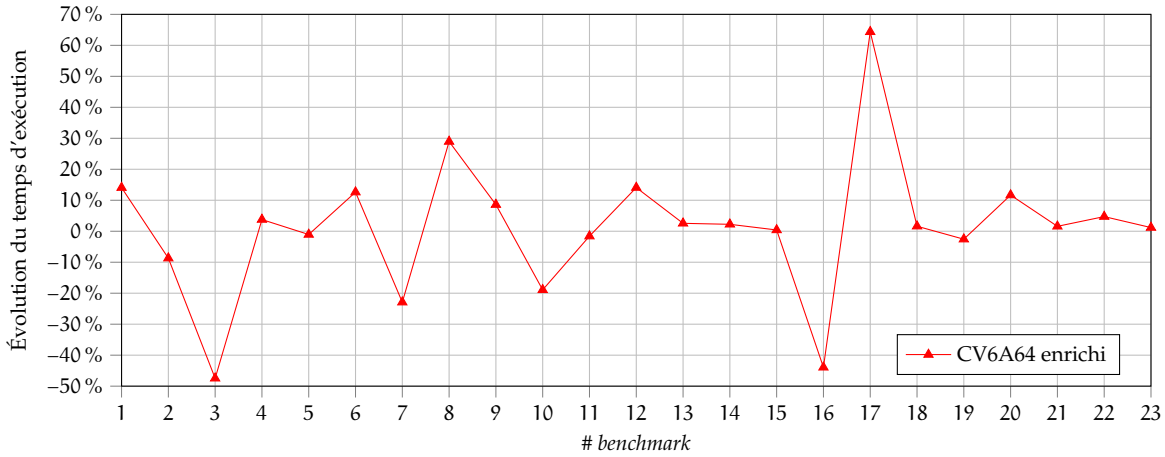

FIGURE 3 – Évolution du temps d'exécution des *benchmarks* exécutés sur carte entre les versions CV6A4 enrichie et d'origine

	LUT	SRL	FF	BRAM36	BRAM18	DSP
Coeur vierge	44 148	0	24 076	36	0	27
Coeur enrichi (2 instr.)	45 880	0	25 078	36	0	27
	+1 732	-	+1 002	-	-	-
	+3,9%	-	+4,2%	-	-	-
Coeur enrichi (64 instr.)	45 937	0	25 068	38	0	27
	+1 789	-	+992	+2	-	-
	+4,1%	-	+4,1%	+5,6%	-	-

TABLE 2 – Comparaison des complexités matérielles du CV6A64 intégrant ou pas l'unité de microdécodage

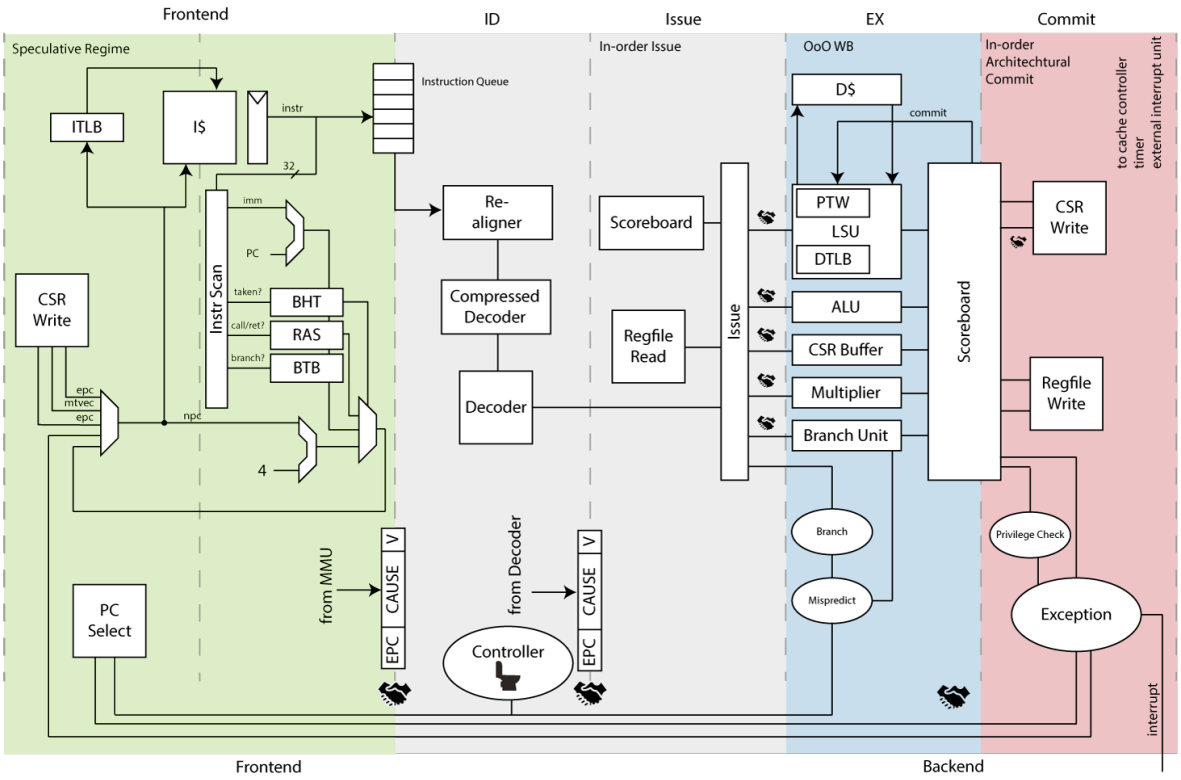

FIGURE 4 – Architecture du coeur CV6A64 de OpenHW Group [14] [13]

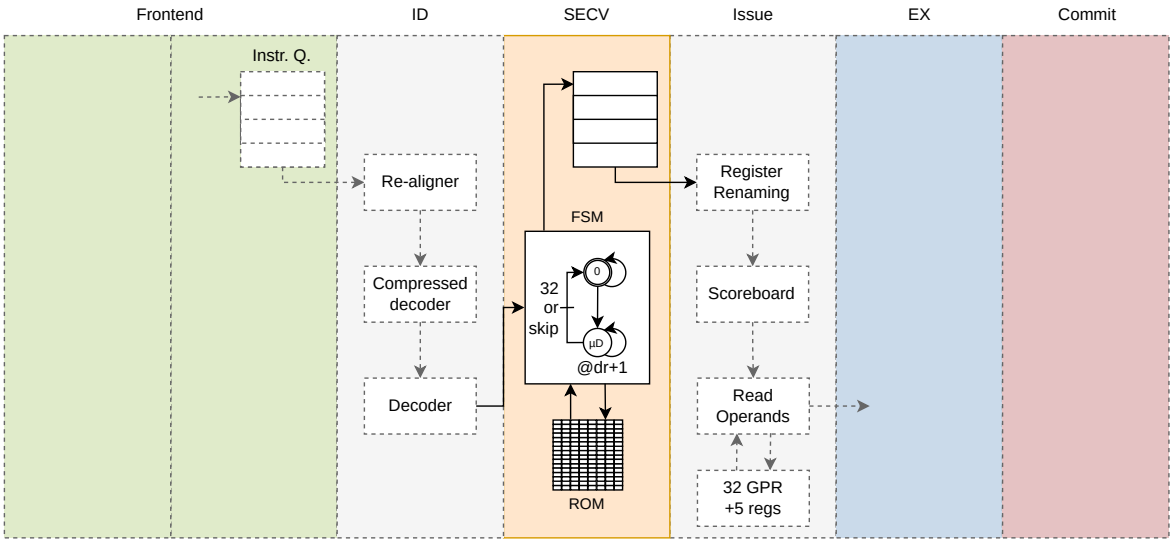

FIGURE 5 – Architecture du coeur CV6A64 enrichie avec notre tranche de *pipeline* et nos registres supplémentaires

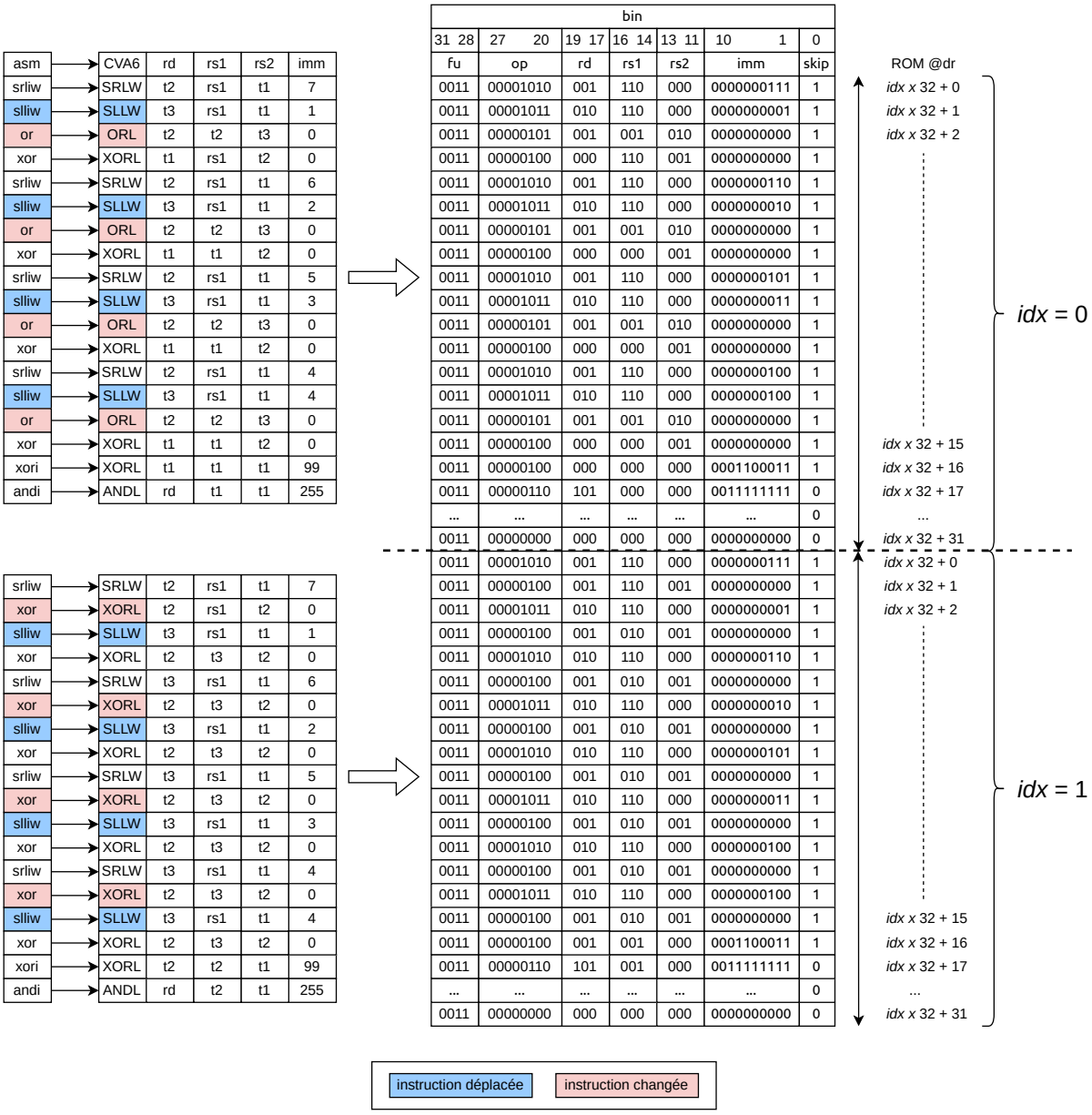

FIGURE 6 – Encodage dans la ROM du microdécodeur des macro-instructions pour le calcul d'élément de la S-Box