Power optimisation of the CV32A6 RISC-V soft-core

Alexandre GORIO
IMT Atlantique
Brest, France
alexandre.gorio@imt-atlantique.net

Quentin KY
IMT Atlantique
Brest, France
quentin.ky@imt-atlantique.net

Karl LA GRASSA
IMT Atlantique
Brest, France
karl.la-grassa@imt-atlantique.net

Julien PORTANGUEN
IMT Atlantique
Brest, France
julien.portanguen@imt-atlantique.net

***Abstract*—Cet article retrace les différentes étapes de notre participation au deuxième concours étudiant RISC-V pour l'année 2021-2022, se focalisant sur l'optimisation d'énergie d'un coeur Ariane, le processeur CV32A6. Notre équipe de quatre étudiants a travaillé sur cette architecture avec une application de réseau de neurones profonds (*CNN*) comme cible de travail. Cette application travaille sur MNIST, une base de données bien connue de chiffres manuscrits sur laquelle les CNN sont régulièrement testés. Cet article revient tout d'abord sur les différentes architectures existantes ainsi que l'intérêt de la réduction de consommation. La majeure partie de cette restitution sera focalisée sur les différentes solutions que nous avons trouvées et se tournera ensuite sur celles qui sont implémentables en pratique dans le cadre de ce concours. Nous reviendrons sur le fonctionnement de quelques unités pour expliquer le chemin de notre réflexion et ainsi rendre compte facilement des idées qui nous sont venues.**

I. Introduction

En 2022, deux ISA (*Instruction Set Architecture*) en particulier dominent sur le marché des processeurs. Le premier, développé par la société *Intel*, est le jeu d'instruction x86 basé sur une architecture CISC (*Complex Instruction Set Computer*) qui faisait sens à l'époque où la mémoire était particulièrement couteuse et peu disponible. La majorité des CPU que ce soit chez Intel avec la gamme des *Intel Core* ou chez AMD avec la gamme des *AMD Ryzen* se basent sur cette architecture afin de supporter notamment Windows et ces applications compilées pour du x86. Le deuxième, développé par la societé *ARM Holdings*, est le jeu d'instruction ARM basé sur une architecture RISC (*Reduced Instruction Set Computer*) dont le but est de réaliser des opérations complexes à partir d'opérations simples et réalisées en un cycle d'horloge (permettant alors un pipelining). Les systèmes embarqués actuels sont quasiment tous équipés d'un ou de plusieurs coeurs ARM notamment pour faire tourner Android sur smartphone, interfacer différents hardwares ou faire face à des contraintes de temps réel. Du fait qu'elles ont été développées en interne par ces entreprises, ces deux architectures sont propriétaires et donc nécessitent non seulement une license d'exploitation pour les utiliser et les vendre sur le marché, mais elles sont aussi non modulables. Dans ce contexte a émergé d'un projet étudiant de *l'Université de Berkeley* une autre architecture RISC open-source, RISC-V.

L'utilisation d'un système autonome est intimement liée à sa consommation : une puce nécessitant beaucoup d'énergie pour fonctionner et donc devant être rechargée régulièrement ne pourra pas être utilisée dans des applications critiques comme la défense, la médecine ou le spatial. Le budget énergétique idéal visé serait de 100µW ; avec une telle consommation, cela permettrait aux systèmes embarqués de tenir plusieurs mois voire années sans besoin d'intervention humaine pour recharger une batterie ou changer les piles. A l'heure actuelle, la consommation est plutôt de l'ordre du mW, descendant difficilement en dessous.

L'ensemble des résultats présenté dans cet article ont été synthétisé avec la version 2020.1 de Vivado de chez Xilinx, et simulé avec la version 2021.3 de Questa de chez Mentor. Le dépôt GitHub originel de Thalès a été cloné depuis le dépôt officiel du concours 2021-2022 depuis cette adresse. Après simulation et validation des modèles, le bitstream est généré avec la même version de Vivado, et programmé sur une carte Zybo Z7, équipé d'un SoC Zynq-7000 dont seule la partie PL (*Programmable Logic*) sera sollicitée. L'application MNIST est enfin exécutée sur l'architecture en baremetal à l'aide d'OpenOCD.

Notre équipe se compose de quatre membres, auteurs de cet article, Alexandre Gorio, Quentin Ky, Karl La Grassa et Julien Portanguen, élèves en dernière année à l'IMT Atlantique sur le campus de Brest en spécialité systèmes embarqués. Nous avons été encadrés par Stefan Weithoffer, maître de conférences, Yehya Nasser, enseignant-chercheur et Amer Baghdadi, professeur, à l'IMT Atlantique. Notre équipe s'est organisée naturellement selon les compétences de chacun, que ce soit pour l'utilisation du GitHub, pour l'installation de l'environnement, la recherche bibliographique ou encore

l'implémentation verilog des différentes idées d'optimisation.

II. CV32A6

Le processeur CV32A6, créé par Thalès, découle du processeur CVA6 ou Ariane crée par OpenHW Group. Le CVA6 implémente une architecture RISC-V 64bits, tandis que le CV32A6 en est une version 32bits plus compacte. Elle implémente les extensions I, M et C. L'extension I qui est une extension de base, indique que le processeur gére les entiers. L'extension M implique d'avoir les multiplications et divions entières. L'extension C quant à elle, donne la possibilité d'utiliser des instructions compressées c'est à dire des versions 16bits des instructions 32bits. On remarquera que celui ci n'implémente par l'exension P (332 instructions) qui permet de faire du SIMD ou un MAC (multiply accumulate), pratiques pour accélérer un réseau de neurone convolutif. Un travail d'implémentation de cette extension a été fait ici: [1]. Le CV32A6 a une profondeur de pipeline de 6. Cependant, le CVA6 a été designé pour des cibles ASIC et non FPGA. C'est pourquoi, il y a des optimisations à faire, notamment pour la consommation énergétique.

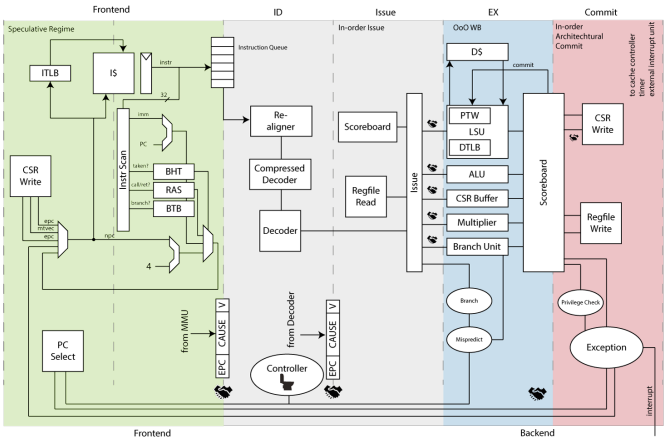

Fig. 1. CVA6 overview

III. L'APPLICATION

Le processeur doit faire tourner un réseau de neurones convolutif qui a été écrit en C et compilé grâce au cross-compiler fournit par les organisateurs du concours. Le compilateur a été mis en place pour les architectures 32bits avec les extensions I, M et A (–with-arch=rv32ima): l'extension C n'est donc pas utilisée. Ce réseau de neurones a été entrainé sur la base de données MNIST qui consiste en des images 24x24 pixels en nuance de gris avec des chiffres écrits à la main. L'application qui reçoit une image représentant un 4 doit, à l'issue de nos simulations et de l'implémentation, avoir une crédence de 82 et renvoyer le chiffres 4. Les poids sont codé en 8bits. Sur la figure 2, on remarque dans la couche conv2, que 70% des poids sont nuls, ce qui nous laisse penser que cette couche a été élagée. On estime que la couche conv1 prend 30K MACs et la conv2 prend 150K MACs. Les couches complétements connectés fc1 et fc2 comptent pour 50K et 9K MACs respectivements.

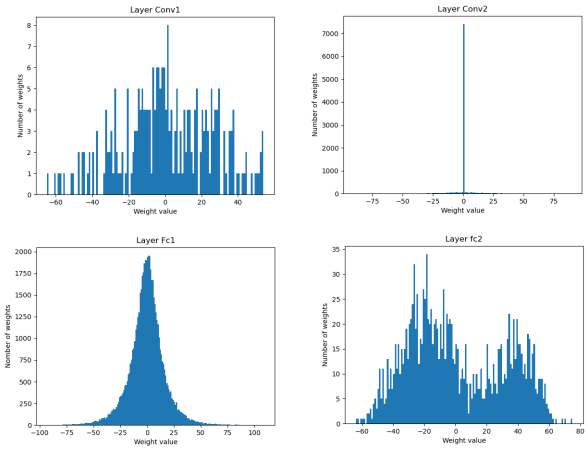

Fig. 2. Distribution des poids dans les couches du CNN

IV. ETAT DE L'ART

La première étape de ce projet a consisté en un état de l'art sur les voies d'optimisaton d'énergie mises en place dans les systèmes embarqués de façon générale. Cette première étape nous a orientés vers plusieurs solutions possibles :

- la réduction de la fréquence de fonctionnement, l'énergie pour un transistor variant en

$$E = \frac{1}{2} f C U^2$$

, celle-ci est uniquement envisageable si nous pouvions parvenir à réduire le nombre de cycles nécessaires à des opérations pour compenser cette diminution,
- la variation de la fréquence et de la tension en fonction de la demande du processeur, qu'on retrouve dans la littérature comme *Dynamic Frequency Scaling*,
- la minimisation des composants, de façon évidente, tous les composants dont le rôle peut être simplifié ou tout simplement supprimé peuvent être retirés de l'architecture,
- l'endormissement du CPU avec la désactivation des CAN (apportant généralement une baisse importante de la consommation ou encore la désactivation du *Brown out* (protocole de détection des tensions basses pour lancer un reset) qui inclue quelques risques,
- l'utilisation du *clock gating*, l'horloge provoquant des changements permanents de polarisation, il est intéressant de la combiner à un autre signal d'activation pour les composants inutilisés, de ce fait, il y a possibilité de les rendre inactifs et de supprimer leur consommation dynamique,
- l'utilisation du *power gating*, plus simplement, il s'agit ici de couper l'alimentation d'une partie du circuit qui serait inutilisée pendant une longue période par rapport au temps de fonctionnement de l'application.

Une fois cette étude préliminaire menée, la question principale fut de savoir lesquelles étaient implémentables en HDL mais aussi lesquelles pourraient entrer dans les limites du concours.

V. Le synthétiseur

Le synthétiseur utilisé est le synthétiseur de Vivado 2020.1. Il peut faire des optimisations et choisit le nombre de dsp utilisé par exemple. Il s'avère que les DSP sont utilisés pour les multiplications. On peut aussi spécifier les paramètres de la synthèse, avec l'option "gated_clock_conversion" par exemple pour convertir les signaux d'horloge avec l'attribut "GATED_CLOCK".

VI. Profiling

A. Profiling dynamique

Une étape de profiling (profilage) a été menée afin d'isoler les parties ayant besoin d'être optimisées. L'idée est de savoir quels sont les opérateurs basiques les plus utilisés pour les optimiser sur le code Verilog. Plusieurs outils ont été utilisés pour réaliser ce profilage :

- gprof : outil de la suite GNU GCC permettant de lire le résultat d'exécution de l'application et de faire des statistiques sur les appels de fonctions etc...
- QEMU : outil de virtualisation qui peut émuler des plate-formes différentes que l'hôte qui lance la virtualisation.
- gobj : outil de la suite GNU GCC permettant d'obtenir le code assembleur d'un exécutable.

La première étape était pour nous d'émuler le comportement de la carte sur QEMU. En effet ce dernier peut produire des logs assembleurs en utilisant les bons plugins. Cependant nous nous sommes heurtés à un problème, l'adresse mémoire de départ pour QEMU n'était pas le même de celui de la carte. Il fallait donc soit améliorer QEMU pour prendre en compte la carte ou modifier la carte pour qu'elle marche avec QEMU. Jugeant que l'une des deux solutions allait prendre trop de temps et au pire des cas produirait des résultats mais trop tard pour être utilisés : nous avons donc abandonnés cette piste. C'est à ce moment que nous avons utilisés la suite GNU, qui permet aussi de faire du profilage. Cependant le même problème persistait : comment stocker le résultat du profilage sur une application qui tourne bare-metal (sans OS) ? La solution "simple" est QEMU, cependant comme dit plus haut, cette solution n'est plus envisageable. La dernière piste est simplement d'utiliser objdump et de repérer les fonctions et ainsi avoir son assembleur, ce qui a été fait. A ce jour, le profilage est considéré comme en suspend. Les travaux d'optimisations continuent.

B. Désassemblage de l'application compilée

Afin de comprendre quelles sont les instructions les plus utilisées lors de la compilation de l'application, grâce à l'outil objdump de la suite GNU GCC il a été possible de retracer les instructions les plus utilisées. Le graphe du nombre d'appel d'instructions est représenté en $Fig.4$.

Ce graphe ne permet que de se donner une idée des instructions les plus utilisée par l'application puisqu'il ne représente que les instructions du set RISC-V utilisées pour réaliser les fonctions implémentées en C, et ne représente en aucun cas les instructions appellées pendant que l'application tourne sur l'architecture : c'est ce qu'on appellera par la suite le *profiling statique*.

TABLE I
Appels de fonction

Fonction	Nombre d'appels
macsOnRange	10274
clamp	2480
sat	2480
saturate	2480
read_mnist_input	4
convcellPropagate1	2
envRead	1
fccellPropagateDATA$_T$	1
fccellPropagateUDATA$_T$	1
feof_mnist_input	1
maxPropagate1	1
processInput	1
propagate	1
readStimulus	1

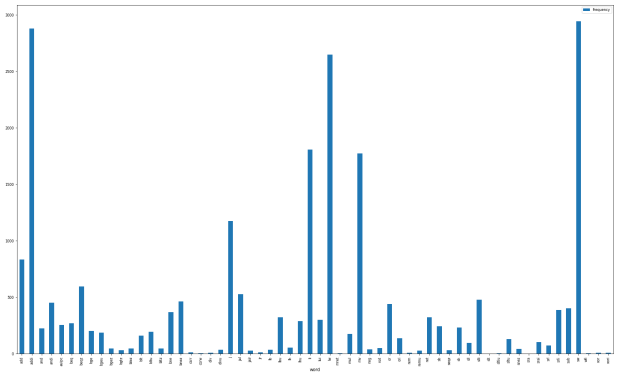

Fig. 3. Nombre d'appels d'instructions (issu du profilage)

Du profiling statique ressortent quelques tendances :

- Les instructions sollicitant le LSU (lw, li, sw) sont particulièrement présente.
- Le nombre d'addition immediate est particulièrement élevé.

On a pu alors en déduire que pour les functionals units (FU) sur lesquel le plus d'attention doivent être portés sont le LSU et l'ALU.

VII. Approximate computing

De part l'étude faite dans la partie précédente, cette idée d'approximate computing vise à optimiser l'unité fonctionnelle ALU (*Arithmetic Logic Unit*).

L'approximate computing est une technique visant à réduire la consommation énergétique en utilisant des operateurs dits "approximatifs" dont la consommation en puissance et en surface est moindre que les operateurs classiques (carry-lookahead par exemple pour les additionneurs), mais dont le résultat de l'opération n'est "qu'approximé", c'est-à-dire qu'il n'est pas exactement juste. Ainsi, un multiplicateur

approximatif pourrait calculer par exemple $35 * 42 = 1475$ au lieu de $35 * 42 = 1470$.

Il existe d'autres types d'opérations d'approximation pour la réduction de consommation telle que l'approximate memory qui consiste à lire le contenu de la mémoire de manière approximée, mais des résultats obtenus (trop grand changement des résultats) il n'était pas envisageable d'implémenter cette solution.

Dans la litterature, il existe des implémentations de CNN avec des operateurs approximatifs, où un gain de puissance à été largement observé, de l'ordre de 20% jusqu'à parfois 80%. En effet, un CNN étant un réseau de neurones entraînés, il paraît raisonnable de penser qu'il soit résistant à quelques erreurs de multiplications et d'additions.

L'idée de cette optimisation consiste alors à créer une seconde version de l'ALU implémentant uniquement des opérateurs approximés afin de limiter au maximum la consommation énergétique du CNN. Les opérateurs approximés ont pu été obtenu à partir de la librairie *EvoApproxLib*, disponible à ce lien. En 4 est présenté la relation entre la puissance consommée par multiplication et sa *Mean Relative Error* pour un multiplicateur 16 bits de la librairie *EvoApproxLib*.

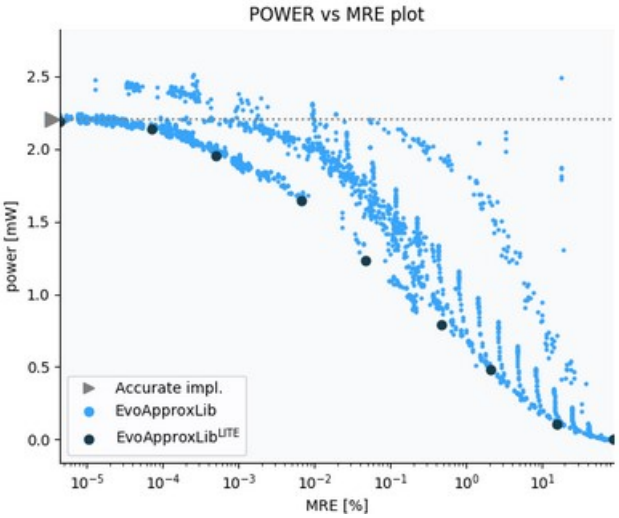

Fig. 4. Puissance en fonction du MRE

La difficulté de cette implémentation consiste à déterminer le moment où il est possible d'utiliser une opération approximée pour le CNN, et où il faut une version précise des opérateurs notamment pour le calcul des adresses. Après plusieurs simulations modélisées avec l'application en C, il est venu que l'opérateur de multiplication au sein de l'architecture CV32A6 est utilisées exclusivement pour les multiplications au sein de CNN et n'est donc pas utilisée ailleurs. De cette constatation, deux solutions ont été envisagées :

- Créer un flag lorsque le multiplicateur est appellé, afin de comprendre que la prochaine instruction d'addition est une addition pour le CNN, et donc peut-être approximée.
- Remplacer simplement les DSP utilisés par l'architecture CV32A6 pour un multiplicateur approximatif.

A l'heure actuelle, seule la deuxième solution a été implémentée. Des gains en puissances ont été observés suite à son implémentation ainsi que des résultats identiques en terme de résultats et nombre d'instructions appellées, mais la crédence, indice de mesure de confiance introduit dans le cadre de ce concours en Février 2022, se trouve très légèrement modifiée, ce qui est contraire aux règles du concours. De ce fait, à moins de réduire le nombre d'erreurs produits par l'opérateur, cette approche par approximate computing est pour l'instant écartée.

VIII. Approximate Memory

De part l'étude faite sur le profiling statique, cette idée d'Approximate Memory vise à optimiser l'unité fonctionelle LSU (*Load Store Unit*).

Toujours dans la même idée d'économiser des ressources en applicant des idées d'approximation, similairement à l'*Approximate Computing*, l'*Approximate Memory* consiste à lire les données de la mémoire (qu'il s'agisse d'un cache ou de la RAM) de manière approximative en fonction des bits. Par exemple, les MSB seraient lus avec précision alors que les LSB seraient lus avec une probabilité d'erreur en échange d'un coup énergétique moindre. Afin de vérifier la fiabilité de cette piste avant de l'implémenter en HDL, des simulations de Monte-Carlo ont été realisé sur l'application MNIST en C pour vérifier l'impact sur la crédence.

Les résultats font apparaître une gaussienne autour de la valeur de crédence 82 dès que le LSB est lu approximativement de la mémoire, et donc cette solution pour les mêmes raisons que la partie précédente ne peut être implémenté dans le cadre de ce concours.

IX. Optimized adder

L'architecture CVA6 utilisée au sein de ce concours est la version 32bit, autrement dénommée CV32A6. Le but de cette optimisation consiste à réduire le nombre d'additions de bits inutiles au bon fonctionnement de l'application.

Les opérations de multiplications du CNN se font avec des nombres entiers codés sur 8 bits, le résultat peut alors être codé sans pertes sur 16 bits. Ce résultat est ensuite sommé par un autre nombre entier codé sur 8 bits, le résultat doit alors être codé sur 17 bits pour ne perdre aucune précision. Enfin, afin de pouvoir coder en complément à deux, un bit de signe doit être rajouté, ce qui fait un total de 18 bits nécessaires.

L'architecture implémente un additionneur synthétisé par le synthétiseur de Vivado qui réalise plusieurs tâches, dont le calcul d'adresse qui doit se faire sur 32 bits. Ce même additionneur est utilisé pour le calcul du CNN, ce qui fait qu'au moins 14 bits sont alors calculés inutilement à chaque addition.

L'idée de cette optimisation, similaire à l'idée précédente présentée à la partie précédente, consiste à implémenter une version optimisée d'un additionneur faite uniquement pour les additions du CNN. En reprennant l'idée du "flag" renvoyé par le multiplicateur, il serait possible de connaître quand l'addition du CNN se produit, et donc d'utiliser la version faite

pour et donc d'économiser des bits à calculer, pour économiser de l'énergie. Cette idée est encore à implémenter.

X. Clock gating

Le clock gating est une pratique commune dans la littérature visant à réduire la consommation dynamique d'un circuit. Elle consiste à retirer le signal d'horloge d'un circuit synchrone lorsqu'il n'est pas sollicité afin qu'il ne gaspille pas d'énergie à, par exemple, faire "clocker" des flip-flops non utilisées. Cependant cela amène des considérations au niveau de la clock stew et des glitchs qui peuvent apparaitre. Afin d'implémenter ce concept dans le cadre de ce concours, plusieurs unités fonctionelles comme le FPU (*Floating Point Unit*) qui ne semblent pas nécessaires pour faire tourner l'application *mnist* ont été désactivés pour mesurer l'impact sur la consommation énergétique afin de voir s'il était avantageux de se concentrer sur son implémentation. Les résultats nous ont indiqués que même en retirant ces unités fonctionelles, la consommation énergétique n'était pas impactés suffisamment pour avoir un impact significatif. De ce fait, l'implémentation du clock gating a été écarté.

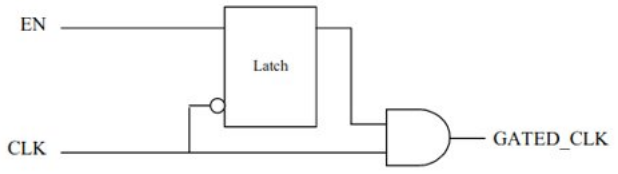

Fig. 5. Principe du clock gating

XI. Simplified multiplier

Une des optimisations les plus simples acceptée à ce jour désactive le multiplieur en cas de multiplication par 0. Afin de déterminer l'opération à réaliser par l'ALU, l'une de ses entrées est le code opération (*codeop*). Il peut alors s'agir :

- d'une addition ou soustraction (avec une seconde entrée, les opérandes étant passés en complément à 2, il s'agit simplement de changer un bit ensuite),
- d'un shift pour les multiplications ou divisions par des puissances de 2,
- des opérations logiques de base (ET, OU, OU exclusif).

Dans la cas d'une multiplication, les opérandes que l'on désignera par A et B sont connectés à l'entrée de l'unité fonctionelle du multiplicateur. Au vu de la partie III, il y a une couche de neurone *conv2* composé en très grande majorité de 0. Puisque le nombre de fois où l'unité fonctionnelle de multiplication est utilisé simplement pour calculer une produit de 0 est particulièrement élevée, il a été envisagé d'économiser de la puissance sur ce calcul: au lieu de solliciter le multiplicateur pour multiplier un nombre par 0 et donc de faire tout le calcul consommant de l'énergie, on a implémenté une idée qui consiste à renvoyer un 0 en sortie du multiplicateur si l'une des deux opérandes A ou B valent 0.

De ce fait, dans le cadre d'un CNN cette optimisation permet d'économiser simplement de la puissance de calcul.

XII. Résultats

Parmi toutes les idées d'implémentations pensées pour réduire la consommation en énergie de l'architecture CVA32A6, seule le *Simplified Multiplier* a pu être implémenté dans les temps respectant les règles, l'idée d'*Approximate computing* permettant effectivement de réduire aussi la consommation mais changeant la crédence (de 82 à 81). Les résultats de la simulations par le simulateur Questa est fournie dans le GitHub. Parmi les paramètres qui nous intéressent dans le cadre de ce concours :

- Le Total On-Chip Power(W) est de **0.306mW**.
- Le nombre de LUT utilisés est de **14693** (selon le rapport cva6.utilization.rpt).
- Le nombre de FFs est de **9318** (selon le rapport cva6.utilization.rpt).
- Le nombre BRAM utilisé est de **36** (selon le rapport cva6.utilization.rpt).
- Le nombre de cycle nécessaire pour faire tourner l'application est de **2098749**.
- La période utilisée est de **22.2 ns**.

De cette donnée, nous pouvons alors extraire les paramètres:

- L'énergie du design de référence : **14.30 mJ**.
- L'énergie de notre design : **14.26 mJ**.
- Le gain en énergie : **-0.33%**.
- Le temps nécessaire : **46.59 ms**.

Les résultats de la simulation sont bien ceux attendus (prédit un 4 pour un 4 avec une crédence de 82). On bien alors une optimisation respectant les contraintes du concours.

Conclusion

Ce projet aura évidemment été pour nous l'occasion de nous améliorer en langage de description matériel, ici le verilog mais aussi dans l'utilisation de GitHub ou encore dans la mise en place d'un tout nouvel environnment de développement et donc de flexibilité. La recherche documentaire faisant aussi partie du projet, elle fut un bon exercice pour ceux d'entre nous qui vont continuer en thèse. Les solutions que nous avons pu trouver ont souvent été la source de frustration, que ce soit à cause d'une implémentation trop difficile à mettre en place ou en raison d'un refus de la solution dans le cadre des limites définies par le concours.

Remerciements

Durant toute la durée de ce projet, nous avons pu compter sur nos encadrants. C'est pourquoi nous tenons à remercier particulièrement Yehya Nasser, enseignant-chercheur, Stefan Weithoffer, maître de conférences et Amer Baghdadi, professeur, tous à IMT Atlantique sur le campus de Brest. Nous remercions aussi Frédéric Pétrot, professeur à l'ENSIMAG, pour son support lors de l'utilisation de *Qemu* ainsi que Jean-Noël Bazin, ingénieur recherche et développement à l'IMT Atlantique pour son support lors de la mise en place de

l'environnement. Nous tenons aussi à remercier les organisateurs du concours à l'origine de ce projet que sont Thalès, CNFM et GDR SOC^2.

REFERENCES

[1] Davy Koene,2021, *Implementation and Evaluation of Packed-SIMD Instructions for a RISC-V Processor*

[2] Min Soo Kim, Alberto A.Del Barrio, Roman Hermida, Nader Bagherzadeh, 2018, *Low power implementation of Mitchell's approximate logarithmic multiplication for convolutional neural networks*

[3] Xilinx, 2020, *Vivado Design Suite User Guide*

[4] Andrew Waterman, Krste Asanovic, SiFive Inc., 2019, *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*

[5] Weixiong Jiang, Heng Yu, Xinzhe Liu, Yajun Ha, *Energy Efficiency Optimization of FPGA-based CNN Accelerators with Full Data Reuse and VFSs*

[6] Satyajit Bora and Roy Paily, 2021 *A High-Performance Core Micro-ArchitectureBased on RISC-V ISA for Low Power Applications*

[7] ETH Zurich and University of Bologna, 2020 *OpenHW Group CVA6 User Manual*