UCLouvain

epl

École polytechnique de Louvain

Comparaison de CPUs pour l'optimisation d'un microcontrôleur à ultra-basse consommation

Auteurs : **Damien Deprez**
Promoteurs : **David Bol**
Lecteurs : **David Bol**, **Jean-Didier Legat**, **Rémi Dekimpe**
Année académique 2018–2019
Master [120] : ingénieur civil électricien

ABSTRACT

Dans les années à venir, le nombre d'objets connectés va augmenter mais pas les ressources disponibles. Une grande partie de ces objets connectés sont des *Wireless Sensor Nodes* (WSN) également appelés *smartsensors*. Dans le futur, ces WSNs devront être totalement autonomes pour leur alimentation en énergie en récupérant de l'énergie dans l'environnement. L'énergie ainsi récupérée est très faible. C'est pourquoi les WSNs doivent consommer le moins d'énergie possible. Dans ce travail, nous allons nous intéresser à l'amélioration d'un WSN en terme d'efficacité énergétique en améliorant le *Micro Controller Unit* (MCU) utilisé par le WSN. Pour ce faire, nous cherchons à diminuer la consommation du processeur central (CPU). Nous comparons différents processeurs afin de déterminer lequel est le plus efficace pour diminuer la consommation d'énergie sans perdre de performances. Pour effectuer cette comparaison, nous utilisons les différents facteurs de mérite suivants pour quantifier un CPU : nombre d'instructions exécutées par seconde (performances), consommation électrique du CPU, surface, taille du code et nombre d'accès à la mémoire. L'analyse de données provenant de la littérature scientifique et celles provenant des fournisseurs montre que ces données ne sont pas comparables entre elles, car les conditions de test sont différentes. C'est pourquoi, afin de comparer différents processeurs, nous mettons en place un banc d'essai basé sur la simulation des processeurs. Il utilise le *benchmark* CoreMark. Nous trouverons dans ce travail le détail de l'architecture de ce banc d'essai et l'extraction des différents facteurs de mérite. Le processeur de référence dans ce travail de comparaison est le Cortex-M0 Design Start qui est présent dans la version actuelle du MCU à ultra-basse consommation développé à l'UCLouvain : SleepRunner. Nous avons choisi de le comparer avec Zero-riscy et RI5CY. Au final, le Cortex-M0 a un concurrent au niveau de l'efficacité énergétique dans certaines conditions. Si dans une première comparaison, Zero-riscy est 29% moins efficace que le Cortex-M0, il devient 21% plus efficace que ce dernier si nous effectuons une comparaison tenant compte de la consommation des mémoires. Nous nous trouvons dans cette deuxième comparaison avec la même configuration de mémoires que celle de SleepRunner. Par contre, RI5CY qui est conçu pour effectuer des tâches de traitement de signaux n'est pas du tout intéressant pour remplacer le Cortex-M0. Le banc d'essai développé pour ce travail de fin d'études est disponible en open source afin de permettre sa réutilisation pour prolonger ce travail de recherche en comparant d'autres processeurs ou en utilisant d'autres *benchmarks*.

Remerciements

Je tiens à remercier mon promoteur, David Bol, pour ses conseils judicieux et pour l'autonomie qu'il m'a laissé concernant mon choix de recherche. Je tiens également à remercier Rémi Dekimpe et Mathieu Xhonneux pour leur aide tout au long de ce mémoire. Et finalement, je remercie Jean-Didier Legat d'avoir accepté de faire partie des lecteurs de ce mémoire.

Table des matières

1 Introduction 5

2 Fondements de l'étude 8

2.1 Processeur 8

2.1.1 Fonctionnement d'un processeur 8

2.1.2 Comparaison de l'architecture Von Neumann et de l'architecture Harvard 10

2.1.3 Facteurs de mérite 11

2.2 État de l'art 15

2.2.1 État de l'art des comparaisons de processeur. 15

2.2.2 Présentation des processeurs utilisés dans ce travail 16

2.3 Benchmarks 17

2.3.1 Qu'est-ce qu'un benchmark ? 17

2.3.2 Classification des benchmarks 18

2.3.3 CoreMark 18

3 Mise en place du banc d'essai 19

3.1 Architecture du banc d'essai 19

3.2 Extraction des facteurs de mérite 22

3.3 Compilation du software 27

4 Caractérisation du Cortex-M0 DS 29

4.1 Analyse des performances 29

4.2 Analyse de la consommation 30

4.3 Analyse de la surface 31

4.4 Analyse des accès à la mémoire 32

5 Comparaison du Cortex-M0 avec les autres processeurs 34

5.1 Zero-riscy 34

5.2 RI5CY 35

5.3 Évaluation de la consommation globale 37

5.3.1 Variation de l'énergie d'accès de l'IMEM avec l'énergie d'accès de la DMEM constante 37

5.3.2 Variation de l'énergie d'accès de la DMEM avec l'énergie d'accès de l'IMEM constante 40

5.3.3 Variation de l'énergie d'accès de l'IMEM et de la DMEM à rapport constant 41

6 Conclusion 43

7 Table des illustrations 45

8 Abréviations utilisées 46
9 Bibliographie 47
10 Annexes 49
10.1 Code source du banc d'essai 49
10.2 Tableau de comparaison global des résultats des simulations 50

1 Introduction

Un grand nombre de visions de l'Internet des Objets (IoT) prévoient entre une vingtaine et une soixantaine de milliards d'objets connectés dans les cinq prochaines années [1, 2]. Tous ces objets connectés ne peuvent pas envoyer en continu toutes les données collectées. En effet, nous disposons de ressources limitées pour envoyer ces données collectées. Dans notre cas, la limitation s'effectue au niveau de la bande de fréquence de communication sans fil. Au plus le débit à transmettre est élevé, au plus la bande de fréquence requise doit être large. Pour diminuer cette consommation en fréquence, il est nécessaire de limiter le volume des données à envoyer. Pour ce faire, les données doivent être traitées avant d'être envoyées sur le réseau.

Une grande partie des objets connectés sont des Wireless Sensors Node (WSNs). Le terme *smartsensor* (capteur intelligent) est aussi utilisé pour désigner ces WSNs.
Les WSNs sont constitués de plusieurs blocs :

- un ou plusieurs capteurs,
- une unité de traitement de signaux ou de données,
- une unité de communication,
- une unité de gestion de l'alimentation.

L'unité de traitement de signaux peut se faire de deux manières différentes : soit en utilisant un DSP (*Digital Signal Processor*) ou un MCU (*Micro Controller Unit*), soit en utilisant un circuit intégré avec un *hardware* spécifique à l'application (ASIC) qui effectue le traitement. Le MCU est conçu pour des applications de contrôle. Pour ce faire, il est optimisé pour les branchements conditionnels et un déplacement flexible des données. Par contre, le DSP est conçu pour des applications de traitement de signaux avec des contraintes sur le temps d'exécution. Pour ce faire, les DPSs sont optimisés pour effectuer très rapidement des opérations arithmétiques et des accès à la mémoire réguliers [3]. Avec un MCU ou un DSP, il est possible de changer le traitement du signal en changeant le code du programme exécuté. Par contre, l'utilisation d'un ASIC ne permet pas de changer le traitement du signal.

Dans un futur proche, les WSNs alimentés par batterie ne seront plus viables [4]. En effet, avec la quantité de WSNs qui seront prochainement en service et en faisant l'hypothèse que chaque batterie doit être remplacée tous les ans, les coûts environnementaux pour la production de ces batteries seront très importants. Les WSNs doivent donc être autonomes pour leur alimentation en énergie. Leur unité de gestion de l'alimentation doit récupérer de l'énergie dans l'environnement. Cela se fait à l'aide d'un bloc appelé *energy harvester*. Une antenne permettant de récupérer de l'énergie des ondes radio est un exemple d'*energy harvester.* La puissance générée par les *energy harvester* reste très faible, de l'ordre de quelques centaines de microwatts tout au plus [5]. C'est pourquoi les WSNs doivent consommer le moins d'énergie possible, car ces quelques centaines de microwatts doivent servir à alimenter les capteurs, à traiter les données provenant des capteurs et à les envoyer via un réseau sans fil. Cette dernière opération consomme la majeure partie de l'énergie collectée par l'*energy harvester* [6].

Dans ce travail, nous allons nous intéresser à l'amélioration d'un WSN en terme d'efficacité énergétique. Pour ce faire, nous nous pencherons sur l'amélioration du bloc de traitement de signaux du WSN. Dans notre cas, ce bloc est constitué d'un MCU pour lequel nous cherchons à diminuer la consommation du processeur central (CPU). Nous comparerons différents processeurs afin de

déterminer lequel est le plus efficace pour diminuer la consommation d'énergie sans perdre de performances. Pour effectuer cette comparaison, nous utilisons les différents facteurs de mérite suivants pour quantifier un CPU : nombre d'instructions exécutées par seconde (performances), consommation électrique du CPU, surface, taille du code et nombre d'accès à la mémoire.

Si nous analysons les données provenant de la littérature scientifique et celles provenant des fournisseurs, nous constatons que les conditions de test sont différentes. Parmi celles-ci, nous retrouvons la technologie, le *benchmark* utilisé… La consommation d'un processeur ne dépend pas de manière linéaire avec la technologie utilisée. Nous connaissons la consommation du Cortex-M0 [7] pour les technologies 90nm et 40nm. Si nous essayons de calculer sa consommation pour une technologie de 65nm à l'aide d'une approximation linéaire, les résultats obtenus diffèrent de 12% en fonction de la technologie de référence choisie. D'autre part, il n'est pas possible de convertir les résultats du *benchmark* Dhrystone [8] en résultats du *benchmark* CoreMark (c'est-à-dire convertir des DMIPS en CoreMark). Pour Zero-riscy [9] les performances sont exprimées uniquement en CoreMark/MHz. Nous ne pouvons donc pas le comparer avec Z-scale [10] où les performances sont données en DMIPS/MHz. Pour pouvoir comparer différents processeurs, nous devons définir et utiliser des conditions de test identiques pour tous.

Ce travail de fin d'études s'inscrit dans le cadre de la recherche effectuée par le groupe *Electronic Circuits & Systems* (ECS) de l'Université catholique de Louvain (UCLouvain) pour le développement d'un nouveau MCU à ultra-basse consommation pour les WSNs : SleepRider. Pour ce nouveau MCU, nous étudions l'impact sur les performances et la consommation du remplacement du processeur (Cortex-M0) présent dans la version actuelle SleepRunner [11] développé par ce même groupe de l'UCLouvain.

Ce travail effectue la comparaison du Cortex-M0 avec différents processeurs utilisant l'ISA (*Instruction Set Architecture*) RISC-V qui est open source. Les processeurs RISC-V choisis dans ce travail sont Zero-riscy et RI5CY. Nous utilisons une technologie 28nm FDSOI avec une tension d'alimentation de 0.4V. Le *benchmark* utilisé est le CoreMark et nous travaillons à une fréquence d'horloge de 80MHz.

Nous verrons dans ce travail que le Cortex-M0 a un concurrent au niveau efficacité énergétique dans certaines conditions. Ce concurrent est Zero-riscy. Par contre RI5CY qui est conçu pour effectuer des tâches de traitement de signaux n'est pas du tout intéressant pour remplacer le Cortex-M0.

Si nous ne prenons pas en compte la consommation des mémoires, le Cortex-M0 a une efficacité de 6.95 CoreMark/µW tandis que Zero-riscy (O3) est 29% moins efficace. Si nous prenons en compte la consommation des mémoires, le Cortex-M0 a une efficacité de 0.82 CoreMark/µW tandis que Zero-riscy est 21% plus efficace. Les mémoires utilisées sont les mêmes que celles utilisées dans SleepRunner pour calculer l'efficacité énergétique.

Le banc d'essai développé pour ce travail de fin d'études est disponible en open source afin de permettre sa réutilisation pour prolonger ce travail de recherche en comparant d'autres processeurs ou en utilisant d'autres *benchmarks*.

La structure de ce mémoire est la suivante :

- Au **Chapitre 2**, nous présentons les fondements de l'étude. Il est divisé en trois parties : la première décrit le fonctionnement d'un processeur. Nous y analysons la différence entre les architectures Von Neumann et Harvard. Nous décrivons par la suite les différents facteurs de mérite que nous utilisons dans la comparaison des processeurs. La deuxième partie reprend l'état de l'art des comparaisons déjà effectuées ainsi que les processeurs utilisés dans ce travail. Dans la troisième partie, nous abordons les *benchmarks*. Après avoir expliqué cet outil de comparaison, nous les classifions et présentons le *benchmark* utilisé dans ce travail.
- Au **Chapitre 3**, nous expliquons les différents choix effectués pour la mise en place des bancs d'essai. Ce chapitre est divisé en trois parties : la première décrit l'architecture du banc d'essai. La deuxième partie détaille l'extraction des différents facteurs de mérite. La troisième partie présente la compilation du code source du *benchmark*.
- Au **Chapitre 4**, nous caractérisons entièrement le Cortex-M0, un processeur de chez ARM. Cette caractérisation se base sur les facteurs de mérite que nous utilisons dans ce travail.
- Au **Chapitre 5**, nous comparons différents processeurs avec le Cortex-M0 pour déterminer s'il existe un processeur qui a une meilleure efficacité énergétique que le Cortex-M0. Nous évaluons également l'impact de la consommation des mémoires sur l'efficacité des différents processeurs que nous comparons.

2 Fondements de l'étude

2.1 Processeur

Dans cette section, nous abordons le fonctionnement d'un processeur. Nous comparons ensuite deux architectures de processeur : l'architecture Von Neumann et Harvard. Nous terminons par la description des différents facteurs de mérite.

2.1.1 Fonctionnement d'un processeur

Un processeur est constitué des différents blocs logiques suivants (voir Figure 1) :

- *instruction memory* (IMEM),
- *control unit,*
- *register file,*
- *arithmetic and logic unit* (ALU),
- *data memory* (DMEM).

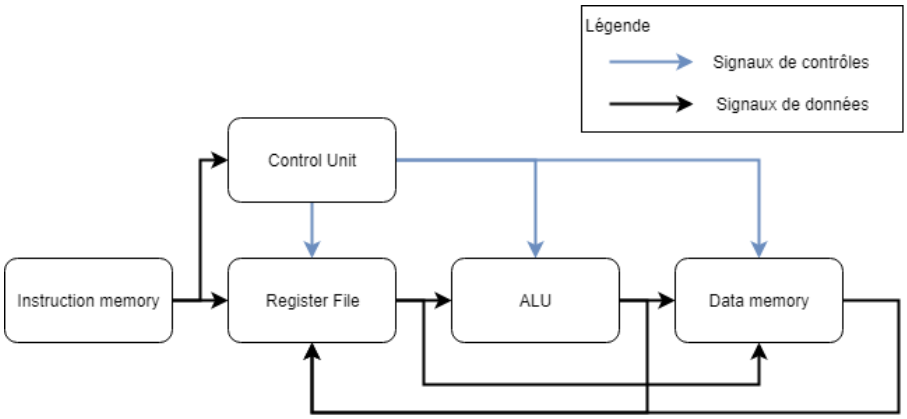

Figure 1 - Schéma des blocs logiques d'un processeur [12]

Le *control unit* décode l'instruction qui a été lue en mémoire. Il envoie alors des signaux de contrôle vers les autres blocs. L'ALU reçoit l'information du calcul à effectuer. Si des données doivent être écrites dans un registre, le *control unit* prévient le *register file*. Si des données doivent être écrites dans la *data memory*, le *control unit* la prévient. Le décodage de l'instruction dépend de l'ISA utilisé. Au plus l'ISA est complexe, au plus le *control unit* est complexe. La complexité de l'ISA va donc augmenter la consommation et la surface du *control unit*.

L'ALU est, comme son nom l'indique, le bloc qui effectue l'ensemble des calculs. Les calculs effectués par l'ALU sont :

- calculs arithmétiques,
- calculs logiques,
- calculs d'adresses pour les accès à la mémoire.

Le type de calcul est donné par le *control unit*. L'ALU a maximum deux opérandes. En fonction de l'instruction, les opérandes proviennent soit des registres, soit directement de l'instruction en elle-même. Les deux lignes de codes reprises dans le Code 1 montrent les provenances pour les opérandes de l'ALU. L'ensemble des opérations que l'ALU peut réaliser change d'un processeur à

l'autre. Certains processeurs ne savent par exemple pas effectuer de multiplication. Plus le nombre d'opérations de l'ALU est grand, plus sa consommation et sa surface augmentent.

```
1.  ADD R3, R2, R1 ; effectue le calcul R3 = R2 + R1 (R2 et R1 proviennent du registre)
2.  ADD R3, R2, #8 ; effectue le calcul R3 = R2 + 8 (R2 provient du registre tandis que
    8 vient de l'instruction)
```

Code 1 - Instructions Assembleur montrant la provenance des opérandes de l'ALU

Le *register file* est la mémoire interne au processeur. Il est composé d'un certain nombre de registres permettant de stocker les données temporaires utilisées par le processeur : résultat de calcul, adresse… Plus il y a de registres présents dans le *register file,* plus sa consommation et sa surface augmentent. Les performances du processeur peuvent également augmenter s'il y a plus de registres dans le *register file.*

L'*instruction memory* est la mémoire dans laquelle est stocké l'ensemble du code du programme exécuté. La *data memory* stocke l'ensemble des données générées par le programme.

Lors de l'exécution d'une instruction, le processeur effectue les 5 étapes suivantes :

1. *Fetch* : Lecture d'une instruction depuis l'*instruction memory*
2. *Decode* : Décodage de l'instruction et lecture des registres
3. *Execute* : Exécution du contenu de l'instruction
4. *Memory Read/Write* : Lecture ou écriture dans la *data memory*
5. *Writeback* : écriture du résultat dans les registres

L'exécution des différentes étapes peut se faire de deux manières différentes : séquentiellement ou en utilisant le mécanisme du *pipeline*. Ce dernier permet de commencer à traiter une nouvelle instruction avant d'avoir terminé de traiter l'instruction précédente. Le *pipeline* découpe une instruction en plusieurs étapes. Le nombre d'étapes différentes définit le nombre d'étages du *pipeline*. La Figure 2 montre un exemple d'exécution de plusieurs instructions avec et sans le mécanisme du *pipeline*.

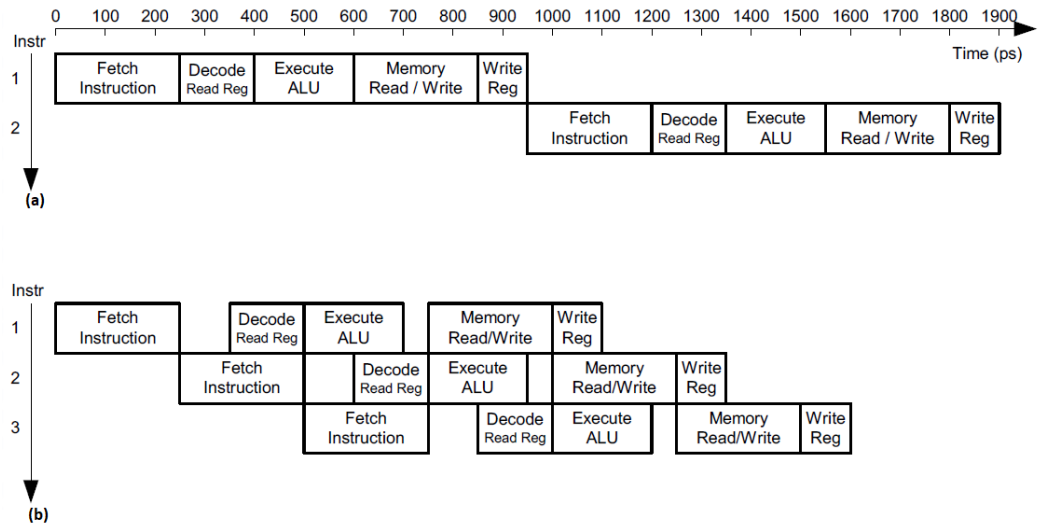

Figure 2 - Diagramme temporel : (a) exécution séquentielle et (b) exécution pipeline [12]

Le mécanisme du *pipeline* est ralenti par certaines opérations comme les branchements. En effet, lors d'un branchement, l'adresse de l'instruction qu'il faudrait lire est calculée à l'étape de l'exécution. De ce fait, quand nous lisons l'instruction qui suit le branchement, nous ne lisons pas la bonne instruction. Il faut attendre que l'étape de l'exécution de l'instruction de branchement soit terminée pour pouvoir lire la bonne instruction. Nous exécutons donc deux lectures d'instruction et un décodage d'instruction inutilement. La Figure 3 montre l'exécution d'un branchement.

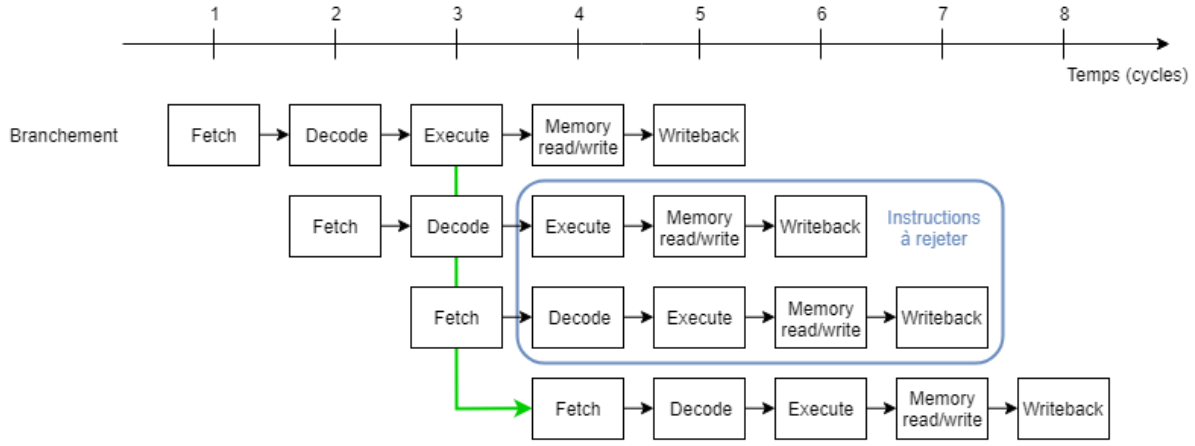

Figure 3 - Exécution d'un branchement en *pipeline* [12]

2.1.2 Comparaison de l'architecture Von Neumann et de l'architecture Harvard

Au niveau du transfert des données entre les mémoires (instructions et données) et le processeur, il y a deux architectures différentes :

- l'architecture Von Neumann,
- l'architecture Harvard.

L'architecture Von Neumann montrée à la Figure 4 consiste à utiliser un seul et unique bus pour transférer les données provenant de la mémoire des instructions mais également de la mémoire des données.

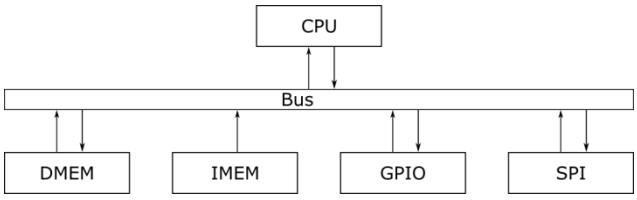

Figure 4 - Architecture Von Neumann

L'architecture Harvard montrée sur la Figure 5 utilise deux bus pour faire passer les données : un réservé aux données provenant de la mémoire des instructions et un autre utilisé conjointement par la mémoire de données et les différentes entrées et sorties.

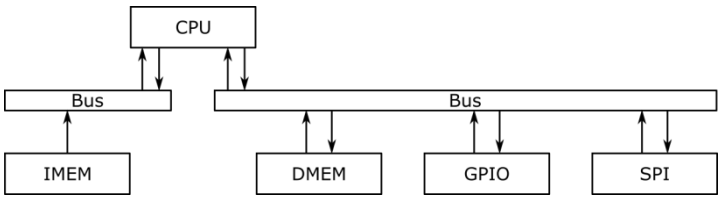

Figure 5 - Architecture Harvard

Étant plus simple, l'architecture Von Neumann utilise moins de transistors et de ce fait consomme moins que l'architecture Harvard. Par contre, l'architecture Von Neumann a un risque de goulot d'étranglement (*bottleneck*). En effet, le bus a un débit limité pour faire passer les données et les instructions. À chaque cycle d'horloge, une seule donnée ou instruction peut être lue et une seule donnée peut être écrite. Dans le cas où il faut traiter beaucoup de données en effectuant très peu de calculs, la lecture de l'instruction et la lecture de la donnée doivent se faire dans le même cycle d'horloge. Cela n'est pas possible.

2.1.3 Facteurs de mérite

Les facteurs de mérite sont des grandeurs utilisées pour caractériser les performances d'un système afin de le comparer à un autre système alternatif. Dans le cas des processeurs *ultra low power*, les facteurs de mérite utilisés sont :

a) **Les performances d'un processeur** : C'est le nombre d'instructions exécutées par seconde. Elles se mesurent en exécutant un code et en mesurant le temps d'exécution de ce dernier. Le code exécuté est appelé un *benchmark*. Pour que le facteur de mérite soit objectif, il faut prendre en compte la fréquence à laquelle le processeur travaille, car la fréquence a une influence linéaire sur le nombre d'instructions exécutées par seconde. La métrique souvent utilisée pour la caractérisation est le nombre d'instructions par seconde divisé par la fréquence du processeur. Pour le *benchmark* CoreMark, l'unité de mesure est le CoreMark et correspond au nombre d'itérations que le *benchmark* peut effectuer en une seconde. Cela équivaut à un nombre d'instructions par seconde, car une itération correspond à un nombre d'instructions qui ne change pas au cours de l'exécution. La métrique utilisée dans ce rapport s'exprime en Coremark/MHz.

b) **La consommation d'un processeur** : C'est la puissance électrique consommée par le processeur pour effectuer une tâche donnée. Pour rester objectif, la consommation est mesurée pendant que le processeur exécute le *benchmark*. À partir de 50MHz dans le cas du Cortex-M0, nous pouvons faire l'approximation que la consommation totale du CPU dépend de manière linéaire de la fréquence du processeur voir Figure 6. C'est pourquoi nous divisons la métrique par la fréquence du processeur. La métrique utilisée dans ce rapport s'exprime en µW/MHz. La non-linéarité de la consommation en fonction de la fréquence provient du fait qu'une partie de la consommation est indépendante de la fréquence. C'est la consommation de fuite. Nous expliquerons cela plus en détail dans la section 3.2 (Extraction de la consommation).

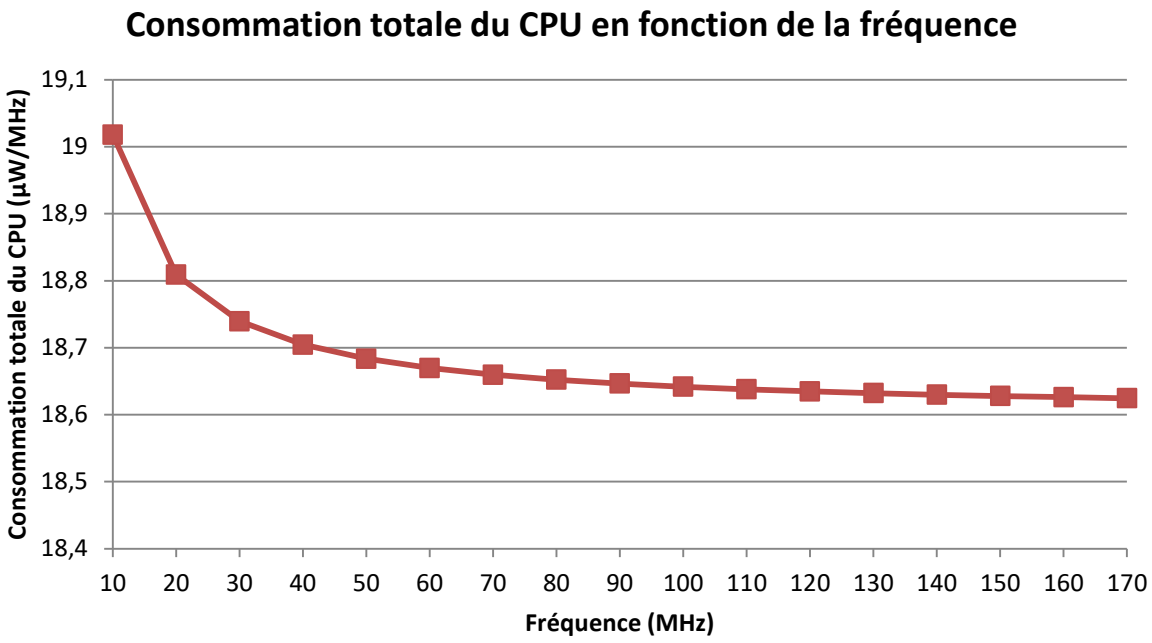

Figure 6 - Calcul de la consommation totale du CPU en fonction de la fréquence sur base de la simulation du Cortex-M0

c) **La surface** : C'est l'espace que le processeur occupe sur la puce de silicium. Cet espace a un impact direct sur les coûts de fabrication. La surface du processeur dépend de la technologie utilisée : le même processeur aura une surface moins grande s'il est gravé avec une technologie en 28 nm au lieu d'une technologie en 65 nm. De plus, toutes les portes logiques n'ont pas la même surface. Pour retirer l'influence de la technologie, nous comparons un nombre de portes logiques équivalentes. Pour ce faire, nous normalisons par rapport à la taille de la NAND2. La métrique utilisée dans ce rapport s'exprime en kGates équivalentes (KGe).

d) **Le nombre et le type d'accès à la mémoire** : Ils ont un impact sur la consommation globale du SoC (*System On Chip*). Le nombre d'accès à la mémoire permet de comparer les processeurs sur leur consommation de mémoire. La répartition des types d'accès (lecture/écriture, IMEM/DMEM) permet de choisir les différentes mémoires à utiliser pour minimiser la consommation totale. La métrique utilisée dans ce rapport s'exprime en nombre d'accès à la mémoire pour la mémoire des instructions et la mémoire des données (lecture/écriture) pour exécuter un programme de *benchmark* donné.

e) **La taille du code** : C'est le volume occupé par le code du *benchmark* dans la mémoire des instructions. Elle a une influence sur la taille des mémoires à utiliser. Au plus les mémoires sont de grande capacité, au plus la consommation nécessaire pour une lecture et une écriture augmente. La métrique utilisée dans ce rapport s'exprime en nombre de mots de 32 bits (*words*).

f) **La fréquence maximale** : C'est la fréquence au-delà de laquelle le processeur ne fonctionne plus correctement. Ce sont les contraintes de *timing* qui sont responsables de ce dysfonctionnement. En effet, il faut que le temps entre deux cycles d'horloge soit plus grand que celui nécessaire pour que les données passent d'un registre à l'autre ($T_{délai}$) comme montré à la Figure 7. $T_{délai}$ dépend de l'architecture du processeur puisque celle-ci définit le nombre de portes logiques du bloc de logique combinatoire. $T_{délai}$ dépend également de la technologie utilisée puisque celle-ci définit le temps qu'il faut pour passer une porte logique. Dans le cas de ce travail, ce facteur de mérite est fixe, car nous travaillons à fréquence constante (80 MHz) inférieure à la fréquence maximale pour chacun des processeurs étudiés. Ces valeurs de fréquence maximale seront présentées dans la section 2.2.2 (Présentation des processeurs utilisés dans ce travail).

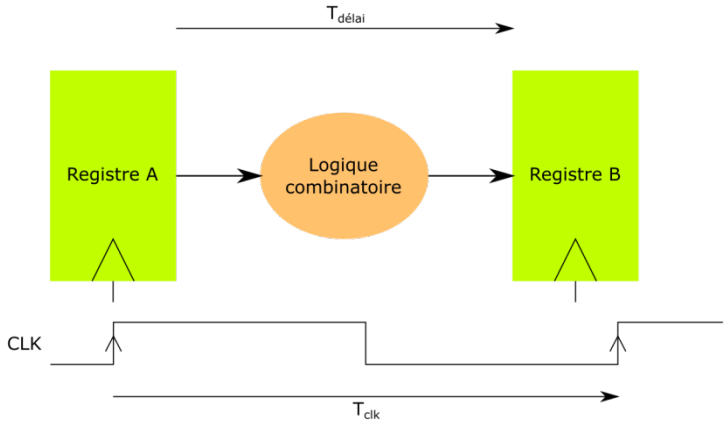

Figure 7 - Contrainte de *timing* [13]

Les différentes conditions du test influencent les facteurs de mérite. La tension d'alimentation va directement influencer la consommation et la fréquence maximale. La technologie et les librairies ont un impact sur la surface, la consommation et la fréquence maximale. La fréquence de l'horloge a un impact sur les performances, la consommation et sur la surface [14].

Analysons l'influence des conditions de test sur les différents facteurs de mérite impactés.

- **Les performances** dépendent de manière linéaire de la fréquence de l'horloge. En effet, si nous augmentons la fréquence de l'horloge, il y a plus de cycles d'horloge et donc d'instructions qui sont exécutées durant un même laps de temps.

- **La consommation** dépend de toutes les conditions de test. Elle dépend du carré de la tension d'alimentation comme nous l'expliquerons dans la section 3.2 (Extraction de la consommation). La technologie influence la capacité d'entrée d'une porte logique en modifiant sa taille, son épaisseur et son diélectrique ainsi que le courant de fuite, car il dépend de la capacité d'entrée et de la largeur minimale des portes logiques. La technologie influence donc la consommation puisque celle-ci dépend de la capacité d'entrée comme nous l'expliquerons également dans la section 3.2. Les librairies influencent le courant de fuite en changeant la tension de seuil. La fréquence de l'horloge influence de manière linéaire la partie dynamique de la consommation. D'autre part, les optimisations effectuées pour respecter les contraintes de *timing* vont également avoir une influence sur la consommation. En effet, si nous augmentons la fréquence de l'horloge, les contraintes de *timing* seront plus difficiles à respecter. Différentes optimisations peuvent être effectuées. Parmi celles-ci, nous avons l'augmentation de la largeur des transistors de la porte logique pour diminuer son délai, l'ajout de *buffers* pour permettre de sortir assez de courant pour l'étage suivant, l'utilisation du principe du *pipeline* sur le bloc de logique combinatoire pour diminuer le nombre de portes logiques et donc le $T_{délai}$ entre deux registres… La Figure 8 montre l'influence de l'augmentation de la fréquence de l'horloge sur la consommation dynamique et la consommation de fuite.

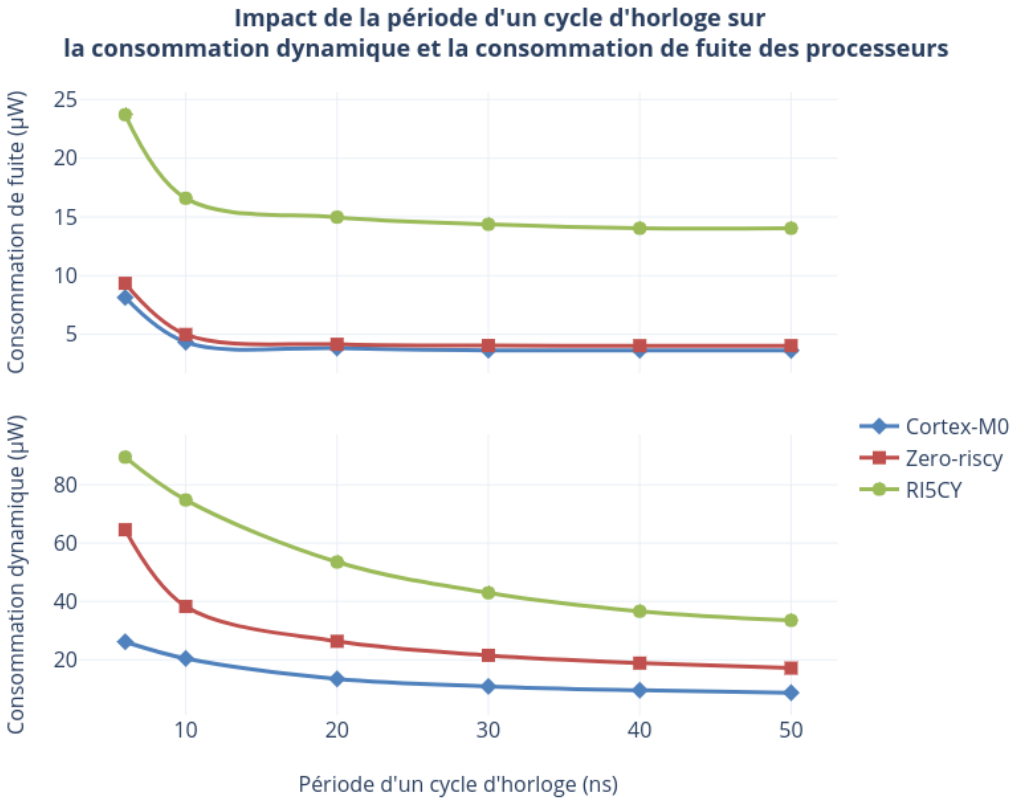

Figure 8 - Influence de la période du cycle de l'horloge sur la consommation dynamique et la consommation de fuite des processeurs sur base des simulations du banc d'essai

- **La surface** dépend du carré de la longueur du transistor. Celle-ci est fixée par la technologie. Les librairies et plus précisément le nombre de pistes de routage vont influencer la surface totale en permettant d'utiliser plus ou moins de transistors en épaisseur. Plus nous pouvons mettre de transistors en épaisseur, plus la surface totale peut diminuer. Comme pour la consommation, les optimisations effectuées pour respecter les contraintes de *timing* vont avoir une influence sur la surface. La Figure 9 montre l'influence de l'augmentation de la fréquence de l'horloge sur la surface. Cette influence suit une loi hyperbolique [15].

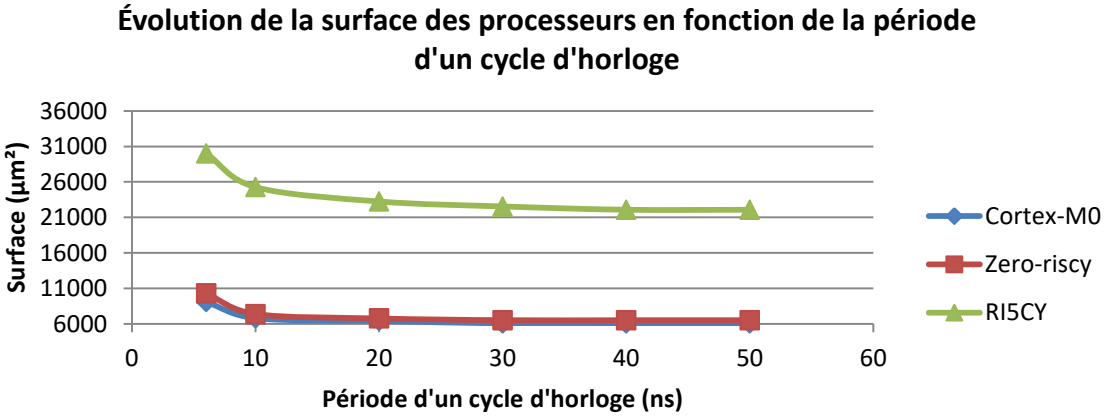

Figure 9 - Influence de la période du cycle de l'horloge sur la surface des processeurs sur base des simulations du banc d'essai

- **La fréquence maximale** dépend du délai de propagation du signal au travers de la logique combinatoire comme expliqué au point f de cette section. Ce délai de propagation est directement proportionnel à la tension d'alimentation et la capacité d'entrée de la porte logique et inversement proportionnel au courant passant (I_{ON}).

2.2 État de l'art

Dans cette section, nous regardons l'état de l'art des différentes comparaisons existantes entre des processeurs. Nous présentons ensuite les différents processeurs étudiés dans ce travail.

2.2.1 État de l'art des comparaisons de processeur.

Nous allons nous intéresser aux différentes comparaisons déjà réalisées entre les processeurs de la famille du Cortex-M0 et les processeurs de type RISC-V.
Le premier article [16] comprend une comparaison entre le Cortex-M0 et le processeur Z-scale. Cette comparaison utilise le *benchmark* Dhrystone pour lequel les résultats ne sont pas comparables avec ceux du *benchmark* CoreMark car il n'est pas possible de convertir les résultats du *benchmark* Dhrystone en résultats du *benchmark* CoreMark. Il ressort de cette comparaison que le processeur Z-scale est un peu plus performant que le Cortex-M0 mais consomme plus au point de consommation minimale. Le processeur Z-scale n'étant plus maintenu, cet article n'est pas intéressant pour ce travail.
Le deuxième article [9] compare les processeurs micro-riscy, Zero-riscy et RI5CY. Cette deuxième comparaison est beaucoup plus intéressante, car notre travail porte également sur les processeurs Zero-riscy et RI5CY. Dans cette étude, les différents processeurs sont comparés pour les facteurs de mérites suivants : performances, consommation, surface et efficacité énergétique. Pour le calcul des performances, trois *benchmarks* différents sont choisis : Convolution 2D, CoreMark et un *benchmark* custom. La convolution 2D représente des tâches de traitement d'images et le *benchmark* custom est constitué d'un set de fonctions systèmes embarqués. Au final, il en ressort que RI5CY est le processeur le plus efficace pour des traitements de données lourds (Convolution 2D). Par contre, Zero-riscy est le plus efficace en contrôle et calcul arithmétique. Quant à Micro-riscy, il est le plus efficace dans tout ce qui est du contrôle pur.
Le troisième article [17] comprend une comparaison entre le Cortex-M0 et un processeur de type RISC-V 32 bits. Dans cette comparaison, nous constatons que le processeur Cortex-M0 a de meilleures performances et une consommation plus basse que le processeur RISC-V. Comme les mesures ne sont pas réalisées avec la même technologie, la comparaison n'est pas totalement objective. Il n'est pas possible de déterminer sur base de ces résultats si le processeur RISC-V 32 bits est plus efficace que le processeur Cortex-M0.
Le quatrième article [18] compare un processeur RISC-V avec le Cortex-M4, un processeur destiné à faire du traitement de signal. Il en ressort que le processeur RISC-V est comparable en taille au Cortex-M4 mais il a des performances plus basses. Cette différence entre les performances est principalement due au coût des branchements, des *cache-miss* et des *load-hazard*. Le *benchmark* utilisé est le CoreMark qui effectue beaucoup de tâches de contrôle. Ces tâches ne peuvent pas être accélérées par les accélérateurs matériels présents dans le processeur RISC-V.

Sur base de la littérature scientifique ([9] et [10]) et des informations des constructeurs ([7], [19] et [20]), nous avons réalisé un tableau (Tableau 1) présentant les différents facteurs de mérite des différents processeurs.

	Cortex-M0[7]	Cortex-M4 [19]	Cortex-M33[20]	RI5CY[9]	Zero-riscy[9]	Z-scale [10]
Technologie	40 nm LP	40 nm LP	28 nm HPC	65 nm	65 nm	40 nm GPLUS
Alimentation	1.1 V	1.1 V	0.9 V	1.2 V	1.2 V	0.99V
Performances (CoreMark/MHz)	2.33	3.4	4.02	3.19	2.44	N.C.
Consommation totale du CPU(µW/MHz)	5.3	12.26	3.8	5.08	2.08	1.8
Consommation totale du CPU normalisée (µW/MHz)	0.49	1.13	0.75	0.24	0.10	0.21

Tableau 1 - Comparaison des différents processeurs sur base d'articles les présentant.

Pour pouvoir effectuer une première comparaison des différents processeurs sur base de leur consommation dynamique, nous devons retirer l'impact de la technologie et de la tension d'alimentation. C'est pourquoi nous normalisons la consommation dynamique à l'aide de la formule suivante (que nous expliquons dans la section 4.2) :

$$P_{norm} = P \frac{28\,nm}{technologie\ d'originie} \left(\frac{0{,}4V}{tension\ d'alimentation\ d'origine}\right)^2$$

De cette façon, nous pourrons comparer ces résultats avec ceux que nous produisons dans ce travail.

De ce tableau, trois processeurs ne sont pas utilisés dans ce travail : Z-scale, le Cortex-M4 et le Cortex-M33. Z-scale n'étant plus maintenu et n'utilisant pas le langage Verilog a été écarté. Le Cortex-M4 n'est pas intéressant, car il consomme plus de deux fois plus que le Cortex-M0 sans avoir des performances deux fois plus élevées. Il est de ce fait moins efficace que le Cortex-M0. Le Cortex-M33 vient de sortir et nous n'avons pas encore accès à son code. Mais il devrait être intéressant vu qu'il s'agit de la continuation du Cortex-M4 avec des améliorations. Il consomme 50% de plus que le Cortex-M0 pour une amélioration des performances de plus de 70%.
Ce tableau nous montre clairement que le processeur Zero-riscy est meilleur que le Cortex-M0. Si les performances de Zero-riscy sont légèrement meilleures que celles du Cortex-M0, la grande amélioration provient de la consommation qui est presque divisée par un facteur 4. Il est donc très intéressant de comparer ces deux processeurs avec des conditions de test identiques.

2.2.2 Présentation des processeurs utilisés dans ce travail

Nous allons présenter dans cette partie les principales caractéristiques des trois processeurs que nous comparons. Le processeur de référence est le Cortex-M0 DS qui est présent dans la version actuelle du MCU : SleepRunner développé par le groupe ECS de l'UCLouvain. Nous avons choisi de le comparer avec Zero-riscy et RI5CY. Tout d'abord parce que ces trois processeurs sont écrits dans le même langage HDL (*Hardware Description Langage*). Ensuite, pour ces trois processeurs, nous connaissons leurs performances pour le *benchmark* utilisé dans ce travail. Cela nous permet donc de vérifier si les résultats que nous obtenons sont cohérents avec les données que nous possédons (données constructeurs ou articles).

Cortex-M0

Le Cortex-M0 [7] est un processeur 32 bits de la famille des Cortex-M. Cette famille se compose de processeurs optimisés pour avoir une consommation très basse et un coût en surface faible. Le Cortex-M0 est optimisé pour avoir une consommation minimale et une surface minimale. Son architecture se base sur trois niveaux de *pipeline*, un multiplieur 32 bits en un cycle optionnel et un *register file* de 16 registres. Le transfert des données avec les mémoires se base sur l'architecture Von Neumann et utilise le protocole AHB Lite. Sa fréquence maximale pour les conditions de test que nous utilisons est de 174 MHz.

Zero-riscy

Zero-riscy [21] est un processeur 32 bits basé sur l'ISA RISC-V développé par ETH-Zurich. Il est optimisé pour avoir une très basse consommation et un coût en surface faible. Son architecture est basée sur une architecture en *pipeline* à deux niveaux et un *register file* de 32 registres. Pour le transfert des données avec les mémoires, il utilise l'architecture Harvard et le protocole utilisé est un protocole fait sur mesure. Le processeur supporte l'ISA RV32IMC. Il supporte l'utilisation d'instructions compressées pour diminuer la taille de l'IMEM si nécessaire. Il est aussi équipé d'un bloc pour faire les multiplications et divisions 32 bits en hardware. Sa fréquence maximale pour les conditions de test que nous utilisons est de 140 MHz.

RI5CY

RI5CY [22] est un processeur 32 bits basé sur l'ISA RISC-V développé par ETH Zurich. Il est optimisé pour les applications basées sur du traitement de signaux. Son architecture est basée une architecture *pipeline* à quatre niveaux et un *register file* de 32 registres. Pour le transfert des données avec les mémoires, l'architecture Harvard est utilisée. Le protocole de communication est le même que celui utilisé par Zero-riscy. Le processeur supporte l'ISA RV32IMFC. Par rapport à Zero-riscy, ce processeur peut être équipé d'un *floating point hardware* permettant d'effectuer les calculs en virgule flottante en un cycle. Il implémente aussi des instructions customs visant à améliorer ses performances dans le traitement de signaux. Sa fréquence maximale pour les conditions de test que nous utilisons est de 132 MHz.

2.3 Benchmarks

Dans cette section, nous abordons ce qu'est un *benchmark*. Nous classifions ensuite les *benchmarks* en trois catégories. Pour terminer, nous détaillons les *benchmarks* utilisés pour la comparaison des processeurs.

2.3.1 Qu'est-ce qu'un benchmark ?

Un *benchmark* est un programme permettant de tester les performances d'un système. Le *benchmark* va effectuer un stress test, c'est-à-dire, soumettre une charge de travail très importante sur le système. Pour que le *benchmark* reflète les performances réelles du système testé, l'influence des paramètres extérieurs à celui-ci doit être négligeable. S'il existe des paramètres qui ont une influence sur le *benchmark*, il faut les garder constants et les mentionner lors du compte-rendu des résultats. L'optimisation du compilateur, le système d'exploitation ainsi que le type de mémoire [23] sont des exemples de paramètres extérieurs qui ont souvent une influence sur les résultats du *benchmark*.

2.3.2 Classification des benchmarks

Les *benchmarks* peuvent être classés en trois catégories distinctes [8] :

- *benchmark* synthétique,
- *benchmark* basé sur des applications,
- *benchmark* basé sur des algorithmes.

Les *benchmarks* synthétiques permettent de mesurer des paramètres spécifiques d'un composant du système mais pas l'ensemble du système. Le temps d'accès à la mémoire ainsi que le nombre d'instructions par seconde exécutées par le processeur sont des exemples de paramètres spécifiques testés par les *benchmarks* synthétiques.

Les *benchmarks* basés sur des applications sont prévus pour tester l'ensemble du système dans des conditions proches des conditions réelles d'utilisation. Ce type de *benchmark* est très utile pour vérifier le bon comportement d'applications real-time.

Les *benchmarks* basés sur des algorithmes permettent de mesurer les performances d'un système lors de l'exécution de tâches très précises. L'encodage, les manipulations de matrice/tableau sont des exemples d'algorithmes utilisés pour tester les performances d'un système.

2.3.3 CoreMark

Le CoreMark [24] est un *benchmark* de *Embedded Microprocessor Benchmark Consortium* (EEMBC) qui a pour but de mesurer les performances des processeurs de manière standardisée. Le CoreMark est un *benchmark* de type synthétique, car il ne fait que mesurer les performances en nombre d'instructions exécutées par seconde. Pour effectuer la mesure, le CoreMark effectue les 4 algorithmes suivants :

- opérations sur les listes (recherche, tri…),
- opérations matricielles (addition, multiplication …),
- calculs de machines à états finis,
- calculs de CRC (Contrôle de redondance cyclique).

Ces différents algorithmes sont souvent utilisés dans un grand nombre d'applications.

Le CoreMark a été conçu pour être utilisable sur des processeurs allant de 8 bits à 64 bits. Pour que cela soit possible, son code compilé est très petit (environ 16 kB). Pour éviter que le compilateur ne fasse une bonne partie du travail du *benchmark* en précalculant les résultats, toutes les opérations effectuées utilisent des données qui ne sont pas disponibles au moment de la compilation. Cela permet de limiter l'influence du compilateur mais pas de la retirer complètement.

Le CoreMark possède des règles de compte-rendu des résultats spécifiques pour pouvoir facilement refaire le test et obtenir les mêmes résultats. Dans le compte-rendu des résultats du *benchmark* se trouvent :

- le compilateur utilisé, sa version et les paramètres de compilation,
- l'endroit où se trouve le code du *benchmark* (SRAM, Cache …),
- le type d'exécution et le nombre de *thread*s (si on utilise une exécution en parallèle).

3 Mise en place du banc d'essai

Dans ce chapitre, nous allons décrivons la mise en place du banc d'essai pour comparer les différents processeurs. Dans un premier temps, nous détaillons l'architecture du banc d'essai. Nous discutons par la suite de l'extraction des différents facteurs de mérite. Nous terminons par les différents paramètres de la compilation du software utilisé.

3.1 Architecture du banc d'essai

Le banc d'essai est constitué des différents blocs suivants :

- processeur à tester,
- mémoire pour les instructions,
- mémoire pour les données,
- bus de transfert entre le processeur et les mémoires,
- GPIO.

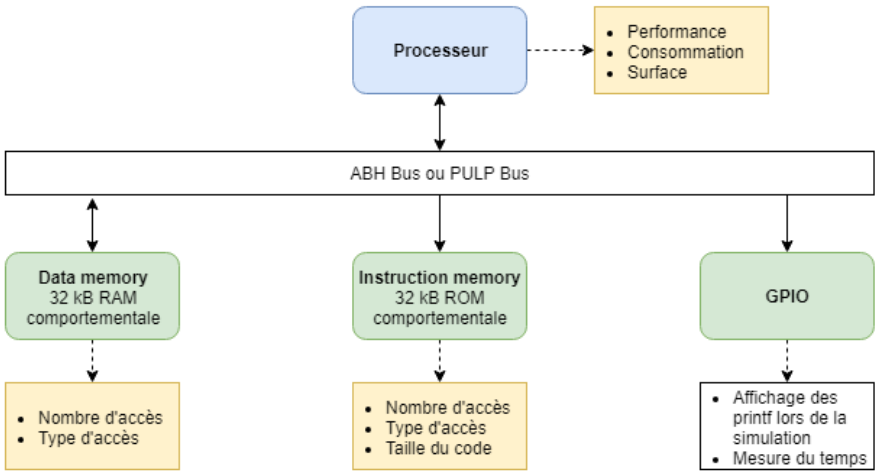

Figure 10 - Architecture du banc d'essai

Les mémoires utilisées dans le banc d'essai sont purement comportementales. Cela permet de ne pas devoir s'occuper des problèmes de délai causés par les mémoires. Ces délais ont un impact direct sur le calcul des performances. En effet, si le processeur doit attendre un cycle de plus avant de recevoir une instruction ou une donnée, le *benchmark* va mettre plus de temps pour s'exécuter.

Dans le banc d'essai, nous utilisons une RAM de 32 kB pour la DMEM et une ROM de 32 kB pour l'IMEM. Sur la Figure 11, nous voyons le plan d'adressage utilisé. Ce plan permet au compilateur, au programme et au processeur de connaître les plages d'adresses qui sont réservées aux instructions, à la mémoire des données et aux différents périphériques.

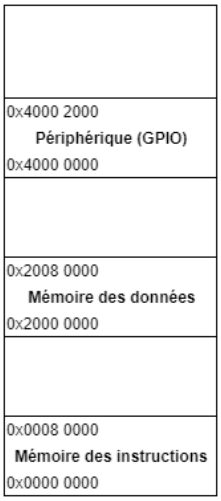

Figure 11 - Plan d'adressage

Le processeur va définir l'architecture du bus de transfert en fonction des signaux d'entrée et de sortie disponibles. Le Cortex-M0 utilise un bus de type AHB-Lite tandis que les processeurs Zero-riscy et RI5CY provenant de chez ETH Zurich utilisent un bus customisé qui présente quelques différences avec le bus AHB-Lite.

Dans le protocole AHB-Lite [25], il y a deux phases distinctes : la phase d'adressage et la phase des données. Dans le protocole PULP, ces deux phases ne sont plus aussi clairement séparées. Pour la lecture, il y a bien deux phases. Pour l'écriture par contre, les données arrivent dès la phase d'adressage et la phase des données n'existe pas.

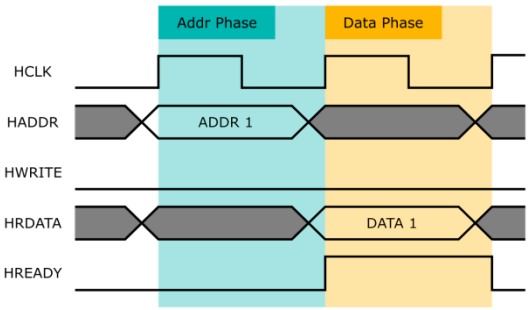

Figure 12 - Lecture sur le bus AHB Lite

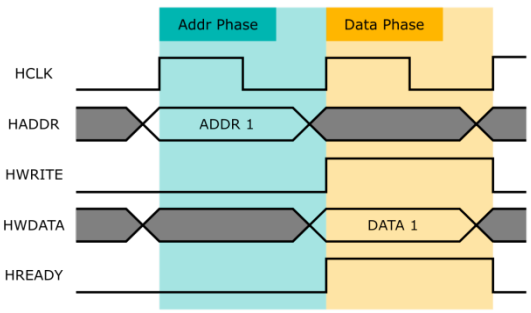

Figure 13 - Écriture sur le bus AHB Lite

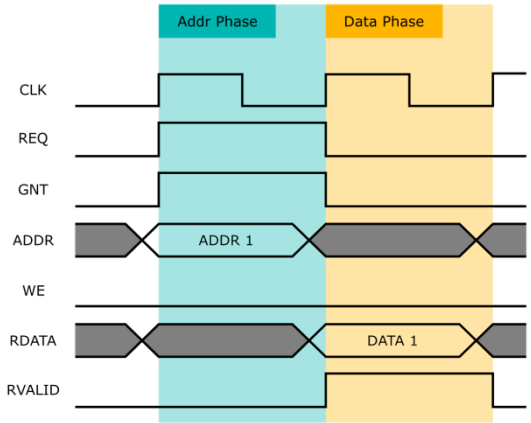

Figure 14 - Lecture sur le bus PULP

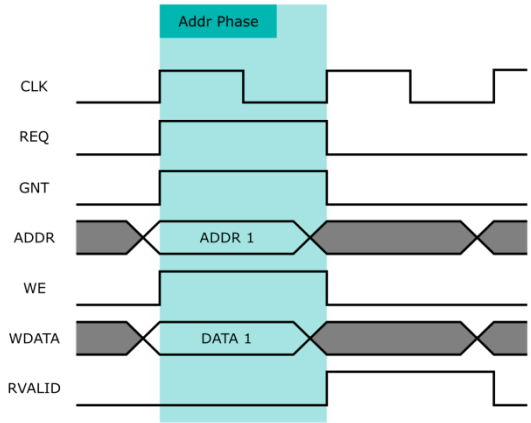

Figure 15 - Écriture sur le bus PULP

Une seconde différence est dans la requête d'accès à la mémoire. Pour l'AHB-Lite, la requête se base juste sur l'envoi d'un signal de requête. Pour PULP, la requête utilise un mécanisme différent. Tout d'abord le signal REQ est mis à 1 pour signaler que l'adresse est valide. Dans le même cycle d'horloge (pour une mémoire comportementale), la mémoire répond en mettant le signal GNT à 1 pour signaler qu'elle est prête à effectuer l'action. Au prochain cycle d'horloge, l'adresse et les données à écrire peuvent être modifiées.

Une troisième différence concerne la sélection de la taille des données à lire ou écrire. Pour le protocole AHB-Lite, nous avons un signal sur 3 bits donnant la taille des données. Pour le protocole PULP, la taille des données dépend du masque sur 4 bits. Chaque bit correspond à l'activation ou non-activation d'un byte en lecture ou en écriture.

Le GPIO nous permet de communiquer avec le processeur depuis le monde extérieur. Le GPIO se comporte comme une mémoire. Pour le Cortex-M0, nous redirigeons la sortie standard sur le GPIO pour permettre l'affichage du texte lors des simulations avec la fonction **`printf`**. Cette redirection se fait en redéfinissant les fonctions permettant d'écrire ou lire des informations depuis un fichier. Ces fonctions sont aussi utilisées pour écrire sur la sortie standard ou lire depuis l'entrée standard. Le Code 2 reprend les différentes fonctions redéfinies.

```
1.  volatile unsigned short *gpout = (volatile unsigned short *) 0x40002000U; // pointeu
    r vers le GPIO
2.
3.  // fonction permettant d'écrire un caractère sur le GPIO
4.  int fputc(int ch, FILE *f){
5.      *gpout = ((GPOUT_ENABLE << GPOUT_PRINTF_SHIFT) | ch); // écriture du caractère s
    ur le GPIO
6.      *gpout = 0x0000;
7.      return ch;
8.  }
9.  // fonction permettant de savoir s'il y a eu une erreur avec le fichier f
10. int ferror(FILE *f) { return 0; }
11. //fonction permettant de lire un caractère depuis le fichier f (Non utilisée)
12. int fgetc(FILE *f) { return -1; }
```

Code 2 - Redirection pour le Cortex-M0

Pour les processeurs Zero-riscy et RI5CY, cette façon de rediriger ne fonctionne pas. Tout d'abord, l'écriture sur le GPIO avec la méthode montrée dans le Code 2 ne fonctionne pas. En effet, lors de l'exécution du programme, l'adresse utilisée pour écrire n'est pas celle définie dans le programme. Il s'agit à première vue d'un bug à la compilation. Cela a pour conséquence une erreur dans l'écriture. Le deuxième problème provient du fait que la fonction **`fputc`** n'est pas utilisée par les librairies lors de l'écriture de texte sur la sortie standard. De ce fait, nous n'envoyons pas de texte sur le GPIO pour les processeurs Zero-riscy et RI5CY. Par contre, nous envoyons quand même manuellement des nombres. Pour ce faire, nous avons utilisé la même technique que pour le Cortex-M0 pour écrire directement les valeurs à l'adresse du GPIO. Cela est montré dans le Code 3.

```
1.  #include "driver.h"
2.
3.  // Peripherals
4.  volatile unsigned short *gpout = (volatile unsigned short *) 0x20002000;
5.  volatile unsigned short *gpin = (volatile unsigned short *) 0x20001000;
6.
7.  void  send_enc_byte(char enc)  { *gpout = (GPOUT_ENCBYTE | enc); *gpout = 0x0000; }
8.
9.  void  send_signal_byte(char enc)  { *gpout = ((GPOUT_ENABLE << GPOUT_SINGAL_SHIFT) |
     enc); *gpout = 0x0000; }
10.
11. void send_enc_int(int value){
12.     *gpout = ( GPOUT_ENCBYTE | ( value & 0x000000FF));
13.     *gpout = ( GPOUT_ENCBYTE | ((value & 0x0000FF00) >> 8));
14.     *gpout = ( GPOUT_ENCBYTE | ((value & 0x00FF0000) >> 16));
15.     *gpout = ( GPOUT_ENCBYTE | (value >> 24));
16.     *gpout = 0x0000;
17. }
```

Code 3 - Écriture sur le GPIO pour Zero-riscy et RI5CY

3.2 Extraction des facteurs de mérite

L'extraction des facteurs de mérite s'effectue lors de certaines étapes de la chaîne de développement d'un *System on Chip* (SoC). Commençons par décrire ces différentes étapes montrées à Figure 16.

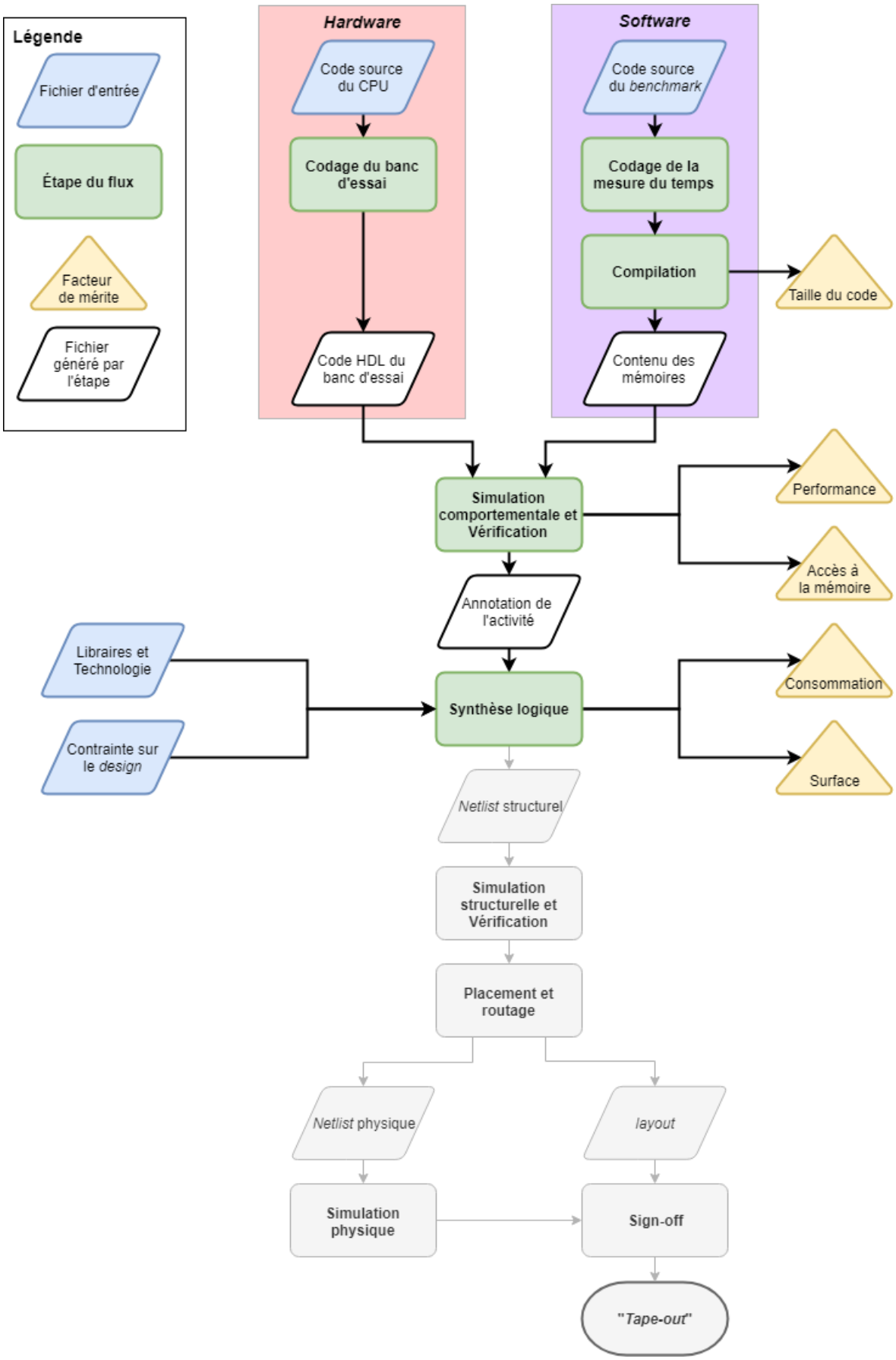

Figure 16 - Flux global de développement d'un SoC [26]

La première étape comporte deux niveaux. D'un point de vue *software*, après avoir adapté la mesure du temps pour le processeur utilisé dans le code source du *benchmark*, nous compilons le code source du *benchmark*. À ce moment, nous pouvons déjà extraire la taille du code. Cette compilation nous donne un fichier avec le contenu de la mémoire. D'un point de vue *hardware*, nous écrivons le code source (code HDL) du banc d'essai sur base du code source du processeur.
L'étape suivante consiste à effectuer la simulation comportementale qui nous permet de vérifier le bon fonctionnement du banc d'essai. Pour effectuer cette simulation comportementale, nous utilisons les fichiers générés à la première étape : le code source du banc d'essai et le fichier avec le contenu des mémoires. Durant cette simulation, nous générons le fichier d'annotation de l'activité et extrayons les performances ainsi que les différents accès à la mémoire.
Ensuite vient l'étape de la synthèse logique qui utilise le fichier d'annotation de l'activité, le code source du processeur, les libraires qui contiennent les informations sur la technologie et les contraintes sur le *design* (*timing* sur les entrées/sortie, consommation maximale, surface maximale). Le programme qui effectue la synthèse remplace le modèle comportemental des portes logiques par leur modèle plus réaliste fourni par les fichiers de la librairie. Ce modèle permet de prendre en compte les contraintes de *timing*. À la fin de cette étape, nous obtenons la consommation dynamique et la consommation de fuite du processeur pour l'exécution du *benchmark* ainsi que sa surface.
Les étapes suivantes dans la chaîne de développement d'un SoC sont grisées dans la Figure 16, car dans le cadre de ce travail, nous ne les effectuons pas. En effet, il n'y a aucun nouveau facteur de mérite à extraire de ces étapes.

Extraction des performances

Les performances se calculent sur base du temps que le processeur met pour exécuter un nombre donné d'itérations du *benchmark*. Dans le cadre de ce travail, nous calculons le temps d'une seule itération. Il y a deux raisons à ce choix. La première est directement liée à la nature des simulations. Il n'y a pas d'évènements aléatoires dans les simulations. De ce fait, chaque itération prend le même temps sauf s'il y a des changements dans les données. La deuxième raison est la durée de la simulation. En effet, pour simuler 10 ms, il faut environ 2 minutes. En conséquence, si nous souhaitons simuler les 10 secondes prévues par le CoreMark, la durée de la simulation serait plus longue qu'une journée et les volumes de données générées seraient énormes. CoreMark prévoit une simulation de 10 secondes afin de moyenner l'ensemble des facteurs externes qui influencent le temps d'exécution du *benchmark*. Avec nos simulations comportementales, nous n'avons pas de facteurs externes au processeur. Nous pouvons donc effectuer une simulation plus courte en durée que celle prévue par le CoreMark.

Pour mesurer le temps d'exécution du *benchmark*, nous procédons de deux manières différentes. Tout d'abord, nous utilisons des compteurs directement disponibles dans le Cortex-M0 et les processeurs RISC-V. Nous comparons ensuite ces valeurs avec celles déduites d'un calcul de temps entre des signaux que nous envoyons par le GPIO.

Pour le Cortex-M0, le calcul de temps se base sur les interruptions système. Le processeur possède un compteur interne dont la valeur initiale est définie dans le programme du *benchmark*. Ce compteur est décrémenté de 1 à chaque cycle d'horloge. Lorsqu'il atteint la valeur 0, une interruption est générée. Il est alors remis à sa valeur initiale au prochain cycle d'horloge. Dans le programme du *benchmark*, à chaque fois que cette interruption se produit, nous incrémentons de 1

un compteur que nous appelons `tick`. Si les interruptions se produisent trop souvent, cela va avoir un impact non négligeable sur le temps mis pour exécuter le *benchmark*. Pour cela, nous avons fixé la durée entre deux interruptions à 200 ms ce qui correspond à 16 millions de cycles d'horloge (presque la valeur maximale autorisée). Pour ce faire, nous avons fixé la valeur initiale du compteur interne au processeur à 16 millions. Pour calculer le temps d'exécution, nous utilisons la formule suivante :

$$T = nombre\ interrupt * 200\ ms + \frac{(16\ 000\ 000 - valeur\ du\ compteur - 1)}{80\ MHz}$$

Le code pour traiter l'interruption est le suivant :

```
1.  void SysTick_Handler(void){tick++;}
```

Code 4 - Traitement de l'interruption SysTick

Le code pour initialiser, lancer et arrêter le compteur est le suivant :

```
1.  SysTick->CTRL = 0; // arrête le compteur
2.  SysTick->LOAD = 16000000-1; // valeur chargée dans le compteur
3.  SysTick->VAL = 0; // reset du compteur
4.  SysTick-
    >CTRL = (SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_TICKINT_Msk | SysTick_CTRL_ENABLE
    _Msk); // active le compteur avec comme source de clock la clock interne et active
    l'interruption
```

Code 5 - Initialisation, lancement et arrêt du compteur

Pour les processeurs Zero-riscy et RI5CY, le calcul de temps se base sur les compteurs de performance qui comptent le nombre de cycles où le processeur est actif. Le temps écoulé est donc calculé avec la formule suivante :

$$T = \frac{\#\ cycle\ fin - \#\ cycle\ début}{80\ MHz}$$

Les instructions pour commander les compteurs de performance pour les processeurs RISC-V sont données dans le Code 6.

```
1.  asm volatile ("csrr %0, 0x780" :"=r" (cycle)); // csrr -
    > lecture du compteur de performance à l'adresse donnée (0x780) et le mettre dans la
     variable cycle
2.  asm volatile ("csrw 0x7A0, %0" :"+r"(eventMask)); // csrw -
    > écriture dans le compteur de performance à l'adresse donnée (0x7A0) avec la valeur
     contenue dans eventMask
```

Code 6 - Accès aux compteurs de performance pour les processeurs RISC-V

Ces instructions sont des encapsulations de code assembleur dans un code C pour être plus simple à utiliser et permettre au compilateur d'utiliser les bons registres. L'adresse 0x7A0 pour les processeurs Zero-riscy et RI5CY est l'adresse permettant de choisir quels compteurs de performance sont actifs. L'adresse 0x780 correspond au compteur du nombre de cycles où le processeur est actif. Les adresses peuvent changer pour d'autres processeurs avec l'architecture RISC-V mais les instructions resteront les mêmes.

Pour confirmer les valeurs obtenues à l'aide des compteurs déjà présents dans les processeurs, nous envoyons un signal via le GPIO au début et à la fin du code dont nous mesurons le temps d'exécution. En calculant la durée entre le signal de début et le signal de fin, nous obtenons également le temps d'exécution. Nous avons constaté que la différence entre les temps d'exécution donnés par les processeurs et les temps calculés à l'aide du GPIO est négligeable.

Extraction de la consommation

La consommation d'un système digital, c'est-à-dire, la puissance électrique consommée par ce système est divisée en deux parties :

- la puissance dynamique,
- la puissance statique.

La puissance dynamique est la partie qui varie en fonction du temps. Pour un processeur, la puissance dynamique se compose de la puissance de transition et de la puissance de court-circuit. La puissance de transition est présente à chaque charge d'un nœud. Ces nœuds sont toujours capacitifs. L'énergie de transition pour un nœud est donnée par la formule :

$$E_{SW,i} = \int_0^T i_{DD} V_{DD} dT = C_{L,i} V_{DD}^2 \text{ [27]}$$

La puissance de transition pour un nœud dépend du nombre de fois que ce nœud est chargé. Pour calculer cela, nous introduisons le facteur d'activité (α_F) de la porte logique. Par défaut, c'est le rapport du nombre de transitions à l'entrée qui donnent une transition de 0 à 1 à la sortie sur le nombre de transitions possibles à l'entrée. La formule suivante donne la puissance de transition pour un nœud.

$$P_{SW,i} = E_{SW,i} \alpha_{F,i} f_{clk} = \alpha_{F,i} f_{clk} C_{L,i} V_{DD}^2 \text{ [27]}$$

La puissance de transition totale se calcule en effectuant la somme des puissances de transition de tous les nœuds du système.

$$P_{SW} = \sum_i E_{SW,i} \alpha_{F,i} f_{clk} = C_L V_{DD}^2 \alpha_F f_{clk} N_{nodes} \text{ [27]}$$

La puissance de court-circuit apparaît uniquement lorsqu'il y a une transition. Lors de la transition, la branche de pull-up et la branche de pull-down (Figure 17) sont passantes et un courant passe directement de l'alimentation à la masse. Pour réduire cette puissance, nous essayons d'avoir des transitions très abruptes et donc de diminuer le temps de transition. Cette puissance est environ 5 à 10% de la puissance de transition.

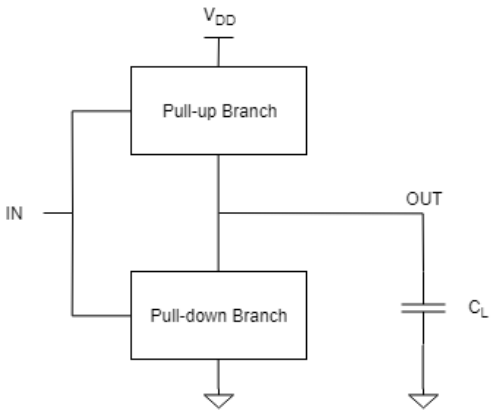

Figure 17 - Schéma du circuit d'une porte logique

La puissance statique dissipée par le processeur est principalement due au courant de fuite (*leakage*) des transistors. Cette puissance se calcule avec la formule suivante : $P_{stat} = I_{leak} V_{DD}$

Pour extraire la consommation, nous commençons par générer un fichier permettant de calculer le facteur d'activité de chaque porte logique ($\alpha_{F,i}$). Ce fichier nous permet d'avoir un calcul de consommation bien plus précis. Ce fichier (appelé Annotation de l'activité dans la Figure 16) est généré durant la simulation comportementale. Une fois la simulation terminée, pour obtenir la consommation, nous procédons à la synthèse qui nous permet de passer d'un niveau de porte logique au niveau des transistors avec leur dimensionnement. Une fois cela fait, le programme utilisé nous donne un rapport avec la consommation dynamique ainsi que la consommation de fuite. Ce rapport s'obtient avec la commande `report_power -hier` une fois que la synthèse est terminée.

Extraction de la surface

L'extraction de la surface se fait au même niveau que l'extraction de la consommation à une exception près. Nous n'avons pas besoin du fichier permettant de calculer le facteur d'activité de chaque porte logique. Pour récupérer la surface occupée par le processeur, nous utilisons un autre rapport généré par le programme (Design Vision) qui effectue la synthèse. Ce rapport s'obtient avec la commande `report_area -nosplit` une fois que la synthèse est finie.

Extraction des accès à la mémoire

L'extraction des accès à la mémoire s'effectue aisément car nous avons des mémoires comportementales. Nous pouvons donc rajouter un morceau de code dans la mémoire qui permet d'écrire dans un fichier texte chaque accès au format suivant : « (Lecture/Écriture)(Taille) (Donnée) @ (Adresse) ». Cela nous donne pour une lecture d'un mot la ligne suivante : RW 0xXXXXXXXX @ 0xXXXXXXXX. À l'aide d'un script en Python il est donc possible de traiter ce fichier texte pour en extraire le nombre d'accès à la mémoire total, le rapport entre les différents accès à la mémoire (lecture/écriture et word/half-word/byte)… Le code Python de ce script est mis en annexe.

3.3 Compilation du software

La compilation du software nous permet de passer du code C du *benchmark* à une série d'instructions compréhensibles par le processeur. La compilation est divisée en trois étapes pour notre banc d'essai :

- compilation,
- *link*,
- conversion en fichier mémoire.

La première étape consiste à compiler tous les fichiers utilisés par le *benchmark*. Lors de cette compilation, nous pouvons choisir le niveau d'optimisation. Au plus le niveau d'optimisation est élevé, au plus les performances sont élevées. Il existe 4 niveaux d'optimisation. Le premier niveau (O0) est le niveau sans optimisation. Les trois autres niveaux (O1, O2 et O3) utilisent des optimisations de plus en plus poussées. Par contre, plus les optimisations sont poussées, plus le temps de compilation augmente et plus la mémoire utilisée pour stocker les instructions augmente. Pour le niveau d'optimisation le plus poussé, le compilateur effectue par défaut un compromis entre la taille du code et les performances. Il est possible de lui dire de ne pas faire ce compromis à l'aide de paramètres (flag). Dans le cas du Cortex-M0, le paramètre Otimes permet de dire au compilateur de ne pas tenir compte de la taille du code [28].
En plus de cela, nous pouvons rajouter des options pour que le compilateur optimise davantage certains points. Le déroulement des boucles est une optimisation qui n'est pas effectuée par défaut

dans le cas O3, car elle augmente significativement la taille du code. Cependant, comme elle augmente les performances, nous avons choisi d'utiliser cette optimisation car, pour nous, les performances sont plus importantes que la taille du code. Quand le compilateur déroule une boucle, il recopie l'intérieur de la boucle plusieurs fois pour réduire le nombre de branchements. Comme expliqué dans la section 2.1.1 (Fonctionnement d'un processeur), à chaque branchement, nous perdons du temps. Si nous réduisons le nombre de branchements, nous réduisons le temps perdu à cause des branchements. De cette manière, nous gagnons en performances. Pour le Cortex-M0, cela correspond au paramètre `loop_optimization_level=2` et pour les processeurs RISC-V au paramètre `-funroll-loops` [29].

La deuxième étape consiste à *linker* tous les fichiers en un seul fichier exécutable. C'est lors de cette étape que le compilateur va effectuer la carte des adresses en mémoire. Pour effectuer cette carte, nous allons indiquer au compilateur les plages d'adresses des différentes mémoires (IMEM/DMEM). Pour ce faire, nous utilisons un script de *link*. Ce script est automatiquement généré par l'environnement de développement Keil µVision pour le Cortex-M0. Par contre pour les processeurs Zero-riscy et RI5CY, ce script n'est pas directement généré. Nous avons dû l'écrire sur base des scripts présents dans les *repository* d'un des microcontrôleurs développés par ETH Zurich : PULPino [30]. Dans cette deuxième étape, nous avons également besoin du code de *boot*. Celui-ci contient deux éléments très importants :

- la table des vecteurs,
- le code pour traiter les interruptions systèmes.

La table des vecteurs contient les adresses des différents codes de traitement des interruptions ainsi que l'adresse du début du code du programme. Pour le Cortex-M0, ce code de *boot* est lui aussi généré par l'environnement de développement que nous utilisons. Pour Zero-riscy et RI5CY, ce code de *boot* se trouve dans le *repository* de PULPino.

La dernière étape consiste à transformer le fichier obtenu à la fin du *link* en un fichier utilisable par le banc d'essai. Le fichier résultant correspond au contenu de la mémoire des instructions à charger. Pour effectuer cette transformation, l'environnement de développement pour le Cortex-M0 embarque un outil qui s'en occupe. Par contre, pour les processeurs Zero-riscy et RI5CY, cet outil n'existe pas. Nous avons donc écrit un script Python qui permet de faire cette transformation.

4 Caractérisation du Cortex-M0 DS

Dans ce chapitre, nous caractérisons le processeur Cortex-M0 DS (*Design Start*). Tout d'abord, nous analysons ses performances. Ensuite, nous regardons sa consommation. Nous continuons par la surface occupée par le processeur. Pour terminer, nous nous intéressons aux accès à la mémoire des données et la mémoire des instructions. Pour les comparaisons, nous avons pris le cas de référence suivant : optimisation du compilateur au niveau 3 sans tenir compte de la taille du code et présence d'un multiplieur 32 bits. C'est le cas qui donne les meilleures performances.

4.1 Analyse des performances

Dans cette section, nous analysons les performances du Cortex-M0 DS. Nous commençons par comparer les performances que nous avons obtenues avec les chiffres fournis par ARM. Nous analysons par la suite l'influence des deux paramètres suivants : optimisation du compilateur et l'influence du multiplieur 32 bits.

	ARM	Ce travail
Performances (CoreMark/MHz)	2.33	1.98
Compilation Flag	-Ohs, --no_size_constraints	-O3, -Otime, --loop_optimization_level=2
Version du compilateur	IAR 6.60	5.06 update 6

Tableau 2 - Comparaison des performances entre les données du fournisseur et ce travail

Le Tableau 2 reprend la comparaison entre les données du fournisseur du processeur et les performances obtenues avec notre banc d'essai. Nous observons une différence de 0.35 CoreMark/MHz (-15%). Il est tout à fait normal d'avoir cette différence, car nous n'utilisons pas la même version du compilateur qu'ARM, ni les mêmes optimisations et nous ne connaissons pas la version du processeur utilisé par le fournisseur pour produire ses chiffres.

Analysons l'influence de l'optimisation du compilateur par rapport à notre cas de référence.

	O3	O2	O1	O0
Performances (CoreMark/MHZ)	1.98	1.35 (-32%)	1.21 (-39%)	0.87 (-56%)
Taille du code (word)	3003	2496 (-17%)	2503 (-17%)	2746 (-9%)

Tableau 3 - Influence de l'optimisation du compilateur

Le Tableau 3 montre l'augmentation des performances et la diminution de la taille du code en fonction de l'augmentation du niveau d'optimisation. Nous remarquons que l'optimisation du compilateur joue un rôle important dans les performances d'un système. Si les optimisations utilisées par le fournisseur sont plus poussées que celles que nous utilisons, la différence de 15% entre les données du fournisseur et notre travail est explicable. Vu que notre version du compilateur est plus ancienne que celle utilisée par ARM, il est possible que les optimisations que nous utilisons soient moins poussées. Conformément à ce que nous avons expliqué dans la section 3.3, la taille du code augmente avec l'augmentation du niveau d'optimisation. La seule exception est le niveau 0 qui n'effectue pas d'optimisation. En effet, quand aucune optimisation n'est effectuée, la taille du code dépend de la manière avec laquelle le code source est écrit.

Pour le Cortex-M0, il existe plusieurs versions du processeur. La principale différence entre ces versions est le multiplieur. Il s'agit soit d'un multiplieur 32 bits en un cycle, soit d'un multiplieur

récursif qui prend 32 cycles pour effectuer une multiplication sur 32 bits. La différence des performances dépend totalement du programme exécuté. Si le programme n'effectue aucune multiplication sur 32 bits, le multiplieur 32 bits en un cycle n'est pas utilisé et est donc inutile. Dans le cas de CoreMark, des multiplications sur 32 bits sont effectuées. Le Tableau 4 montre l'influence de ce multiplieur sur le Cortex-M0 DS. La différence de performances est loin d'être négligeable. Sans le multiplieur 32 bits en un cycle, nous perdons 37% de performances. Pour savoir s'il augmente l'efficacité du processeur, il faut comparer son influence sur la consommation du processeur.

	Avec multiplieur	Sans multiplieur
Performances (CoreMark/MHz)	1.98	1.24 (-37%)

Tableau 4 - Influence du multiplieur 32 bits en un cycle sur les performances

4.2 Analyse de la consommation

Dans cette section, nous analysons la consommation de puissance du Cortex-M0 DS. Nous commençons par comparer la consommation déduite de nos simulations avec les chiffres fournis par ARM. Nous regardons également l'influence du multiplieur 32 bits sur la consommation du Cortex-M0 DS.

	ARM	Ce travail
Technologie	40 nm LP @ 1.1V	28nm @ 0.4V
Consommation totale du CPU	5.3 µW/MHz	0.28 µW/MHz
Consommation totale du CPU normalisée	0.49 µW/MHz	0.28 µW/MHz

Tableau 5 - Comparaison de la consommation entre les données du fournisseur et ce travail

Le Tableau 5 montre la consommation donnée par le fournisseur du processeur (ARM) et la consommation déduite de nos simulations. Pour comparer ces deux consommations, nous normalisons par rapport à la tension utilisée et à la technologie utilisée à l'aide de la formule suivante :

$$P_{norm} = P \frac{28\,nm}{technologie\ d'originie} \left(\frac{0{,}4V}{tension\ d'alimentation\ d'origine}\right)^2$$

Nous observons une différence de 0.21 µW/MHz (soit 42.9%) entre les chiffres du fournisseur et ceux déduits de nos simulations. Cette différence s'explique par :

- la normalisation,
- le facteur d'activité,
- la version du processeur,
- l'utilisation du *clock gating*.

La normalisation effectuée se base sur le postulat que la consommation dépend linéairement de la technologie. En pratique, cette influence n'est pas totalement linéaire. Il y a un certain nombre de phénomènes d'ordres supérieurs qui ne sont pas pris en compte. Cette approximation explique une grande partie de la différence de consommation.

Le facteur d'activité influence linéairement la consommation dynamique. Le facteur d'activité peut soit être le facteur d'activité par défaut de chaque porte logique, soit être calculé en fonction du

nombre de transitions effectuées par la porte logique. Le facteur d'activité utilisé par ARM n'est pas connu. Il nous est donc impossible de reproduire ses chiffres de consommation.

Le facteur d'activité dépend du programme exécuté. Comme chaque programme n'effectue pas les mêmes types d'opérations, l'activité de certains blocs comme l'ALU change. Par exemple, si le programme effectue beaucoup d'additions, mais très peu de multiplications, l'ensemble des portes logiques du multiplieur ne change pas et leurs facteurs d'activité sont très bas. Par contre, les portes logiques de l'additionneur changent beaucoup et leurs facteurs d'activité sont très élevés. Avec un programme qui effectue beaucoup de multiplications et très peu d'additions, les facteurs d'activité du multiplieur et de l'additionneur sont différents. Entre ces deux programmes, il y a une différence de consommation provenant de l'activité des différents blocs constituant le processeur.
Dans ce travail, nous utilisons un facteur d'activité qui correspond à l'exécution du CoreMark.

L'utilisation du *clock gating* permet de diminuer la consommation dynamique en désactivant l'horloge des registres qui contiennent des valeurs qui n'ont pas besoin d'être mises à jour. De ce fait, nous diminuons le facteur d'activité de ces registres et du bloc de logique qui les suit, car il n'y a plus de changement.

Le processeur Cortex-M0 DS possède plusieurs versions. Comme vu dans l'analyse des performances, le multiplieur 32 bits en un cycle a une influence sur les performances. Il a aussi une influence sur la consommation car c'est un ajout de portes logiques. Le Tableau 6 montre cette influence.

	Avec multiplieur	Sans multiplieur
Consommation dynamique (µW)	18.6	24.48 (32%)
Consommation de fuite (µW)	4.18	3.44 (-23%)

Tableau 6 - Influence du multiplieur 32 bits en un cycle sur la consommation du Cortex-M0 DS

Nous remarquons qu'en retirant le multiplieur 32 bits en un cycle, la consommation dynamique augmente de 32%. Cette augmentation s'explique par l'augmentation du nombre de cycles qu'il faut pour effectuer les multiplications en 32 bits. Durant tous ces cycles, un grand nombre de registres ne peuvent pas être désactivés. En conclusion, l'utilisation du multiplieur a bien un impact positif sur l'efficacité du processeur.

Dans notre cas, l'impact sur la consommation totale de l'augmentation de la consommation de fuite est moindre que celui créé par l'augmentation de la consommation dynamique. En effet, la consommation dynamique (à 80 MHz) est 4 fois plus grande que la consommation de fuite avec multiplieur et 7 fois sans multiplieur. Par contre, si la fréquence de l'horloge diminue, l'impact de la consommation de fuite sur la consommation totale augmente.

L'augmentation de la consommation de fuite est liée à l'augmentation de la surface du processeur. En effet, par définition, la consommation de fuite dépend du courant de fuite total qui dépend du courant de fuite de chaque porte logique. Si nous ajoutons des portes logiques, comme c'est le cas avec le multiplieur 32 bits, le courant de fuite augmente et donc la consommation de fuite aussi.

4.3 Analyse de la surface

Dans cette section, nous allons analyser l'influence du multiplieur 32 bits sur la surface du processeur. Le Tableau 7 reprend la surface du processeur avec et sans le multiplieur 32 bits en un cycle.

	Avec multiplieur	Sans multiplieur
Surface (kGe)	20.84	15.60 (-25%)

Tableau 7 - Influence du multiplieur 32 bits en un cycle sur la surface du Cortex-M0

En retirant le multiplieur 32 bits en un cycle, nous économisons un quart de la surface totale du processeur. Mais ce n'est pas intéressant pour notre cas, car sans le multiplieur nous diminuons trop fortement les performances comme nous pouvons le déduire du Tableau 4 et nous augmentons la consommation totale comme le montre le Tableau 6.

4.4 Analyse des accès à la mémoire

Dans cette section, nous analysons la répartition des accès à la mémoire en fonction du type de mémoire (IMEM/DMEM) et du type d'accès (Lecture/Écriture). Cette analyse nous permet de savoir quelle mémoire est la plus sollicitée. De cette manière, nous savons où concentrer nos efforts pour diminuer la consommation totale du SoC. Nous analysons ensuite l'influence du multiplieur 32 bits sur le nombre d'accès à la mémoire effectués par le processeur.

La Figure 18 montre la répartition des accès à la mémoire en fonction du type de mémoire, du type d'accès et de la taille des données lues ou écrites. Les accès à la mémoire sont donnés selon le pourcentage des cycles exécutés par le processeur. Cela nous permet de savoir quelle proportion de temps le processeur passe à lire ou écrire des données en mémoire. La Figure 18 nous permet de déduire que la mémoire des instructions est la plus lue par le processeur.

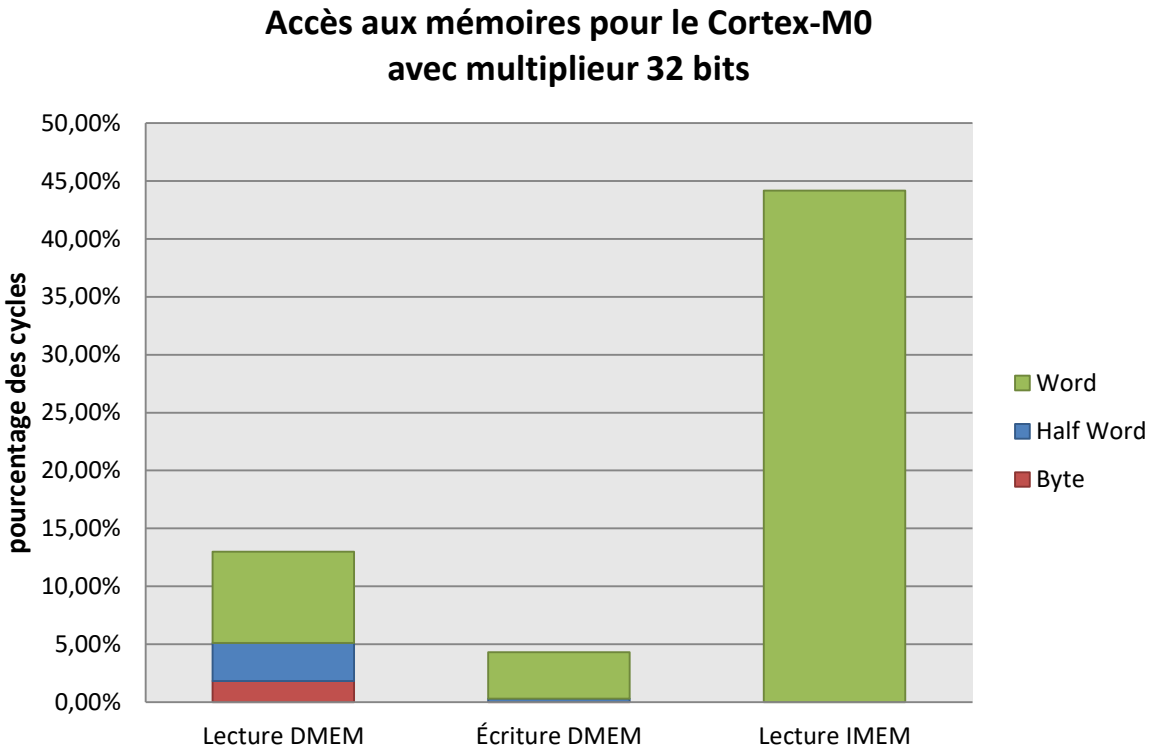

Figure 18 - Répartition des accès aux mémoires pour le Cortex-M0 avec multiplieur 32 bits en un cycle

Le Tableau 8 reprend la comparaison des accès à la mémoire en fonction de la présence du multiplieur 32 bits en un cycle ainsi que le pourcentage du nombre de cycles passés à accéder aux mémoires.

	Avec multiplieur	Sans multiplieur
Nombre de lectures dans l'IMEM	224.72 k	224.72 k
Nombre de lectures dans la DMEM	66.12 k	66.12 k
Nombre d'écritures dans la DMEM	21.95 k	21.95 k
Pourcentage du nombre de cycles	61.47 %	43.5 %

Tableau 8 - Influence du multiplieur 32 bits en un cycle sur les accès à la mémoire

Il est tout à fait normal que le multiplieur 32 bits en un cycle n'impacte pas les accès à la mémoire. En effet, lorsque le processeur effectue une multiplication dans le cas où le multiplieur est présent, il lit l'instruction en mémoire et transmet à l'ALU les opérandes et l'opération à effectuer. L'ALU effectue la multiplication en un cycle. Dans le cas où le multiplieur 32 bits en un cycle n'est pas présent, c'est juste au niveau de l'ALU qu'il y a une différence. Au lieu de prendre un cycle pour effectuer la multiplication, l'ALU utilise 32 cycles. Dans aucun cas, l'ALU n'ira écrire ou lire des données dans la mémoire.

La différence dans le pourcentage du nombre de cycles provient de la diminution du nombre de cycles pour faire exécuter CoreMark. Si le processeur utilise moins de cycles, mais qu'il n'y a aucun changement dans les accès à la mémoire, le pourcentage du nombre de cycles d'accès à la mémoire est plus élevé.

5 Comparaison du Cortex-M0 avec les autres processeurs

Dans ce chapitre, nous comparons le Cortex-M0 avec les processeurs de type RISC-V suivants :

- Zero-riscy,
- RI5CY.

Nous évaluons ensuite l'impact de la consommation des mémoires sur l'efficacité des différents processeurs que nous comparons.

5.1 Zero-riscy

Zero-riscy est un processeur supportant l'ISA RV32IMC. Il s'agit donc d'un processeur 32 bits utilisant la version 2.0 de l'ISA de base. L'extension M signifie que le processeur supporte les multiplications et les divisions. La dernière extension est celle qui permet d'utiliser des instructions compressées sur 16 bits et, ainsi, de prendre moins de place en mémoire [31].

Pour commencer, nous allons étudier l'influence des paramètres suivants : niveau d'optimisation du compilateur et utilisation des instructions compressées pour déterminer quel est le cas de référence que nous allons utiliser pour notre comparaison.

Le Tableau 9 compare les performances du Zero-riscy en fonction des paramètres cités ci-dessus. Nous remarquons clairement que c'est le cas O3 qui est le plus performant. Cependant, le gain en performances n'est que de 4.3% par rapport au cas O3C alors que l'augmentation de la taille du code est de 19%. Nous allons donc prendre deux cas comme référence pour la comparaison. Le premier cas de référence (O3) est le cas où nous ne faisons pas de compromis entre les performances et la taille du code. Pour le deuxième cas (O3C), nous acceptons de perdre un peu de performances pour diminuer de manière significative la taille du code.

	O3	O3C	O2C	O1C	O0C
Performances (CoreMark/MHz)	2.43	2.33	2.1	1.85	0.52
Taille du code (word)	7387	6193	2553	3520	3496
Instructions compressées	Non	Oui	Oui	Oui	Oui
Niveau d'optimisation	Niveau 3	Niveau 3	Niveau 2	Niveau 1	Niveau 0

Tableau 9 - Influence du niveau d'optimisation et des instructions compressées sur les performances du Zero-riscy

Si nous comparons les résultats que nous avons obtenus dans le Tableau 9 avec les informations que nous avons dans l'état de l'art (Tableau 1), nous remarquons qu'au niveau des performances pour le cas O3, nous avons exactement les mêmes valeurs.

	Cortex-M0	Zero-riscy (O3C)	Zero-riscy (O3)
Performances (CoreMark/MHz)	1.98	2.33 (17.7%)	2.43 (22.7%)
Consommation dynamique (µW)	18.6	34.89 (87.9%)	34.89 (87.9%)
Consommation de fuite (µW)	4.18	4.4 (5.2%)	4.4 (5.2%)
Surface (kGe)	20.835	21.399 (2.7%)	21.399 (2.7%)
Nombre de lectures dans l'IMEM	224.72 k	300.78 k (33.8%)	258.54k (15%)
Nombre de lectures dans la DMEM	66.12 k	56.87 k (-14%)	39.95k (-39.6%)
Nombre d'écritures dans la DMEM	21.95k	12.73 k (-42.0%)	7.39k (-66.3%)
Efficacité (CoreMark/µW)	6.95	4.74 (-31.8%)	4.95 (-28.8%)

Tableau 10 - Comparaison du Cortex-M0 et de Zero-riscy

Si Zero-riscy est plus performant que le Cortex-M0, il consomme aussi beaucoup plus. De ce fait que ce soit avec les instructions compressées (O3C) ou sans les instructions compressées (O3), Zero-riscy est moins efficace que le Cortex-M0. Par contre, si nous effectuons la même comparaison avec les données de l'état de l'art, nous obtenons le résultat inverse (voir Tableau 11).

	Cortex-M0 (ce travail)	Cortex-M0 [7]	Zero-riscy (O3) (ce travail)	Zero-riscy [9]
Performances (CoreMark/MHz)	1.98	2.33	2.43	2.44
Consommation totale du CPU (µW/MHz)	0.285	0.49	0.491	0.10
Efficacité (CoreMark/µW)	6.95	4.76	4.95	24.4

Tableau 11 - Comparaison de ce travail avec les données de l'état de l'art pour Zero-riscy

Dans le chapitre 4, nous avons expliqué les différences entre nos mesures et les informations du constructeur pour le Cortex-M0. Pour Zero-riscy, la différence de consommation entre nos mesures et les informations de l'article [9] peut également s'expliquer par la normalisation effectuée, le facteur d'activité utilisé et l'utilisation du *clock gating* (voir section 4.2).

Au niveau des accès à la mémoire, nous remarquons une différence entre les deux cas de références utilisés pour Zero-riscy. Avec les instructions compressées, le processeur effectue 16% de lecture en plus. Cela est provient du fait que toutes les instructions de l'ISA RISC-V ne peuvent pas être compressées sur 16 bits au lieu de 32 bits. De ce fait, nous pouvons avoir des instructions non compressées qui se retrouvent coupées en deux. Une partie sur un mot et la deuxième partie sur le mot suivant. Pour aller lire ces instructions, il faut deux accès à la mémoire différents et deux cycles d'horloge. C'est de là que provient la perte de performances avec l'utilisation des instructions compressées et l'augmentation du nombre d'accès à la mémoire des instructions.

5.2 RI5CY

Comme décris dans la section 2.2.2, RI5CY est un processeur supportant l'ISA RV32IMFC. Comparé à Zero-riscy, il supporte les instructions de calculs avec des nombres en virgule flottante. Il embarque aussi un certain nombre d'accélérateurs matériels.

Pour commencer, nous allons étudier l'influence des paramètres suivants : niveau d'optimisation du compilateur et l'utilisation des instructions compressées pour déterminer quel cas de référence nous allons utiliser pour le comparer avec le Cortex-M0.

Le Tableau 12 compare les performances de RI5CY en fonction des paramètres cités ci-dessus. Nous remarquons clairement que c'est le cas O3 qui est le plus performant. Cependant, le gain en performances n'est que de 2% comparé à l'augmentation de la taille du code qui est de 18%. Tout comme pour Zero-riscy, nous allons prendre deux cas de référence distincts. Le premier cas de référence (O3) est le cas où nous ne faisons pas de compromis entre les performances et la taille du code. Pour le deuxième cas (O3C), nous acceptons de perdre un peu de performances pour diminuer de manière significative la taille du code.

	O3	O3C	O2C	O1C	O0C
Performances (CoreMark/MHz)	3.14	3.08	2.76	2.14	0.64
Taille du code (word)	6919	5859	296	2732	4321
Instructions compressées	Non	Oui	Oui	Oui	Oui
Niveau d'optimisation	Niveau 3	Niveau 3	Niveau 2	Niveau 1	Niveau 0

Tableau 12 - Influence du niveau d'optimisation et des instructions compressées sur les performances de RI5CY

Au niveau des performances, nous obtenons des valeurs très proche de celles provenant de l'article [9].

	Cortex-M0	RI5CY (O3C)	RI5CY (O3)
Performances (CoreMark/MHz)	1.98	3.08 (55.5%)	3.14 (58.6%)
Consommation dynamique (µW)	18.6	70.08 (277.6%)	70.08 (277.6%)
Consommation de fuite (µW)	4.18	15.71 (275.7%)	15.71 (275.7%)
Surface (kGe)	20.835	75.7 (263.3%)	75.7 (263.3%)
Nombre de lectures dans l'IMEM	224.72 k	311.12 k (38.5%)	317.77k (41%)
Nombre de lectures dans la DMEM	66.12 k	56.52 k (-14.5%)	56.52k (-14.5%)
Nombre d'écritures dans la DMEM	21.95k	13.12 k (-40.2%)	13.12k (-40.2%)
Efficacité (CoreMark/µW)	6.95	2.87 (-58.7%)	2.93 (-57.9 %)

Tableau 13 - Comparaison du Cortex-M0 et de RI5CY

Si RI5CY est bien plus performant que le Cortex-M0, il consomme également beaucoup plus que le Cortex-M0. En conséquence, RI5CY n'est pas plus efficace que le Cortex-M0 dans notre cas. En effet, le gain en performances que nous avons ne compense pas du tout l'augmentation de la consommation. Par contre, si nous effectuons la même comparaison avec les données de l'état de l'art, nous obtenons le résultat inverse (voir Tableau 14).

	Cortex-M0 (ce travail)	Cortex-M0 [7]	RI5CY (O3) (ce travail)	RI5CY [9]
Performances (CoreMark/MHz)	1.98	2.33	3.14	3.19
Consommation totale du CPU (µW/MHz)	0.285	0.49	1.07	0.24
Efficacité (CoreMark/µW)	6.95	4.76	2.93	13.29

Tableau 14 - Comparaison de ce travail avec les données de l'état de l'art pour RI5CY

Dans le chapitre 4, nous avons expliqué les différences entre nos mesures et les informations du constructeur pour le Cortex-M0. Pour RI5CY, la différence de consommation entre nos mesures et les informations de l'article [9] peut également s'expliquer par la normalisation effectuée, le facteur d'activité utilisé et l'utilisation du *clock gating* (voir section 4.2).

Au niveau des accès à la mémoire, nous remarquons une légère différence entre les deux cas de références utilisés pour RI5CY sur le nombre de lectures dans la mémoire des instructions et aucun changement sur les accès à la mémoire des données.

5.3 Évaluation de la consommation globale

Dans cette section, nous allons comparer l'efficacité des processeurs en prenant en compte la consommation des mémoires utilisées. Tout d'abord, nous regardons l'impact sur l'efficacité si nous utilisons des mémoires différentes pour la mémoire des instructions (IMEM) et la mémoire des données (DMEM) avec des énergies d'accès différentes. Ensuite, nous analysons l'impact sur l'efficacité si nous faisons varier l'énergie d'accès tout en gardant le même rapport entre les énergies d'accès des deux mémoires. Lorsque nous fixons une énergie d'accès pour la DMEM ou l'IMEM, nous utilisons la valeur de 1pJ/accès. Cette valeur arbitraire correspond à une énergie d'accès à la mémoire pour une mémoire à très basse consommation.

5.3.1 Variation de l'énergie d'accès de l'IMEM avec l'énergie d'accès de la DMEM constante

Nous commençons par comparer l'efficacité des différents processeurs en faisant varier l'énergie d'accès de l'IMEM tout en gardant l'énergie d'accès de la DMEM constante (1 pJ/accès).

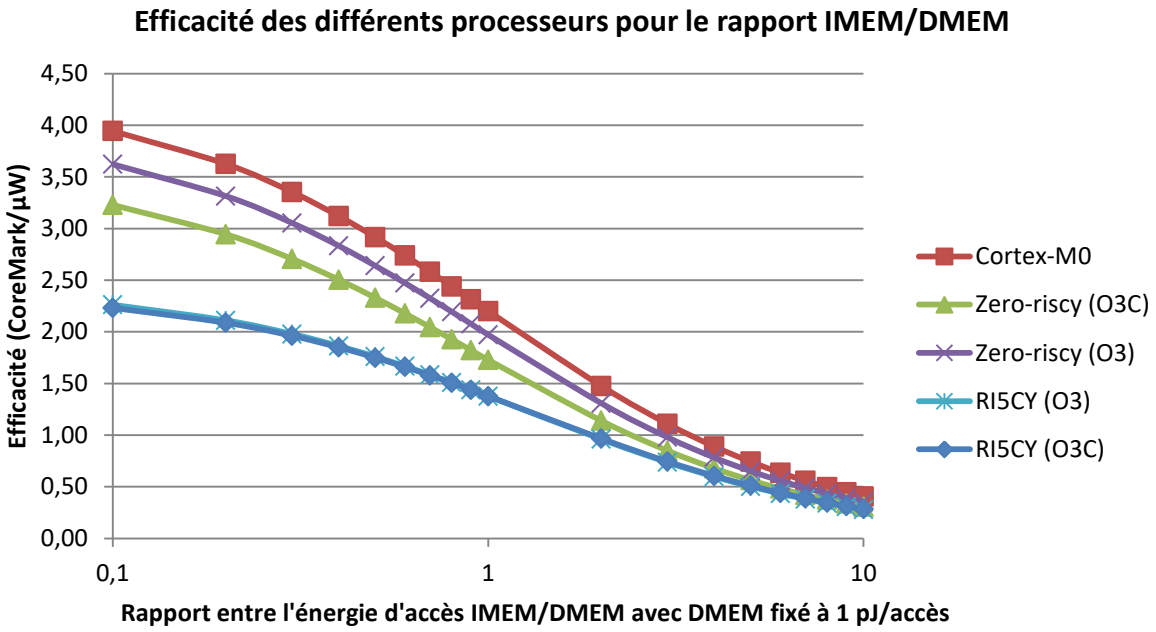

Figure 19 - Efficacité des différents processeurs en fonction du rapport d'énergie d'accès des mémoires IMEM et DMEM avec l'énergie d'accès de la DMEM fixé à 1 pJ/accès

Cela nous donne la Figure 19 où nous pouvons voir que le Cortex-M0 reste toujours le processeur le plus efficace.

Analysons maintenant en détail les différentes composantes de l'efficacité. Comme nous l'avons vu dans les sections 5.1 et 5.2, le Cortex-M0 est le processeur qui est le plus efficace sans prendre en compte la consommation des mémoires. Si nous tenons compte de la consommation des mémoires, l'efficacité du processeur va également dépendre du nombre d'accès effectués à l'IMEM et à la DMEM.
L'efficacité se calcule avec la formule suivante :

$$Efficacité = \frac{Performance}{Consommation\ globale}$$

$$= \frac{Performance}{Consommation\ IMEM + Consommation\ DMEM + Consommation\ totale\ du\ CPU}$$

La consommation de l'IMEM ou de la DMEM est directement proportionnelle à l'énergie d'accès de la mémoire et au nombre d'accès à la mémoire effectués par le processeur et inversement proportionnel au temps mis par le processeur pour effectuer l'ensemble des accès.

Le Cortex-M0 est le processeur qui a le moins d'accès à l'IMEM comme le montre la Figure 20. Par contre, le Cortex-M0 effectue plus d'accès à la DMEM que Zero-riscy (O3).

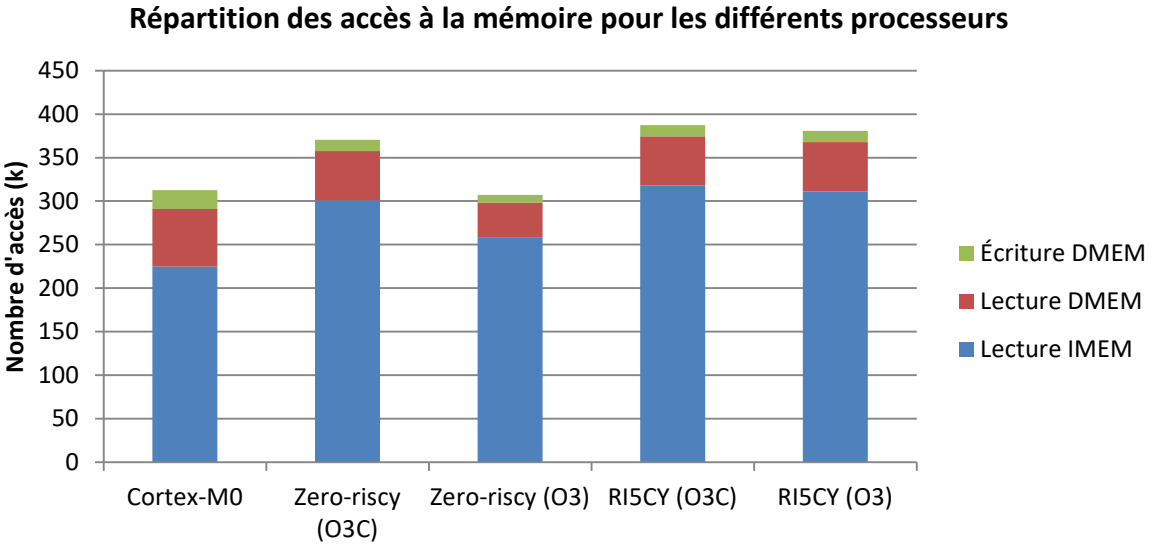

Figure 20 – Accès à la mémoire pour les différents processeurs

Illustrons la formule de l'efficacité par un exemple chiffré. Dans le Tableau 15, nous pouvons voir la consommation de l'IMEM, de la DMEM et du CPU pour le Cortex-M0 et Zero-riscy avec des énergies d'accès de 1 pJ/accès pour l'IMEM et la DMEM.

	Cortex-M0	Zero-riscy (O3)
Consommation de l'IMEM (µW/MHz)	0.441	0.624
Consommation de la DMEM (µW/MHz)	0.173	0.117
Consommation totale du CPU (µW/MHz)	0.285	0.491
Consommation globale (µW/MHz)	**0.899**	**1.232**
Performances (CoreMark/MHz)	1.98	2.43
Efficacité (CoreMark/µW)	2.2	1.97

Tableau 15 – Comparaison du Cortex-M0 et Zero-riscy pour des énergies d'accès aux mémoires de 1pJ/accès

La consommation totale du CPU est calculée sur base de la consommation dynamique et de la consommation de fuite venant du Tableau 10. Si le Cortex-M0 a des performances plus faibles que Zero-riscy, il a également une consommation totale moins élevée. Il en résulte que son efficacité est meilleure que celle de Zero-riscy. Si l'énergie par accès de l'IMEM diminue (comprise entre 0.1 et 1 pJ/accès), la consommation totale reste plus basse pour le Cortex-M0 et par conséquence son efficacité reste plus élevée comme le montre la Figure 19. Si l'énergie par accès de l'IMEM augmente (comprise entre 1 et 10 pJ/accès), le poids de la consommation de l'IMEM est plus important dans la consommation totale. Comme le Cortex-M0 effectue moins d'accès à l'IMEM que Zero-riscy, la consommation de l'IMEM est plus basse pour le Cortex-M0 que pour Zero-riscy. De ce fait, l'efficacité du Cortex-M0 reste plus élevée que celle de Zero-riscy.

5.3.2 Variation de l'énergie d'accès de la DMEM avec l'énergie d'accès de l'IMEM constante

Lorsque nous faisons varier la consommation de la DMEM tout en conservant la consommation de l'IMEM constante (1pJ/accès), nous obtenons la Figure 21.

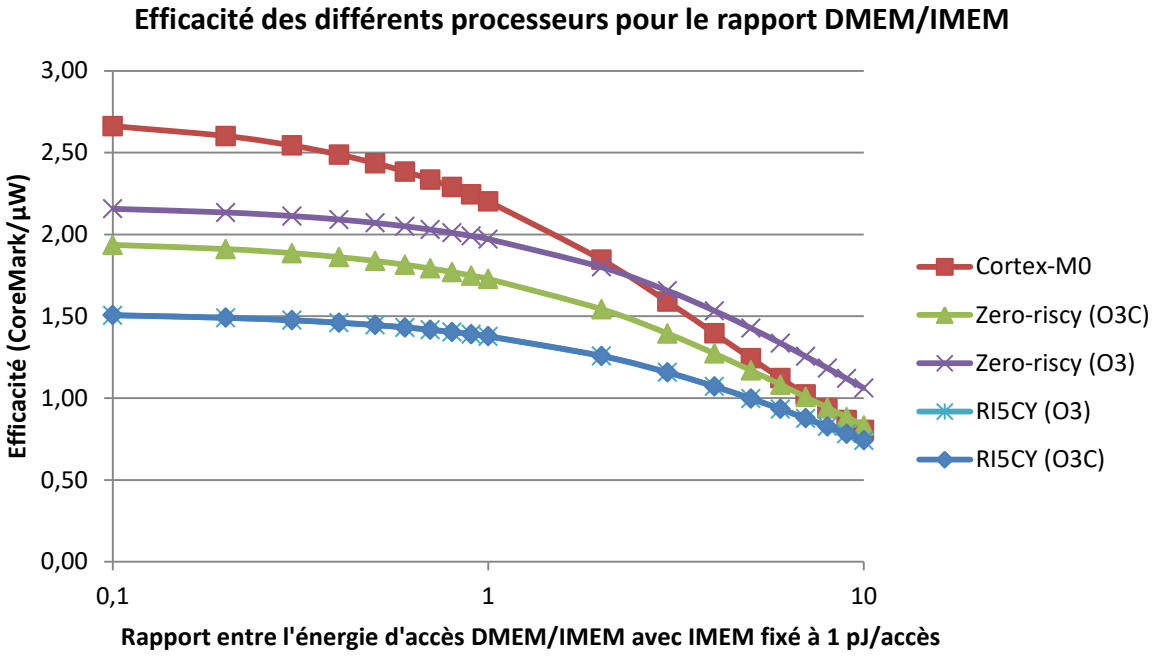

Figure 21 - Efficacité des différents processeurs en fonction du rapport d'énergie d'accès des mémoires DMEM et IMEM avec l'énergie d'accès de l'IMEM fixé à 1 pJ/accès

Si l'énergie par accès de la DMEM diminue (comprise entre 0.1 et 1 pJ/accès), la consommation totale reste plus basse pour le Cortex-M0 et par conséquence son efficacité reste plus élevée comme le montre la Figure 21. Si l'énergie par accès de la DMEM augmente (comprise entre 1 et 10 pJ/accès), le poids de la consommation de la DMEM est plus important dans la consommation totale. Comme Zero-riscy (O3) effectue moins d'accès à la DMEM que le Cortex-M0, la consommation de la DMEM de Zero-riscy (O3) augmente moins vite que celle du Cortex-M0. De ce fait, à un certain point, il y a inversion : l'efficacité de Zero-riscy (O3) devient plus élevée que celle du Cortex-M0.

Pour SleepRunner, l'IMEM est une macro faite sur mesure qui consomme 1.6 pJ/accès et la DMEM est une mémoire haute densité qui consomme 8.192 pJ/accès [32]. La Figure 22 montre l'efficacité des différents processeurs en fonction du rapport d'énergie d'accès des mémoires DMEM et IMEM avec l'énergie d'accès de l'IMEM fixé à 1.6 pJ/accès, c'est-à-dire la valeur actuelle pour l'énergie d'accès de l'IMEM de SleepRunner. Nous constatons que l'aspect général des Figure 21 et Figure 22 est similaire. Nous avons un rapport consommation de la DMEM sur la consommation de l'IMEM de 5.12 (ligne verticale orange sur la Figure 22). Nous pouvons déduire que Zero-riscy (O3) est plus efficace que l'ensemble des autres processeurs étudiés avec cette configuration de mémoire.

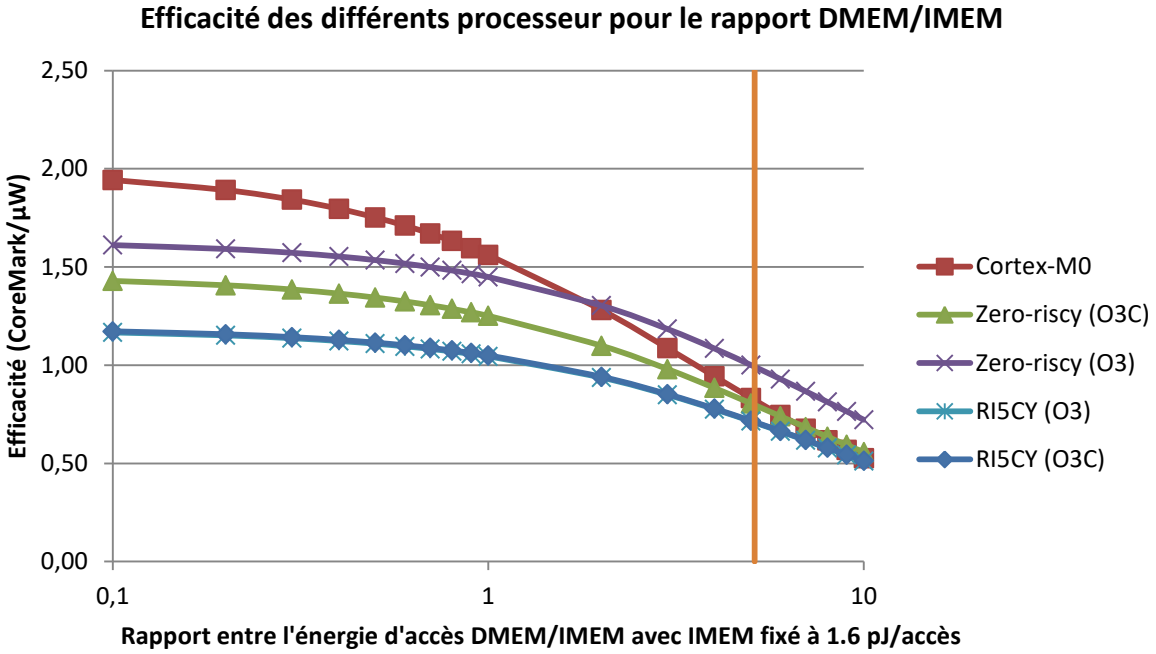

Figure 22 - Efficacité des différents processeurs en fonction du rapport d'énergie d'accès des mémoires DMEM et IMEM avec l'énergie d'accès de l'IMEM fixé à 1.6 pJ/accès

Actuellement, dans un travail de recherche effectué dans le groupe *Electronic Circuits & Systems* de l'UCLouvain, on tente de diminuer l'énergie d'accès de l'IMEM. Nous pouvons déjà prédire qu'avec une énergie d'accès de l'IMEM comprise entre 1.6 pJ/accès et 1 pJ/accès, Zero-riscy (O3) sera plus efficace que l'ensemble des autres processeurs étudiés.

5.3.3 Variation de l'énergie d'accès de l'IMEM et de la DMEM à rapport constant

Pour la configuration de mémoire utilisée dans SleepRunner, nous avons vu que le rapport entre les énergies d'accès à la DMEM et à l'IMEM est de 5.12. Si nous conservons ce rapport et que nous faisons varier l'énergie d'accès de l'IMEM, nous obtenons la Figure 23. Sur cette figure, nous remarquons que Zero-riscy est le plus efficace tant que l'énergie d'accès de l'IMEM est inférieure à 0.4 pJ/accès.

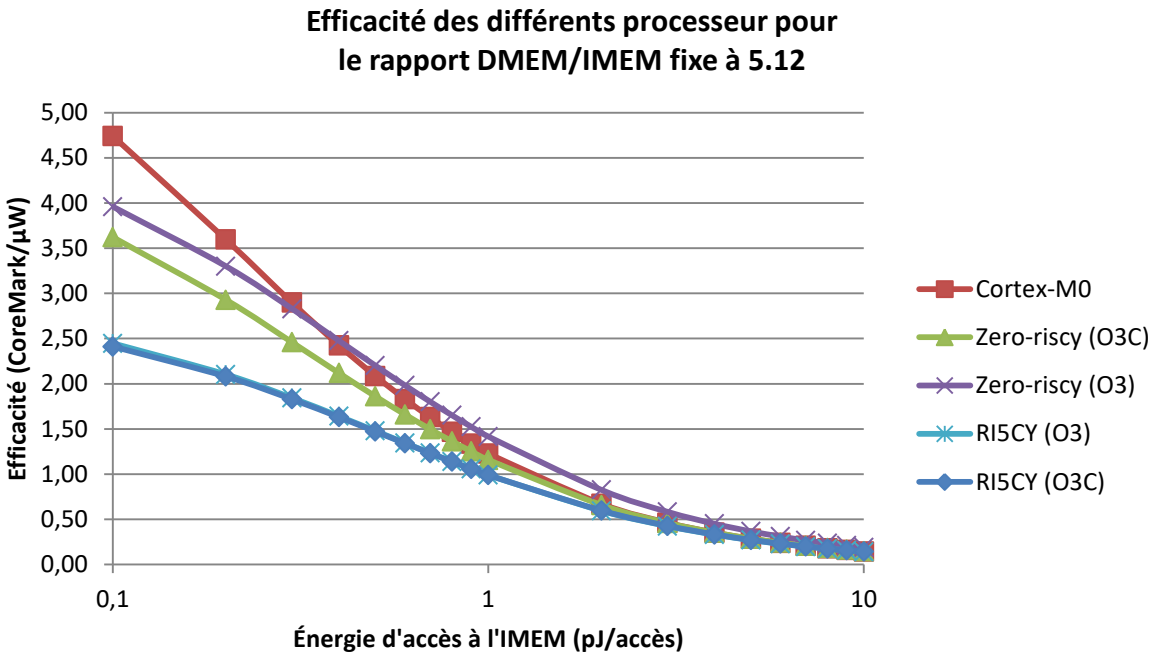

Figure 23 - Efficacité des différents processeurs pour un rapport DMEM/IMEM fixé à 5.12

Sur la Figure 24, le rapport entre les énergies d'accès de la DMEM et l'IMEM est fixé à 1. Cela correspond à l'utilisation de deux mémoires identiques pour la DMEM et l'IMEM. Dans ce cas, nous remarquons que le Cortex-M0 est toujours le plus efficace.

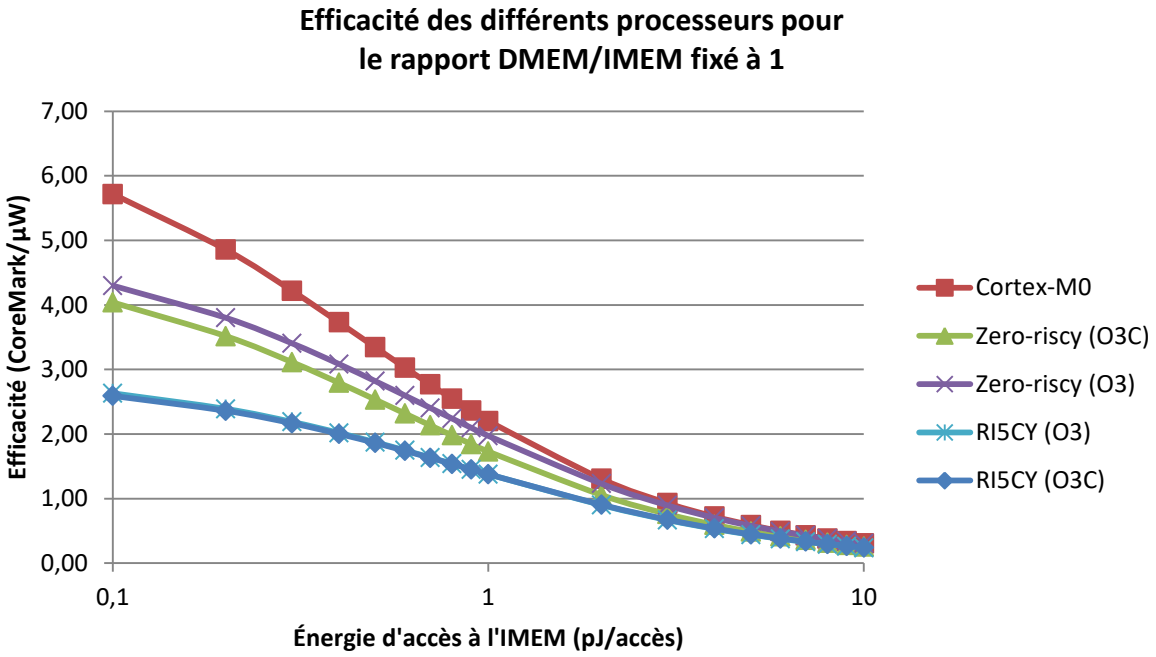

Figure 24 - Efficacité des différents processeurs pour un rapport DMEM/IMEM fixé à 1

6 Conclusion

Le but de ce travail est de voir s'il existe des processeurs très basse consommation basés sur l'architecture RISC-V qui sont plus efficaces que le Cortex-M0. Pour ce faire, nous avons développé un banc d'essai afin d'être en mesure d'effectuer une comparaison objective.

Pour réaliser ce banc d'essai, il y a plusieurs étapes clés :

- La première est la modification à apporter dans le code du *benchmark* pour pouvoir mesurer le temps d'exécution du *benchmark*. Pour ce faire, la documentation associée à chaque processeur est très utile. Pour tous les processeurs étudiés, il y a des compteurs internes qui permettent d'extraire ce temps d'exécution en comptant le nombre de cycles. Pour chacun d'eux, nous avons dû développer des fonctions pour récupérer les valeurs de ces compteurs et calculer le temps d'exécution. Ces développements que nous avons effectués sont réutilisables si nous souhaitons étudier un autre processeur de la famille RISC-V ou Cortex-M.
- La deuxième étape concerne la compilation du code source du *benchmark*. Pour les processeurs de chez ARM, l'outil fourni (keil µVision) est très simple à prendre en main et s'occupe de générer tous les fichiers nécessaires. Par contre pour les processeurs RISC-V, nous n'avons pas trouvé d'outil permettant de générer ces fichiers. Nous avons donc dû les créer sur base d'exemples provenant de ETH Zurich.
- La troisième étape clé est l'écriture du banc d'essai et, plus particulièrement, du bus de transfert. Pour ce faire, il faut se référer à la documentation du processeur pour déterminer le bus utilisé.

Une fois ces trois étapes terminées, nous pouvons commencer à effectuer les simulations et la synthèse pour extraire l'ensemble des facteurs de mérite : performances, consommation, surface, taille du code et accès à la mémoire.

Nous avons analysé l'impact des conditions de test (tension d'alimentation, fréquence de l'horloge, technologie et librairies) sur les différents facteurs de mérite. De cette analyse, nous pouvons conclure que tester les processeurs avec des conditions de test différentes et simplement normaliser au premier ordre par rapport à ces conditions de test ne permet pas d'effectuer une comparaison objective.

Nous avons étudié l'effet des optimisations effectuées par le compilateur sur les différents facteurs de mérite : performances, taille du code et nombre d'accès à la mémoire. De cette étude, nous pouvons conclure qu'il est très intéressant d'utiliser le niveau d'optimisation maximum, car il permet de fortement augmenter les performances du code exécuté sans avoir un impact trop important sur la taille du code. Plus le niveau d'optimisation est élevé, plus le compilateur va essayer de réduire le nombre d'accès à la mémoire.

Nous avons examiné l'impact du multiplieur 32 bits en un cycle sur les différents facteurs de mérite du Cortex-M0. De cet examen, nous pouvons conclure que le multiplieur 32 bits a un impact positif sur l'efficacité du processeur. Avec nos conditions de tests, il augmente grandement les performances, diminue la consommation totale du processeur et par conséquent améliore grandement l'efficacité énergétique.

Nous avons regardé l'impact des instructions compressées sur les facteurs de mérite de Zero-riscy et de RI5CY. Nous en concluons que les instructions compressées ont un impact négatif sur l'efficacité

du processeur. Elles diminuent fortement la taille du code tout en diminuant légèrement les performances.

Nous avons évalué l'impact de la consommation des mémoires sur les différents facteurs de mérite des processeurs. De cette évaluation, nous pouvons conclure que pour effectuer une comparaison objective, nous devons prendre en compte la consommation des mémoires. Étant donné que ces consommations dépendent de l'énergie d'accès de la mémoire et du nombre d'accès à la mémoire effectués par le processeur, il en résulte que l'efficacité du processeur va également en dépendre.

De la comparaison de l'efficacité du Cortex-M0 avec Zero-riscy et RI5CY, il ressort que Zero-riscy est le processeur le plus efficace si l'énergie d'accès de la mémoire des données est au moins deux fois plus élevée que celle de la mémoire des instructions. Dans tous les autres cas, le Cortex-M0 est le processeur le plus efficace parmi les processeurs étudiés.

De cette comparaison, nous pouvons également conclure que RI5CY est toujours le processeur le moins efficace pour le *benchmark* que nous utilisons. Le *benchmark* effectue des tâches de contrôle. RI5CY est optimisé pour effectuer du traitement de signaux et est donc moins efficace pour les tâches de contrôle.

Si dans un futur travail de recherche, on tente de diminuer l'énergie d'accès de la mémoire des instructions. Zero-riscy (optimisation de niveau 3) sera le processeur ayant le meilleur potentiel d'utilisation pour les tâches de contrôle en très basse consommation. Une piste de recherche ouverte sera l'amélioration de ce processeur en diminuant l'impact négatif de l'utilisation des instructions compressées.

Le banc d'essai développé pour ce travail de fin d'études est disponible en open source afin de permettre sa réutilisation pour prolonger ce travail de recherche en comparant d'autres processeurs ou en utilisant d'autres *benchmarks*.

7 Table des illustrations

Figure 1 - Schéma des blocs logiques d'un processeur [12] 8
Figure 2 - Diagramme temporel : (a) exécution séquentielle et (b) exécution pipeline [12] 9
Figure 3 - Exécution d'un branchement en *pipeline* [12] 10
Figure 4 - Architecture Von Neumann 10
Figure 5 - Architecture Harvard 10
Figure 6 - Calcul de la consommation totale du CPU en fonction de la fréquence sur base de la simulation du Cortex-M0 11
Figure 7 - Contrainte de *timing* [13] 12
Figure 8 - Influence de la période du cycle de l'horloge sur la consommation dynamique et la consommation de fuite des processeurs sur base des simulations du banc d'essai 14
Figure 9 - Influence de la période du cycle de l'horloge sur la surface des processeurs sur base des simulations du banc d'essai 14
Figure 10 - Architecture du banc d'essai 19
Figure 11 - Plan d'adressage 20
Figure 12 - Lecture sur le bus AHB Lite 20
Figure 13 - Écriture sur le bus AHB Lite 20
Figure 14 - Lecture sur le bus PULP 20
Figure 15 - Écriture sur le bus PULP 20
Figure 16 - Flux global de développement d'un SoC [26] 23
Figure 17 - Schéma du circuit d'une porte logique 26
Figure 18 - Répartition des accès aux mémoires pour le Cortex-M0 avec multiplieur 32 bits en un cycle 32
Figure 19 - Efficacité des différents processeurs en fonction du rapport d'énergie d'accès des mémoires IMEM et DMEM avec l'énergie d'accès de la DMEM fixé à 1 pJ/accès 38
Figure 20 – Accès à la mémoire pour les différents processeurs 39
Figure 21 - Efficacité des différents processeurs en fonction du rapport d'énergie d'accès des mémoires DMEM et IMEM avec l'énergie d'accès de l'IMEM fixé à 1 pJ/accès 40
Figure 22 - Efficacité des différents processeurs en fonction du rapport d'énergie d'accès des mémoires DMEM et IMEM avec l'énergie d'accès de l'IMEM fixé à 1.6 pJ/accès 41
Figure 23 - Efficacité des différents processeurs pour un rapport DMEM/IMEM fixé à 5.12 42
Figure 24 - Efficacité des différents processeurs pour un rapport DMEM/IMEM fixé à 1 42

Tableau 1 - Comparaison des différents processeurs sur base d'articles les présentant. 16
Tableau 2 - Comparaison des performances entre les données du fournisseur et ce travail 29
Tableau 3 - Influence de l'optimisation du compilateur 29
Tableau 4 - Influence du multiplieur 32 bits en un cycle sur les performances 30
Tableau 5 - Comparaison de la consommation entre les données du fournisseur et ce travail 30
Tableau 6 - Influence du multiplieur 32 bits en un cycle sur la consommation du Cortex-M0 DS 31
Tableau 7 - Influence du multiplieur 32 bits en un cycle sur la surface du Cortex-M0 32
Tableau 8 - Influence du multiplieur 32 bits en un cycle sur les accès à la mémoire 33
Tableau 9 - Influence du niveau d'optimisation et des instructions compressées sur les performances du Zero-riscy 34
Tableau 10 - Comparaison du Cortex-M0 et de Zero-riscy 35
Tableau 11 - Comparaison de ce travail avec les données de l'état de l'art pour Zero-riscy 35

Tableau 12 - Influence du niveau d'optimisation et des instructions compressées sur les performances de RI5CY 36
Tableau 13 - Comparaison du Cortex-M0 et de RI5CY 36
Tableau 14 - Comparaison de ce travail avec les données de l'état de l'art pour RI5CY 37
Tableau 15 – Comparaison du Cortex-M0 et Zero-riscy pour des énergies d'accès aux mémoires de 1pJ/accès 39

8 Abréviations utilisées

ALU : Arithmetic and Logic Unit

ASIC : Application Specific Integrated Circuit

CRC : Cyclic Redundancy Check

DMEM : Data MEMory

DPS : Digital Signal Processor

GPIO : General Purpose Input Output

IMEM : Instruction MEMory

ISA : Instruction Set Architecture

MCU : Micro Controller Unit

SoC : System on Chip

WSN : Wireless Sensor Node

9 Bibliographie

[1] Sam Lucero, "IoT platforms: enabling the Internet of Things," ed, 2016.

[2] Philip Sparks, "The route to a trillion devices," in *The outlook for IoT investment to 2035*, ed: ARM Limited, 2017.

[3] D. Bol, "ELEC2570 Slide - DSP SoC architectures," ed, 2017.

[4] David Bol, Guerric de Streel, and Denis Flandre, "Can we connect trillions of IoT sensors in a sustainable way? A technology/circuit perspective (Invited)," presented at the 2015 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S), 2015.

[5] Guenole Lallement, Fady Abouzeid, Martin Cochet, Jean-Marc Daveau, Philippe Roche, and Jean-Luc Autran, "A 2.7 pJ/cycle 16 MHz, 0.7 µW Deep Sleep Power ARM Cortex-M0+ Core SoC in 28 nm FD-SOI," *IEEE Journal of Solid-State Circuits,* vol. 53, no. 7, pp. 2088-2100, 2018.

[6] Taoufik Bouguera, Jean-François Diouris, Jean-Jacques Chaillout, Randa Jaouadi, and Guillaume Andrieux, "Energy Consumption Model for Sensor Nodes Based on LoRa and LoRaWAN," *Sensors,* vol. 18, no. 7, 2018.

[7] ARM Limited. *Cortex-M0 – Arm Developer*. Available: https://developer.arm.com/ip-products/processors/cortex-m/cortex-m0

[8] Alan R. Weiss, "Dhrystone Benchmark: History, Analysis, Scores and Recommendations," ed.

[9] P. Davide Schiavone *et al.*, "Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications," in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2017, pp. 1-8.

[10] Lee Yunsup, Ou Albert, and Maryar Albert, "Z-scale: Tiny 32-bit RISC-V Systems," presented at the 2nd RISC-V Workshop, 2015.

[11] David Bol *et al.*, "19.6 A 40-to-80MHz Sub-4µW/MHz ULV Cortex-M0 MCU SoC in 28nm FDSOI With Dual-Loop Adaptive Back-Bias Generator for 20µs Wake-Up From Deep Fully Retentive Sleep Mode," presented at the 2019 IEEE International Solid- State Circuits Conference - (ISSCC), 2019.

[12] David Money Harris and Sarah L. Harris, *Digital Design and Computer Architecture*, 2nd Edition ed. 2012.

[13] D. Bol, "ELEC2570 Slide - Design clocking & timing closure," ed, 2017.

[14] D. Bol, R. Ambroise, D. Flandre, and J. D. Legat, "Interests and Limitations of Technology Scaling for Subthreshold Logic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 17, no. 10, pp. 1508-1519, 2009.

[15] R.I.M.P. Meijer, "Body bias aware digital design : a design strategy for area- and performance-efficient CMOS integrated circuits," Department of Electrical Engineering, Technische Universiteit Eindhoven, Eindhoven, 2011.

[16] Roel Uytterhoeven and Wim Dehaene, "A sub 10 pJ/Cycle Over a 2 to 200 MHz Performance Range RISC- V Microprocessor in 28 nm FDSOI," presented at the ESSCIRC 2018 - IEEE 44th European Solid State Circuits Conference (ESSCIRC), 2018. Available: https://ieeexplore.ieee.org/document/8494259/

[17] Hans Reyserhove and Wim Dehaene, "Ultra-Low Voltage Microcontrollers," in *Efficient Design of Variation-Resilient Ultra-Low Energy Digital Processors*Cham: Springer International Publishing, 2019, pp. 87-126.

[18] Michael Gautschi *et al.*, "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 25, no. 10, pp. 2700-2713, 2017.

[19] ARM Limited. *Cortex-M4 – Arm Developer*. Available: https://developer.arm.com/ip-products/processors/cortex-m/cortex-m4

[20] ARM Limited. *Cortex-M33 – Arm Developer*. Available: https://developer.arm.com/ip-products/processors/cortex-m/cortex-m33

[21] PULP Platform. (2018). *zero-riscy*. Available: https://github.com/pulp-platform/zero-riscy
[22] PULP Platform. (2019). *riscy*. Available: https://github.com/pulp-platform/riscv
[23] Klaus-Dietrich Kramer, Thomas Stolze, and Alexander Oppelt, "Microprocessor Benchmarks - A Detailed Look at Techniques, Problems and Solutions," presented at the 2011 21st International Conference on Systems Engineering, 2011.
[24] EEMBC. (2018). *CPU Benchmark – MCU Benchmark – CoreMark – EEMBC Embedded Microprocessor Benchmark Consortium*. Available: https://www.eembc.org/coremark
[25] ARM Limited, "AMBA® 3 AHB-Lite Protocol," ed, 2006.
[26] D. Bol, "ELEC2570 Slide - Logic synthesis and robust HDL," ed, 2017.
[27] David Bol, "Power and energy consumption of digital circuits," 2008.
[28] ARM Limited. (2010). *ARM Compiler toolchain*. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.swdev.coretools/index.html
[29] Inc. Free Software Foundation. (2019). *Using the GNU Compiler Collection (GCC): Optimize Options*. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
[30] PULP Platform. (2018). *PULPino*. Available: https://github.com/pulp-platform/pulpino
[31] A. Waterman and K. Asanovic, Eds. *The RISC-V Instruction Set Manual* (Volume I: User-Level ISA). RISC-V Foundation, 2017.
[32] Thomas Haine, "Large array-based circuits in ULV SoCs: design and statistical assessment of SRAMs and CMOS imagers," ed, 2018.

10 Annexes

10.1 Code source du banc d'essai

Le code source du banc d'essai ce trouve à l'adresse suivante :
https://github.com/DamienDeprez/Test-bench-Damien-Deprez

Dans le dossier *hardware*, nous retrouvons un dossier par processeur qui contient le code source du banc d'essai, les scripts pour exécuter la simulation comportementale et la synthèse du processeur. Dans le dossier de chaque processeur, nous avons la structure suivante :

- **sim_behav** : Ce dossier contient les scripts nécessaires pour lancer la simulation comportementale. Il y a un script qui permet de générer le fichier d'annotation de l'activité (VCD) et un autre qui effectue la simulation comportementale sans générer le fichier d'annotation de l'activité.
- **src** : Ce dossier contient l'ensemble du code source du banc d'essai. Il contient également le contenu des mémoires obtenu à la fin de la compilation.
- **synth** : Ce dossier contient les scripts nécessaires pour lancer la synthèse.

Dans le dossier *software*, nous retrouvons un dossier par processeur qui contient le code source adapté du *benchmark* du processeur. Pour Zero-risy et RI5CY, ce dossier contient également un Makefile pour compiler ce code source. Pour le Cortex-M0, ce dossier contient le projet pour Keil µVision qui est l'outil utilisé pour effectuer la compilation.

Dans le dossier *librairies*, nous retrouvons le fichier des librairies utilisées lors de la synthèse.

Dans le dossier *analyse*, nous retrouvons les scripts Python permettant d'analyser les différents accès à la mémoire.

10.2 Tableau de comparaison global des résultats des simulations

	Cortex –M0	Zero-riscy (O3C)	Zero-riscy (O3)	RI5CY (O3C)	RI5CY (O3)
Performances (CoreMarkI/µW)	1.98	2.33	2.43	3.08	3.14
Consommation dynamique (µW)	18.6	34.89	34.89	70.08	70.08
Consommation de fuite (µW)	4.182	4.4	4.4	15.71	15.71
Consommation totale du CPU (µW/MHz)	0.285	0.491	0.491	1.07	1.07
Surface (kGe)	20.835	21.399	21.399	75.699	75.699
Nombre de lectures dans l'IMEM	224 723	300 782	258 549	311 189	317 772
Nombre de lectures dans la DMEM	66 121	56 868	39 947	56 519	56 519
Nombre d'écritures dans la DMEM	21 948	12 730	8 666	13 122	13 122
Taille du code (word)	3003	6193	7387	5859	6919
Niveau d'optimisation	3	3	3	3	3
Efficacité (CoreMark/µW)	6.95	4.74	4.95	2.87	2.93
Efficacité tenant compte de la consommation des mémoires (CoreMark/µW)	0.82	0.8	0.99	0.71	0.71
Options	Multiplieur	Instructions compressés	Aucune	Instructions compressées	Aucune

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**