UNIVERSITÉ DE SHERBROOKE
Faculté de génie
Département de génie électrique et de génie informatique

RAPPORT APP4

Architecture des ordinateurs
GIF310

Présenté à
Marc-André Tétrault

Présenté par
Pascal-Emmanuel Lachance – LACP3102
Yan Ha Routhier-Chevrier - ROUY2404

Site web S4 – Sherbrooke – 30 juin 2022

Table des matières

1. PERFORMANCE DE RÉFÉRENCE

UNICYCLE & SRAM

Pour le calcul de la performance d'un processeur unicycle avec accès de mémoire instantané, le calcul du temps d'exécution est très simple, il ne faut que compter les instructions exécutées, en faisant attention aux boucles. Une légère réécriture du code de référence a été réalisée pour étendre des pseudo-instructions et optimiser les lectures mémoire.

On compte 14 instructions dans la boucle interne, qui se répète 4 fois. Il ne faut pas oublier de compter la comparaison finale de la boucle, qui nous renvoie à la sortie de cette dernière, mais qui prend une instruction supplémentaire.

La boucle externe a 8 instructions, en plus des 57 instructions de la boucle interne, et se répète elle aussi 4 fois, et a une instruction supplémentaire à compter pour la sortie de la boucle.

Il faut ensuite rajouter les instructions entourant les boucles, celles du main (qui comprennent des instructions *la* qui sont des pseudo-instructions prenant 2 cycles d'horloges) et les instructions de fin de programme, ce qui nous ramène à un total de 271 instructions.

Avec 271 instructions, et une fréquence d'horloge de 25ns, notre code prendrait 6.775µs à s'exécuter.

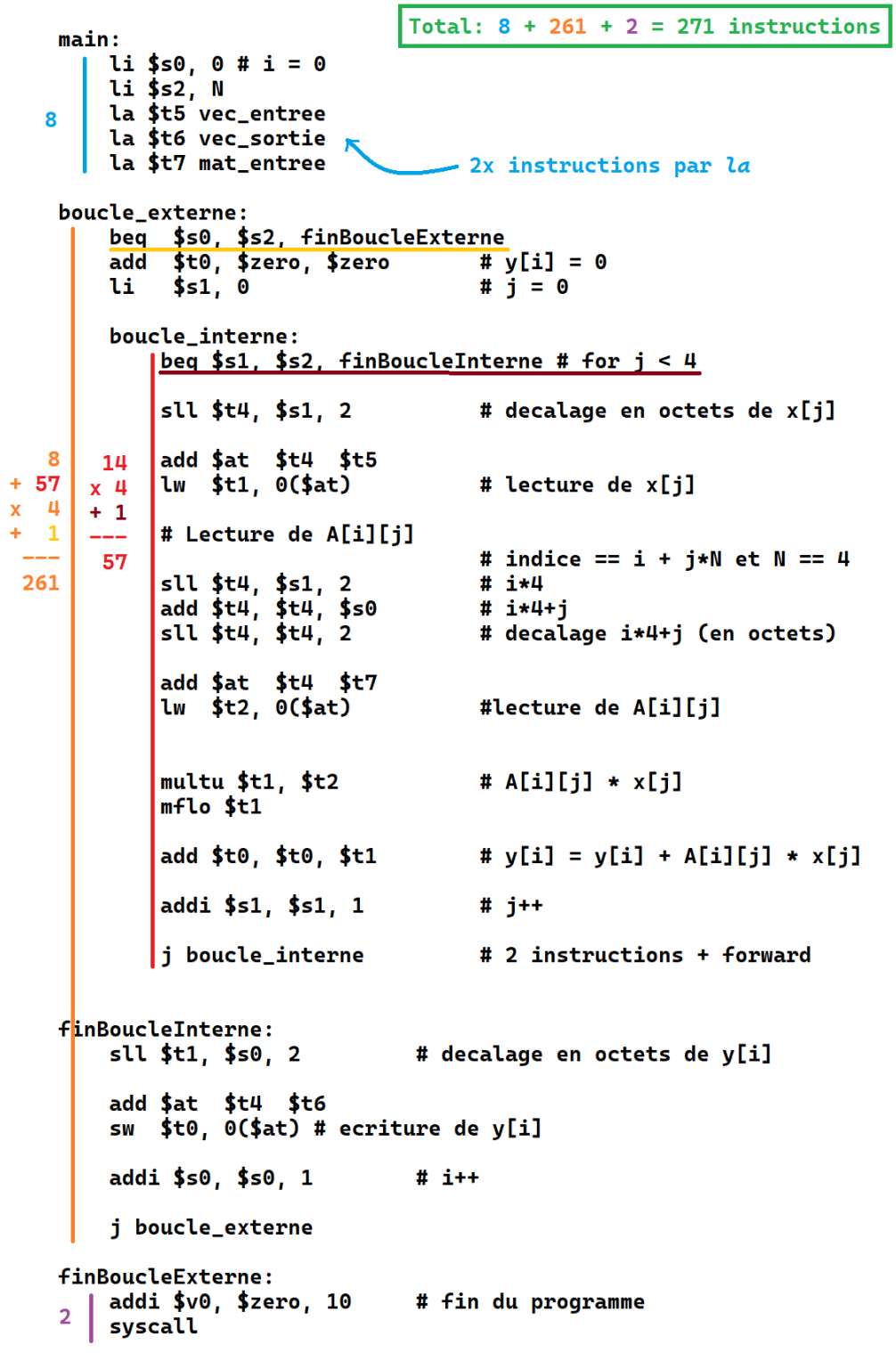

Figure 1 - Calculs d'instructions pour unicycle SRAM

PIPELINE & SRAM

Pour le calcul du temps d'exécution du processeur avec architecture *pipeline*, le même code de référence a été utilisé, avec des commentaires pertinents aux instructions *nop* ajoutées par le module de détection des aléas, qui insère automatiquement ces instructions aux endroits pertinents; après des sauts, après un *syscall*, ou après la lecture à partir de la mémoire de données. Nos calculs considèrent qu'en plus de l'unité de détection d'aléas, une unité d'avancement est présente pour permettre d'utiliser immédiatement à l'entrée de l'ALU la sortie de l'ALU de l'opération précédente.

Aux 14 instructions de la boucle interne se rajoutent 5 instructions *nop*, 2 provenant de la lecture mémoire (de *x[j]* et de *A[i][j]* respectivement), et 3 provenant du saut vers le *beq* de la boucle interne. Un pipeline plus avancé pourrait détecter l'adresse des instructions *j* et précharger les instructions pertinentes, plutôt que de *flush* les instructions préchargées inutiles et d'insérer des bulles à la place, mais notre implémentation ne va pas jusque-là.
Il faut aussi prendre en compte les 3 *nop* ajoutées après le saut du *beq* à la sortie de la boucle, ce qui amène notre total d'instructions de la boucle interne à 80.

Pour la boucle externe, le même calcul se fait, avec les 8 instructions de base auxquelles s'ajoutent cette fois-ci seulement 3 *nop*, et les 80 instructions de la boucle internes, le tout se répétant 4 fois et auquel on rajoute des *nop* après le dernier *beq* de comparaison, pour 368 instructions pour la boucle extérieure.

On rajoute à ce total 8 instructions pour le début du *main*, et puis 6 instructions pour la fin de la boucle externe, y compris 4 instructions *nop* après le *syscall*, ce qui nous donne un grand total de 382 instructions.

Un total de 382 instructions semble être beaucoup, et une régression par rapport à l'architecture unicycle, mais il y a un élément important à considérer : L'architecture *pipeline* permet de rouler à une fréquence plus élevée qu'une architecture unicycle équivalente.

Il est possible de calculer le nombre de coups d'horloges requis pour notre système *pipeline* a 5 couches; il nous faudrait 382 coups d'horloges + 4 autres coups, pour un total de 386 coups d'horloges requis pour rouler notre programme.

Avec 10ns par cycle d'horloge, le programme roulant dans l'architecture *pipeline* prendrait 3.86µs à s'exécuter, ce qui est environ 1.75x plus rapide que son équivalent unicycle.

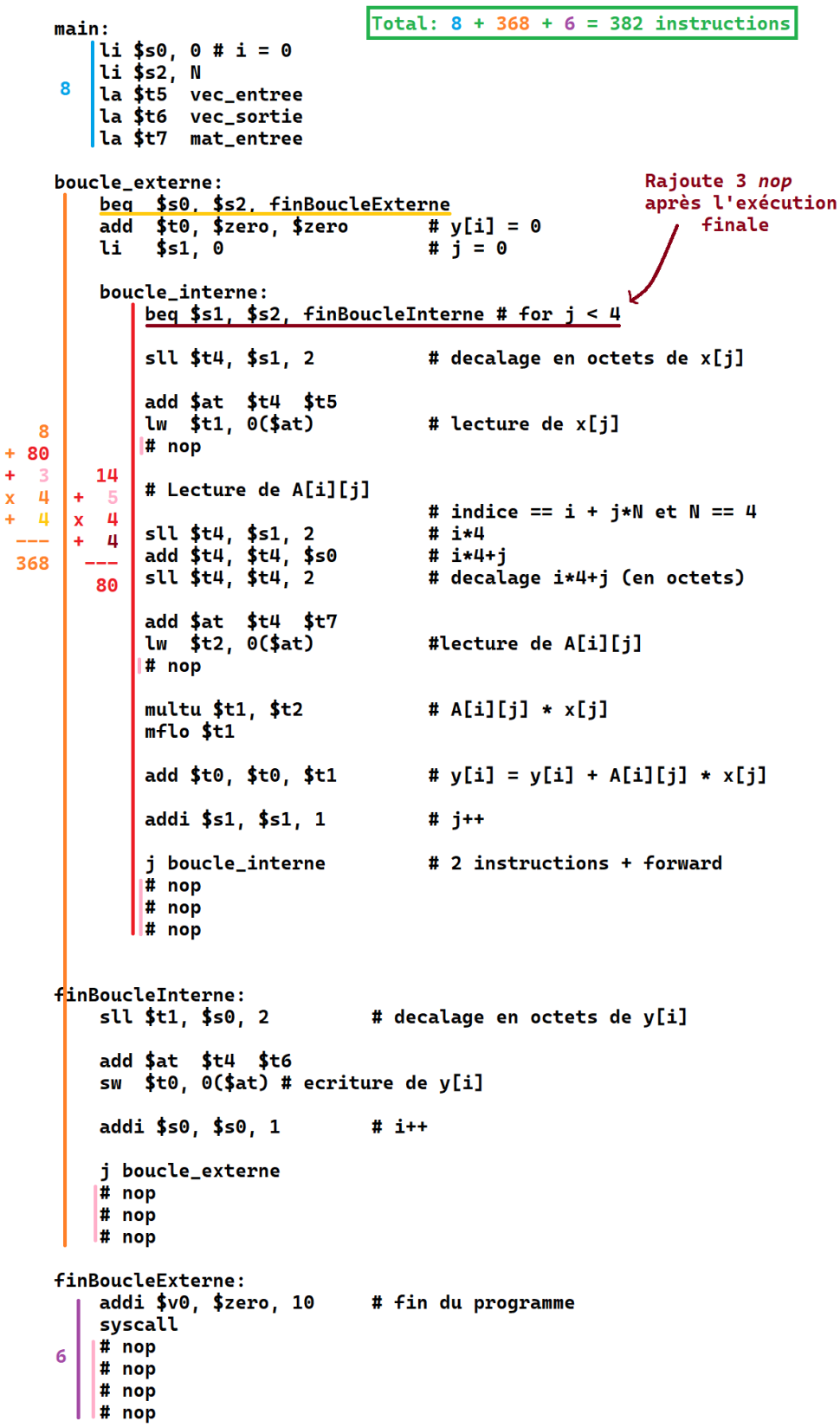

Figure 2 - Calculs instructions pour pipeline SRAM

UNICYCLE & DRAM

Changer la mémoire d'une SRAM à une DRAM a un effet négatif important sur la vitesse du programme, car un délai est rajouté pour toutes les instructions et les accès à la mémoire de données. La mémoire étant partagée entre les données et les instructions, il est possible de simplifier les calculs, en comptant tous les délais introduit par les accès mémoires comme une instruction *nop*, les deux ayant le même coût additionnel de 10 cycles d'instructions.
Les calculs sont plus simples à réaliser dans le cas d'une architecture unicycle, mais les cycles d'instructions sont beaucoup plus longs, et une architecture *pipeline* ayant des délais d'accès aux instructions plus long que son nombre de couches n'a plus besoin d'unités d'avancement ou de détection d'aléas, car 90% des opérations sont déjà des *nop*.

La stratégie de considérer les délais supplémentaires d'accès à la mémoire de donnée comme des *nop* rajoute 2 instructions dans la boucle intérieure, et une autre instruction dans la boucle extérieure, ce qui donne un total de 36 « instructions » supplémentaires dans le cas de la mémoire DRAM, soit 307.
Mais ce n'est pas tout, en effet, chacune de ces instructions prend 10 cycles d'horloges à exécuter, plutôt qu'une; donc il faut calculer 250ns par instructions, ce qui nous donne un temps d'exécution de 76.750µs, un temps atroce.

Si seulement c'était possible d'avoir une petite mémoire SRAM qui sert de tampon d'accès pour une plus grosse mémoire DRAM…

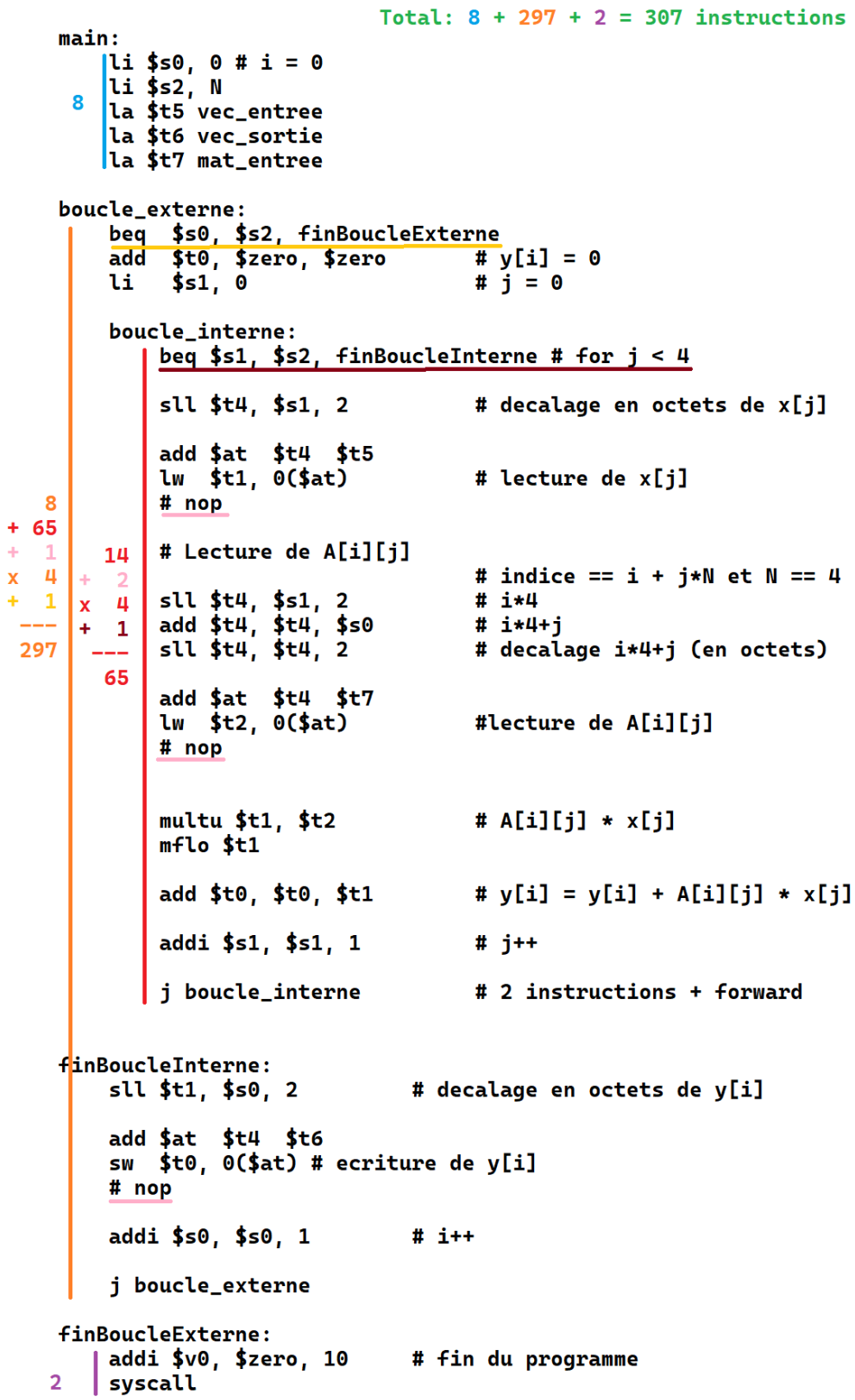

Figure 3 - Calculs instructions pour unicycle DRAM

UNICYCLE & CACHE

En effet utiliser la DRAM pour le programme engendre un effet négatif, par contre ajouter l'utilisation de la cache engendre un effet positif. Car à la place de toujours consulter la DRAM, le CPU peut consulter la cache qui réduira grandement le temps d'exécution du programme si les données s'y retrouvent déjà. Démontré dans la deuxième section (la cache), a la place d'avoir 36 instructions supplémentaires dû à l'accès à la mémoire, l'ajout de la cache le réduit à un total de 6 instructions supplémentaire, au lieu de 36, donc 277 instructions. Avec 250ns par instruction, le programme s'exécute en 69.25µs, lequel est une amélioration de 11%.

2. CACHE

Dans le but de rendre le programme efficace avec l'implémentation de la cache, nous avons déterminé qu'il fallait rassembler 4 mots de 32 bits chacun ensemble soit 128 bits pour la manipulation des données, donc une cache ayant des blocs de 128 bits. Au début de la boucle interne du programme, suite à une tentative manquée, à cause de la localisation spatiale des données et de la façon dont le programme est conçu, les 3 mots de cette suite seront trouvés avec succès. Par contre, il faut prévoir que l'algorithme utilise d'autres données à plusieurs endroits dans la mémoire, avant de poursuivre avec la suite de métriques, en d'autres mots il faut de plus choisir la façon dont les blocs mémoires seront gérés lorsque toutes les espaces disponibles dans la cache seront utilisés, afin de maximiser le nombre de *hits*. Pour ce cas présent, il sera préférable d'utiliser une cache associative, qui lorsque la cache est pleine et qu'un *miss* survient, la cache utilisera le dernier bloc mémoire modifié pour rentrer la nouvelle donnée. Au départ du programme, avec une cache réinitialisée, le programme essayera d'accéder au premier mot de vec_entree, qui causera le premier *miss* et chargera dans la cache les 4 mots de ce dernier. Ils resteront accessibles tout au long de la vie du programme. Par la suite, l'algorithme tentera d'accéder au premier mot de mat_entree, un autre *miss* s'en suivra (2 total). vec_sortie s'ensuit avec un *miss* (3 total), par contre les 4 mots seront accessibles jusqu'à la fin du programme. C'est le 4e espace qui sera constamment écrasé pour tous les prochains *miss*, soit lorsque le programme essaie d'accéder aux 4 prochains mots dans la table mat_entree, donc il y aura 3 autres miss (6 total). Donc une pénalité de 60 coups d'horloge tout au long du programme.

Pour résumer, l'information de vec_entree demeurera toujours disponible dans le premier bloc de la cache, et l'information de vec_sortie sera également toujours disponible dans le 3e bloc de la cache. Par contre, l'information de mat_entree, qui change à chaque itération de la boucle externe, se retrouvera constamment écrasé dans le 4e bloc de la cache.

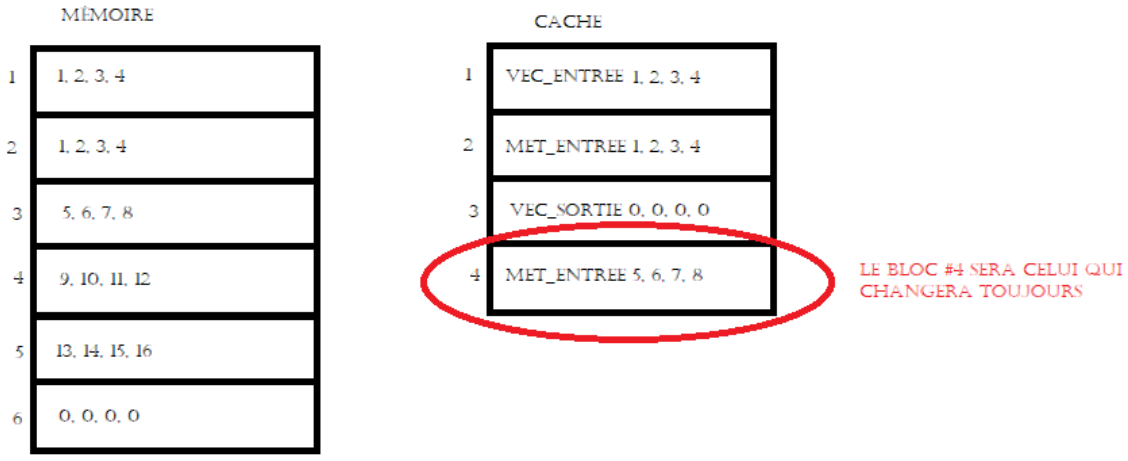

Figure 4 - Représentation de la cache de données

Configuration de la cache	
Spécification	**Valeur**
Taille de la cache (bytes)	64
Taille des blocs (bytes)	16
Nombre de bloc	4
Nombre de mots par bloc	4
Type de de cache	Pleinement associative – Dernière écrasée

3. EXTENSIONS ET MODIFICATION DE L'ALGORITHME

L'implémentation du devoir rajoute certains éléments coûteux, tels que l'utilisation de la pile, ainsi qu'une branche supplémentaire dans la boucle interne. Le chemin le plus long a été pris en considération pour le calcul de la performance de cette implémentation.

La boucle de l'acs contient 2 branches qui sont des pseudo-instructions et prennent 2 cycles d'horloges à exécuter. Le chemin le plus long au travers de la 2e branche est considéré toujours pris pour considérer le pire scénario. La pseudo-branche gérant la boucle coûte également 2 instructions supplémentaires pour la dernière comparaison pour sortir de la boucle. À la boucle principale de l'acs se rajoute un peu de logique pour gérer la boucle et le retour de fonction, ce qui donne 57 instructions.

Dans la fonction du calcul des survivants, une boucle externe contient également une pseudo-branche, ainsi que la pseudo-instruction *subiu*, utilisée pour allouer de l'espace sur la pile, qui se fait remplir avec les données des registres que l'on souhaite préserver, avant d'appeler la fonction acs. Un peu de logique de contrôle d'adresses et de contrôle de boucle se rajoute à la boucle externe, ce qui donne 324 instructions.

Dans la fonction *main*, les adresses des tableaux sont chargées comme paramètres de fonction avant d'appeler la fonction CalculSurvivant, pour un grand total de 333 instructions.
Avec un cycle d'horloge de 25ns, l'implémentation de l'algorithme du devoir prend 8.325µs.

Figure 5 - Calculs instructions devoir

L'optimisation par SIMD de l'algorithme est réalisée en permettant à certains registres, les registres *$s0* jusqu'à *$s7*, de tenir des informations vectorielles; des groupes de 4 mots de 32 bits. L'ALU a été étendu en rajoutant 3 copies de l'ALU et en élargissant le bus de données; ce qui permet la compatibilité entre toutes les instructions arithmétiques, qui se comportent toutes exactement comme normal et qui n'ont pas besoin d'adaptations. Des nouvelles instructions pour charger et sauvegarder des vecteurs à partir de la mémoire de données ont été créées, ainsi qu'une opération de déplacement conditionnel *movnv.* Finalement, une instruction vectorielle verticale a été implémentée, *sml*, qui permet de récupérer la valeur la plus petite parmi un vecteur de mots de 31 bits.

C'est cette dernière instruction qui est la clé de notre optimisation SIMD, et qui permet de combiner toutes les comparaisons et la sélection de l'élément le plus petit du tableau, en une seule instruction.

La boucle principale de l'acs fait donc uniquement 42 instructions, un total qui monte à 46 avec la logique de contrôle et de retour. La fonction *main* y rajoute 9 instructions, pour un total de 55 instructions, ce qui, avec une implémentation unicycle avec un temps d'horloge de 25ns, prend 1.375µs à s'exécuter, presque 5x plus vite que l'implémentation du code de référence. Le code pourrait être encore plus rapide en utilisant une instruction *beq* au lieu de la pseudo-instruction *bge,* ce qui nous sauverait 125ns. Un 20ns de plus pourrait également être cherché en intégrant à la fonction *main* le contenu de la fonction *acs.* Ces deux modifications n'ont pas été jugées pertinentes par l'équipe, qui a préféré se concentrer sur l'implémentation VHDL des extensions SIMD.

Un aspect important de l'implémentation est que le tableau de sortie est initialisé avec des valeurs maximales de 0xffffffff; toutes les valeurs calculées sont donc certainement plus petites ou égales à celles du tableau de sortie, ce qui permet d'écraser ces dernières sans avoir à effectuer de comparaison (ce qui aurait nécessité 2 instructions supplémentaires dans la boucle de l'asc).

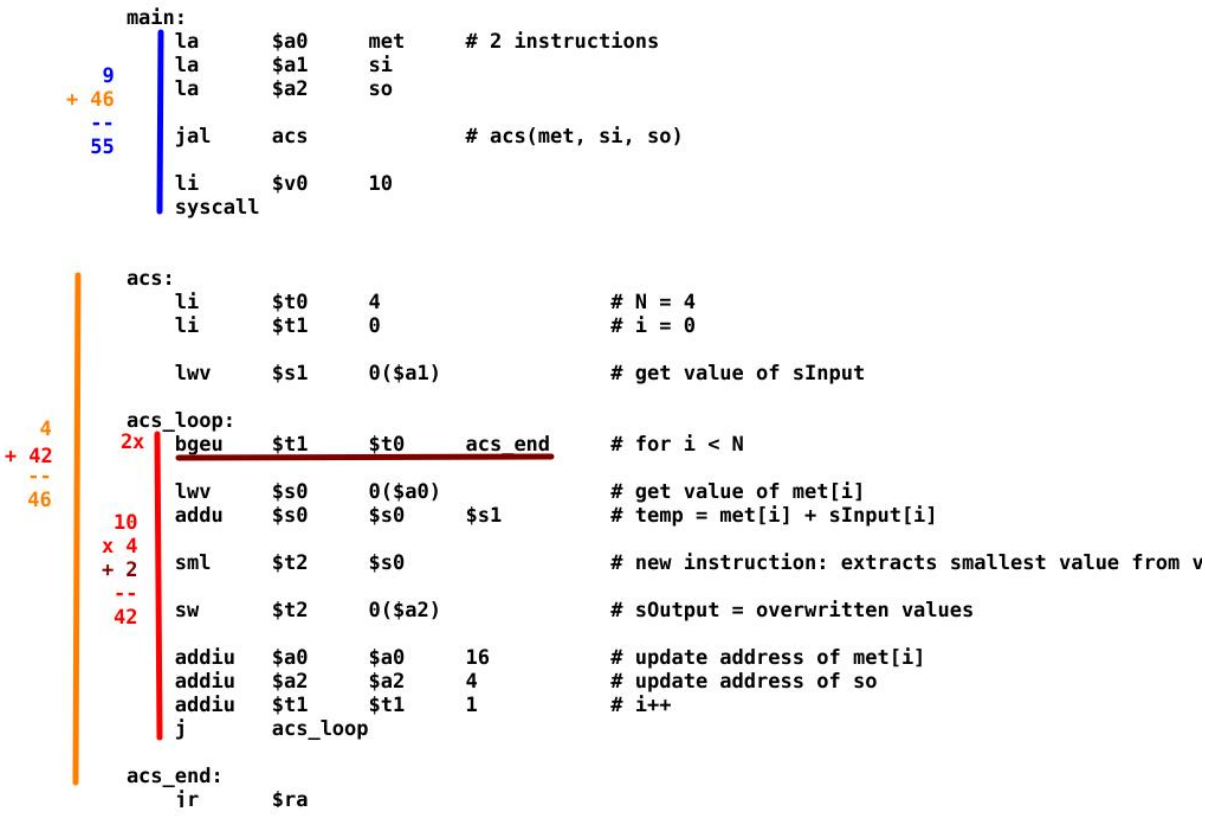

Figure 6 - Calculs instruction implémentation SIMD

4. PROCESSEURS

Pour ce qui est modification apportée du processeur, il a fallu agrandir les registres $s0 à $s7 à 128 bits lesquelles contiendra 4 mots de 32 bits chacun. C'est pour cette raison que nous voyons à la sortie des registres un bus de 128 bits sortant de Read data 1, Read data 2 et Read WrReg. Au control, il a fallu ajouter un signal afin d'indiquer si l'instruction est vectorielle au data memory, qui lira différemment les données dans la mémoire, soit en 4 mots de 32 bits ou bien 1 seul mot. Il y a 4 ALU, qui sont utilisés pour les opérations vectorielles. Le registres à de plus une sortie de plus Read WrReg lequel est la valeur associée au registre dans lequel il sera écrit le résultat de l'opération. Ce dernier rentre dans un multiplexeur et sera choisi, si jamais l'opération est un movnv et que les valeurs dans le vecteur pointé

ne sont pas égales à zéro. Pour terminer, il y a un bus de 128 bits sortant à la fin du chemin, après le *DataMemory*, pour être inscrit dans les registres.

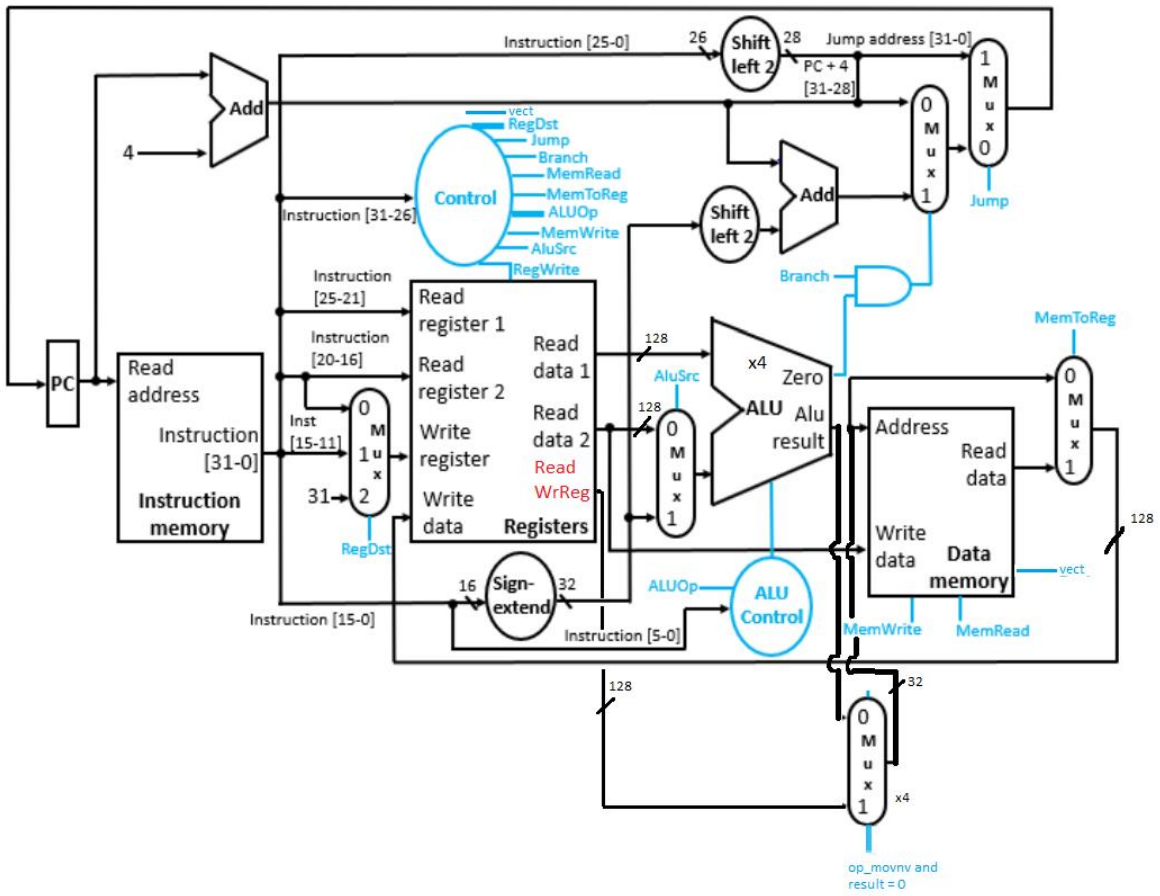

Figure 7 - Architecture modifiée du processeur

5. INTÉGRATION

L'implémentation optimale de l'algorithme profiterai des accélérations de calculs fournies par les extensions SIMD, ainsi que des accélérations aux accès mémoires fournis par une cache de 4 blocs de 128 bits. Cependant, un facteur ralentissant important est que, selon l'énoncé, la mémoire des instructions est également stockée en DRAM. Avec les ralentissements d'accès aux instructions forcée par ces dernières étant stockés en DRAM, une implémentation de type *pipeline* perd beaucoup de ses avantages.

Une manière possible de contourner le problème, avec une implémentation unicycle serait d'implémenter une "cache", à part de la première, stockant par exemple les 8 prochaines instructions, et n'ayant en moyenne qu'un *miss* à tous les 4 instructions et chaque branche, au lieu d'à chaque instruction.

Rajouter une cache avec les spécifications de la section 2 nous permet également de rendre beaucoup plus rapide nos accès à la mémoire de données. Un premier *cache miss* se produit lors du chargement initial du tableau *sInput*, puis un autre se produit à l'écriture des données vers le tableau *sOutput*. 3 autres *cache miss* se produisent pour la lecture des blocs du tableau de métriques, pour un total de 6 *miss* pour l'accès aux données. Ces 6 *miss* ont un coût de 10 cycles d'horloge.

Au niveau de la "cache" d'instructions, qui est plus proche d'un simple tampon d'instruction que d'une cache; 8 instructions consécutives y sont chargées à la fois, si le programme cherche à accéder à une adresse en dehors des 8 chargées, l'adresse demandée sera chargée, avec les 7 instructions qui la suivent. Il s'agit d'une approche très simple et naïve, mais qui fonctionne bien dans notre cas. La différence entre les accès à la cache de données et la cache d'instruction peuvent soit être gérés à même les modules de gestion de la mémoire, ou avec un masque au niveau de l'adresse demandée.
Si la cache d'instruction peut contenir 8 instructions, un premier *miss* se produit au chargement du programme, puis un autre au *jal* vers la fonction acs. Un autre *cache miss* se produit au niveau de l'instruction *sw*, puis un autre au *j* vers de début de la boucle de l'acs. Pour les 3 prochaines boucles, 2 *cache miss* se produisent à chaque boucle, une au niveau de l'avant-dernière addition, et l'autre au *j* vers le début de la boucle, 2 instructions plus loin. À la fin du programme, 2 *miss* supplémentaires se produisent pour aller vers la sortie de la fonction, et pour charger la fonction *main*. Au total, 12 *miss* se produiraient au niveau des instructions, ce qui monterait un total des accès mémoires de 18, à 10 cycles de 25ns, un coût de 4.5µs.

Les accès mémoires gérés, en utilisant les extensions SIMD dans une architecture unicycle, le temps calculé à la section 3 de 1.375µs pour les calculs serait gardé, ce qui amène notre total à 5.875µs, ce qui demeure plus rapide que le code de base unicycle avec SRAM.

Bien entendu, notre nouvelle "cache" d'instruction permet à nouveau à une architecture de type *pipeline* d'être pertinente, car elle peut charger les instructions de la cache au fur et à mesure. Un aspect intéressant de l'implémentation d'un pipeline avec notre modèle de "cache" d'instruction est que l'insertion de bulles de *nop* lors d'aléas de contrôle n'est plus nécessaire, car tous les aléas de contrôle coïncident avec des *cache miss*, qui prennent déjà 10 cycles d'instructions à s'exécuter. Les seules bulles à insérer sont celles nécessaire pour les *lw*, qui ne sont pas complètement couverts par l'unité d'avancement, et qui nécessitent un *nop.* Les *nop* générés par l'unité de détection d'aléas ne comptent pas non plus dans les instructions prisent dans la "cache" d'instruction, ce qui veut dire que les *cache miss* calculés 3 paragraphes plus haut pour une architecture unicycle demeurent à la même position. L'instruction *lwv* n'est exécutée que 5 fois au total dans la durée du programme, on peut donc compter que notre total initial de 55 instructions est maintenant de 60 instructions. Dans un pipeline à 5 étages, cela consiste en 64 coups d'horloges.

Par contre, l'utilisation d'une architecture *pipeline* permet d'utiliser une vitesse d'horloge plus élevée que dans une architecture unicycle, spécifiquement 10ns au lieu de 25ns. L'impact le plus important que cette nouvelle cadence amène est sur le temps d'accès à la DRAM, qui demeure de 10 cycles d'horloge, mais à une fréquence de plus élevée, ne coûtant maintenant que 1.8µs (au lieu des 4.5µs en unicycle). Le reste du programme voit également une accélération, se produisant en 640ns, pour un temps d'exécution total de 2.44µs, environ 3x plus vite que le code de référence unicycle avec SRAM.

Pour résumer, l'implémentation idéale avec DRAM serait une implémentation pipeline 5 étages, avec des extensions SIMD rajoutant des opérations vectorielles sur 4 mots de 32 bits, notamment *sml*, une

instruction vectorielle verticale. Pour minimiser les impacts d'accès à la mémoire DRAM, deux modèles de caches différentes seraient utilisés; une cache simple agissant comme buffer de 8 instructions, ainsi qu'une cache pleinement associative pouvant stocker 4 blocs de 128 bits permettant d'accélérer les accès aux données. Cette configuration, avec un code fortement optimisé, permet un temps d'exécution théorique total de 2.44µs.