

INDEX

DOC → [Config] [MIPS U] [MIPS K] [markdown] [CR.md]
 COURS → [1] (+code) (+outils) [2] [3] [4] [5] [6] [7] [8] [9]
 TME → [1] [2] [3] [4] [5] [6] [7] [8] [9]
 CODE → [gcc + soc] [1] [2] [3] [4] [5] [6] [7] [8] [9]

A. Questions de cours
 B. Exercices
 B.1. Remplissage des cases de cache
 B.2. Cas de collision de lignes
 C. Travaux pratiques
 C.1. Calcul du taux de MISS dans le cache d'instructions
 C.2. Analyse de trace
 C.3. Mesure du taux de MISS
 C.4. Optimisation du code pour minimiser le taux de MISS

4 - Cache L1 à correspondance directe - principes

Vous pouvez lire les slides de cours pour voir les détails, mais voici le résumé des principes en quelques lignes.

Dans un ordinateur (un SoC), l'accès à la mémoire principale par le, ou les, processeur(s) est souvent coûteux en nombre de cycles d'horloge, car la mémoire principale est souvent située en dehors du SoC. Pour améliorer les performances, on place des *mémoires*, petites, mais très rapides, entre le processeur et la mémoire.

Ces mémoires, nommées *cache*, contiennent les données ou les instructions récemment utilisées par le processeur. Les *caches* ne contiennent que des copies de données ou d'instructions de la mémoire principale. L'efficacité des *caches* est liée aux propriétés de localité spatiale et temporelle des applications logicielles. En effet, il est fort probable que, après avoir accédé à une certaine adresse mémoire x , l'application accède à nouveau à l'adresse x dans un futur proche (localité temporelle) ou qu'elle accède à une autre adresse mémoire proche de x (localité spatiale).

Un cache lit la mémoire par *ligne de cache*. Une *ligne de cache* est un *segment d'adresse aligné* dans l'espace d'adressage dont la taille est une puissance de 2 en mots. Une ligne de cache peut faire 2 mots, 4 mots, 8 mots voire 16 mots, mais rarement plus (parce que la propriété de localité spatiale est moins grande si on est trop loin l'adresse lue). Un *segment d'adresse aligné* signifie que l'adresse du premier octet du segment est un multiple de la taille du segment. Pour le MIPS, les mots font 4 octets, si les lignes de cache font 2 mots alors les lignes commencent à des adresses multiples de 8. Les lignes sont numérotées. Pour savoir à quelle ligne appartient un octet quelconque de l'espace d'adressage de la mémoire, il suffit de diviser l'adresse de l'octet par la taille d'une ligne. À titre d'exemple, pour des lignes de 8 octets, les octets de 0 à 7 sont dans la ligne n°0, les octets de 8 à 15 sont dans la ligne 1, etc.

Le cache lit des **lignes de caches** et les range dans des **cases de caches**. Attention *case de cache* et *ligne de cache* ne sont pas synonymes. Une ligne de cache est un contenu alors qu'une case de cache est un contenant. À chaque fois qu'un *composant-cache* lit une ligne de cache en mémoire, il doit la ranger dans l'une de ces cases. Le choix le plus simple est que ce soit le numéro de la ligne qui définisse directement le numéro de la case. Ce choix est celui des caches à correspondance directe : un numéro de ligne impose un numéro de case. Le nombre de cases d'un cache est toujours une puissance de 2 (2 cases, 4 cases, 8 cases, [...], 1024 cases, etc.), ainsi un numéro de cases est toujours défini par un nombre entier de bits (1 bit s'il y a 2 cases, 2 bits s'il y a 4 cases, 3 bits s'il y a 8 cases, [...], 10 bits s'il y a 1024 cases, etc.).

Nous allons étudier les mémoires cache à *correspondance directe* pour lesquels le numéro de la case de cache est simplement défini par le n bits de poids faible du numéro de ligne de cache, avec 2^n = nombre de cases de cache.

Le but de cette première séance sur les caches est de répondre à trois questions :

- Comment les caches se remplissent en fonction de la position des instructions et des données en mémoire (leurs adresses) et en fonction de la taille des caches.
- Comment calculer le taux de miss du cache instruction lors de l'exécution d'une boucle d'instructions
- Comment calculer le taux de miss du cache de données de l'usage de donnée par les instructions de leur position en mémoire

A. Questions de cours

1. Que signifie bus système ?

- Le bus système, c'est le média de communication présent entre le processeur et l'ensemble des composants interne du SoC, c'est-à-dire la mémoire et les contrôleurs de périphériques.
- Le bus système achemine (route) les requêtes de lecture et d'écriture du processeur (et des contrôleurs de périphérique initiateurs) vers la mémoire ou les périphériques cibles et il achemine en retour les réponses des cibles vers les initiateurs.
- Le bus système reçoit des requêtes dans un espace d'adressage et il y a toujours au plus un composant pour gérer une adresse.

2. À quoi sert l'arbitre du bus système ?

- Lorsque le SoC contient plusieurs initiateurs, et donc plusieurs composants capables d'initier (démarrer) une requête en même temps, il faut gérer un arbitrage pour autoriser un seul des initiateurs d'envoyer sa requête.
- C'est le rôle de l'arbitre qui reçoit des demandes de propriété et qui donne des autorisations suivant une politique établie à l'avance, le plus souvent équitable, à tour de rôle (round robin).

3. Qu'est un contrôleur mémoire ?

- La mémoire, quand elle est externe, est un composant complexe qui utilise un protocole de communication spécifique (DDR, SRAM, SPI, et tant d'autres). Le but du contrôleur mémoire est de traduire les requêtes des initiateurs du bus système en requêtes compréhensibles par le composant de mémoire externe.

4. La "localité spatiale" et la "localité temporelle" sont deux propriétés des programmes, que représentent-elles ?

- La localité spatiale représente le fait que lorsqu'on accède à une donnée ou une instruction à une adresse X , la probabilité d'avoir une requête aux adresses voisines de X est grande.
- La localité temporelle représente le fait que lorsqu'on accède à une donnée ou une instruction à la date T , la probabilité d'avoir une requête à la même donnée ou à la même instruction est dans les cycles futurs est grande.

5. Où sont placés les caches ?

- Les caches de premier niveau sont placés entre les processeurs et le bus système.
- Le cache de deuxième niveau est placé entre le bus système et le contrôleur de mémoire externe.

6. Peut-on avoir plusieurs niveaux de cache ?

- Oui, on peut en avoir 1, 2, 3 voire 4.
- En fait, à chaque fois qu'une donnée se trouve sur un support lent, on peut ajouter un cache qui va accélérer les accès, à la condition que les propriétés de localité spatiale et temporelle soient présentes. Sinon on ne gagne rien.

7. Que signifient "caches séparés" et "cache unifié" ?

- Cela fait référence au mélange des instructions et des données.
- Pour le MIPS, il y a deux caches L1 par MIPS et ces caches L1 sont séparés, il y a un cache L1 pour les instructions et un cache L1 pour les données.
- Pour le MIPS, il n'y a qu'un seul cache L2 qui mélange les instructions et les données, on dit que le cache L2 est unifié.

8. Quelle est la répartition des types d'instructions dans un programme ?

- Évidemment, ça dépend des programmes, il s'agit d'ordre de grandeur pour les programmes "standards"
 - 40% d'instructions de type ALU (p. ex. `addu`)
 - 30% de branchement (rupture de séquence) (p. ex. `beq`, `jal`)
 - 20% de lecture de données (p. ex. `lw`)
 - 10% d'écriture de données (p. ex. `sw`)

9. Pourquoi les lectures sont bloquantes et les écritures sont non bloquantes ?

- Si un programme a besoin d'une donnée pour avancer, il doit attendre qu'elle arrive pour continuer.
- Si un programme a besoin d'écrire une donnée, il lui suffit de la poster dans un tampon d'écriture et il peut continuer son travail. Il n'a pas à attendre que la donnée soit écrite dans la mémoire. Si le tampon d'écriture est plein, là, il faut attendre.

10. Qu'est un taux de MISS et quels sont les taux normaux ?

- un taux de MISS est une probabilité que le cache n'ait pas la donnée ou l'instruction qu'on lui demande.
- Le taux de MISS dépend de la taille du cache et aussi du profil d'accès à la mémoire.
 - Pour les instructions, un bon taux de miss est de l'ordre de 2%, voire moins.
 - Pour les données, un bon taux de miss est de l'ordre de 5%, voire moins.
 - Il est assez normal que les taux de miss des caches instructions soient meilleurs que ceux des caches de données, parce que les localités spatiales et temporelles des programmes sont généralement meilleures (on reste dans des boucles d'instructions par exemple, qui font rester le processeur longtemps au même endroit).

11. Qu'est-ce qu'une "ligne de cache" ?

- Une ligne de cache est un segment d'adresse aligné en mémoire et d'une puissance de 2 mots (8, 16, 32 ou 64 mots).
- *aligné* signifie que l'adresse du premier octet est multiple de la taille de la ligne.
 - multiple de 8 pour les lignes de 2 mots (2*4 octets)
 - multiple de 16 pour les lignes de 4 mots (4*4 octets)
 - etc.

12. Qu'est-ce qu'une "case de cache" ?

- Une case de cache est une mémoire dans le cache pour stocker une ligne de cache.
- Évidemment, une case de cache a la même taille qu'une ligne de cache.

13. Que veut dire cache à correspondance directe ?

- La correspondance directe fait référence au choix de la case pour une ligne de cache.
- On dit que le cache est à correspondance directe si le choix de la case pour une ligne dépend directement du numéro de la ligne.
- Dit autrement, si vous avez une adresse dans l'espace d'adressage, et si vous connaissez le nombre de cases du cache, alors vous savez dans quelle case, la ligne dans laquelle se trouve l'adresse se rangée si le processeur lit cette adresse.
- Pour faciliter le calcul du numéro de la case, le nombre de cases du cache est une puissance de 2 (2, 4, 8, 16, ... cases). Ainsi, pour calculer le numéro de case, on utilise les bits de poids faible du numéro de ligne.

14. Que sont un numéro de ligne, un tag, un index et un offset de cache ?

- Si on voit l'espace d'adressage du MIPS comme un tableau de lignes, où une ligne est un segment d'adresses alignées sur une puissance de 2, alors la ligne numéro 0 est la première ligne, la ligne numéro 2 est la deuxième ligne, etc.
- Pour calculer le numéro de ligne d'une adresse quelconque, il suffit de diviser cette adresse par la taille d'une ligne.
- L'index désigne les bits de poids faibles d'un numéro de ligne qui définissent le numéro de case pour la ligne, dans le cas d'un cache à correspondance directe.
- et le tag désigne les bits restants du numéro de ligne quand on a retiré les bits d'index.

15. Qu'est-ce que le tampon d'écriture ?

- C'est une petite mémoire qui contient les requêtes d'écriture faites par le processeur à destination de la mémoire, plus généralement de l'espace d'adressage, mais le plus souvent pour écrire dans la mémoire.

16. Quel est le principal inconvénient des caches à correspondance directe ?

- Le fait qu'il n'y ait qu'une case unique pour chaque ligne a pour conséquence que si dans une boucle de programme, on utilise deux lignes avec le même numéro d'index et donc deux lignes utilisant la même case du cache, il y a une collision et on va perdre le bénéfice du cache.
- Dans le cas précédent, on va même perdre beaucoup, parce qu'on lit des données ou des lignes en mémoire pour rien.

B. Exercices

B.1. Remplissage des cases de cache

Vous avez le corrigé de l'exercice, mais nous vous conseillons vraiment de ne pas le regarder avant d'avoir vraiment essayé de répondre aux questions, parce que vous aurez ce genre d'exercice à l'examen et ce sera sans document.

Soit un processeur **MIPS32** associé à un cache d'instructions et à un cache de données séparés. Les deux caches ont une capacité de stockage de 32 octets et sont à *correspondance directe*. La largeur d'une ligne de cache est de 8 octets (soit 2 mots).

Rappel : toutes les adresses émises par le processeur sont des adresses octets, et les adresses sont codées sur 32 bits.

- Dites comment le contrôleur du cache interprète une adresse :
 - quel est le nombre de bits de l'index ?
 - quel est le nombre de bits du déplacement (*offset*) ?
 - quel est le nombre de bits de l'étiquette (*tag*) ?
- Nombre de bits de l'index : $\log_2 (\text{capacité du cache} / \text{largeur de la ligne}) = \log_2 (32/8) = 2$.
 - Nombre de bits du déplacement : $\log_2 (\text{nombre d'octets dans la ligne de cache}) = \log_2 (2*4) = 3$.
 - Nombre de bits de l'étiquette : $\text{taille totale de l'adresse} - \text{taille de l'index} - \text{taille de l'offset} = 32 - 2 - 3 = 27$.

Vous devez comprendre et connaître ces formules pour l'examen !

Considérons la séquence d'instructions suivante dont la première instruction est stockée à l'adresse **loop = 0x00000010** :

```
loop:
lw      $8, 0($16)
lw      $9, 4($16)
addu    $10, $8, $9
sw      $10, 512($16)
addiu   $16, $16, 8
bne     $16, $12, loop
```

- Pour chacune des instructions de cette séquence, donnez les valeurs (en base 2) du numéro de ligne, index et déplacement de l'adresse de l'instruction, en complétant la table suivante : (pour ne pas avoir à faire de calcul à la place du numéro de ligne, vous indiquerez l'adresse du premier octet de la ligne, que nous appellerons Adresse de ligne)

Instruction	Adresse de ligne	Index	Offset

Instruction	Adresse de ligne	Index	Déplacement
lw \$8, 0(\$16)	0x10	0b10	0b000
lw \$9, 4(\$16)	0x10	0b10	0b100
addu \$10, \$8, \$9	0x18	0b11	0b000
sw \$10, 512(\$16)	0x18	0b11	0b100
addiu \$16, \$16, 8	0x20	0b00	0b000
bne \$16, \$12, loop	0x20	0b00	0b100

- Complétez le tableau suivant en précisant si le chargement de l'instruction est un échec ou un succès.

Instruction	Échec ou succès

Instruction	Échec ou succès
lw \$8, 0(\$16)	échec
lw \$9, 4(\$16)	succès
addu \$10, \$8, \$9	échec
sw \$10, 512(\$16)	succès
addiu \$16, \$16, 8	échec
bne \$16, \$12, loop	succès

- Considérez une deuxième itération de la séquence d'instructions précédente (la condition de branchement **bne** est vérifiée et donc l'instruction située à l'adresse **0x00000010** est de nouveau exécutée). Modifiez le tableau précédent pour la 2^e itération de la boucle.

Instruction	Échec ou succès
lw \$8, 0(\$16)	succès
lw \$9, 4(\$16)	succès
addu \$10, \$8, \$9	succès
sw \$10, 512(\$16)	succès

addiu \$16, \$16, 8	succès
bne \$16, \$12, loop	succès

Considérons maintenant les lectures de données générées par la séquence d'instructions précédente. On suppose que le registre `$16` contient la valeur `0x00000110`, qui est l'adresse de base d'un tableau d'entiers `Tab[]` dont les valeurs ont été initialisées telles que `Tab[i] = i`. Au démarrage de la boucle, le cache de données est supposé vide (ce qui signifie que les 4 lignes sont marquées invalides).

- Complétez le tableau ci-dessous pour décrire le contenu du cache de données à la fin de la première itération de la boucle ? À la fin de la deuxième itération ? À la fin de la troisième ?

Validité	Adresse de ligne	Donnée 1	Donnée 0
0			
0			
0			
0			

`0x00000110 = 0...0 0001 0001 00002` : index = 2, déplacement = 0.

- À la fin de la première itération :

Validité	Adresse de ligne	Donnée 1	Donnée 0
0			
0			
1	0x110	0x00000001	0x00000000
0			

- À la fin de la deuxième itération :

Validité	Adresse de ligne	Donnée 1	Donnée 0
0			
0			
1	0x110	0x00000001	0x00000000
1	0x118	0x00000003	0x00000002

- À la fin de la troisième itération :

Validité	Adresse de ligne	Donnée 1	Donnée 0
1	0x120	0x00000005	0x00000004
0			
1	0x110	0x00000001	0x00000000
1	0x118	0x00000003	0x00000002

B.2. Cas de collision de lignes

On considère maintenant un processeur `MIPS32` possédant un cache de données à correspondance directe de 8 kibi octets organisé en lignes de 32 octets.

Soit deux tableaux de 4096 entiers (un entier équivaut à 32 bits), implantés en mémoire aux adresses suivantes :

- `X` : `0x00010000`
- `Y` : `0x00014000`
- Donnez les éléments des tableaux `X` et `Y` qui peuvent occuper le mot n°0 de la case n°3 du cache de données.

- Nombre de bits de l'index : \log_2 (taille du cache / taille de la ligne) = $\log_2 (8192/32) = 8$.
 - il y a 256 lignes dans le cache.
- Nombre de bits du déplacement : \log_2 (nombre d'octets dans ligne de cache) = $\log_2 (32) = 5$.
- Mot n°0 : déplacement = `00000`
- Case n°3 : index = `00000011`
- Dans le tableau `X` :
 - `0x10060` (`X[24]`)
 - `0x12060` (`X[2072]`)
- Dans le tableau `Y` :
 - `0x14060` (`Y[24]`)
 - `0x16060` (`Y[2072]`)

- Calculez le taux d'échecs dans le cache de données pour la boucle suivante (on suppose que la variable scalaire `S` est contenue dans un registre - donc jamais d'échec de cache lors de sa lecture) :

```
for (i = 0; i < 4096; i++) {
    S = S + X[i] + Y[i];
}
```

- Nombre d'accès : $2 * N$ (2 accès par itération).
- Nombre d'échecs : à chaque accès, car `X[i]` et `Y[i]` sont en compétition pour la même case du cache.
- Taux d'échecs = Nombre d'échecs / Nombre d'accès = 1.
- Si l'on suppose que le tableau `Y` est maintenant rangé à l'adresse `0x00014020`, calculez le nouveau taux d'échecs.
 - Nombre d'accès : $2 * N$
 - Nombre d'échecs : à chaque nouvelle ligne, c'est-à-dire tous les 8 accès = $2 * N / 8$
 - Taux d'échecs = Nombre d'échecs / Nombre d'accès = 0.125.

Le tableau `Y` a en fait été décalé d'une ligne de cache : on a fait du "padding" (bourrage).

C. Travaux pratiques

Ce TP a pour but l'observation (en simulation) du fonctionnement des mémoires caches, et des mouvements de données entre les caches et la mémoire principale, plus précisément l'étude du taux de miss de cache L1 en fonction du programme exécuté.

On a choisi des lignes de cache de 16 octets et des caches de très faible capacité : chaque cache (cache d'instructions et cache de données) possède une capacité de 128 octets (soit 8 cases, pouvant contenir chacune une ligne de cache de 16 octets). Les deux caches du processeur sont à *correspondance directe* (sous-entendu correspondance directe entre le numéro de ligne de cache et le numéro de case de cache). On ne s'intéresse pas, dans ce TP, au fonctionnement du cache L2, qui peut être vu comme un accélérateur d'accès à la mémoire externe : grâce au cache L2, un accès à la mémoire, en cas de MISS sur un cache L1 va coûter en moyenne quelques dizaines de cycles au lieu de quelques centaines de cycles s'il est absent.

Pour ce TP, vous utiliserez le simulateur `almol.x`, qui peut produire des fichiers d'instrumentation permettant de suivre l'évolution des caches L1 au cours du temps. Commencez par recopier le code du tp4 dans votre répertoire de travail (l'archive est accessible dans l'INDEX en haut de cette page).

```
tp4
├── Makefile
└── src
    ├── harch.c
    ├── harch.h
    ├── hcpu.S
    ├── hcpu.h
    ├── kernel.ld
    ├── kinit.c
    ├── klibc.c
    └── klibc.h
```

C.1. Calcul du taux de MISS dans le cache d'instructions

Ce répertoire tp4 contient 1 répertoire. Il va permettre de voir l'évolution des miss de cache. Tous les fichiers nécessaires à la génération du code binaire `kernel.x` se trouvent dans le fichier `src`, le fichier `Makefile` permet de générer l'exécutable et les fichiers de trace. Ces fichiers représentent une version minimaliste du système (vu au tp1), il n'y a presque rien, mais le but est d'analyser le comportement des caches donc, moins il y a de code à exécuter avant la fonction que vous allez analyser, mieux c'est. Dans un premier temps vous utiliserez le code sans modification.

Allez dans le répertoire `tp4`

1. Ouvrez le fichier `src/kinit.c` et expliquez ce que fait la fonction `kinit()` dans le cadre de ce TP ?

- La fonction `kinit()` déclare un tableau de 20 entiers. Les valeurs sont initialisées dans une première boucle `for`, puis une seconde boucle `for` est exécutée 1000 fois. À chaque itération de cette seconde boucle, chaque élément du tableau est incrémenté d'une valeur égale à son indice de tableau. Finalement, les valeurs finales des 20 éléments du tableau sont affichées sur le terminal grâce à une troisième boucle `for`.

2. Lancez l'exécution du `Makefile` (make compil), puis examinez le code assembleur correspondant à l'application logicielle (`kernel.x.s`). Déterminez les adresses de début et de fin de la boucle de calcul (seconde boucle `for`).

- Combien d'instructions sont exécutées à chaque itération de cette boucle ?
 - Toutes les instructions de la boucle de calcul peuvent-elles être simultanément stockées dans le cache ?
 - Que pouvez-vous en conclure ?
- La boucle commence à l'instruction `lw v0,24(sp)` et se termine à l'instruction `nop` qui suit l'instruction `bnez v0, kinit+0x50`.
 - Cette boucle exécute 51 instructions à chaque itération. Si les étudiants posent la question, on peut expliquer que l'instruction `nop` qui suit le branchement est toujours exécutée à cause de l'effet retardé du branchement (`delayed slot` dû à l'architecture *pipeline*).
 - Comme le cache ne peut contenir que 32 instructions au max (8 cases contenant chacune 4 instructions), la boucle ne tient pas entièrement dans le cache. Il y aura donc des MISS et des évincements à chaque itération dans la boucle.

3. Vous allez renommer le fichier `kernel.x.s` en `kernel.myx.s` et y ajouter des commentaires (ce renommage permet de ne pas perdre vos commentaires lors du `make clean`), déterminez, pour chaque instruction de la boucle de calcul, dans quelle case du cache sera rangée la ligne de cache à laquelle cette instruction appartient. La boucle `for` fait 51 instructions, vous devez grouper les instructions par 4 (puisque une ligne de cache contient 4 instructions).

- Un exemple de modification du fichier `kernel.x.s` pour l'ajout de commentaires est donné dans la réponse de la question suivante.

4. En analysant la valeur du champ `index` de l'adresse, calculez pour chacune de ces 13 lignes de cache, dans quelle case du cache elle va être stockée.

```
80000210 <kinit>:
[...]
```

8000024c:	1440fff5	bnez	v0,80000224 <kinit+0x14>
80000250:	00000000	nop	
80000254:	afa00010	sw	zero,16(sp)
80000258:	10000030	b	8000031c <kinit+0x10c>
8000025c:	00000000	nop	

ligne de cache (ci-dessous) : case n°6

80000260:	8fa20018	lw	v0,24(sp)
80000264:	afa20018	sw	v0,24(sp)
80000268:	8fa2001c	lw	v0,28(sp)
8000026c:	24420001	addiu	v0,v0,1

ligne de cache (ci-dessous) : case n°7

80000270:	afa2001c	sw	v0,28(sp)
-----------	----------	----	-----------

```

80000274: 8fa20020 lw v0,32(sp)
80000278: 24420002 addiu v0,v0,2
8000027c: afa20020 sw v0,32(sp)

# ligne de cache (ci-dessous) : case n°0
80000280: 8fa20024 lw v0,36(sp)
80000284: 24420003 addiu v0,v0,3
80000288: afa20024 sw v0,36(sp)
8000028c: 8fa20028 lw v0,40(sp)

# ligne de cache (ci-dessous) : case n°1
80000290: 24420004 addiu v0,v0,4
80000294: afa20028 sw v0,40(sp)
80000298: 8fa2002c lw v0,44(sp)
8000029c: 24420005 addiu v0,v0,5

# ligne de cache (ci-dessous) : case n°2
800002a0: afa2002c sw v0,44(sp)
800002a4: 8fa20030 lw v0,48(sp)
800002a8: 24420006 addiu v0,v0,6
800002ac: afa20030 sw v0,48(sp)

# ligne de cache (ci-dessous) : case n°3
800002b0: 8fa20034 lw v0,52(sp)
800002b4: 24420007 addiu v0,v0,7
800002b8: afa20034 sw v0,52(sp)
800002bc: 8fa20038 lw v0,56(sp)

# ligne de cache (ci-dessous) : case n°4
800002c0: 24420008 addiu v0,v0,8
800002c4: afa20038 sw v0,56(sp)
800002c8: 8fa2003c lw v0,60(sp)
800002cc: 24420009 addiu v0,v0,9

# ligne de cache (ci-dessous) : case n°5
800002d0: afa2003c sw v0,60(sp)
800002d4: 8fa20040 lw v0,64(sp)
800002d8: 2442000a addiu v0,v0,10
800002dc: afa20040 sw v0,64(sp)

# ligne de cache (ci-dessous) : case n°6
800002e0: 8fa20044 lw v0,68(sp)
800002e4: 2442000b addiu v0,v0,11
800002e8: afa20044 sw v0,68(sp)
800002ec: 8fa20048 lw v0,72(sp)

# ligne de cache (ci-dessous) : case n°7
800002f0: 2442000c addiu v0,v0,12
800002f4: afa20048 sw v0,72(sp)
800002f8: 8fa2004c lw v0,76(sp)
800002fc: 2442000d addiu v0,v0,13

# ligne de cache (ci-dessous) : case n°0
80000300: afa2004c sw v0,76(sp)
80000304: 8fa20050 lw v0,80(sp)
80000308: 2442000e addiu v0,v0,14
8000030c: afa20050 sw v0,80(sp)

# ligne de cache (ci-dessous) : case n°1
80000310: 8fa20010 lw v0,16(sp)
80000314: 24420001 addiu v0,v0,1
80000318: afa20010 sw v0,16(sp)
8000031c: 8fa20010 lw v0,16(sp)

# ligne de cache (ci-dessous) : case n°2
80000320: 2c4203e8 sltiu v0,v0,1000
80000324: 1440ffce bnez v0,80000260 <kinit+0x50>
80000328: 00000000 nop
8000032c: afa00014 sw zero,20(sp)
[...]
```

5. Évaluez le nombre de MISS instruction lors de l'exécution de la première itération ? Lors de la deuxième itération ? En déduire une valeur estimée du *taux de MISS* moyen après 1000 itérations.

- 1re itération : En exécutant la boucle `for` la première fois, le processeur va provoquer le chargement de 13 lignes de caches aux index successifs suivants : 6, 7, 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2. Il y a donc 13 MISS lors de la 1re itération.
- Itérations suivantes : Au début de la 2e itération, les instructions contenues dans les cases 6, 7, 0, 1, 2 font MISS, car elles ont été écrasées. Les instructions contenues dans les cases 3, 4, 5 ne font pas MISS. À la fin de l'itération, les instructions contenues dans les cases 0, 1, 2 font de nouveau MISS. On a donc 10 MISS pour 51 instructions lors de la 2e itération, et il en va de même pour les itérations suivantes. Ceci correspond à un taux de MISS de 10/51, légèrement inférieur à 20%.

C.2. Analyse de trace

Vous allez maintenant tenter de valider ce calcul du taux de MISS par la simulation. Dans le fichier `Makefile`, vous pouvez voir de nouvelles règles : `cachetrace` et `cachestats`, qui lancent le simulateur en lui demandant d'afficher les états successifs des caches et des statistiques. Elles imposent aussi les caractéristiques du cache :

- -NICACHELEN : nombre de mots par case dans le cache instruction
- -NDCACHELEN : nombre de mots par case dans le cache data
- -NICACHESET : nombre de cases dans le cache instruction
- -NDCACHESET : nombre de cases dans le cache data

Pour observer précisément le comportement des caches, le simulateur dispose donc d'une option d'instrumentation `-TRACE cache.txt` qui produit le fichier `cache.txt` permettant de visualiser le contenu des caches instructions et data au cours du temps. Les fichiers de trace du cache étant très volumineux, on a limité à 5000 le nombre de cycles simulés en utilisant l'option `-NCYCLES 5000`.

Vous pouvez ouvrir le fichier `Makefile` pour voir la commande du simulateur avec ces options. La règle `cachetrace` du `Makefile` lance le simulateur avec le mode `TRACE` afin d'obtenir la **trace de remplissage du cache**, mais également avec le mode `DEBUG` que vous connaissez déjà afin de générer la **trace d'exécution du programme**. Comprenez bien que ce sont deux traces du même programme, mais de nature très différente.

A titre d'illustration, si vous exécutez la commande : `make cachetrace`, voici ce que vous pouvez observer au tout début des fichiers `trace0.s` et `cache.txt`, relativement à l'exécution de la première instruction du code de boot :

`trace0.s`

```
K 12: <boot> ----- ./src/hcpu.S
K 12: 0xbfc00000 0x3c1d8040 lui sp,0x8040
K 13: 0xbfc00004 0x27bd0000 addiu sp,sp,0
K 14: 0xbfc00008 0x3c1a8000 lui k0,0x8000
K 15: 0xbfc0000c 0x275a0210 addiu k0,k0,528
K 26: 0xbfc00010 0x03400008 jr k0
K 27: 0xbfc00014 0x00000000 nop
K 37: <kinit> ----- ./src/kinit.c
K 37: 0x80000210 0x27bdfafa addiu sp,sp,-96
[...]
```

`cache.txt`

```
***** cycle 10 INSTRUCTION
V / way 0 / set 0 / @ = BFC00000 / 275A0210 3C1A8000 27BD0000 3C1D8040
/ way 0 / set 1 / @ = 00000010 / 00000000 00000000 00000000 00000000
/ way 0 / set 2 / @ = 00000020 / 00000000 00000000 00000000 00000000
/ way 0 / set 3 / @ = 00000030 / 00000000 00000000 00000000 00000000
/ way 0 / set 4 / @ = 00000040 / 00000000 00000000 00000000 00000000
/ way 0 / set 5 / @ = 00000050 / 00000000 00000000 00000000 00000000
/ way 0 / set 6 / @ = 00000060 / 00000000 00000000 00000000 00000000
/ way 0 / set 7 / @ = 00000070 / 00000000 00000000 00000000 00000000
[...]
```

On voit que la première instruction `lui sp,0x8040` a le code binaire `0x3c1d8040` et que cette instruction a été rangée dans le mot 0 (à gauche) dans la case 0 du cache (c'est le numéro de set). Lors de la lecture de cette instruction, les 3 autres instructions de la même ligne ont aussi été chargée.

Le fichier `cache.txt` représente l'état, ici au cycle 10, des 8 cases du cache. La case d'index 0 (notée ici set 0) est valide `V`, c'est-à-dire qu'elle contient la ligne de l'espace d'adressage dont l'adresse du premier octet est `BFC00000` (on n'indique pas le numéro de ligne, mais l'adresse de ligne par souci de lisibilité, si ce n'est pas clair pour vous, demandez-moi ou relisez le cours).

Le numéro de `way` c'est pour les caches n-way-set-associative, qui permettent d'avoir plusieurs `way`s (c'est-à-dire plusieurs cases) possibles pour chaque ligne. En fait, un cache direct-mapped est un cache 1-way-set-associative.

Notez aussi, qu'au cycle 10 la ligne est déjà dans le cache, mais que l'instruction n'est exécutée que 2 cycles plus tard.

1. Commencez par lancer la simulation *normalement* avec la commande : `make exec`

Vous devriez voir les résultats s'afficher dans la fenêtre du TTY, avec la date à laquelle le programme est arrivé au `exit()`. Pour arrêter le simulateur, il faut taper le caractère `CTRL + C` dans la fenêtre du terminal où a été lancée la simulation. A quelle numéro de cycles s'affiche le message EXIT

- 230892 (autour de 24000 cycles)

2. Relancez le simulateur pour avoir l'histoire du cache avec `make tracecache` :

Une fois la simulation terminée, ouvrez dans 4 fenêtres différentes (1) le fichier source `src/kinit.c` ; (2) le fichier `kernel.x.s` contenant le code désassemblé ; (3) le fichier de trace d'exécution du processeur `trace0.txt` contenant la séquence des instructions exécutées par le MIPS au cours du temps et, enfin (4) le fichier de trace du cache `cache.txt` contenant les états successifs des caches data et instruction au cours du temps.

Observez le remplissage progressif des deux caches au fur et à mesure de l'exécution de l'application.

- À quel cycle est chargée dans le cache d'instructions la première instruction de la fonction `kinit()` ?
 - Dans le `cache.txt`. On voit que la première ligne de cache correspondant aux instructions de la fonction `kinit()` est copiée dans la case n°0 du cache au cycle 35. Dans le fichier `trace0.txt`, la première instruction est exécutée au cycle 37.
- À quel cycle est chargée la première ligne de cache contenant des instructions du corps de la boucle de calcul ? (on peut la repérer parce la boucle de calcul fait beaucoup de fois la même chose)
 - Dans `kernel.x.s`, on peut repérer la première instruction de la boucle de calcul :
`80000260: 8fa20018 lw v0,24(sp)`
 Dans `trace0.s`, on peut voir que cette instruction est exécutée au cycle 337 :
`K 371: 0x80000260 0x8fa20018 lw v0,24(sp)`
 - La première ligne de cache correspondant aux premières instructions de la boucle est copiée dans la case n°6 du cache au cycle 369.
`V / way 0 / set 6 / @ = 80000260 / 24420001 8FA2001C AFA20018 8FA20018`
- À quel cycle cette première ligne est-elle évincée par le chargement d'une autre ligne de cache ?
 - Elle est évincée au cycle 533 pour stocker d'autres instructions situées vers la fin de la boucle. Il faut observer quand la case 6 change de ligne.
- À quel cycle cette première ligne est-elle rechargée pour exécuter la deuxième itération de la boucle ? Et à quel cycle cette première instruction est ré-exécutée pour la seconde itération ?

- Elle est rechargée pour la deuxième itération au cycle 618 (`cache.txt`) et ré-exécutée en 620 (`trace0.s`).
- Quelle est la durée (en nombre de cycles) de la première itération?
 - Il s'est donc écoulé $(620 - 371) = 249$ cycles entre la 1^e et la 2^e itérations. Cela signifie qu'il a fallu 249 cycles pour exécuter 50 instructions, soit 5 cycles par instruction.
- À quel cycle est-elle ré-exécutée à la troisième itération?
 - l'instruction est re-exécutée au cycle 794, il faut chercher l'adresse `0x80000260` dans `trace0.s`.
- Quelle est la durée des itérations suivantes?
 - Il s'est donc écoulé $(794 - 620) = 174$ cycles entre la 2^e et la 3^e itérations, soit 3,5 cycles par instruction.

C.3. Mesure du taux de MISS

Pour mesurer le taux de MISS sur le cache instruction, nous allons activer l'option d'instrumentation `-STATS stats.txt` du simulateur de la machine `almo1.x`. Cette option permet de produire un fichier nommé `stats.txt`. Ce fichier `stats.txt` contient des informations statistiques de comportement du cache. Pour ce faire, le simulateur relève à intervalles réguliers (tous les 10 cycles) différents compteurs du simulateur permettant de caractériser l'activité des caches L1.

- Chaque ligne de ce fichier de statistiques contient 8 valeurs :
 - Le nombre de cycles simulés depuis le démarrage de la machine (incrément de 10 à chaque ligne),
 - Le nombre d'instructions exécutées depuis le démarrage de la machine,
 - Le nombre de MISS sur le cache d'instructions depuis le démarrage de la machine,
 - Le nombre de lectures de données depuis le démarrage de la machine,
 - Le nombre de MISS sur le cache de données depuis le démarrage de la machine,
 - Le taux de MISS sur le cache d'instructions,
 - Le taux de MISS sur le cache de données,
 - Le CPI, qui est le nombre moyen de cycles par instruction.

Relancez la simulation avec la commande shell suivante : `make cachestats`

Vous pouvez ouvrir le fichier `Makefile` pour voir comment est appliquée l'option `STATS` au simulateur.

À l'aide de l'outil `'gnuplot'` (s'il n'est pas installé sur votre machine personnelle, vous devrez l'installer), c'est un logiciel de visualisation de courbes, vous allez afficher l'évolution du taux de MISS sur le cache d'instructions au cours du temps. Pour cela, lancez la commande :

```
$ gnuplot
```

Une fois dans ce logiciel (indiqué par l'invite de commande `'gnuplot> '`), vous pouvez entrer la commande :

```
plot 'stats.txt' using 1:6
```

Note : cette commande signifie que vous souhaitez afficher la courbe où la colonne n°1 du fichier `stats.txt` (le nombre de cycles écoulés) est en abscisse et la colonne n°6 (le taux de MISS sur le cache d'instructions) est en ordonnée.

Attention : les valeurs mesurées sont des moyennes cumulées depuis le début de la simulation...

- Comment expliquez-vous l'évolution du taux de MISS au cours du temps ?

- Au début le cache est vide, et il n'y a que des MISS. Puis le cache d'instructions est rempli avec le code de `reset`, puis avec le code du `kinit`, et enfin avec le code de la boucle. Le taux de MISS dans le cache remonte pour se rapprocher de 20% car 5 des 8 lignes de cache sont systématiquement MISS pendant l'exécution de la boucle (10 MISS pour 51 instructions lues dans le cache).

C.4. Optimisation du code pour minimiser le taux de MISS

Pour minimiser le taux de MISS, il faut modifier l'application logicielle pour que les 1000 itérations de la boucle de calcul puissent s'exécuter sans MISS sur le cache d'instructions. Pour cela, on peut remplacer les 15 lignes calculant les 15 nouvelles valeurs du tableau par une boucle `for` interne portant sur l'index dans le tableau, de façon à obtenir un code plus compact, qui tienne entièrement dans le cache.

- Copiez le fichier `kinit.c` actuel dans un autre fichier (par exemple, `kinit_orig.c`) afin de garder une sauvegarde du fichier original. Puis, ouvrez le fichier `kinit.c` et modifiez la fonction `kinit()` comme indiqué ci-dessus.
- Éditez le fichier exécutable de l'application logicielle (`kernel.x.s`), et vérifiez que votre nouvelle boucle de calcul a bien une longueur inférieure à 32 instructions (afin d'être contenue entièrement dans le cache).
- Éditez le fichier `Makefile` pour que la simulation avec statistique produise le fichier `stats_nomiss.txt`.
- Relancez la simulation pour 100000 cycles, en changeant le nom du fichier de statistiques : `make cachestat`
- Enfin, à l'aide de `gnuplot`, affichez sur le même graphique les résultats des exercices **C.3.** et **C.4.**, afin de les comparer. Pour cela, entrez les deux commandes suivantes :

```
plot 'stats.txt' using 1:6
replot 'stats_nomiss.txt' using 1:6
```

- Comment expliquez-vous l'évolution du taux de MISS pour cette nouvelle version de l'application ?

- Au début, le comportement des deux versions de l'application est identique. Vers les cycles 300/400, le cache est complètement chargé. Quand le processeur commence à exécuter la boucle de calcul, les deux courbes commencent à diverger, puisque la courbe verte correspond au cas où toutes les instructions de la boucle tiennent dans le cache : le taux de MISS instruction est nul.