Fonctionnement des Ordinateurs

TP3 - Hiérarchie mémoire

B. QUOITIN
Faculté des Sciences
Université de Mons

Résumé

L'objectif de ce TP est de renforcer votre compréhension de l'organisation de la mémoire, de la hiérarchie mémoire, des caches et de la mémoire virtuelle.

Table des matières

1 Organisation des mémoires

1.1 Quantité de mémoire

Un ordinateur peut être muni d'une quantité de mémoire de, par exemple, 1073741824 octets. Pourquoi un fabricant choisirait-il une quantité aussi bizarre, plutôt que 1000000000 octets ?

Q1) Pourquoi ?

Le nombre d'adresses d'une mémoire est un exposant de 2. Si m représente le nombre de bits d'une adresse, il y aura 2^m adresses possibles.

De plus, chaque cellule de la mémoire contient un nombre d'octets qui est lui aussi un exposant de 2, p.ex. 4 ou 8 octets. Si n représente le nombre d'octets d'une cellule, alors la taille totale de la mémoire, en octets, sera égale à $n \times 2^m$. Comme n est aussi un exposant de 2, la taille de la mémoire est aussi un exposant de 2.

Dans cet exercice, la taille de la mémoire est égale à 2^{30} octets.

1.2 Endianness

Supposons une mémoire de 4 Ko organisée sous la forme de 4096 cellules de 8 bits. Une partie de son contenu est donnée dans la Table 1. Cette mémoire est accédée par un processeur qui ne peut lire ou écrire que 8 bits à la fois. Le processeur est de type *little-endian*.

Adresse mémoire	Donnée
...	...
0x120	0x8F
0x121	0xA2
0x122	0xB8
0x123	0x2A
...	...

TABLE 1 – Contenu de la mémoire. Les adresses et valeurs des mots sont représentées en hexadécimal.

Un programme qui effectue des lectures d'entiers en mémoire est exécuté sur ce système. Les entiers lus peuvent avoir des tailles différentes et peuvent être signés ou non-signés. Les nombres signés sont représentés en complément à 2. On distingue les types suivants : entier signé de 8 bits (`char`), entier non-signé de 8 bits (`unsigned char`), entier signé de 16 bits (`short`) et entier non-signé de 16 bits (`unsigned short`). Pour chaque lecture, donnez en décimal la valeur de l'entier lu.

Q2) `char` à l'adresse `0x120`

Le mot lu est `0x8F`. L'endianness n'a pas d'importance dans ce premier cas car on ne lit qu'un octet. Si on interprète ce mot comme la représentation d'un entier (signé), sous-entendu en complément à 2, alors cet entier vaut -113.

En effet, l'observation de la représentation binaire du mot, `10001111`, permet de constater que le bit de poids fort vaut 1, associé au poids -128, tandis que les 7 bits restants représentent 15. Le résultat vaut donc -128 + 15 = -113.

Q3) `unsigned char` à l'adresse `0x121`

Le mot lu est `0xA2`. Interpreté comme un naturel, cela donne 162.

Q4) `unsigned short` à l'adresse `0x122`

On effectue la lecture de 2 octets consécutifs aux adresses `0x122` et `0x123`. Ces octets sont respectivement `0xB8` et `0x2A`. Il est nécessaire de se préoccuper de l'*endianness* pour re-composer le mot de 16 bits. En *little-endian*, on stocke la "petite" partie d'un mot, sous-entendu les octets de poids faible, aux adresses les plus basses. Le mot recomposé est donc `0x2AB8`.

Si on interprète ce mot comme un naturel, cela donne 10936.

Q5) `short` à l'adresse `0x120`

Le mot lu est `0xA28F`.

Interpreté comme un entier, cela donne -32768 + 8847 = -23921. Observer la représentation binaire du mot, `1010 0010 1000 1111`, permet de s'en convaincre.

1.3 Détection de l'endianness

Le morceau de programme suivant peut être utilisé pour déterminer si un ordinateur est *little-endian* ou *big-endian*. Expliquez comment.

```
1 li $t0, 0xABCD9876
2 sw $t0, 100($0)
3 lb $t1, 100($0)
```

Q6) Comment ?

L'idée du programme ci-dessus est d'écrire à une adresse arbitraire (100 dans cet exemple), un mot connu de 32 bits et de relire les 8 premier bits en mémoire de ce mot. Si le résultat correspond aux 8 bits de poids faible du mot (`0x76`), alors le système est *little-endian*, sinon (`0xAB`) le système est *big-endian*.

Le code donné en exemple ci-dessus ne s'exécute pas correctement dans QtSpim. La raison est que l'adresse 100 n'est pas utilisable et son accès génère une exception. Pour pouvoir l'exécuter dans QtSpim, il faut modifier légèrement le programme de façon à réserver une zone mémoire à une adresse située dans la zone `data` du simulateur. Le code ci-dessous donne un exemple de code modifié.

```
1         .data
2 addr:
3         .word 0
4
5         .text
6 main:
7         la $t2, addr
8         li $t0, 0xABCD9876
9         sw $t0, 0($t2)
10        lb $t1, 0($t2)
11        jr $ra
```

1.4 Problèmes d'alignement

Le programme suivant, écrit en langage C, effectue plusieurs transferts entre des variables situées en mémoire. Le type `uint16_t` (resp. `uint32_t`) correspond à un entier non-signé représenté sur 16-bits (resp. 32-bits). Pour rappel, l'opérateur `&` permet d'obtenir l'adresse en mémoire d'une variable, tandis que l'opérateur `*` permet d'obtenir la valeur située à une adresse donnée. Par exemple, `&a` permet de récupérer l'adresse de la variable `a`, alors que `*(&a)` permet de lire la valeur située à l'adresse de `a`.

```
1  #include <inttypes.h>
2  #include <stdio.h>
3
4  uint32_t a = 3;
5  uint16_t b;
6  uint16_t c[] = { 1, 3 };
7
8  int main() {
9    b = *((uint16_t *) &a);
10   a = *((uint32_t *) c);
11
12   printf("b = %" PRIu16 "\n", b);
13   printf("a = %" PRIu32 "\n", a);
14
15   return 0;
```

```
16 | }
```

Note : la documentation des types `uintn_t` et leur format d'affichage `PRIun` peut être obtenue dans les pages de manuel (`man inttypes.h`).

Sur certains systèmes, certains transferts mémoire effectués par le programme ci-dessus s'exécuteront de façon peu performante. Sur d'autres systèmes, ils mèneront à une erreur, voire à un *crash*. Identifiez les accès problématiques et décrivez pourquoi ils pourraient poser problème.

Q7) Accès mémoires problématiques et conséquences.

Le programme ci-dessous utilise le transtypage pour copier le contenu de mots mémoires de tailles différentes.

- L'affectation de `b` en ligne 9 prend les 16 bits de poids faible de `a`. En conséquence de quoi, `b` doit prendre la valeur 3 si l'architecture est de type *little-endian*.
- L'affectation de `a` en ligne 10 considère les deux cellules de 16-bits du tableau `c` comme un seul mot de 32-bits. En conséquence de quoi, la variable `a` prend la valeur 196609 si l'architecture est de type *little-endian*.

Si les données ne sont pas alignées correctement en mémoire, les accès en mémoire de ce programme peuvent avoir des performances dégradées voire conduire à des erreurs. Supposons par exemple que les cellules du tableau soient alignées sur 2 octets (en raison du type `uint16_t`). Lorsque le programme effectue la lecture d'un mot de 4 octets à l'adresse du tableau, on fait l'hypothèse implicite que cette adresse est alignée sur 4 octets, ce qui pourrait ne pas être le cas.

Comment traduiriez-vous (manuellement) le programme ci-dessus en langage d'assemblage MIPS ? Pour garder la traduction simple, omettez les appels à la fonction `printf`.

Q8) Traduction en langage d'assemblage MIPS.

```
1         .data
2 _a:     .word 3
3 _b:     .space 2
4 _c:     .short 1, 3
5
6         .text
7 main:
8         la $t0, _a
9         la $t1, _b
10        la $t2, _c
11        lh $t3, 0($t0)
12        sh $t3, 0($t1)
13        lw $t3, 0($t2)
14        sw $t3, 0($t0)
15        jr $ra
```

Enregistrez le programme en langage C ci-dessus dans un fichier nommé `align-pblm.c`. Ensuite, compilez le et exécutez le sur votre propre ordinateur. Essayez d'interpréter les instructions du programme généré. Pouvez-vous en particulier y repérer les transferts mémoires en question. Utilisez à cet effet les commandes montrées ci-dessous. Note : il est également possible d'utiliser le site web "compiler explorer" pour effectuer la compilation voire l'exécution de ce programme.

```
gcc -o align-pblm align-pblm.c
./align-pblm
objdump -d align-pblm | less
```

1.5 Temps d'accès DRAM

Supposons une mémoire SDRAM DDR organisée en mots de 32 bits et connectée au processeur par un bus système à f_{bus} =200MHz. Son délai *RAS to CAS* vaut t_{RCD} =2 cycles, tandis que son délai *CAS* vaut t_{CL}=2 cycles.

La mémoire SDRAM permet d'effectuer la lecture séquentielle de jusqu'à 8 mots de 32 bits en un seul accès, dans un mode appelé *sequential burst*. Les cellules auxquelles le processeur accède dans ce mode font partie d'un même bloc de 8 mots et dont le premier mot est situé à une adresse mémoire alignée sur 8 mots. La lecture commence à une adresse mémoire située dans le bloc et se poursuit avec les adresses suivantes (incrémentées de 1). Si le bout du bloc est atteint, la lecture continue au début du bloc. Supposons par exemple que la lecture commence à l'adresse mémoire 6 et que 3 cellules soient lues en mode *sequential burst*, les cellules 6, 7 et 0 seront retournées.

Ce qui vous est demandé : Un programme doit lire les octets situés en mémoire aux adresses CPU suivantes `0x12340014`, `0x12340018`, `0x12340024`, `0x12340025` et `0x1234002A`. Déterminez le temps minimum pour obtenir ces données. Attention aux différences entre adresses mémoire (en mots) et processeur (en octets).

Q9) Temps d'accès minimum.

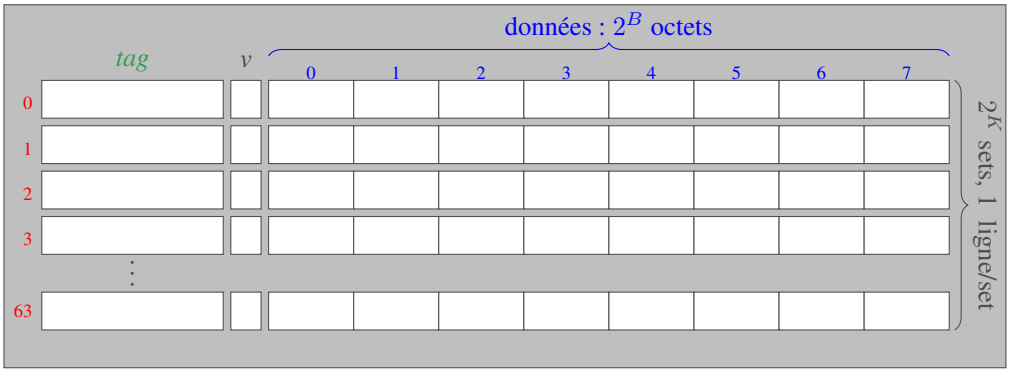
2 Caches

2.1 Usage d'une adresse avec différents formats de cache

Un processeur donné accède la mémoire avec des adresses de 32-bits. Supposons qu'il y ait une cache entre le processeur et la mémoire. La mémoire est adressable par octet. La cache a une taille de 512 octets et des lignes de 8 octets. Pour chacune des architectures de cache suivantes, indiquez comment les adresses de 32-bits seront utilisées. Par exemple, pour la cache *direct-mapped*, vous devriez indiquer combien de bits de l'adresse sont utilisés pour le *tag*, pour le *set* et pour l'*offset*.

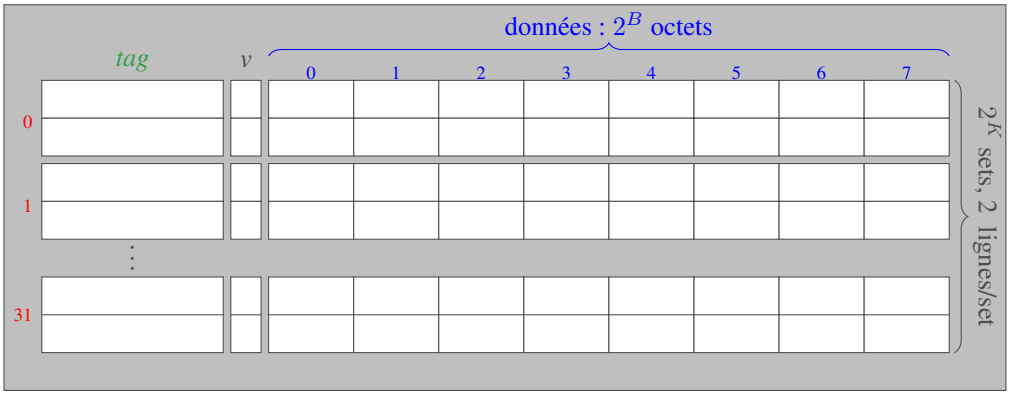
Q10) cache *Direct-mapped* : nombre de bits de *tag*, *set* et *offset*.

Le nombre d'octets par ligne est égal 8. Il faut donc $B = log_2(8) = 3$ bits pour l'*offset*.
La taille de la cache est égale à 512 octets. Il y a donc $512/8 = 64$ lignes dans la cache. Dans le cas d'une cache *direct-mapped*, il y a une ligne par set. Il y a donc 64 sets. Il est nécessaire d'utiliser $K = log_2(64) = 6$ bits pour *set*.
Finalement, la longueur du *tag* est égale à $32 - K - B = 23$ bits.
Une adresse a donc le format suivant : TTTTTTTTTTTTTTTTTTTTTTTSSSSSSOOO.



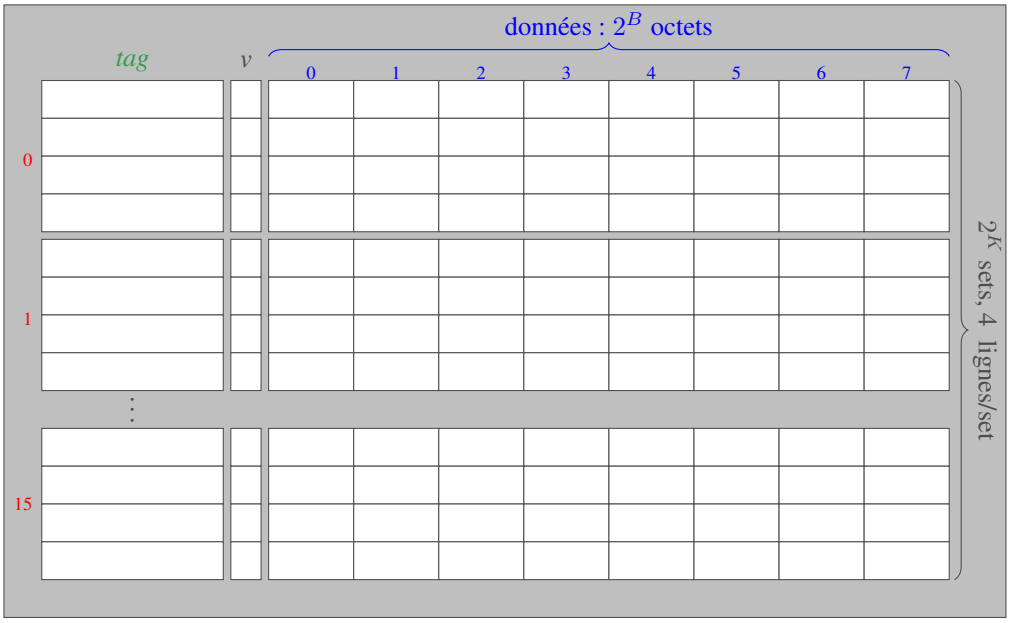
Q11) cache *2-way set-associative* : nombre de bits de *tag*, *set* et *offset*.

Dans le cas d'une cache *2-way set-associative*, il y a $S = 2$ lignes par set. Il est donc possible de stocker en même temps dans cette cache deux adresses qui ont la même valeur de *set*.
Le nombre d'octets par ligne reste égal à 8. Par conséquent, $B = 3$.
Le nombre de lignes de cache reste égal à 64.
En revanche, il y a deux fois moins de sets : $64/S = 32$. Le nombre de bits de *set* vaut donc $K = log_2(32) = 5$.
Par conséquent, la longueur de *tag* est égale à $32 - 5 - 3 = 24$.
Une adresse a donc le format suivant : TTTTTTTTTTTTTTTTTTTTTTTTSSSSSOOO.



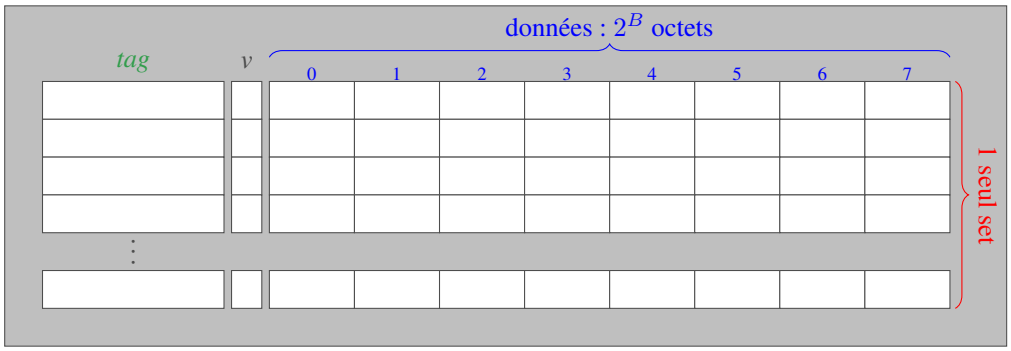
Q12) cache *4-way set-associative* : nombre de bits de *tag*, *set* et *offset*.

Même principe, mais le nombre de lignes par set devient $S = 4$. Le nombre de sets est donc $512/8/4 = 16$ et $K = 4$.
Par conséquent, la longueur de *tag* est égale à $32 - 5 - 4 = 25$.
Une adresse a donc le format suivant : TTTTTTTTTTTTTTTTTTTTTTTTTSSSSOOO.



Q13) cache *fully-associative* : nombre de bits de *tag*, *set* et *offset*.

Dans une cache *fully-associative*, il n'y a qu'un seul set contenant toutes les lignes.
Les paramètres de cette cache sont donc $B = 3$, $K = 0$ et le tag fait $32 - 3 = 29$ bits de long.
Une adresse a donc le format suivant : TTTTTTTTTTTTTTTTTTTTTTTTTTTTTOOO.



2.2 Implémentation d'une cache

Calculer le nombre total de bits nécessaires pour implémenter les caches suivantes dans un système où les adresses sont représentées sur 32 bits. Attention, le nombre de bits nécessaires est différent de la capacité de stockage de la cache. Le nombre de bits nécessaire représente la quantité totale de mémoire nécessaire pour stocker toutes les informations de la cache.

Q14) *Direct-mapped* : 16 lignes de 8 octets.

8 octets par ligne $\Rightarrow B = 3$.
16 lignes, 1ligne / set $\Rightarrow K = 4$.
La longueur du tag est par conséquent $32 - 3 - 4 = 25$.
Chaque ligne de la cache doit stocker un tag, un bit de validité et 8 octets, pour un total de $25 + 1 + 8 \times 8 = 90$ bits.
La taille totale de la cache est donc égale à $16 \times 90 = 1440$ bits.
La capacité de la cache est de $16 \times 8 \times 8 = 1024$ bits.

Q15) ***Direct-mapped*** **: 8 lignes de 32 octets.**

$B = 5$, $K = 3$, tag de longueur 24.
Nombre de bits par ligne : $24 + 1 + 32 \times 8 = 281$ bits.
Nombre total de bits : $8 \times 281 = 2248$.
La capacité de la cache est de $8 \times 32 \times 8 = 2048$ bits.

Q16) ***Fully-associative*** **: 16 lignes de 8 octets.**

$B = 3$, $K = 0$, tag de longueur 29.
Nombre de bits par ligne : $29 + 1 + 8 \times 8 = 94$ bits.
Nombre total de bits : $16 \times 94 = 1504$.
La capacité de la cache est de $16 \times 8 \times 8 = 1024$ bits.

Q17) ***Fully-associative*** **: 8 lignes de 32 octets.**

$B = 5$, $K = 0$, tag de longueur 27.
Nombre de bits par ligne : $27 + 1 + 32 \times 8 = 284$ bits.
Nombre total de bits : $8 \times 284 = 2272$.
La capacité de la cache est de $8 \times 32 \times 8 = 2048$ bits.

Q18) ***8-way-associative*** **: 16 lignes de 8 octets.**

$B = 3$, $K = 1$ (16 lignes, 8 lignes par set), tag de longueur 28.
Nombre de bits par ligne : $28 + 1 + 8 \times 8 = 93$ bits.
Nombre total de bits : $16 \times 93 = 1488$.
La capacité de la cache est de $16 \times 8 \times 8 = 1024$ bits.

Q19) ***2-way-associative*** **: 8 lignes de 32 octets.**

$B = 5$, $K = 2$ (8 lignes, 2 lignes par set), tag de longueur 25.
Nombre de bits par ligne : $25 + 1 + 32 \times 8 = 282$ bits.
Nombre total de bits : $8 \times 282 = 2256$ bits.
La capacité de la cache est de $8 \times 32 \times 8 = 2048$ bits.

2.3 Correspondance adresse ↔ (tag, set, offset)

Soit une cache *4-way-associative* de taille 128 octets organisée en 16 lignes de 8 octets.
Quelles sont les valeurs des paramètres suivants ?

Q20) K **nombre de bits identifiant un** ***set***

$K = 2$

Q21) B **nombre de bits d'***offset*

$B = 3$

Donnez pour les adresses suivantes les valeurs du triplet (*tag*, *set*, *offset*). Toutes les adresses sont encodées sur 16 bits.

Q22) **`0xA3C9`**

Le format d'une adresse est `TTTTTTTTTTTSSOOO`.
En binaire, l'adresse est égale à `1010001111001001`.
En conséquence de quoi, le tag vaut `0x51E`, le set vaut 1 et l'offset vaut 1.

Q23) 0xA3CB

Adresse en binaire : 1010001111001011
En conséquence de quoi, le tag vaut 0x51E, le set vaut 1 et l'offset vaut 3.

Q24) 0xB5EA

Adresse en binaire : 1011010111101010
En conséquence de quoi, le tag vaut 0x5AF, le set vaut 1 et l'offset vaut 2.

Q25) 0xB5E1

Adresse en binaire : 1011010111100001
En conséquence de quoi, le tag vaut 0x5AF, le set vaut 0 et l'offset vaut 1.

2.4 Cache direct-mapped

Un processeur de 32-bits possède une cache L1 unifiée de 1 Ko, organisée en lignes de 4 octets et de type *direct-mapped*. Un programmeur écrit, compile et exécute le programme suivant sur ce processeur :

```
1 #define N 32
2 char A[N], B[N], C[N];
3
4 int main()
5 {
6   for (int i = 0; i < N; i++)
7     A[i] = B[i] + C[i];
8   return 0;
9 }
```

Calculez le nombre de bits nécessaires dans une adresse pour localiser chaque élément dans la cache : *tag*, *set* et *offset*. Montrez également comment ces bits sont pris dans une adresse en identifiant avec T pour *tag*, S pour *set* et O pour *offset* les 32 bits de l'adresse.

Q26) Nombre de bits de *tag*, *set* et *offset*.

$B = 2$, $K = 8$ et tag de longueur 22.

Q27) Correspondance des 32 bits d'une adresse avec *tag*, *set* et *offset*.

Le format d'une adresse est TTTTTTTTTTTTTTTTTTTTTTSSSSSSSSOO

Pour chaque accès mémoire effectué par le programme (limitez-vous aux 6 premières itérations de la boucle), indiquez à quelle **adresse** a lieu l'accès, s'il s'agit d'un ***hit*** ou d'un ***miss*** et quelle **ligne** de la cache est accédée. On considère que la cache est de type *no-write allocate*, ce qui signifie que lorsque l'on écrit à une adresse qui n'est pas située en cache, la cache n'est pas mise à jour. On considère que les tableaux A, B et C sont alloués en mémoire dans cet ordre et en commençant à l'adresse 0.

Par exemple, lors de la première itération, la première cellule du tableau B est accédée. Son adresse est 32. Dérivez-en les valeurs de *tag*, *set* et *offset*. Déterminez sur cette base quelle est la ligne de la cache dans laquelle chercher. Ici, il s'agira d'un *(compulsory) miss* car c'est la première fois que cette adresse est lue. Des accès à la mémoire sont donc nécessaires pour charger la ligne de cache correspondante avec les données situées à l'adresse de B. Attention, il y a plusieurs octets dans une ligne de cache.

Q28) Exécution des 6 premières itérations de la boucle.

Supposons d'abord $N = 32$:
Les tableaux `A`, `B` et `C` sont situés en mémoire aux adresses respectives suivantes : 0, $N = 32$ et $2N = 64$.

Itération	Accès	Adresse	(t, s, o)	Opération
1	lire `B[0]`	32	$(0, 8, 0)$	*miss* ; charger 32..35 en ligne 8
	lire `C[0]`	64	$(0, 16, 0)$	*miss* ; charger 64..67 en ligne 16
	écrire `A[0]`	0	$(0, 0, 0)$	pas dans la cache
2	lire `B[1]`	33	$(0, 8, 1)$	*hit* ; déjà en ligne 8
	lire `C[1]`	65	$(0, 16, 1)$	*hit* ; déjà en ligne 16
	écrire `A[1]`	1	$(0, 0, 1)$	pas dans la cache
3	lire `B[2]`	34	$(0, 8, 2)$	*hit* ; déjà en ligne 8
	lire `C[2]`	66	$(0, 16, 2)$	*hit* ; déjà en ligne 16
	écrire `A[2]`	2	$(0, 0, 2)$	pas dans la cache
4	lire `B[3]`	35	$(0, 8, 3)$	*hit* ; déjà en ligne 8
	lire `C[3]`	67	$(0, 16, 3)$	*hit* ; déjà en ligne 16
	écrire `A[3]`	3	$(0, 0, 3)$	pas dans la cache
5	lire `B[4]`	36	$(0, 9, 0)$	*miss* ; charger 36..39 en ligne 9
	lire `C[4]`	68	$(0, 17, 0)$	*miss* ; charger 68..71 en ligne 17
	écrire `A[4]`	4	$(0, 1, 0)$	pas dans la cache
6	lire `B[5]`	37	$(0, 9, 1)$	*hit* ; déjà en ligne 9
	lire `C[5]`	69	$(0, 17, 1)$	*hit* ; déjà en ligne 17
	écrire `A[5]`	5	$(0, 1, 1)$	pas dans la cache

Supposons ensuite $N = 1024$:
Les tableaux `A`, `B` et `C` sont désormais situés en mémoire aux adresses respectives suivantes : 0, $N = 1024$ et $2N = 2048$.

Itération	Accès	Adresse	(t, s, o)	Opération
1	lire `B[0]`	1024	$(1, 0, 0)$	*miss* ; charger 1024..1027 en ligne 0
	lire `C[0]`	2048	$(2, 0, 0)$	*miss* ; charger 2048..2051 en ligne 0
	écrire `A[0]`	0	$(0, 0, 0)$	pas dans la cache
2	lire `B[1]`	1025	$(1, 0, 1)$	*miss* ; charger 1024..1027 en ligne 0
	lire `C[1]`	2049	$(2, 0, 1)$	*miss* ; charger 2048..2051 en ligne 0
	écrire `A[1]`	1	$(0, 0, 1)$	pas dans la cache
3	lire `B[2]`	1026	$(1, 0, 2)$	*miss* ; charger 1024..1027 en ligne 0
	lire `C[2]`	2050	$(2, 0, 2)$	*miss* ; charger 2048..2051 en ligne 0
	écrire `A[2]`	2	$(0, 0, 2)$	pas dans la cache
4	lire `B[3]`	1027	$(1, 0, 3)$	*miss* ; charger 1024..1027 en ligne 0
	lire `C[3]`	2051	$(2, 0, 3)$	*miss* ; charger 2048..2051 en ligne 0
	écrire `A[3]`	3	$(0, 0, 3)$	pas dans la cache
5	lire `B[4]`	1028	$(1, 1, 0)$	*miss* ; charger 1028..1031 en ligne 1
	lire `C[4]`	2052	$(2, 1, 0)$	*miss* ; charger 2052..2055 en ligne 1
	écrire `A[4]`	4	$(0, 1, 0)$	pas dans la cache
6	lire `B[5]`	1029	$(1, 1, 1)$	*miss* ; charger 1028..1031 en ligne 1
	lire `C[5]`	2053	$(2, 1, 1)$	*miss* ; charger 2052..2055 en ligne 1
	écrire `A[5]`	5	$(0, 1, 1)$	pas dans la cache

Quel seront les *hit ratio* et *miss ratio* atteints avec ce programme et cette cache ?

Q29) *hit ratio* et *miss ratio*.

Avec $N = 32$, il y aura deux *misses* toutes les 4 itérations (8 accès). Il s'agit uniquement de *compulsory misses*. Par conséquent, le *miss ratio* vaudra $\frac{2}{8} = 25\%$ et le *hit ratio* $\frac{6}{8} = 75\%$.
Avec $N = 1024$, tous les accès sont des *misses*. Le *miss ratio* vaut donc 100% et le *hit ratio* 0 %. Parmi les *misses*, 25 % sont des *compulsory misses* alors que les 75% restant sont des *conflict misses*.

Comment pourriez-vous modifier le programme pour améliorer ses performances, tout en gardant la même cache ?

Q30) Programme offrant de meilleures performances.

Le programme ci-dessous évite les conflits en cache. Pour cela, il ne procède plus octet par octet, mais par 4 octets (longueur d'une ligne de cache), en lisant d'abord `B[i*4]` à `B[i*4+3]` dans des variables locales (registres). La première lecture en `B[i*4]` devrait causer un miss qui charge `B[i*4]` à `B[i*4+3]` en cache. Les lectures de `B[i*4+1]` à `B[i*4+3]` sont donc des hits.

```
1  #define N 1024
2  char A[N], B[N], C[N];
3
4  int main()
5  {
6    for (int i = 0; i < N/4; i++) {
7      char tmp[4];
8      for (int j = 0; j < 4; j++)
9        tmp[j] = B[i*4 + j];
10     for (int j = 0; j < 4; j++)
11       A[i*4 + j] = tmp[j] + C[i*4 + j];
12   }
13   return 0;
14 }
```

Comment pourriez-vous modifier le type de cache pour améliorer les performances, tout en gardant le même programme, la même taille de cache et la même taille de ligne de cache ?

Q31) Autre type de cache.

Le problème rencontré par le programme initial est qu'il faut pouvoir maintenir `B[i]` et `C[i]` simultanément en cache.
Une cache *fully-associative* apporterait une solution facile. Cependant, il faut garder à l'esprit qu'une cache *fully-associative* est coûteuse en raison de l'usage d'un comparateur par ligne pour comparer tous les *tags* simultanément.
Une meilleure solution consiste donc à utiliser une cache *set-associative*. Une telle cache permet de garder S *tags* différents dans un même *set*. Dans ce cas-ci, une cache 2-way set-associative ferait l'affaire.

Si vous avez opté pour une cache *fully-associative* ci-dessus, retournez à la question et proposez encore un autre type de cache.

2.5 Types de misses

Les *cache misses* peuvent être de plusieurs types : *compulsory*, *capacity* et *conflict*. Pour chaque type de *cache miss*, expliquez pourquoi il peut se produire et comment des changements à l'architecture de la cache peuvent réduire leur fréquence.

Q32) *Compulsory miss*.

Q33) *Capacity miss*.

Q34) ***Conflict miss.***

2.6 Simulation de caches

On considère deux caches différentes. La stratégie de remplacement est *least-recently-used* (LRU), i.e. la ligne de cache la plus anciennement accédée est éjectée.

— **Cache 1** : *4-way-associative* 128 octets, 16 lignes de 8 octets
— **Cache 2** : *direct mapped* 128 octets, 16 lignes de 8 octets

Le processeur effectue des lectures d'un octet aux adresses 16-bits suivantes et dans l'ordre indiqué : `0xA3C9`, `0xA3CB`, `0xB5EA`, `0xB5E1`, `0xB7E5`, `0xB9C9`, `0xA3C9`, `0xB5E1`, `0xB5EA`, `0xB5E1`, `0x12AA`, `0x122A`, `0xA3C8`, `0xA3CE`

Déterminez le *hit ratio* obtenu avec les deux caches.

En préliminaire à cette question, observons la structure des adresses dans le cas de chaque type de cache.

Adresse	Adresse en binaire	Cache 1 $B=3, K=2$ TTTTTTTTTTSSOOO	Cache 2 $B=3, K=4$ TTTTTTTTSSSSOOO
0xA3C9	1010001111001001	(0x51E,1,1)	(0x147,9,1)
0xA3CB	1010001111001011	(0x51E,1,3)	(0x147,9,3)
0xB5EA	1011010111101010	(0x5AF, 1, 2)	(0x16B,13,2)
0xB5E1	1011010111100001	(0x5AF, 0,1)	(0x16B,12,1)
0xB7E5	1011011111100101	(0x5BF, 0, 5)	(0x16F,12,5)
0xB9C9	1011100111001001	(0x5CE, 1, 1)	(0x173,9,1)
0xA3C9	1010001111001001	(0x51E, 1, 1)	(0x147,9,1)
0xB5E1	1011010111100001	(0x5AF, 0, 1)	(0x16B,12,1)
0xB5EA	1011010111101010	(0x5AF, 1, 2)	(0x16B,13,2)
0xB5E1	1011010111100001	(0x5AF, 0, 1)	(0x16B,12,1)
0x12AA	0001001010101010	(0x095, 1, 2)	(0x025,5,2)
0x122A	0001001000101010	(0x091, 1, 2)	(0x024,5,2)
0xA3C8	1010001111001000	(0x51E, 1, 0)	(0x147,9,0)
0xA3CE	1010001111001110	(0x51E, 1, 6)	(0x147,9,6)

Regardons ensuite ce qui se passe lors de chacun de ces accès, pour la Cache 1.

Adresse	Résultat	Set / ligne	Détails
0xA3C9	*compulsory miss*	1 / 0	chargement 0xA3C8..0xA3CF
0xA3CB	*hit*	1 / 0	
0xB5EA	*compulsory miss*	1 / 1	chargement 0xB5E8..0xB5EF
0xB5E1	*compulsory miss*	0 / 0	chargement 0xB5E0..0xB5E7
0xB7E5	*compulsory miss*	0 / 1	chargement 0xB7E0..0xB7E7
0xB9C9	*compulsory miss*	1 / 2	chargement 0xB9C8..0xB9CF
0xA3C9	*hit*	1 / 0	
0xB5E1	*hit*	0 / 0	
0xB5EA	*hit*	1 / 1	
0xB5E1	*hit*	0 / 0	
0x12AA	*compulsory miss*	1 / 3	chargement 0x12A8..0x12AF
0x122A	*compulsory miss*	1 / 2	éjection + chargement 0x1228..0x122F
0xA3C8	*hit*	1 / 0	
0xA3CE	*hit*	1 / 0	

Puis pour la Cache 2.

Adresse	Résultat	Ligne	Détails
0xA3C9	*compulsory miss*	9	chargement 0xA3C8..F
0xA3CB	*hit*	9	
0xB5EA	*compulsory miss*	13	chargement 0xB5E8..F
0xB5E1	*compulsory miss*	12	chargement 0xB5E0..7
0xB7E5	*compulsory miss*	12	éjection + chargement 0xB7E0..7
0xB9C9	*compulsory miss*	9	éjection + chargement 0xB9C8..F
0xA3C9	*conflict miss*	9	éjection + chargement 0xA3C8..F
0xB5E1	*conflict miss*	12	éjection + chargement 0xB5E0..7
0xB5EA	*hit*	13	
0xB5E1	*hit*	12	
0x12AA	*compulsory miss*	5	chargement 0x12A8..F
0x122A	*compulsory miss*	5	éjection + chargement 0x1228..F
0xA3C8	*hit*	9	
0xA3CE	*hit*	9	

Q35) Nombre de hits Cache 1

7

Q36) Hit ratio Cache 1

50 %

Q37) Nombre de hits Cache 2

5

Q38) Hit ratio Cache 2

$\sim$ 35,7 %

Indiquez quels sont les adresses (tags) contenues dans les caches à la fin de la séquence de lecture ci-dessus.

Q39) Contenu Cache 1

	tag	*v*	données : 2^B octets 0	1	2	3	4	5	6	7
0	0x5AF		0xB5E0							
	0x5BF		0xB7E0							
1	0x51E		0xA3C8							
	0x5AF		0xB5E0							
	0x091		0x1228							
	0x095		0x12A8							
2										
3										

Q40) Contenu Cache 2

	tag	*v*	données : 2^B octets 0	1	2	3	4	5	6	7
0										
1										
2										
3										
4										
5	0x24		0x1228							
6										
7										
8										
9	0x147		0xA3C8							
10										
11										
12	0x16B		0xB5E0							
13	0x16B		0xB5E8							
14										
15										

2.7 Implémentation d'un comparateur

Les caches nécessitent des comparateurs pour tester si le *tag* d'une entrée correspond à l'adresse recherchée. Fournissez le circuit logique d'un comparateur de 8 bits.

Comparateur de 8-bits.