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 ?

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`

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

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

Q5) `short` à l'adresse `0x120`

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 ?

.....

.....

.....

.....

.....

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 }
```

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.

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.

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.

```
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*.**

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

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

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

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.**

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

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

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

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

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

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***

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

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

Q22) **`0xA3C9`**

Q23) **`0xA3CB`**

Q24) **`0xB5EA`**

Q25) **`0xB5E1`**

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*.

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

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.

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

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

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

Q30) **Programme offrant de meilleures performances.**

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.**

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*, 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.

Q35) Nombre de hits Cache 1

Q36) Hit ratio Cache 1

Q37) Nombre de hits Cache 2

Q38) Hit ratio Cache 2

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

Q39) Contenu Cache 1

Q40) Contenu Cache 2

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.