# UNIVERSITÉ PARIS SACLAY
# Master E3A – SETI

## T1 Architectures avancées
## TD2 Hiérarchie mémoire

### 1 Étiquettes et index de cache

Un processeur a 2 Go de mémoire principale.
On considérera les les cas suivant :

1. Cache de 2 Mo à correspondance directe et écriture simultanée avec des lignes de 16 octets
2. Cache de 4 Mo à correspondance directe, réécriture et lignes de 32 octets.
3. Cache de 4 Mo associatif 4 voies (4 lignes par ensemble), réécriture et lignes de 32 octets.

Pour ces différents caches, on demande :

**1.1** Quelle est la décomposition d'une adresse mémoire (figure 1) ? Donner le nombre de bits pour les parties étiquettes, index et déplacement dans la ligne.

*Rep :*
1. *Déplacement dans la ligne : 4, Index : 17 ($2^{21}/2^4$ bloc de cache), Étiquette : 31-17-4=10*
2. *Déplacement dans la ligne : 5, Index : 17 ($2^{22}/2^5$), Étiquette:31-17-5=9*
3. *Déplacement dans la ligne : 5, Index : 15($2^{-2} \times 2^{22}/2^5$), Étiquette:31-15-5=11*

| étiquette | index | déplacement dans la ligne |
|---|---|---|

FIGURE 1 – Décomposition d'une adresse mémoire

**1.2** Donner les différentes parties d'une ligne (bloc) de cache (figure 2). Combien y a-t-il de bits pour le contrôle, l'étiquette et la partie données ? Quel est le nombre total de bits du cache ? Par rapport à la partie données du cache, quel est le surcoût lié aux bits de contrôle et d'étiquette ?

| étiquette | ctl | ligne d'instruction ou de données |
|---|---|---|

FIGURE 2 – Ligne de cache

*Rep :* *Pour le contrôle, il faut (toujours) un bit de validité (mis à 0 au démarrage et validé quand la ligne contient de vraie données).*
*Pour les caches à réécriture, il faut un bit supplémentaire indiquant si la ligne a été modifiée.*
*(et pour les politiques de remplacement un peu sophistiquées des informations supplémentaires non considérées ici).*

1. *par ligne : étiq. 10, ctrl : 1, Ligne : 16×8 Total cache 2Mo+11×(2Mo/16)b=2M×8+11×128k (en bits)*
2. *par ligne : étiq. 9, ctrl : 2, Ligne : 32×8 Total cache 4Mo+11×(4Mo/32)b=4M×8+11×128k (en bits)*
3. *par ligne : étiq. 11, ctrl : 2, Ligne : 32×8 Total cache 4Mo+13×(4Mo/32)b=4M×8+13×128k (en bits)*

### 2 Caches données

On considère une architecture possédant un cache de données de 8K octets organisé en lignes de 32 octets. Les exercices suivants seront traités dans 2 cas : correspondance directe et associativité par ensembles de 2 lignes, avec pseudo-LRU. On considère des tableaux de 4096 flottants simple précision (32 bits), implantés aux adresses suivantes :

| X | Y | Z | X1 | Y1 | X2 | Y2 |
|---|---|---|---|---|---|---|
| 0x10000 | 0x14000 | 0x18000 | 0x1C000 | 0x20000 | 0x24000 | 0x28000 |

**2.1** Quels sont les éléments des tableaux `X` et `Y` qui peuvent occuper le mot 0 de la ligne 0 du cache ?

*Rep :* ***Correspondance directe*** *Dépl./ligne : 5, index 8. Une ligne sera dans le bloc d'adresse 0 ssi son index est 0 et donc son adresse multiple de* $2^{13}$*. Comme les données font 4 octets (`float`), les éléments 0 et 2048 de chaque tableau se retrouveront toujours dans le mot 0 de la ligne 0 .*
***Associativité*** *2 lignes/bloc Dépl./ligne : 5, index 7. Une ligne sera à dans le bloc d'adresse 0 ssi son index est 0 et son adresse multiple de* $2^{12}$*. Les éléments 0, 1024, 2048 et 3072 de chaque tableau peuvent se retrouver dans le mot 0 de la ligne 0 (ou de la ligne 1) de l'ensemble 0.*

**2.2** Combien de défauts de cache de données par itération interviennent dans chacune des boucles $b_1 \cdots b_4$ ci-dessous ? On suppose que les variables scalaires sont toujours en registre.

$b_1$

```
for(i=0;i<N;i++)
 S+=X[i]*Y[i];
```

$b_2$

```
for(i=0;i<N;i++){
  S1+=X1[i]*Y1[i];
  S2+=X2[i]*Y2[i];
}
```

$b_3$

```
for(i=0;i<N;i++)
  S1+=X1[i]*Y1[i];
for (i=0;i<N;i++)
  S2+=X2[i]*Y2[i];
```

$b_4$

```
for(i=0;i<N;i++){
  S1+=X[i]*Y[i];
  S2+=X[i]*Z[i];
}
```

*Rep :* *Corrrespondance directe :*
*$b_1$) les lignes des éléments `X[i]` et `Y[i]` sont à la même adresse quel que soit `i`. 2 défauts de cache/itération*
*$b_2$) 4 défauts de cache/itération*
*$b_3$) 2 défauts/itération*
*$b_4$) 3 défauts/itération*
*Associativité 2 voies*
*$b_1$) Du fait de l'associativité, `X[i]` et `Y[i]` se retrouvent dans le même bloc, mais celui-ci contient 2 lignes. Deux défauts de cache seulement en début de ligne (`i` multiple de 8). 0.25 defaut/itération*
*$b_2$) `X1[i]`, `X2[i]`, `Y1[i]` et `Y2[i]` sont tous à la même adresse. Tout accès fait un défaut. 4 défauts de cache/iteration*
*$b_3$) Identique à $b_1$ (et donc préférable à $b_2$ pour la même fonctionnalité).*
*$b_4$) En début de ligne défaut de cache sur `X[i]`. L'accès à `Y[i]` fait un défaut de cache qui va dans la deuxième ligne de l'ensemble. Le nouvel accès à `X[i]` est un succès. L'accès à `Z[i]` est un défaut ; du fait de la politique de remplacement LRU, c'est `Y[i]` qui est remplacé. Donc, à chaque itération défauts de cache sur `Y[i]` et `Z[i]` et succès sur `X[i]` (sauf en début de ligne). 2.125 défauts de cache/itération*

## 3 Caches instructions

Un processeur a un jeu d'instructions RISC, avec des instructions de longueur fixe d'un mot. Il a un cache instructions de 2 Kmots, avec des lignes de 8 mots. Il utilise la correspondance directe. Il exécute le code de la Figure 3, constitué de deux boucles imbriquées. Les seuls branchements du programme sont les deux branchements de boucle, aux adresses 239 et 1200.
Le temps pour un succès cache est $t$ et un défaut de cache coûte $8t$.

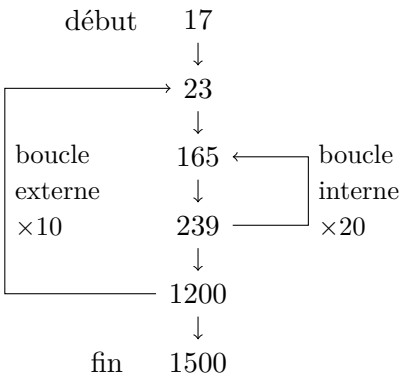


FIGURE 3 – Programme

**3.1** En négligeant l'effet des défauts de caches pour les données, quel est le temps d'exécution du programme de la Figure 3.

*Rep :* *Le cache comprend 256 lignes et chaque ligne contient 8 instructions.*
*La première instruction est à l'adresse 17 et dans la ligne 2 du cache.*
*La dernière est à l'adresse 1500 et dans la ligne 187 du cache.*
*Il n'y a aucun conflit. Défauts uniquement au démarrage. : 186 défauts de cache.*

*Le nombre d'instructions exécuté est :*
*17 à 22 et 1201 à 1500 exécutées 1 fois : 306*
*23 à 164 et 240 à 1200 exécutées 10 fois, soit (142+961)*10 =11030 instructions*
*165 à 269 exécutées 10*20 fois, soit 75*200 instruction = 15000*

*T exécution : (306+11030+15000)t+186*8t=27824t*

**3.2** Reprendre la question précédente en supposant un cache de 1 Kmots avec correspondance directe, puis l'associativité 2 voies (2 lignes par ensemble)

*Rep :* *Dans le cas d'un cache de 1k mot, tout le programme ne tient pas dans le cache (128 lignes de 8 instr). Le programme utilise pour les instructions 17-1023 les lignes 2-127 et pour les instructions 1024-1500 les lignes 1 à 59. Conflit sur les lignes 2 à 59.*
*Analyse des défauts de cache :*
*Un défaut lors du premier passage pour chacun des blocs*
*Pas de défaut dans la boucle interne*

*Lors du 2ème passage dans la boucle externe, les blocs 2 à 22 (correspond au instructions 1040-1200) auront remplacé des blocs existants (des instructions 23-183). 2*21 défaut par itération → 21*2*9=378 défauts.*
*Total : 186+378=564 défauts*
*Temps : 16336t+564*8t=30848T*

*Cache associatif 2 voies de 1k mots => 64 blocs dans le cache (de 2 lignes chacun) Les conflits sont plus nombreux. Notamment les blocs 2 à 22 seront utilisés par les instructions d'adresse 23 à 183, 528 à 695 et 1040 à 1207.*
*Défaut lors du premier accès à chaque ligne d'instructions.*
*Deuxième itération de la boucle externe (et suivante). Pour les blocs 2 à 22, a chaque nouvel accès, il y aura un défaut de cache du fait du mécanisme LRU qui enlève la ligne la plus ancienne. 3 défauts/itération pour chacune des lignes 2 à 22, soit 21*3*9 défauts (567) Soit un total de défauts de 186+567=753*
*Temps d'exécution : 26336t+753*8t=32360*

## 4 Mémoire virtuelle et TLB

Soit le code suivant de multiplication de matrices.

```
1  float a[1024][1024], b[1024][1024], c[1024][1024];
2  multiply()
3  {
4    int i, j, k;
5    for(i = 0; i < 1024; i++)
6      for(j = 0; j < 1024; j++){
7        c[i][j]=0.0;
8        for(k = 0; k < 1024; k++)
9          c[i][j] += a[i][k] * b[k][j];
10     }
11 }
```

On suppose que le code binaire pour exécuter cette fonction tient dans une page de 4 Ko et que la pile tient elle aussi dans une page.

**4.1** Quel est le nombre de défauts de TLB avec un TLB de 8 entrées utilisant LRU comme politique de remplacement ?.

NB : Les pages pour le code binaire et pour la pile seront en permanence en mémoire, et les entrées correspondantes du TLB seront donc constamment utilisées. 6 entrées du TLB restent disponibles pour l'exécution du programme.

*Rep :* *Au sein de la boucle interne, pour exécuter*
*`c[i][j] += a[i][k] * b[k][j];`*
*il faut 3 entrées de TLB valides pour `a`, `b` et `c`. Chaque entrée de TLB permet d'adresser 4ko, donc 1k `floats` (et donc une ligne de chaque matrice).*

*Défauts de TLB :*
*ligne 7 sur `c` pour chaque nouvelle valeur `i` (1k TLB miss)*
*ligne 9 sur `a` idéalement un TLB miss par nouvelle valeur de `i`. Comme `a` est utilisé systématiquement, le mécanisme LRU assure que l'entrée de TLB va rester pour les valeurs successives `j`. (1k TLB miss)*
*ligne 9 sur `b` pour chaque nouvelle valeur de `k` (1k*1k*1k TLB miss !!)*
*Soit un total de 1024*(2+1024*1024)=1073743872 défauts de TLB*

**4.2** Reprendre la question précédente en utilisant l'algorithme ikj (optionnel)

```
1  float a[1024][1024], b[1024][1024], c[1024][1024];
2  multiply_ikj()
3  {
4    int i, j, k;
5    for(i = 0; i < 1024; i++)
6      for(k = 0; k < 1024; k++){
7        float aa = a[i][k];
8        for(j = 0; j < 1024; j++)
9          c[i][j] += aa * b[k][j];
10     }
11 }
```

*Rep :* *Ligne 7 `a[i][k]` va faire un défaut de TLB pour chaque nouvelle valeur de `i`. (1k defaut)*
*Pour la ligne 9, `b[k][j]` va faire un défaut de TLB pour chaque nouvelle valeur de `k` (1k*1k défauts)*
*`c[i][j]` va faire un défaut pour chaque nouvelle valeur de `i` (1k défauts).*
*Soit un total de 1024*(2+1024)=1050624 défauts de TLB (1000 fois inférieur au cas précédent !).*