UNIVERSITÉ PARIS SACLAY
Master E3A – SETI

Examen Architecture T1
Décembre 2020

3h – Documents autorisés pour la deuxième partie de l'examen.

Partie 2

1 Optimisation de boucles

On utilise le processeur superscalaire défini dans l'annexe 1, ou sa version scalaire.

Soit la boucle suivante écrite en assembleur, qui travaille sur des tableaux de `floats` `X[256]` et `Y[256]`. Au démarrage `R1` contient l'adresse de `X[0]` et `R2` contient l'adresse de `Y[0]`. `R4` contient la valeur 256. `F0` contient la valeur 0.5.

```
1       ADDI R4,R4, -2
2   Boucle:
3       LF   F1, 0(R1)
4       LF   F2, 8(R1)
5       FADD F1,F1,F2
6       FMUL F1,F1,F0
7       SF   F1, 4(R2)
8       ADDI R1, R1,4
9       ADDI R2, R2,4
10      ADDI R4, R4,-1
11      BNE  R4, R0, Boucle
```

1.1 Donner le programme C équivalent.

Rép. :

```
float X[256],Y[256];
for(i=1; i$<$255; i++)
  Y[i]=(X[i-1]+X[i+1])*0.5f;
```

1.2 Indiquer les dépendances du programme et l'optimiser.

Rép. : Dépendance du programme :

- *RAW sur `F2` entre 4 et 5*
- *RAW (et WAW) sur `F1` entre 5 et 6*
- *RAW sur `F1` entre 6 et 7*
- *RAW sur `R4` entre 10 et 11 (résolu sans suspension)*

Donner l'exécution cycle par cycle de la boucle avant et après optimisation dans la version scalaire du processeur. Quel est, en nombre de cycles, le temps d'exécution par itération de la boucle ?

Rép. :

```
1       ADDI R4,R0, -2
2   Boucle :
3       LF F1, 0(R1)
4       LF F2 , 8(R1)
5       NOP
6       FADD F1,F1,F2
7       NOP
8       NOP
9       FMUL F1,F1,F0
10      NOP
11      NOP
```

```
12      SF F1, 4(R2)
13      ADDI R1,R1,4
14      ADDI R2,R2,4
15      ADDI R4,R4,-1
16      BNE R4, Boucle
```

Soit 14 cyles/itération.
Dépendances : D'ou la version optimisée en déplaçant les `addi` *de gestion de boucle :*

```
1       ADDI R4,R0, -2
2 Boucle :
3       LF F1, 0(R1)
4       LF F2, 8(R1)
5       ADDI R1,R1,4
6       FADD F1,F1,F2
7       ADDI R2,R2,4
8       ADDI R4,R4,-1
9       FMUL F1,F1,F0
10      NOP
11      NOP
12      SF F1, 0(R2)
13      BNE R4, Boucle
```

11 cycles

1.3 On considère maintenant la version surperscalaire du processeur. Donner l'exécution cycle par cycle de la boucle optimisée en plaçant les instructions dans les différents pipelines E0, E1, FA et FM comme sur le tableau ci-dessous. Quel est, en nombre de cycles, le temps d'exécution par itération de la boucle ?

cycles	E0	E1	FA	FM
1				
2				
...				

Rép. :

cycles	*E0*	*E1*	*FA*	*FM*
1	*LF F1, 0(R1)*	*LF F2, 8(R1)*		
2	*ADDI R1,R1,4*	*ADDI R2,R2,4*		
3	*ADDI R4,R4,-1*		*FADD F1,F1,F2*	
4				
5				
6				*FMUL F1,F1,F0*
7				
8				
9	*SF F1, 0(R2)*	*BNE R4, Boucle*		

9 cycles

1.4 Toujours dans la version superscalaire, réaliser un déroulage de boucle d'ordre 4 sur le programme précédent. Donner l'exécution cycle par cycle de la boucle en indiquant l'occupation des différents pipelines.

Rép. : Déroulage d'ordre 4

```
1       ADDI R4,R0, -2
2 Boucle :
3       LF F1, 0(R1)
4       LF F2, 4(R1)
5       LF F3, 8(R1)
6       LF F4, 12(R1)
7       LF F5, 16(R1)
8       LF F6, 20(R1)
9       ADDI R1,R1,16
10      ADDI R2,R2,16
11      ADDI R4,R4,-4
12      FADD F1,F1,F3
13      FADD F2,F2,F4
14      FADD F3,F3,F5
15      FADD F4,F4,F6
16      FMUL F1,F1,F0
```

```
17        FMUL F2,F2,F0
18        FMUL F3,F3,F0
19        FMUL F4,F4,F0
20        SF F1, -12(R2)
21        SF F2, -8(R2)
22        SF F3, -4(R2)
23        SF F4, 0(R2)
24        BNE R4, Boucle
```

(Rem : on peut même supprimer certains LF, mais le code est un peu plus compliqué sans gain en performances)

cycles	E0	E1	FA	FM
1	LF F1, 0(R1)	LF F2, 4(R1)		
2	LF F3, 8(R1)	LF F4, 12(R1)		
3	LF F5, 16(R1)	LF F6, 20(R1)	FADD F1,F1,F3	
4	ADDI R1,R1,16	ADDI R2,R2,16	FADD F2,F2,F4	
5	ADDI R4,R4,-4		FADD F3,F3,F5	
6			FADD F4,F4,F6	FMUL F1,F1,F0
7				FMUL F2,F2,F0
8				FMUL F3,F3,F0
9	SF F1, -12(R2)			FMUL F4,F4,F0
10	SF F2, -8(R2)			
11	SF F3, -4(R2)			
12	SF F4, 0(R2)	BNE R4, Boucle		

12 cy/4 itérations

2 Caches

On suppose qu'un processeur a un cache données de 64 Ko, avec des lignes de 64 octets. Le cache utilise la réécriture avec écriture allouée (il y a des défauts de cache en écriture) Le processeur a des adresses sur 32 bits.

2.1 Quel est pour ce cache le nombre de bits pour le déplacement dans la ligne, le nombre de bits d'index et le nombre de bits d'étiquette pour un cache à correspondance directe ?

Rép. : *64ko et 64o/line => 1k ligne*
1k ligne et correspondance directe => Index=10bits
64o/ligne => deplacement=6bits
Etiquette=32-6-10=16

2.2 `X` est un vecteur de `floats`. Sachant que `X[0]` est à l'adresse 1000 0000_H, dans quelles lignes du cache vont les flottants `X[0]` et `X[2048]` ?

Rép. : *`X[0]` est à l'adresse 1000 0000_H et son index vaut 0*
*`X[2048]` est à l'adresse 1000 0000_H +4*2048=1000 2000H=... 0010000000 000000_b et son index vaut 80h=128*

2.3 Quel est le nombre total de défauts de cache lors de l'exécution des trois boucles ci-dessous pour le cache à correspondance directe ?

```
float X[4096] ;
/*(A)*/
for (i=0 ; i<2048 ; i++)
   S+= X[i];
/*(B)*/
for (i=0 ; i<2032 ; i++)
   S+= X[i] + X[i+16];
/*(C)*/
for (i=0 ; i<2048 ; i++)
   S+= X[i] + X[i+2048];
```

Rép. : *Boucle A : A l'itération 0, défaut de cache et chargement de la ligne 0 du cache avec les 64 octets `X[0]...X[15]`.*
Pas de défaut jusqu'à l'itération 15.
Pour `i=16` nouveau défaut.
1 défaut toutes les 16 itérations, 128 défauts en tout.

Boucle B : A l'itération 0, 2 défauts de cache et chargement dans la ligne 0 du cache de `X[0]...X[15]` et dans la ligne 1 de `X[16]...X[31]`.
Pas de défaut jusqu'à l'itération 15.

Pour `i=16` un défaut et chargement de la ligne 2 du cache.
En tout 2+127=128 défauts.

Boucle C : A l'itération 0, 2 défauts de cache et chargement dans la ligne 0 du cache `X[0]...X[15]` et dans la ligne 128 de `X[2048]...X[2063]`.
Pas de défaut jusqu'à l'itération 15.
Pour `i=16` deux défauts et chargement des lignes 1 et 129 du cache.
Deux défauts pour 16 itérations soit 256 défauts en tout.

2.4 On suppose maintenant un cache de 8 Ko à correspondance directe avec des lignes de 64 octets et la boucle suivante :

```
float X[4096] ;
for (i=0 ; i<2048 ; i++)
   S+= X[i] + X[i+2048];
```

Quel est maintenant le nombre de défauts de cache lors de l'exécution de la boucle ?

Rép. : Le cache a maintenant 128 lignes et `X[2048]` est comme `X[0]` dans la ligne 0 du cache. A l'itération 0, défaut sur `X[0]` qui charge la ligne 0 du cache, puis défaut sur `X[2048]` qui écrase la ligne 0 du cache, et répétition pour tous les `i`
2 défauts/itération soit 4096 défauts en tout.

3 Prédiction de branchement

Soit le programme C suivant :

```
int array[1000] = { /* valeurs quelconques */ };
int sum1 = 0, sum2 = 0, sum3 = 0, sum4 = 0;
for (i = 0; i < 1000; i ++) // BR0 : branchement de boucle
{
  if (i % 4 == 0)      // BR1 : IF CONDITION 1
    sum1 += array[i]; // Pris
  else
    sum2 += array[i]; // Non pris

  if (i % 2 == 0)      // BR2 : IF CONDITION 2
    sum3 += array[i]; // Pris
  else
    sum4 += array[i]; // Non pris
}
```

On rappelle qu'en C, `x % y` renvoie le reste de la division entière de `x` par `y`.

Le but de l'exercice est de déterminer la qualité de la prédiction (% de prédictions correctes) pour les branchements BR0 (pris dans la boucle et non pris en sortie de boucle), et les branchements BR1 et BR2 à l'intérieur de la boucle, pour différents types de prédicteurs

3.1 Quelle est la qualité de la prédiction de chacun des branchements BR0, BR1 et BR2 pour un prédicteur 1 bit (pris/non pris ou P/N). On supposera que tous les prédicteurs sont initialisés à “non pris” (N).

Rép. :

BR0 prédiction	*N*	*P*	*P*	*P*	*P*	*P*	*P*	*P*	*P*	*P*	...	*P*	*P*
BR0 comportement	**P**	*P*	*P*	*P*	*P*	*P*	*P*	*P*	*P*	*P*	...	*P*	**N**

2 erreurs en tout, 2/1000 erreur/itération

BR1 prédiction	*N*	*P*	*N*	*N*	*N*	*P*	*N*	*N*	*N*	*P*	...	*N*	*N*
BR1 comportement	**P**	**N**	*N*	*N*	**P**	**N**	*N*	*N*	**P**	**N**	...	*N*	*N*

0.5 erreur/itération

BR1 prédiction	*N*	*P*	*N*	*P*	*N*	*P*	*N*	*P*	*N*	*P*	...	*N*	*P*
BR1 comportement	**P**	**N**	**P**	**N**	**P**	**N**	**P**	**N**	**P**	**N**	...	**P**	**N**

1 erreur/itération

3.2 Quelle est la qualité de prédiction de chacun des branchements BR0, BR1 et BR2 pour un prédicteur 2 bits (fortement pris P/faiblement pris p/faiblement non pris n/fortement non pris N). On supposera que tous les prédicteurs sont initialisés à “fortement non pris”.

Rép. :

BR0 prédiction	*N*	*n*	*p*	*P*	*P*	*P*	*P*	*P*	*P*	*P*	...	*P*	*P*
BR0 comportement	**P**	**P**	*P*	*P*	*P*	*P*	*P*	*P*	*P*	*P*	...	*P*	**N**

3 erreurs en tout, 3/1000 erreur/itération

BR1 prédiction	N	n	N	N	N	n	N	N	N	n	...	N	N
BR1 comportement	**P**	N	N	N	**P**	N	N	N	**P**	N	...	N	N

0.25 erreur/itération

BR2 prédiction	N	n	N	n	N	n	N	n	N	n	...	N	n
BR2 comportement	**P**	N	**P**	N	**P**	N	**P**	N	**P**	N	...	**P**	N

0,5 erreur/itération

3.3 Même question quand les prédicteurs sont initialisés à “faiblement non pris”.

Rép. :

BR0 prédiction	n	P	P	P	P	P	P	P	P	P	...	P	P
BR0 comportement	**P**	P	P	P	P	P	P	P	P	P	...	P	**N**

2 erreurs en tout, 2/1000 erreur/itération

BR1 prédiction	n	p	n	N	n	N	n	N	n	N	...	n	N
BR1 comportement	**P**	**N**	N	N	**P**	N	N	N	**P**	N	...	N	N

0.25 erreur/itération (+1/1000 erreur/itération)

BR2 prédiction	n	p	n	p	n	p	n	p	n	p	...	n	p
BR2 comportement	**P**	**N**	**P**	**N**	**P**	**N**	**P**	**N**	**P**	**N**	...	**P**	**N**

1 erreur/itération

3.4 On suppose maintenant que l’on utilise un prédicteur avec historique avec 2 bits d’historique (prise en compte des deux derniers branchements exécutés) et 2 bits de prédiction comme ci dessus. Il y aura ainsi un prédicteur différent pour chacun des historiques des deux banchements précédents. Comme BR0 est (presque) toujours pris, en pratique, pour BR1, il y aura un prédicteur différent suivant que BR2 (à l’étape $i-1$) a été pris ou non pris. De même pour BR2, il y aura deux prédicteurs suivant que BR1 (à l’étape i) a été pris ou non pris.

On suppose tous les prédicteurs initialisés à “fortement non pris” et tout l’historique des branchements intialisé à “non pris”.

Dans ces conditions, déterminer la précision de la prédiction pour BR1 et BR2.

Rép. :

BR1 :

Comportement BR1		P	N	N	N	P	N	N	N	...	P	N	N	N
Comportement BR2		P	N	P	N	P	N	P	N	...	P	N	P	N
Historique (BR2/i-1)	N	N	P	N	P	N	P	N	P	...	N	P	N	P
Préd. BR2/i-1 NP	N	**N**	**N**	N	**N**	N	**N**	N	**N**	...	**N**	N	**N**	N
Préd. BR2/i-1 P	N	N	N	**N**	N	**n**	n	**N**	N	...	N	**n**	n	**N**
Qualité		0	1	1	1	0	1	1	1	...	0	1	1	1

0.25 erreur/itération

BR2 :

Comportement BR2		P	N	P	N	P	N	P	N	...	P	N	P	N
Comportement BR1		P	N	N	N	P	N	N	N	...	P	N	N	N
Historique (BR1/i)	N	P	N	N	N	P	N	N	N	...	P	N	N	N
Préd. BR1/i NP	N	n	**N**	**n**	**N**	N	**N**	**n**	**N**	...	N	**N**	**n**	**N**
Préd. BR1/i P	N	**n**	n	n	n	**p**	p	p	p	...	**P**	P	P	P
Qualité		0	1	0	1	1	1	0	1	...	1	1	0	1

0.25 erreur/itération (+1/1000 à cause des premières étapes où la prédiction quand BR1/i est pris est différente de p ou P. Après, on prédira correctement que quand i est mutiple de 4 (BR1 pris), il est aussi multiple de 2 (BR2 pris).)

3.5 Montrer qu’un déroulage de boucle permet d’éviter d’effectuer des branchements. Donner une version en C de ce programme optimisé.

Rép. : On peut faire un déroulage d’ordre 4.

```
int array[1000] = { /* valeurs quelconques */ };
int sum1 = 0, sum2 = 0, sum3 = 0, sum4 = 0;
for (i = 0; i < 1000; i+=4)
{
  // i=0,4,...
    sum1 += array[i];
    sum3 += array[i];
  // i=1,5,...
    sum2 += array[i+1];
    sum4 += array[i+1];
```

```
    // i=2,6,...
      sum2 += array[i+2];
      sum3 += array[i+2];
    // i=3,7,...
      sum2 += array[i+3];
      sum4 += array[i+3];
}
```

Que l'on peut réécrire de la manière suivante (mais avec un bon compilateur, les deux sont équivalents en performances).

```
int array[1000] = { /* valeurs quelconques */ };
int sum1 = 0, sum2 = 0, sum3 = 0, sum4 = 0;
for (i = 0; i < 1000; i+=4)
{
    sum1 += array[i];
    sum2 += array[i+1]+array[i+2]+array[i+3];
    sum3 += array[i]+array[i+2];
    sum4 += array[i+1]+array[i+3];
}
```

4 Mémoire virtuelle et TLB

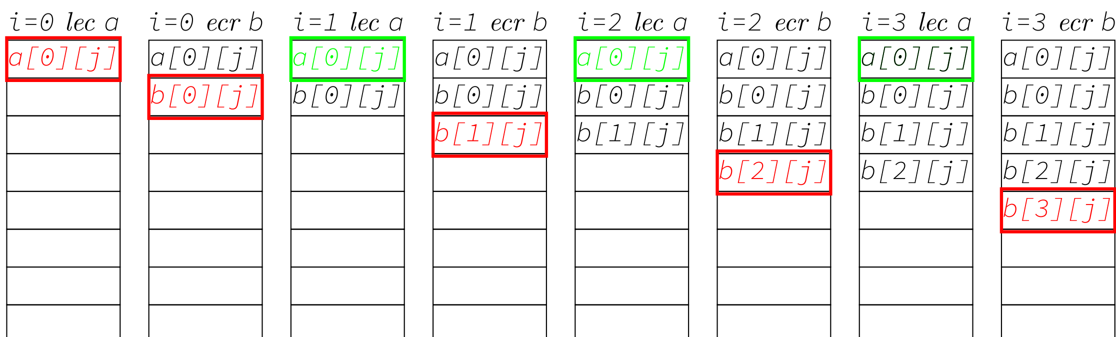
Soit le programme suivant de transposition de matrices :

```
float a[1024][1024], b[1024][1024];
...
int i, j;
for(i = 0; i < 1024; i++)
  for(j = 0; j < 1024; j++){
     b[j][i]=a[i][j];
```

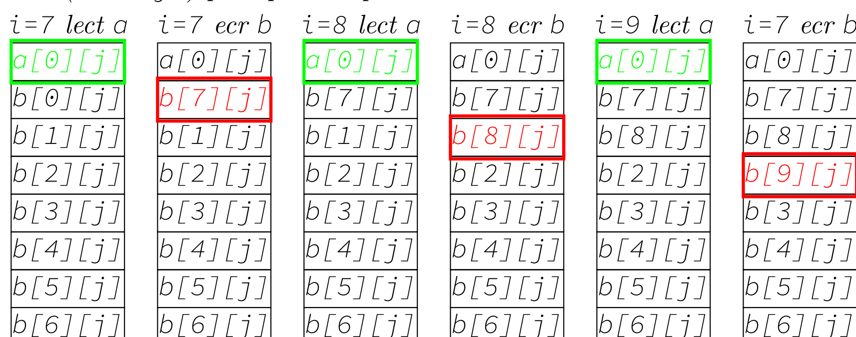
4.1 On considère un processeur dont les pages font 4ko et on supposera que les tableaux sont disposé successivement en mémoire et qu'ils ont leur élément 0 en début d'une page. On supposera que le TLB comprend 8 entrées disponibles pour les données et qu'il est totalement associatif avec un gestion des remplacements LRU.

Calculer le nombre de défauts de TLB lors de l'exécution de ce programme.

Rép. : *Une ligne de* `a` *et* `b` *fait 1024* `floats`*, soit 4ko, soit une page. On aura donc un changement de page à chaque changement de ligne.*



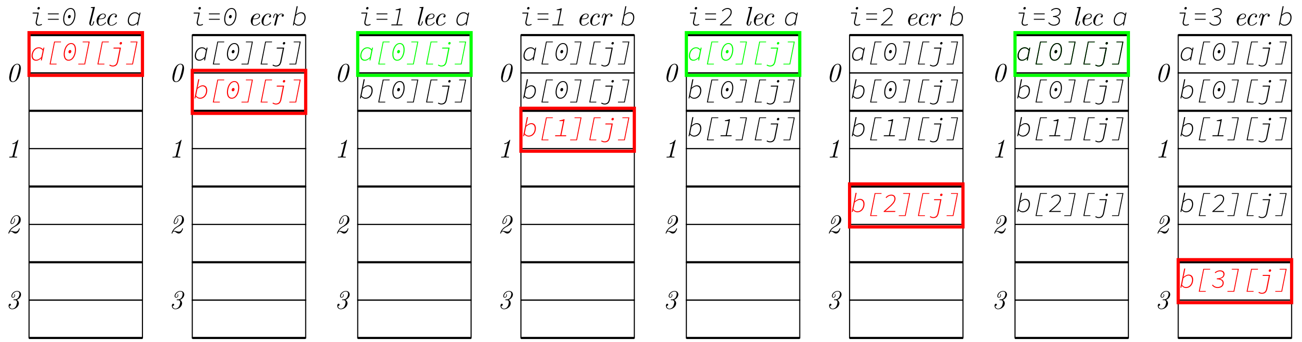
A l'itération `i=7`*, le TLB est plein. La politique LRU va remplacer l'entrée la moins récemment utilisée* (`b[0][j]`) *pour pouvoir placer* `b[7][0]`.



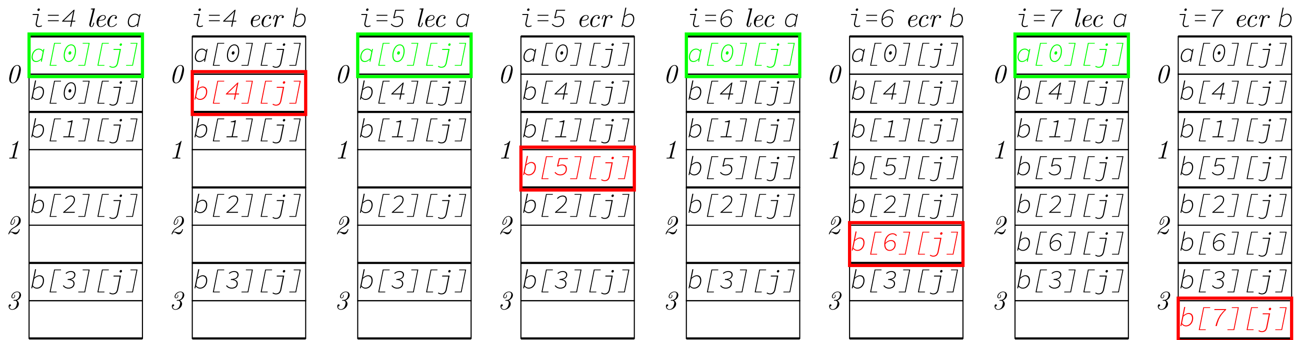
Nous aurons donc un défaut sur `a` *pour chaque nouvelle valeur de* `i` *(toutes les 1024 itérations), et un défaut de TLB sur* `b` *à chaque itération.*

4.2 Même question si le TLB comprend 4 ensembles de 2 voies. On pourra supposer que l'adresse de `a[0][0]` correspond à l'ensemble 0 du TLB.

Rép. : L'entrée de TLB de `a[0][j]` *sera dans le même ensemble 0 que celles de* `b[0][i]`, `b[4][i]`, *etc.*



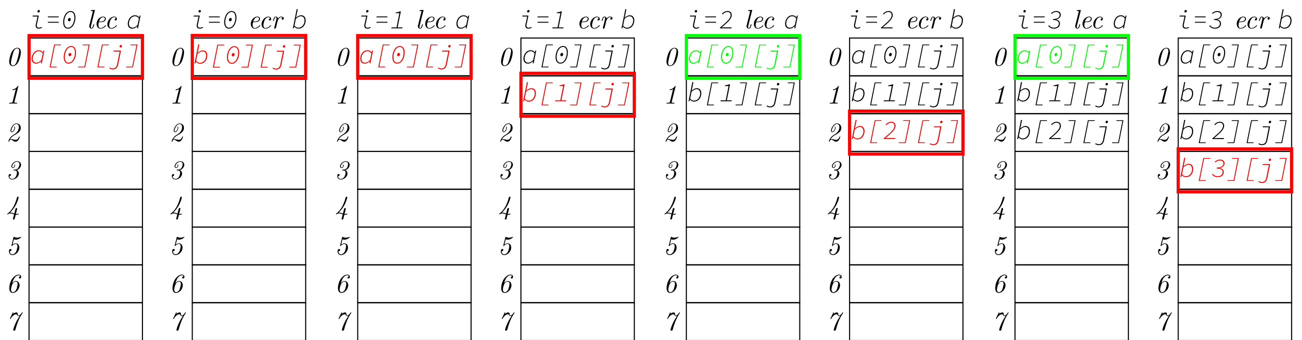
A l'itération 4, comme `a` *a été récemment lu, la politique LRU va remplacer* `b[0][j]` *par* `b[4][j]` *dans l'ensemble 0 du TLB sans créer de défaut sur* `a[0][j]`*.*



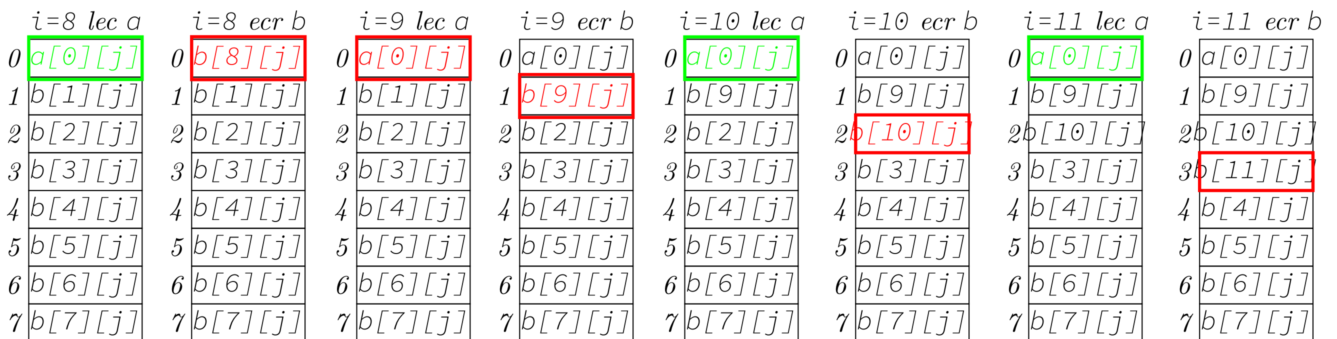
Même résultat que précédemment. Un défaut sur `a` *toutes les 1024 itérations, un défaut sur* `b` *à chaque itération.*

4.3 Même question avec un TLB à correspondance directe.

Rép. : `a[0][j]` *et* `b[0][j]` *vont se retrouver dans la même entrée 0 du TLB, ce qui va créer des défauts supplémentaires.*



A l'itération 8, c'est `b[8][j]` *qui va occuper l'entrée 0 du TLB.*



Donc un défaut de TLB sur `a` *toutes les 1024 itérations, un autre défaut sur* `a` *toutes les 8 itérations, et un défaut par itération sur* `b`*.*

5 Caches et associativité

On considère un cache à 4 entrées, avec des lignes de 1 octet. On veut comparer 3 politiques différentes : cache totalement associatif (4 voies) (TA), cache partiellement associatif (2 ensemble de 2 voies) (PA), cache à correspondance directe (CD). Pour les caches associatifs, on supposera que la politique de remplacement

est LRU. Dans ces différents cas, indiquer le contenu des différentes lignes du cache avec l'adresse associée à chaque ligne pour le motif d'accès 0, 1, 2, 3, 4, 0, 4, 0, 2, 1. On remplira les tableaux joints. Pour les caches associatifs, on mettra la dernière ligne accédée dans chaque ensemble en haut. Dans tous les cas, on indiquera si la dernière ligne accédée a conduit à un **succès** ou un **échec**.

Rép. :

cycle 1 : 0

	TA	*PA*	*CD*
0	*0 échec*	*0 échec*	*0 échec*
1			
2			
3			

cycle 2 : 1

	TA	*PA*	*CD*
0	*1 échec*	*0*	*0*
1	*0*		*1 échec*
2		*1 échec*	
3			

cycle 3 : 2

	TA	*PA*	*CD*
0	*2échec*	*2 échec*	*0*
1	*1*	*0*	*1*
2	*0*	*1*	*2 échec*
3			

cycle 4 : 3

	TA	*PA*	*CD*
0	*3 échec*	*2*	*0*
1	*2*	*0*	*1*
2	*1*	*3 échec*	*2*
3	*0*	*1*	*3 échec*

cycle 5 : 4

	TA	*PA*	*CD*
0	*4 échec*	*4 échec*	*4 échec*
1	*3*	*2*	*1*
2	*2*	*3*	*2*
3	*1*	*1*	*3*

cycle 6 : 0

	TA	*PA*	*CD*
0	*0 échec*	*0 échec*	*0 échec*
1	*4*	*4*	*1*
2	*2*	*3*	*2*
3	*1*	*1*	*3*

cycle 7 : 4

	TA	*PA*	*CD*
0	*4 hit*	*4 hit*	*4 échec*
1	*0*	*0*	*1*
2	*2*	*3*	*2*
3	*1*	*1*	*3*

cycle 8 : 0

	TA	*PA*	*CD*
0	*0 hit*	*0 hit*	*0 échec*
1	*4*	*4*	*1*
2	*2*	*3*	*2*
3	*1*	*1*	*3*

cycle 9 : 2

	TA	*PA*	*CD*
0	*2 hit*	*2 échec*	*0*
1	*0*	*0*	*1*
2	*4*	*3*	*2 hit*
3	*1*	*1*	*3*

cycle 10 : 1

	TA	*PA*	*CD*
0	*1 hit*	*0*	*0*
1	*2*	*0*	*1 hit*
2	*0*	*1 hit*	*2*
3	*4*	*3*	*3*

ANNEXE 1

Soit un processeur superscalaire à ordonnancement statique qui a les caractéristiques suivantes :

— les instructions sont de longueur fixe (32 bits)

— Il a 32 registres entiers (dont R0=0) de 32 bits et 32 registres flottants (de F0 à F31) de 32 bits.

— Il peut lire et exécuter 4 instructions par cycle.

— L'unité entière contient deux pipelines d'exécution entière sur 32 bits, soit deux additionneurs, deux décaleurs. Tous les court-circuits possibles sont implantés.

— L'unité flottante contient un pipeline flottant pour l'addition et un pipeline flottant pour la multiplication.

— L'unité Load/Store peut exécuter jusqu'à deux chargements par cycle, mais ne peut effectuer qu'un *load* et un `store` simultanément. Elle ne peut effectuer qu'un seul `store` par cycle.

— Le processeur dispose d'un mécanisme de prédiction de branchement qui permet de *brancher* en 1 cycle si la prédiction est correcte. Les sauts et branchements ne sont pas retardés.

La Table 1 donne les instructions disponibles et le pipeline qu'elles utilisent : E0 et E1 sont les deux pipelines entiers, FA est le pipeline flottant de l'addition et FM le pipeline flottant de la multiplication. Les instructions peuvent être exécutées simultanément si elles utilisent chacune un pipeline séparé. L'addition et la multiplication flottante sont pipelinées. La division flottante n'est pas pipelinée (une division ne peut commencer que lorsque la division précédente est terminée). L'ordonnancement est statique.

JEU D'INSTRUCTIONS (extrait)

Code	Instruction	Latence	Pipeline	Signification
LF	LF Fi, dép.(Ra)	2	E0 ou E1	Fi $\leftarrow$ M (Ra + dépl.16 bits signé)
SF	SF Fi, dép.(Ra)	0	E0	Fi $\rightarrow$ M (Ra + dépl.16 bits signé)
ADD	ADD Rd,Ra, Rb	1	E0 ou E1	Rd $\leftarrow$ Ra + Rb
ADDI	ADDI Rd, Ra, IMM	1	E0 ou E1	Rd $\leftarrow$ Ra + IMM (16 bits signé)
SUB	SUB Rd,Ra, Rb	1	E0 ou E1	Rd $\leftarrow$ Ra - Rb
FADD	FADD Fd, Fa, Fb	3	FA	Fd $\leftarrow$ Fa + Fb
FMUL	FMUL Fd, Fa, Fb	3	FM	Fd $\leftarrow$ Fa $\times$ Fb
BEQ	BEQ Ri, dépl	1	E1	si Ri=0 alors CP $\leftarrow$ NCP + depl
BNE	BNE Ri, dépl	1	E1	si Ri$\neq$ 0 alors CP $\leftarrow$ NCP + depl