MU5IN100 — Cache et hiérarchie des mémoires — TD 8

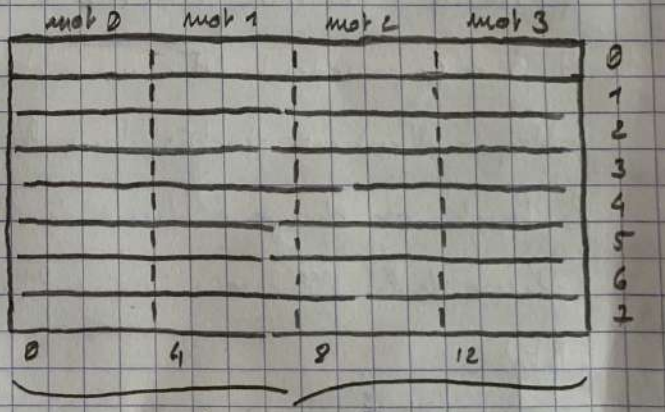
Exo 1 : Cache Direct-Mapped en Write-Through avec tampon d'écriture postée

On étudie un système contenant un cache d'instruction et un cache de donnée à correspondance directe, ils sont tout deux indépendants.
Les adresses sont sur 32 bits. La stratégie d'écriture en mémoire est de type Write Through : toute requête d'écriture émise par le processeur se traduit par une écriture en mémoire et le cache n'est mis à jour que si le bloc est présent dans le cache (cas de Hit).
Un tampon d'écriture postée permet d'éviter de geler le processeur lors des instructions d'écriture tant qu'il n'y a pas un trop grand nombre d'écritures consécutives.

Q1/ Structure des adresses : on considère des caches Direct Mapped ayant une capacité de 8 lignes de 4 mots.

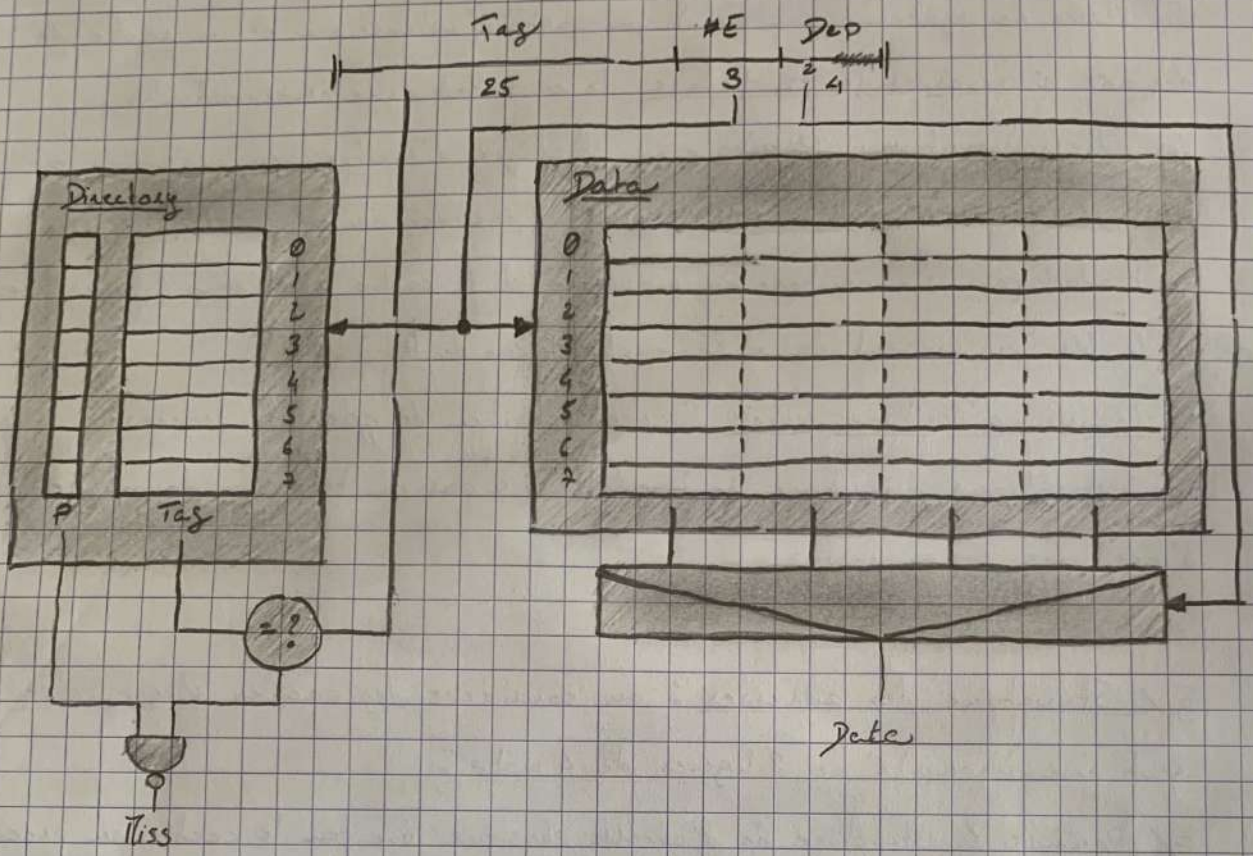
a/ Donnez la structure de l'adresse physique vue par le cache en précisant le nombre de bits dans chaque champ. Quel est le nombre total de lignes de cache dans l'espace d'adressage ?

Notre cache a une capacité de 8 lignes où chaque ligne correspond à N = 4 mots = 16 octets.
Le nombre de bits nécessaires pour décrire un déplacement dans une ligne/bloc est donc égal à log(N) donc 4 bits. De plus, comme nous sommes sur un cache Direct Mapped, son degré d'associativité est donc #deg-Ass = 1 Empl/Ens., donc on a $\#Ens = \frac{\#Empl}{\#deg\text{-}Ass} = \frac{8}{1} = 8$ ensembles, il faut donc $\log(\#Ens) = \log(8) = 3$ bits pour décrire le numéro d'ensemble d'un bloc. Pour le Tag, il suffit de compléter : $32 - \log(N) - \log(\#Ens) = 25$ bits.



b/ Donnez un schéma du cache mettant en évidence l'utilisation de chaque champ.

Schéma de l'utilisation de l'adresse par le cache



Q2/ Analyse d'une application : Soit le programme assembleur sur le sujet qui réalise l'addition vectorielle de deux vecteurs : C = A + B. Les 3 vecteurs A, B et C possèdent 20 composantes chacun et sont représentés par des tableaux d'entiers stockés dans le segment "data". Avant d'entrer dans la boucle, on charge l'adresse de l'élément A[0] dans le registre R8 et l'adresse de la 1ère case mémoire libre après le tableau A dans le registre R9.

```
.data
A:  .align 2
    .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
    .word 0, 12, 13, 14, 15, 16, 17, 18, 19, 20
    .space 48
B:  .word 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
    .word 111, 112, 113, 114, 115, 116, 117, 118, 119, 120
    .space 48
C:  .word 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
    .word 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
.text
    .align 2
    .globl main
    .ent main
main:   .set noreorder
        lui   R8, 0x1000
        addiu R9, R8, 80
loop:   lw    R10, 0(R8)
        lw    R11, 128(R8)
        addi  R8, R8, 4
        add   R10, R10, R11
        bne   R8, R9, loop
        sw    R10 256(R8)
exit:   ori   R2, R0, 10
        syscall
```

c/ L'adresse de base du segment "text" est 0x0040 0000. La 1ère instruction rangée à cette adresse est donc l'instruction associée à l'étiquette "main". Quelle est l'adresse de la 1ère instruction de la boucle (instruction associée à l'étiquette loop) ?

D'après la description en assembleur du segment "text", on a le tableau :

Etiquette	Instruction	Adresse
main	lui	0x0040 0000
	addiu	0x0040 0004
loop	lw	0x____ 0008
	lw	0x____ 000C
	addi	0x____ 0010
	add	0x____ 0014
	Bne	0x____ 0018
	Sw	0x____ 001C
exit	ori	0x____ 0020
	syscall	0x____ 0024

Les instructions sont codés sur 32 bits <=> 4 octets, donc il suffit d'incrémenter de 4 l'adresse d'une instr. pour passer à la suivante. L'adresse de la 1ère instr. de la boucle est 0x0040 0008.

d/ L'adresse de base du segment "data" est 0x1000 0000. Quelles sont les adresses des éléments A[0], B[0] et C[0] des trois tableaux ?

D'après la description en assembleur du segment "data", on a le tableau :

Etiquette	Data (Little-Endian) @ +3 +2 +1 +0	Adresse
A	0x 00 00 00 01	0x1000 0000
	0x 00 00 00 02	0x1000 0004
	...	...
	0x 00 00 00 14	0x1000 0050
	0x ø ø ø ø	0x1000 0054
	...	...
	0x ø ø ø ø	0x1000 007C
B	0x 00 00 00 65	0x1000 0080
	0x 00 00 00 66	0x1000 0084
	...	...
	0x 00 00 00 78	0x1000 00DD
	0x ø ø ø ø	0x1000 00D4
	...	...
	0x ø ø ø ø	0x1000 00FC
C	0x 00 00 00 00	0x1000 0100
	0x 00 00 00 00	0x1000 0104
	...	...
	0x 00 00 00 00	0x1000 0150

Notre boutisme est dit Little-Endian, c'est à dire que l'adresse désigne l'octet de poids faible et on progresse vers les octets de poids fort. Nos données sont bien alignés sur des adresses multiples de la taille de la donnée ; nous avons donc :

@A[0] = 0x1000 0000

@B[0] = 0x1000 0080

@C[0] = 0x1000 0100

Q3/ Analyse du fonctionnement du cache d'instructions :

a/ Représenter dans le tableau ci-dessous l'état du cache d'instructions à la fin de la 1ère itération de la boucle en supposant que le cache est vide avant l'exécution de la 1ère instruction du programme.

Tag (25 bits)	V	Mot 3	Mot 2	Mot 1	Mot 0	N°Empl
0x400000	1	Lw	Lw	Addiu	Lui	0
0x400000	1	Sw	Bne	Add	Addi	1
0x400000	1	?	?	Syscall	Ori	2
	0					3
	0					4
	0					5
	0					6
	0					7

Pour remplir le cache, il faut se baser sur la structure de l'adresse physique définie de l'instruction que l'on veut charger.

b/ Quelles instructions ont déclenché un Miss sur le cache d'instructions pour atteindre cet état ? Quel est l'état du cache à la fin de la 2ème itération ? Qu'en est-il à la fin de la 20ème itération ?

Lors de la 1ère itération, nous avons 3 Miss compulsifs qui nous remplissent le cache, il s'agit des instructions de la colonne "Mot 0", à savoir : Lui, Addi et Ori. De plus, comme nous n'avons aucun Miss de conflit lors des 1ère et 2ème itérations, nous aurons que des Hits jusqu'à la fin de la 20ème itération.

c/ Calculer le taux de Miss.

Sur un Mips pipeliné normal, on a $\#IFC = 2 + 20 \times 6 + 2$.

On a donc $\text{Taux Miss} = \frac{3}{124} \approx 0,024 \iff 2,4\ \%$ de Miss

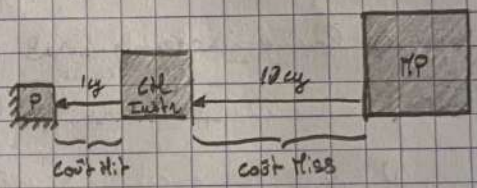
$$\text{Taux Miss}_{inst.} = \frac{\#\text{Miss}}{\#\text{Acces}_{inst.}}$$

avec $\#\text{Acces}_{inst.} = \#IFC$

h/ Le coût d'un Miss instruction (nbre de cycles de gel du processeur) est de 10 cycles. en déduire la durée d'exécution du programme (entre le branchement à la 1ère instruction du programme et la fin de la dernière instruction), si l'on néglige le coût des Miss sur le cache de données.

Le coût d'un Hit instruction est de 1 cycle, donc à chaque IFC nous avons au moins 1 cycle. Le coût d'un Miss instruction est la durée d'une transaction pour réaliser les transferts de la Mem. Prim vers le cache.

D'après les question precedentes, on a eu au total 3 Miss instr, donc la durée d'execution du programme est de $124 + 3*10 = 154$ cycles.



Q4/ Analyse du fonctionnement du cache de données :

i&j/ Représentez dans le tableau l'état du cache de données à la fin de la 1ère itération de la boucle, en précisant quelles instructions entraînent un Miss sur le cache de données et donc un gel du processeur. Faire de même pour la 20ème itération

Tag (25 bits)	V	Mot 3	Mot 2	Mot 1	Mot 0	N° Empl	
1 0 0 0 0 0	1	68	67	66	65	0	iter 1 à 4
1 0 0 0 0 0	1	6C	6B	6A	69	1	iter 5 à 8
1 0 0 0 0 0	1	70	6F	6E	6D	2	iter 9 à 12
1 0 0 0 0 0	1	74	73	72	71	3	iter 13 à 16
1 0 0 0 0 0	1	78	77	76	75	4	iter 17 à 20
	0					5	
	0					6	
	0					7	

Pour chaque emplacement : lw R10, 0(R8) provoque un Miss compulsif mais lw R11, 128(R8) provoque un miss de conflit qui écrase la valeur du load précédent. En write through les Sw ne font pas de Miss.

k/ Combien y a-t-il de Miss par iteration ? En déduire le nombre total de cycles de gel du processeur pour l'execution complète du programme, en tenant compte du fait qu'un Miss de donnée coûte 2 cycles de plus qu'un Miss d'instruction.

D'après les questions précédentes, un Miss d'instruction a un coût de 10 cycle, donc un Miss de donnée coûte 12 cycles.

A chaque itération, nous avons exactement 2 Miss provoqués par les lw, on a donc un total de $20 \times 2 = 40$ Miss pour 40 accès mémoires.

On a donc un Taux Miss$_{data}$ $= \frac{\#Miss}{\#Acces} = \frac{40}{40} = 100\%$ de Miss !

l/ Quelle est finalement la durée totale d'exécution du programme, en prenant compte tous les Miss sur les deux caches ? Calculez le CPI.

D'après les questions précédentes, on a 124 IFC sur un Mips normal pipeliné, 3 Miss d'instructions et 40 Miss de données sur tout le programme ; comme un Miss d'instruction coûte 10 cycles et un Miss de donnée en coûte 12, on a :

$\#Cycles = 124 + 3*10 + 40*12 = 124 + 510 = 634$ cycles au total.

Donc $CPI = \frac{\#Cycles}{\#IFC} = \frac{634}{124} \approx 5,1$ cycles/instructions.

Q6/ Optimisation du fonctionnement du cache de donnée :

m/ Pour cette application, quel pourcentage du temps d'execution les cycles de gel du processeur dû aux Miss de conflit sur le cache de données représentent-ils ?

Sur un total de 634 cycles, nous en avons $(40-1)*12 = 470$ dus aux conflits, on a donc $\frac{470}{634} \times 100 \approx 74,13\%$.

n/ Comment réduire le taux de Miss sur le cache de donnée sans modifier l'architecture matérielle, ni la structure du programme ?

Il suffit de déplacer la structure B d'un bloc vers le haut, c'est à dire faire commencer le tableau B à l'adresse 0x1000 0070.

o/ Recalculez la durée moyenne du programme avec la proposition faite :

Avec cette proposition, on a 2 Miss de données toutes les 4 itérations donc 10 Miss de données pour toute la boucle $\Rightarrow$ $\#C = 124 + 3*10 + 10*12 = 274$ cy.

Le CPI est donc de $\frac{274}{124} \approx 2,2$ cycles/instructions.

Exe 2 : Cache D.M / S.A en Write-Back :

On considère un cache de 1024 octets avec une taille de bloc (ou ligne) de 16 octets. La mémoire est adressée par octets et les adresses sont sur 32 bits.
Le cache est géré en écriture allouée (un défaut en écriture entraine un chargement du bloc correspondant), en réécriture (Write-Back, une donnée n'est modifiée en mémoire principale que lorsque la ligne de cache correspondante est éjectée du cache) et avec une politique de remplacement LRU (least Recently Used).

Q1/ Structure des adresses :

a/ Donner la structure de l'adresse physique vue par le cache en précisant le nombre de bits de chaque champ dans le cas d'un cache Direct Mapped, d'un cache Set-Associative de degré 2, pareil pour un degré 4 :

Taille_Bloc = 16 octets et Taille_cache = 1024 octets = 64 lignes de 16 octets.

- Dans le cas d'un cache en Direct-Mapping :
  - $Nb\_bits(Dep) = \log_2(Taille\_Bloc) = \log_2(16) = \log_2(2^4) = 4$ bits.
  - $Nb\_bits(\#Ens) = \log_2\left(\frac{Taille\_cache}{Taille\_Bloc * Deg\_A}\right) = \log_2\left(\frac{2^{10}}{2^4 * 2^0}\right) = 6$ bits.
  - $Nb\_bits(Tag) = 32 - Nb\_bits(Dep) - Nb\_bits(\#Ens) = 32 - 10 = 22$ bits.
- Dans le cas d'un cache en Set-Associativ de degré 2 :
  - $Nb\_bits(Dep) = 4$ bits ; $Nb\_bits(\#Ens) = \log_2\left(\frac{2^{10}}{2^4 * 2^1}\right) = 5$ bits ; $Nb\_bits(Tag) = 23$ bits.
- Dans le cas d'un cache en Set-Associativ de degré 4 :
  - $Nb\_bits(Dep) = 4$ bits ; $Nb\_bits(\#Ens) = 4$ bits ; $Nb\_bits(Tag) = 24$ bits.

b/ Donner un schéma du cache mettant en évidence l'utilisation de chaque champs dans le cas d'un cache associatif par ensemble de 2 blocs (degré 2).
Quelle est l'utilité des deux niveaux d'associativité ?

Pour le schéma, il suffit de se référé à celui de l'Exe 1, Q1, b en dédoublant le Répertoire et la section Data.
L'utilité des deux niveaux d'associativité est de réduire les Miss de conflits en exploitant la localité temporelle.

Q2/ Analyse d'une application : Soient X, Y et Z des tableaux de réels en double-précision (8 octets). X est implanté à partir de l'adresse 0x10010, Y à partir de 0x20210 et Z à partir de 0x020410. On étudie la boucle :

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

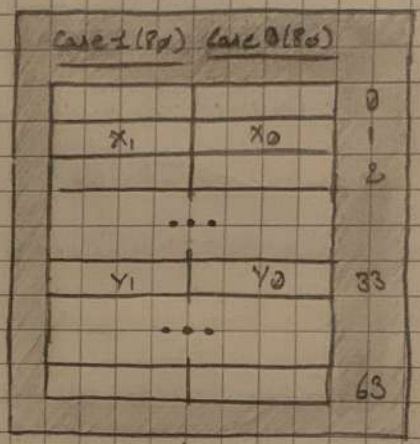
En supposant que N=3, indiquez pour chaque accès mémoire (lecture & écriture), si c'est un succès ou un échec (on traite les accès dans l'ordre X_i, Y_i, Z_i).

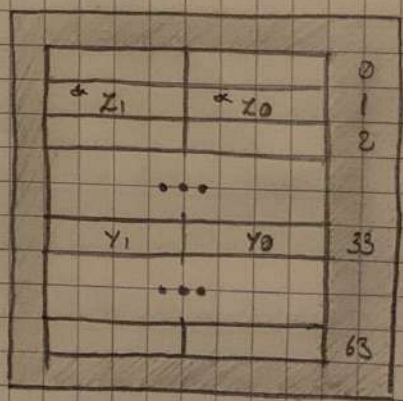
c/ Dans le cas d'un cache en Direct-Mapping en donnant pour chaque référence le numéro de la case de cache où elle est stockée.

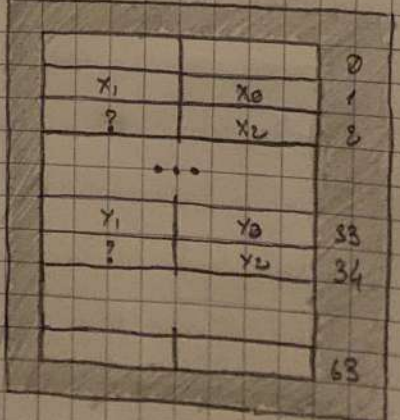
Structure_Adresse : 22, 6, 4

- Pour X : X_0 0x10010 — X_1 0x10018 — X_2 0x10020
- Pour Y : Y_0 0x20210 — Y_1 0x20218 — Y_2 0x20220
- Pour Z : Z_0 0x20410 — Z_1 0x20418 — Z_2 0x20420

Direct-Mapping







Effets des instructions et détail de (#Evs, Dep) :

Load X_0 → (0b 00 0001, 0b 0000) ⇒ Miss comp. clean.

Load Y_0 → (0b 10 0001, 0b 0000) ⇒ Miss comp. clean.

Store Z_0 → (0b 00 0001, 0b 0000) ⇒ Miss comp. clean.

Load X_1 → (0b 00 0001, 0b 1000) ⇒ Miss conf. dirty.

Load Y_1 → (0b 10 0001, 0b 1000) ⇒ Hit.

Store Z_1 → (0b 00 0001, 0b 1000) ⇒ Miss conf. clean.

Load X_2 → (0b 00 0010, 0b 0000) ⇒ Miss comp. clean.

Load Y_2 → (0b 10 0010, 0b 0000) ⇒ Miss comp. clean.

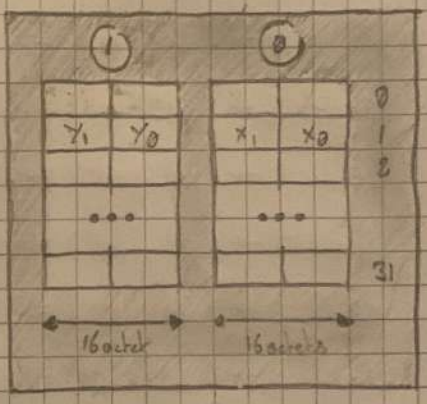
Store Z_2 → (0b 00 0010, 0b 0000) ⇒ Miss comp. clean.

On a 9 accès : 1 Hit et 8 Miss (dont 7 clean et 1 dirty).

d/ Dans le cas d'un cache en Set-Associatif de degré 2 en donnant pour chaque référence le numéro d'ensemble où elle est stockée.

Structure Adresse : 23, 5, 4

2-Way Associative



Effet des instructions et détail de (#Ens, Dep) :

Load X_0 → (0b 0 0001, 0b 0000) => Miss comp clean.

Load Y_0 → (0b 0 0001, 0b 0000) => Miss comp clean.

Store Z_0 → (0b 0 0001, 0b 0000) => Miss comp clean.

Load X_1 → (0b 0 0001, 0b 1000) => Miss conf clean.

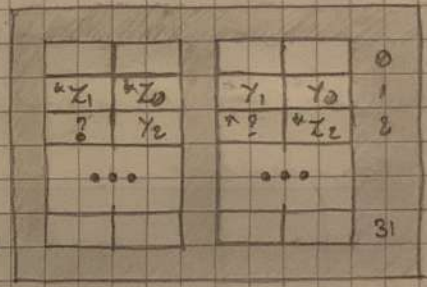
Load Y_1 → (0b 0 0001, 0b 1000) => Miss conf dirty.

Store Z_1 → (0b 0 0001, 0b 1000) => Miss conf clean.

Load X_2 → (0b 0 0001, 0b 0000) => Miss comp clean.

Load Y_2 → (0b 0 0001, 0b 0000) => Miss comp clean.

Store Z_2 → (0b 0 0001, 0b 0000) => Miss comp clean.



Etat des cases de cache de la famille 1 :

Instruction i :	init	LX_0	LY_0	SZ_0	LX_1	LY_1	SZ_1	LX_2	LY_2	SZ_2
Case 00 après i	∅	X_0	X_0	$*Z_0$	$*Z_0$	Y_0	Y_0	Y_0	Y_0	Y_0
Case 01 après i	∅	X_1	X_1	$*Z_1$	$*Z_1$	Y_1	Y_1	Y_1	Y_1	Y_1
Case 10 après i	∅	∅	Y_0	Y_0	X_0	X_0	$*Z_0$	$*Z_0$	$*Z_0$	$*Z_0$
Case 11 après i	∅	∅	Y_1	Y_1	X_1	X_1	$*Z_1$	$*Z_1$	$*Z_1$	$*Z_1$

Etat des cases de cache de la famille 2 :

Instruction i :	init	LX_0	LY_0	SZ_0	LX_1	LY_1	SZ_1	LX_2	LY_2	SZ_2
Case 00 après i	∅	∅	∅	∅	∅	∅	∅	X_2	X_2	$*Z_2$
Case 01 après i	∅	∅	∅	∅	∅	∅	∅	?	?	$*?$
Case 10 après i	∅	∅	∅	∅	∅	∅	∅	∅	Y_2	Y_2
Case 11 après i	∅	∅	∅	∅	∅	∅	∅	∅	?	?

On a 9 accès : 9 Miss dont 8 clean et 1 dirty.

e/ Dans le cas d'un cache associatif par ensemble de 4 blocs et donnez pour chaque référence le numéro de l'ensemble où elle est stockée.

Structure Adresse : 24, 4, 4

Effet instruction et détail de (#Ens, Dep) :

Load X_0 → (01, 00) ⇒ Miss comp clean.

Load Y_0 → (01, 00) ⇒ Miss comp clean.

Store Z_0 → (01, 00) ⇒ Miss comp clean.

Load X_1 → (01, 08) ⇒ Hit.

Load Y_1 → (01, 08) ⇒ Hit.

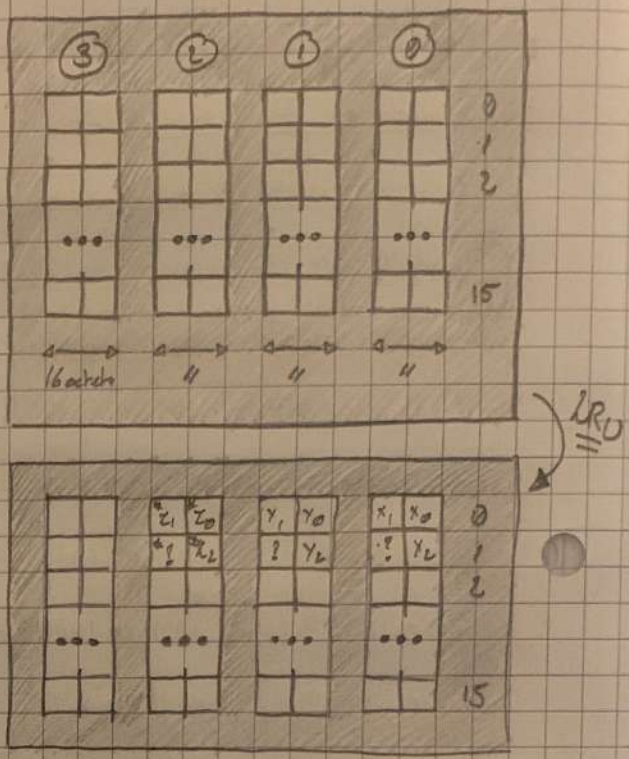
Store Z_1 → (01, 08) ⇒ Hit.

Load X_2 → (02, 00) ⇒ Miss comp clean.

Load Y_2 → (02, 00) ⇒ Miss comp clean.

Store Z_2 → (02, 00) ⇒ Miss comp clean.

4-Way Associative



Etat des cases de caches des familles 1 et 2 :

Instruction	init	LX_0	LY_0	SZ_0	LX_1	LY_1	SZ_1	LX_2	LY_2	SZ_2
Famille 1 — Case 00 après :	∅	X_0	〃	〃	〃	〃	〃	〃	〃	〃
Case 01 après :	∅	X_1	〃	〃	〃	〃	〃	〃	〃	〃
Case 10 après :	∅	∅	Y_0	〃	〃	〃	〃	〃	〃	〃
Case 11 après :	∅	∅	Y_1	〃	〃	〃	〃	〃	〃	〃
Case 20 après :	∅	∅	∅	$*Z_0$	〃	〃	〃	〃	〃	〃
Case 21 après :	∅	∅	∅	$*Z_1$	〃	〃	〃	〃	〃	〃
Case 30 après :	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅
Case 31 après :	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅
Famille 2 — Case 00 après :	∅	∅	∅	∅	∅	∅	∅	X_2	〃	〃
Case 01 après :	∅	∅	∅	∅	∅	∅	∅	?	〃	〃
Case 10 après :	∅	∅	∅	∅	∅	∅	∅	∅	Y_2	〃
Case 11 après :	∅	∅	∅	∅	∅	∅	∅	∅	?	〃
Case 20 après :	∅	∅	∅	∅	∅	∅	∅	∅	∅	$*Z_2$
Case 21 après :	∅	∅	∅	∅	∅	∅	∅	∅	∅	$*?$