Aurélien LEMAIRE
Sébastien LEMOUSSU

Groupe A

# Devoir d'architecture

## Première partie : Architecture logicielle

*Exercice 1. Conversion*

a)

```
/* le caractère à convertir est placé dans l'octet bas du registre R5 */

1  SUBIcc R0,R5,0x40 /* <= 0 si code d'un caractère numérique (0 à 9) */
2  BLEU inchange     /* on ne change donc pas le caractère */
3  SUBIcc R0,R5,0x5A /* > 0 si code d'un caractère minuscule (a à z) */
4  BGU inchange      /* on ne change donc pas le caractère */
5  ADDI R5,R5,0x20   /* convertit en minuscule */

inchange: ...
```

Remarquons que le SUBI ne fait pas partie du jeu d'instructions.
Il faut donc trouver une autre méthode.
On a utilisé, dans la méthode ci-dessus, le fait que pour passer d'une majuscule à son minuscule, il faut ajouter 0x20, soit 32 en décimal ; ce qui revient, en raisonnant en binaire , à changer le $6^{ème}$ bit (de 0 à 1). Si le caractère considéré est déjà numérique ou minuscule, alors ce bit est déjà à 1.
Il faut donc faire un OU bit à bit du caractère considéré avec 0010 0000. Ainsi les majuscules seront transformés en minuscules et les autres caractères resteront inchangés.

On obtient : **`ORI R5, R5, 0x20`**

b)

```
/* R1 contient l'adresse de la chaîne de départ et R2 celle de la chaîne
convertie */

Boucle :
         1  LDUB R5,0(R1)  /* charge le code du caractère */
         2  SUBcc R0,R5,R0 /* = 0 si le caractère est 0 */
         3  BEQ Fin        /* on a alors fini */
         4  ORI R5,R5,0x20 /* convertit le caractère */
         5  STB R5,0(R2)   /* met le caractère converti ou non dans R2*/
                           /* met les 8 bits de poids faible de R5...
                              ...à l'emplacement mémoire de R2 */
         6  ADDI R1,R1,1   /* pour mettre à jour...
         7  ADDI R2,R2,1      ...les pointeurs sur R1 et R2 */
         8  BA Boucle      /* on recommence */
Fin :
         9  STB R5,0(R2)   /* met le 0 de fin dans R2 */
```

Test avec la chaîne « De0 » :

R1 adresse de 0x44 65 30 00

```
LDUB R5,0(R1)                   R5 <- 0x00 00 00 44
SUBcc R0,R5,R0                  R5 – R0 != 0
ORI R5,R5,0x20                  0x00000044 or 0x00000020 ⇒ R5 <- 0x00000064
STB R5,0(R2)                    0(R2) <- 0x64

On fait les mises à jour et on boucle

LDUB R5,0(R1)                   R5 <- 0x00 00 00 65
SUBcc R0,R5,R0                  R5 – R0 != 0
ORI R5,R5,0x20                  0x00000065 or 0x00000020 ⇒ R5 <- 0x00000065
STB R5,0(R2)                    0(R2) <- 0x65

On fait les mises à jour et on boucle

LDUB R5,0(R1)                   R5 <- 0x00 00 00 30
SUBcc R0,R5,R0                  R5 – R0 != 0
ORI R5,R5,0x20                  0x00000030 or 0x00000020 ⇒ R5 <- 0x00000030
STB R5,0(R2)                    0(R2) <- 0x30

On fait les mises à jour et on boucle

LDUB R5,0(R1)                   R5 <- 0x00 00 00 00
SUBcc R0,R5,R0                  R5 – R0 = 0

STB R5,0(R2)                    0(R2) <- 0x00
```

On a donc R2 adresse de 0x64 65 30 00.
Ce qui correspond à la chaîne « de0 ».

Test avec la chaîne vide :

Le branchement BEQ est fait directement (ça se déroule de la même manière que la dernière étape de l'exemple précédent).

c)

Au lieu de faire des accès mémoire registre-immédiat, on va utiliser le mode registre-registre. On va utiliser un compteur :

```
1  ADD R3,R0,R0              /* met le compteur à 0 */
Boucle :
          2  LDUB R5,(R1+R3) /* charge le caractère */
          3  SUBcc R0,R5,R0  /* = 0 si le caractère est 0 */
          4  BEQ Fin         /* on a alors fini */
          5  ORI R5,R5,0x20  /* convertit le caractère */
          6  STB R5,(R2+R3)  /* met le caractère converti ou non dans R2*/
          7  ADDI R3,R3,1    /* pour mettre à jour le compteur */
          8  BA Boucle       /* on recommence */
Fin :
          9  STB R5,(R2+R3)  /* met le 0 de fin dans R2 */
```

d)

Hypothèse : la chaîne initiale doit avoir une longueur multiple de 4.

```
1  LUI R3,0x2020     /* pour mettre 0x20202020...
2  ORI R3,R3,Ox2020     ...dans R3 (vu en TD) */
3  LUI R4,0xFF00     /* met 0xFF000000 dans R4 */
Boucle :
          4  LD R5,0(R1)    /* charge les 4 caractères dans R5 */
          5  ANDcc R0,R5,R4 /* =0 si les 8 bits de poids fort sont à 0*/
          6  BEQ Fin        /* on a alors fini */
          7  OR R5,R5,R3    /* convertit les caractères */
          8  ST R5,0(R2)    /* met les caractères convertis dans R2 */
          9  ADDI R1,R1,4   /* pour mettre à jour...
          10 ADDI R2,R2,4      ...les pointeurs sur R1 et R2 */
          11 BA Boucle      /* on recommence */
Fin :
          12 STB R0,0(R2)   /* met le 0 de fin dans R2 */
```

A l'intérieur de la Boucle, on a 8 instructions comme dans b) mais on traite 4 caractères par passage au lieu de 1. On a donc considérablement amélioré les performances.

Test avec la chaîne « Dev2 » :

```
LUI R3,0x2020               R3 <- 0x20 20 00 00
ORI R3,R3,Ox2020            R3 <- 0x20 20 20 20
LUI R4,0xFF00               R4 <- 0xFF 00 00 00

LD R5,0(R1)                 R5 <- 0x44 65 76 32
ANDcc R0,R5,R4              0x44657632 and 0xFF000000 != 0
OR R5,R5,R3                 R5 <- 0x64657632
ST R5,0(R2)                 0(R2) <- 0x64657632

On fait les mises à jour et on boucle

LD R5,0(R1)                 R5 <- 0x00 .. .. ..
ANDcc R0,R5,R4              0x00..... and 0xFF000000 = 0

STB R0,0(R2)                0(R2) <- 0x00
```

On a donc R2 adresse de 0x64 65 76 32 00.
Ce qui correspond à la chaîne « dev2 ».

*Exercice 2. Palindromes*

On appelle la fonction palin avec les arguments i (numéro d'appel), n (longueur de la chaîne de caractères) et x (la chaîne de caractères).

```
/* R1 contient i, R2 contient n et R3 contient l'adresse de x */

/* on fait de la place sur la pile (R30 est le pointeur de pile) pour y
mettre les registres dont on aura besoin : R31 (adresse de retour), R4 et
R5 pour manipuler les différents paramètres. On considère que pour réserver
de la place pour la pile, on fait un ADDI puis on stocke au dessus. */
Palin :
        1  ADDI R30,R30,12
        2  ST R31,-12(R30)
        3  ST R4,-8(R30)
        4  ST R5,-4(R30)

/* si ( n - 2*i <= 1 ) */
5  SLL R4,1,R1    /* décalage à gauche de 1 bit de R1...
                     ...(multiplication par 2) : r4 <- 2*i */
6  SUB R4,R2,R4   /* r4 <- n - 2*i */
7  SUBIcc R0,R4,1 /* si n - 2*i – 1 > 0 */
8  BG Suite1      /* on continue sinon on retourne vrai (ligne suivante) */

/* retourne vrai */
9  LD R4,-8(R30)    /* on restaure le seul registre modifié (R4) */
10 ADDI R1,R0,1     /* on convient que 0 est faux et 1 est vrai...
                       ...(valeur de retour de palin dans R1) */
11 SUBI R30,R30,12  /* "on libère la pile" */
12 JMPL R0,R31,0    /* on saute à la dernière adresse entrée dans R31 */

/* si ( x[i] != x[n-i-1] ) */
Suite1 :
        13 SUB R5,R2,R1   /* r5 <- n - i
        14 SUBI R5,R5,1   /* r5 <- n - i - 1
        15 ADD R5,R3,R5   /* on met l'adresse de X[n-i-1] dans R5 */
        16 LDUB R5,0(R5)  /* on charge X[n-i-1] dans R5 */
        17 ADD R4,R3,R1   /* on met l'adresse de X[i] dans R4 */
        18 LDUB R4,0(R4)  /* on charge X[i] dans R4 */
       19 SUBcc R0,R4,R5 /* si X[i] - X[n-i-1] = 0 */
        20 BEQ Suite2     /* on continue sinon on retourne faux (ligne...
                             ...suivante) */

/* retourne faux */
21 LD R4,-8(R30)    /* on restaure les...
22 LD R5,-4(R30)       ...registres modifiés */
23 ADD R1,R0,R0     /* comme convenu 0 dans R1 (pour faux) */
24 SUBI R30,R30,12  /* "on libère la pile" */
25 JMPL R0,R31,0    /* on saute à la dernière adresse entrée dans R31 */

/* palin(i+1,n,x) */
Suite2 :
         26 ADD R1,R1,1    /* r1 <- i+1 */
         27 JMPL R31,Palin /* met dans R31 l'adresse de l'instruction...
                              ...suivante et saute à Palin */
/* retourne palin */
28 LD R31,-12(R30) /* on restaure...
29 LD R4,-8(R30)      ...les registres...
30 LD R5,-4(R30)      ...modifiés */
31 SUBI R30,R30,12 /* "on libère la pile" */
32 JMPL R0,R31,0   /* retour au programme appelant */
```

On considère qu'un caractère est codé sur 8 bits comme dans l'exercice 1.
Les tests qui suivent utilisent les caractères a, b et c ; on utilisera les codes respectifs 0x61, 0x62 et 0x63.

Test avec la chaîne « aba » :

```
Initialement :
R1 contient 0 (i), R2 contient 3 (n) et R3 contient l'adresse de « aba ».

1 2 3 4     on réserve la pile (prend 12 octets)
5           R4 <- 0
6           R4 <- 3
7           R4 - 1 = 2 > 0
8           Suite1
13          R5 <- 3
14          R5 <- 2
15          R5 <- adresse de X[2]
16          R5 <- 0x00000061
17          R4 <- adresse de X[0]
18          R4 <- 0x00000061
19          R4 - R5 = 0
20          Suite2
26          R1 <- 1
27          Palin (et R31 <- adresse de l'instruction 28)
1 2 3 4     on réserve à nouveau de la place pour agrandir la pile
            (en tout 12 + 12 = 24 octets)
5           R4 <- 2
6           R4 <- 1
7 8         R4 – 1 = 0 <= 0
9           restauration de R4
10          R1 <- 1 (Vrai)
11          libération de la place prise dans la pile par les derniers
            paramètres entrés (en tout 24 – 12 = 12 octets)
12          saut à l'instruction 28
28 29 30    restauration des registres R31, R4 et R5
31          libération de la place dans la pile (en tout 12 – 12 = 0 octet)
32          on retourne au programme qui a appelé la fonction palin
```

On récupère bien dans R1 le booléen vrai, ce qui est correct puisque la chaîne « aba » est un palindrome.

Test avec la chaîne « abc » :

```
1 2 3 4     on réserve la pile (prend 12 octets)
5           R4 <- 0
6           R4 <- 3
7           R4 - 1 = 2 > 0
8           Suite1
13          R5 <- 3
14          R5 <- 2
15          R5 <- adresse de X[2]
16          R5 <- 0x00000063
17          R4 <- adresse de X[0]
18          R4 <- 0x00000061
19 20       R4 - R5 != 0
21 22       restauration de R4 et R5
23          R1 <- 0 (Faux)
24          libération de la place dans la pile (en tout 12 – 12 = 0 octet)
25          on retourne au programme qui a appelé la fonction palin
```

On récupère bien dans R1 le booléen faux, ce qui est correct puisque la chaîne « abc » n'est pas un palindrome.

Test avec la chaîne « abba » :

```
1 2 3 4     on réserve la pile (prend 12 octets)
5           R4 <- 0
6           R4 <- 4
7           R4 – 1 = 3 > 0
8           Suite1
13          R5 <- 4
14          R5 <- 3
15          R5 <- adresse de X[3]
16          R5 <- 0x00000061
17          R4 <- adresse de X[0]
18          R4 <- 0x00000061
19          R4 – R5 = 0
20          Suite2
26          R1 <- 1
27          Palin (et R31 <- adresse de l'instruction 28)
1 2 3 4     on réserve à nouveau de la place pour agrandir la pile
            (en tout 12 + 12 = 24 octets)
5           R4 <- 2
6           R4 <- 2
7           R4 – 1 = 1 > 0
8           Suite1
13          R5 <- 3
14          R5 <- 2
15          R5 <- adresse de X[2]
16          R5 <- 0x00000062
17          R4 <- adresse de X[1]
18          R4 <- 0x00000062
19          R4 – R5 = 0
20          Suite2
26          R1 <- 2
27          Palin (et R31 <- adresse de l'instruction 28)
1 2 3 4     on réserve à nouveau de la place pour agrandir la pile
            (en tout 24 + 12 = 36 octets)
5           R4 <- 4
6           R4 <- 0
7 8         R4 – 1 = -1 <= 0
9           restauration de R4
10          R1 <- 1 (Vrai)
11          libération de la place dans la pile (en tout 36 – 12 = 24
            octets)
12          saut à l'instruction 28
28 29 30    restauration des registres R31, R4 et R5
31          libération de la place dans la pile (en tout 24 – 12 = 12
            octets)
32          saut à l'instruction 28
28 29 30    restauration des registres R31, R4 et R5
31          libération de la place dans la pile (en tout 12 – 12 = 0 octet)
32          on retourne au programme qui a appelé la fonction palin
```

On récupère bien dans R1 le booléen vrai, ce qui est correct puisque la chaîne « abba » est un palindrome.

**Deuxième partie : Caches**

*Exercice 3. Produit matrice – vecteur*

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

a)

Lors de la boucle sur i de 0 à N-1, j n'est pas modifié.
Au lieu de recharger à chaque fois Y[j], on peut le sortir de la boucle i et l'utiliser comme un registre. Ainsi, il n'y aura des accès à Y[j] que lors des passages dans la boucle j (1 accès par passage).

```
for (j = 0 ; j < N ; j++){
     s = 0 ;
     R = Y[j] ;
     for (i = 0 ; i < N ; i++)
          Z[i] = Z[i] + X[i][j]*R ;
}
```

En conséquence, le programme Prog32.c doit être modifié :

```
for (i = 0 ; i < N ; i++){
     s = 0 ;
     ac((unsigned long) &y[i]) ;
     for (k = 0; k < N ; k++){
          ac((unsigned long) &x[k][i]) ;
          ac((unsigned long) &z[k]);

          z[k] = z[k] + x[k][i]*y[i] ;
     }
}
```

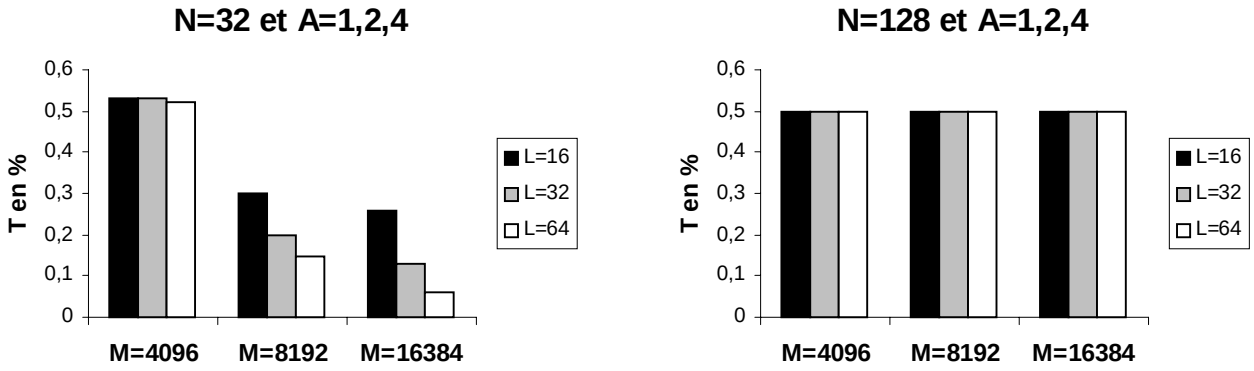
b)

Voici une table des résultats obtenues pour N = 32 et N = 128 :

Taille du cache (M) : 4 Koctets ou 8 Koctets ou 16 Koctets
Associativité (A) : 1 ou 2 ou 4 bloc(s) par ensemble
Taille du bloc (L) : 16 octets ou 32 octets ou 64 octets
Taille des vecteurs ou des matrices (N)
Taux d'échecs (T)

| M | A | L | N | T |
|---|---|---|---|---|
| 4096 | 1 | 16 | 32 | 0.534615 |
| 4096 | 2 | 16 | 32 | 0.532692 |
| 4096 | 4 | 16 | 32 | 0.531731 |
| 4096 | 1 | 32 | 32 | 0.530769 |
| 4096 | 2 | 32 | 32 | 0.528846 |
| 4096 | 4 | 32 | 32 | 0.528365 |
| 4096 | 1 | 64 | 32 | 0.529808 |
| 4096 | 2 | 64 | 32 | 0.526442 |
| 4096 | 4 | 64 | 32 | 0.525962 |
| 8192 | 1 | 16 | 32 | 0.300000 |
| 8192 | 2 | 16 | 32 | 0.311538 |
| 8192 | 4 | 16 | 32 | 0.340385 |
| 8192 | 1 | 32 | 32 | 0.182212 |
| 8192 | 2 | 32 | 32 | 0.202885 |
| 8192 | 4 | 32 | 32 | 0.249038 |
| 8192 | 1 | 64 | 32 | 0.122596 |
| 8192 | 2 | 64 | 32 | 0.146154 |
| 8192 | 4 | 64 | 32 | 0.200000 |
| 16384 | 1 | 16 | 32 | 0.261538 |
| 16384 | 2 | 16 | 32 | 0.261538 |
| 16384 | 4 | 16 | 32 | 0.261538 |
| 16384 | 1 | 32 | 32 | 0.131250 |
| 16384 | 2 | 32 | 32 | 0.131250 |
| 16384 | 4 | 32 | 32 | 0.131250 |
| 16384 | 1 | 64 | 32 | 0.065865 |
| 16384 | 2 | 64 | 32 | 0.065865 |
| 16384 | 4 | 64 | 32 | 0.065865 |
| 4096 | 1 | 16 | 128 | 0.509667 |
| 4096 | 2 | 16 | 128 | 0.509636 |
| 4096 | 4 | 16 | 128 | 0.509636 |
| 4096 | 1 | 32 | 128 | 0.507934 |
| 4096 | 2 | 32 | 128 | 0.507782 |
| 4096 | 4 | 32 | 128 | 0.507782 |
| 4096 | 1 | 64 | 128 | 0.510974 |
| 4096 | 2 | 64 | 128 | 0.506809 |
| 4096 | 4 | 64 | 128 | 0.506809 |
| 8192 | 1 | 16 | 128 | 0.509606 |
| 8192 | 2 | 16 | 128 | 0.509545 |
| 8192 | 4 | 16 | 128 | 0.509515 |
| 8192 | 1 | 32 | 128 | 0.507843 |
| 8192 | 2 | 32 | 128 | 0.507752 |
| 8192 | 4 | 32 | 128 | 0.507721 |
| 8192 | 1 | 64 | 128 | 0.507083 |
| 8192 | 2 | 64 | 128 | 0.506809 |
| 8192 | 4 | 64 | 128 | 0.506809 |
| 16384 | 1 | 16 | 128 | 0.509484 |
| 16384 | 2 | 16 | 128 | 0.509363 |
| 16384 | 4 | 16 | 128 | 0.509302 |
| 16384 | 1 | 32 | 128 | 0.507782 |
| 16384 | 2 | 32 | 128 | 0.507661 |
| 16384 | 4 | 32 | 128 | 0.507630 |
| 16384 | 1 | 64 | 128 | 0.506992 |
| 16384 | 2 | 64 | 128 | 0.506779 |
| 16384 | 4 | 64 | 128 | 0.506749 |

Voici une représentation graphique des résultats du tableau :




c)

On se place dans le cas idéal d'un cache de taille non bornée complètement associatif.

**<u>Pour j = 0</u> :**

On accède à Y[0], 1 échec, on charge les éléments Y[0] à Y[(L/8)-1]

<u>Dans la boucle i (0 à N-1) :</u>
On accède à Z[0], 1 échec, on charge les éléments Z[0] à Z[(L/8)-1] ; **DONC** 1 échec tous les L/8 accès.
On accède à X[0][0], 1 échec, on charge les éléments X[0][0] à X[0][(L/8)-1] ; **DONC** 1 échec tous les accès (pour X[1][0], X[2][0], …, X[N-1][0]).

Pour j = 0, on a `N + N/(L/8) + 1` échecs.

**<u>Pour les accès à X, Y et Z, calcul du nombre d'échecs total</u> :**

<u>Regardons le cas j = 1 :</u>
Pour Y[1], succès car chargé pour j = 0.
Pour Z[i], que des succès car réutilisation des éléments chargés pour j = 0.
Pour X[i][1], que des succès car déjà chargés dans la boucle i quand j = 0.

Nombre d'échecs pour j = 1 : `0.`

<u>Pour un j multiple de L/8, on aura</u> : `N` échecs pour X `+ 1` échec pour Y.

<u>Au total</u> :
Pour Y, 1 échec pour chaque j multiple de L/8 `-> N/(L/8)` échecs.
Pour Z, `N/(L/8)` échecs dans j = 0.
Pour X, `N` échecs pour chaque j multiple de L/8 `-> N`$^2$`/(L/8)`.

Nombre d'échecs total `= (N`$^2$ `+ 2N)/(L/8)`

**Taux d'échecs :**

Le nombre d'accès est de N pour Y, $N^2$ pour Z et $N^2$ pour X : $2N^2 + N$ accès.

Le taux d'échecs est alors : $((N^2 + 2N)/(L/8)) \ / \ (2N^2 + N)$

Ou encore : $(2 + N) \ / \ ((L/8)(2N + 1))$

**Application numérique :**

```
Si N = 32 :
     pour L = 16 : T = 26 %
     pour L = 32 : T = 13 %
     pour L = 64 : T =  6 %

Si N = 128 :
     pour L = 16 : T = 25 %
     pour L = 32 : T = 12 %
     pour L = 64 : T =  6 %
```

d)

Analysons les résultats pour N = 32 et N = 128 pour M = 8K et A = 1.

Les résultats du tableau ne sont pas les mêmes que les résultats théoriques obtenus en c).
Il faut rappeler que nous nous étions placé dans le cas idéal, il n'y avait donc pas de conflit (recouvrement) lors de nos calculs.

La réalité est différente !
La taille de cache minimum qu'il nous faut est de $8(N^2 + 2N)$ (nombre d'éléments de X + nombre d'éléments de Y + nombre d'éléments de Z).
Donc pour N = 32, il faut un cache de 8704 octets et pour N = 128, de 133 120 octets.

Nous avons ici un cache de 8K, soit 8192 octets ; le cache n'est donc pas assez grand pour contenir tous les éléments accédés donc on va avoir des défauts de conflit en plus des défauts d'initialisation.

Rappelons les résultats expérimentaux :

| M | A | L | N | T |
|---|---|---|---|---|
| 8192 | 1 | 16 | 32 | 0.300000 |
| 8192 | 1 | 32 | 32 | 0.182212 |
| 8192 | 1 | 64 | 32 | 0.122596 |
| 8192 | 1 | 16 | 128 | 0.509606 |
| 8192 | 1 | 32 | 128 | 0.507843 |
| 8192 | 1 | 64 | 128 | 0.507083 |

Ce cache peut contenir 1024 éléments de 8 octets et nous en avons 1088 à accéder (si N = 32).

*Exercice 4. Produit de matrices*

```
for (i = 0 ; i < N ; i++)
    for (j = 0 ; j < N ; j++){
          s = 0 ;
          for (k = 0 ; k < N ; k++)
                s = s + X[i][k]*Y[k][j] ;
          Z[i][j] = s ;
}
```

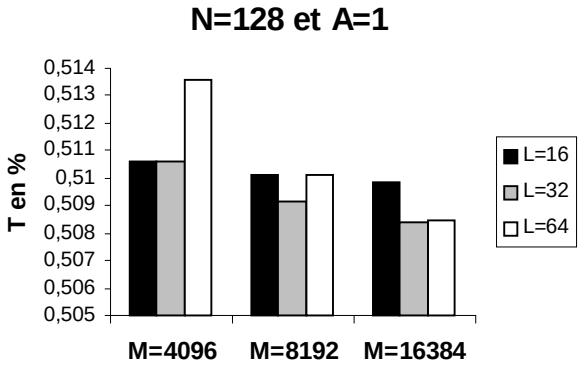
a)

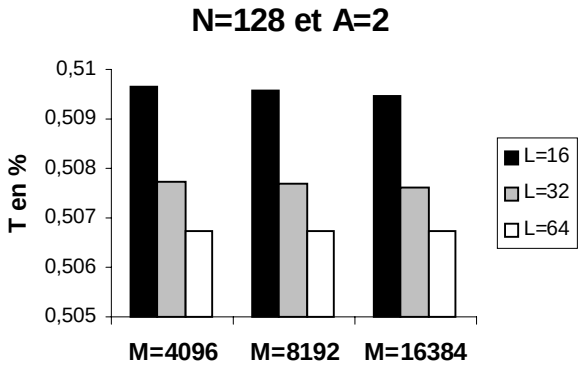
Voici une table des résultats obtenues pour N = 128 :

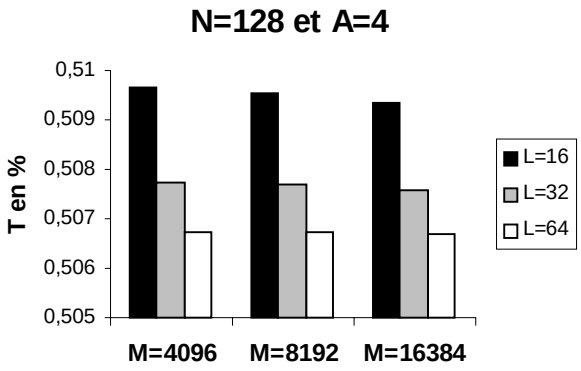
Taille du cache (M) : 4 Koctets ou 8 Koctets ou 16 Koctets
Associativité (A) : 1 ou 2 ou 4 bloc(s) par ensemble
Taille du bloc (L) : 16 octets ou 32 octets ou 64 octets
Taille d'une ligne de matrice (N)
Taux d'échecs (T)

| **M** | **A** | **L** | **N** | **T** |
|---|---|---|---|---|
| 4096 | 1 | 16 | 128 | 0.510624 |
| 4096 | 2 | 16 | 128 | 0.509636 |
| 4096 | 4 | 16 | 128 | 0.509636 |
| 4096 | 1 | 32 | 128 | 0.510640 |
| 4096 | 2 | 32 | 128 | 0.507721 |
| 4096 | 4 | 32 | 128 | 0.507721 |
| 4096 | 1 | 64 | 128 | 0.513558 |
| 4096 | 2 | 64 | 128 | 0.506749 |
| 4096 | 4 | 64 | 128 | 0.506749 |
| 8192 | 1 | 16 | 128 | 0.510108 |
| 8192 | 2 | 16 | 128 | 0.509583 |
| 8192 | 4 | 16 | 128 | 0.509530 |
| 8192 | 1 | 32 | 128 | 0.509165 |
| 8192 | 2 | 32 | 128 | 0.507691 |
| 8192 | 4 | 32 | 128 | 0.507676 |
| 8192 | 1 | 64 | 128 | 0.510153 |
| 8192 | 2 | 64 | 128 | 0.506749 |
| 8192 | 4 | 64 | 128 | 0.506749 |
| 16384 | 1 | 16 | 128 | 0.509804 |
| 16384 | 2 | 16 | 128 | 0.509465 |
| 16384 | 4 | 16 | 128 | 0.509348 |
| 16384 | 1 | 32 | 128 | 0.508405 |
| 16384 | 2 | 32 | 128 | 0.507634 |
| 16384 | 4 | 32 | 128 | 0.507577 |
| 16384 | 1 | 64 | 128 | 0.508436 |
| 16384 | 2 | 64 | 128 | 0.506718 |
| 16384 | 4 | 64 | 128 | 0.506695 |

Voici une représentation graphique des résultats du tableau :




b)

Observons dans un premier temps le graphe pour A=1. Premièrement, on remarque qu'en augmentant la taille du cache, on obtient de "meilleurs résultats" (avec des guillemets parce que on a une baisse de l'ordre des 1 % quand on double la taille du cache !). Ces remarques s'appliquent aussi aux graphes A=2 et 4.

Le point le plus important à regarder est l'évolution du taux d'échecs en fonction de la taille d'un bloc (L). Cette fois, les différences sont claires.
En effet, pour A=1, on remarque que le taux d'échecs a plutôt tendance à augmenter lorsque la taille du bloc augmente ; alors que pour A=2 et 4, le taux d'échecs baisse nettement.
Cela s'explique par le fait qu'en correspondance directe, augmenter la taille du bloc diminue les défauts d'initialisation mais augmente le nombre de défauts de conflit (à cause du nombre de blocs qui diminue).

Par contre, lorsque l'on a 2 ou 4 blocs par ensemble, les défauts de conflit sont diminués (et les conflits d'initialisation diminuent aussi pour les mêmes raisons que pour A=1. Ce qui explique la diminution du taux d'échecs pour A=2 et 4, lorsque L augmente.