UNIVERSITÉ DE PARIS-DAUPHINE **DE MI2E 2e année**

Architecture des ordinateurs (E. Lazard)
Examen du 27 janvier 2011

CORRECTION

(durée 2 heures)

I. Mémoire cache

Notre machine possède un cache d'une taille de 12 instructions regroupées en 4 blocs de 3 instructions. Le cache est à correspondance directe. On rappelle que cela veut dire que les mots mémoire 1-3, 13-15... vont dans le premier bloc du cache, que les mots 4-6, 16-18... vont dans le deuxième, etc. Le temps de cycle de la mémoire principale est M et le temps de cycle du cache est C.

1. Un programme se compose d'une boucle de 33 instructions à exécuter 3 fois ; les instructions se trouvant, dans l'ordre, aux adresses mémoire 1 à 15, 16 à 18, 1 à 3, 16 à 18, 1 à 3, 16 à 18, 1 à 3. Quel est le temps total d'exécution du programme en ne tenant pas compte des temps de calcul ? Le cache est vide au départ.
2. Le compilateur a réussi à optimiser votre programme et en a réduit la taille en gagnant quelques intructions. Il se compose maintenant d'une boucle de 30 instructions à exécuter 3 fois ; les instructions se trouvant, dans l'ordre, aux adresses mémoire 1 à 12, 13 à 15, 1 à 3, 13 à 15, 1 à 3, 13 à 15, 1 à 3. Quel est le temps total d'exécution du programme en ne tenant pas compte des temps de calcul ? Le cache est vide au départ.
3. Que concluez-vous des deux précédents résultats ?

Rappel : Lorsque l'on va chercher un mot en mémoire, pendant M on ramène le mot pour le processeur et tout le bloc correspondant dans le cache.

Corrigé :

1.

$1 \to 3$	$M+2C$	bloc 1	$1 \to 3$	$3C$	bloc 1	$1 \to 3$	$3C$	bloc 1
$4 \to 6$	$M+2C$	bloc 2	$4 \to 6$	$M+2C$	bloc 2	$4 \to 6$	$M+2C$	bloc 2
$7 \to 9$	$M+2C$	bloc 3	$7 \to 9$	$3C$	bloc 3	$7 \to 9$	$3C$	bloc 3
$10 \to 12$	$M+2C$	bloc 4	$10 \to 12$	$3C$	bloc 4	$10 \to 12$	$3C$	bloc 4
$13 \to 15$	$M+2C$	bloc 1	$13 \to 15$	$M+2C$	bloc 1	$13 \to 15$	$M+2C$	bloc 1
$16 \to 18$	$M+2C$	bloc 2	$16 \to 18$	$M+2C$	bloc 2	$16 \to 18$	$M+2C$	bloc 2
$1 \to 3$	$M+2C$	bloc 1	$1 \to 3$	$M+2C$	bloc 1	$1 \to 3$	$M+2C$	bloc 1
$16 \to 18$	$3C$	bloc 2	$16 \to 18$	$3C$	bloc 2	$16 \to 18$	$3C$	bloc 2
$1 \to 3$	$3C$	bloc 1	$1 \to 3$	$3C$	bloc 1	$1 \to 3$	$3C$	bloc 1
$16 \to 18$	$3C$	bloc 2	$16 \to 18$	$3C$	bloc 2	$16 \to 18$	$3C$	bloc 2
$1 \to 3$	$3C$	bloc 1	$1 \to 3$	$3C$	bloc 1	$1 \to 3$	$3C$	bloc 1

Soit un total de $15M + 84C$.

2.

$1 \to 3$	$M+2C$	bloc 1	$1 \to 3$	$3C$	bloc 1	$1 \to 3$	$3C$	bloc 1
$4 \to 6$	$M+2C$	bloc 2	$4 \to 6$	$3C$	bloc 2	$4 \to 6$	$3C$	bloc 2
$7 \to 9$	$M+2C$	bloc 3	$7 \to 9$	$3C$	bloc 3	$7 \to 9$	$3C$	bloc 3
$10 \to 12$	$M+2C$	bloc 4	$10 \to 12$	$3C$	bloc 4	$10 \to 12$	$3C$	bloc 4
$13 \to 15$	$M+2C$	bloc 1	$13 \to 15$	$M+2C$	bloc 1	$13 \to 15$	$M+2C$	bloc 1
$1 \to 3$	$M+2C$	bloc 1	$1 \to 3$	$M+2C$	bloc 1	$1 \to 3$	$M+2C$	bloc 1
$13 \to 15$	$M+2C$	bloc 1	$13 \to 15$	$M+2C$	bloc 1	$13 \to 15$	$M+2C$	bloc 1
$1 \to 3$	$M+2C$	bloc 1	$1 \to 3$	$M+2C$	bloc 1	$1 \to 3$	$M+2C$	bloc 1
$13 \to 15$	$M+2C$	bloc 1	$13 \to 15$	$M+2C$	bloc 1	$13 \to 15$	$M+2C$	bloc 1
$1 \to 3$	$M+2C$	bloc 1	$1 \to 3$	$M+2C$	bloc 1	$1 \to 3$	$M+2C$	bloc 1

Soit un total de $22M + 68C$.

3. On voit que « l'amélioration » effectuée par le compilateur a augmenté le nombre d'accès mémoire par une mauvaise répartition des instructions dans les blocs, même si on a gagné en nombre d'instructions. On a $7M$ en plus et $16C$ en moins, ce qui fait que si $7M > 16C$, le programme « optimisé » s'exécute plus lentement !

II. Assembleur *(Les deux questions sont indépendantes)*

1. Une chaîne de caractères est stockée en mémoire et représente un nombre entre 0 et 32767 (par exemple `"3456"`) ; son adresse est dans le registre `r0` et elle est non-vide et terminée par un octet nul. On souhaite lire cette chaîne et obtenir sa valeur numérique dans le registre r1 (ainsi avec l'exemple précédent, le registre `r1` aura pour valeur 3456 à la fin de la procédure). Écrire la procédure assembleur correspondante.
2. Un nombre entier compris entre -2^{31} et $2^{31} - 1$ est stocké en mémoire sur quatre octets, son adresse étant dans le registre `r0`. On souhaite générer une chaîne de caractères, se terminant par l'octet nul, représentant l'écriture **octale** de ce nombre. Cette chaîne sera donc composée de caractères de `'0'` à `'7'` (chacun sur un octet) et sera stockée à partir de l'adresse contenue dans le registre `r1`. Pour simplifier, on écrira la chaîne « à l'envers » : le caractère représentant les trois bits de poids faible sera mis à l'adresse pointée par `r1`, le caractère représentant les trois bits suivants à l'adresse `r1+1`,... Il y a au maximum 11 chiffres octals mais on n'affichera pas les éventuels zéros de tête. En revanche on n'omettra pas le signe moins si nécessaire (et on le mettra en dernier, après le caractère de poids fort). Par exemple, la valeur (décimale) 123 a pour valeur octale 173_8, on créera donc la chaîne `"371"` ; la valeur décimal -270 a pour valeur octale -416_8, on créera donc la chaîne `"614-"`. Écrire la procédure assembleur correspondante.

Corrigé :

1.
```
          MVI   r1,#0          ; mise à zéro initiale
loop:     LDB   r2,(r0)        ; charger caractère
          JZ    r2,fin         ; fini?
          ADD   r0,r0,#1       ; avancer pointeur
          MUL   r1,r1,#10      ; 10 fois ce qu'on avait avant
          SUB   r2,r2,#'0'     ; revenir à un chiffre ;
          ADD   r1 r1,r2       ; qu'on additionne ;
          JMP   loop           ; et au suivant
fin:
```

2.
```
            MVI   r31,#0
            LDW   r2,(r0)          ; charger le nombre
            JGE   r2,positif       ; ne rien faire si positif
            NEG   r2,r2            ; valeur absolue
            MVI   r31,#'-'         ; signe moins
positif:    AND   r3,r2,#7         ; calculer le modulo 8 (r3)
            LLS   r2,r2,#-3        ; et le quotient (r2) de la div. par 8
            ADD   r3,r3,#'0'       ; caractère 0 ajouté au modulo
            STB   (r1),r3          ; et stocké
            ADD   r1,r1,#1         ; avancer le pointeur
            JNZ   r2,positif       ; reste-t-il des caractères?
            STB   (r1),r31         ; signe moins ou zéro final
            JZ    r31,fin
            ADD   r1,r1,#1
            STB   (r1),r2          ; un zéro final après le moins
fin:
```

III. Nombres flottants

On considère une représentation simplifiée des réels en virgule flottante.

Un nombre réel X est représenté par 10 bits $\boxed{\text{s eeee mmmmm}}$ où $X = (-1)^s * 1{,}m * 2^{e-7}$ avec un exposant sur 4 bits ($0 < e \leq 15$, un exposant égal à 0 désigne le nombre 0 quelle que soit la pseudo-mantisse) et une pseudo-mantisse sur 5 bits (représentant les puissances négatives de 2 ; elles ont pour valeur $2^{-1} = 0{,}5$, $2^{-2} = 0{,}25$, $2^{-3} = 0{,}125$, $2^{-4} = 0{,}0625$ et $2^{-5} = 0{,}03125$).

Les calculs se font sur tous les chiffres significatifs et les arrondis s'effectuent ensuite inférieurement.

1. Représenter 8, 9 et 72 en virgule flottante.
2. *Rappel : pour additionner deux nombres en virgule flottante, il faut décaler la mantisse d'un des deux nombres pour égaliser les exposants, additionner les mantisses (sans oublier le* 1 *débutant la mantisse), puis renormaliser le nombre en arrondissant éventuellement la pseudo-mantisse. Ainsi, on a* $7 = 1{,}11000_2 \times 2^2$ *et* $1{,}25 = 1{,}01000_2 \times 2^0 = 0{,}0101000_2 \times 2^2$ *dont l'addition donne* $10{,}0001_2 \times 2^2$ *qui se normalise en* $1{,}00001_2 \times 2^3 = 8{,}25$.

   On écrit le code suivant :

```
float f = 9.0;
for (i=0 ; i<2 ; i++)
    f = 9.0*(f-8.0);
```

   - Qu'obtient-on comme valeur de `f` après la boucle ?
   - Le compilateur optimise le code et remplace la ligne de la boucle par `f = 9.0*f-72.0;` Quel est maintenant le résultat après la boucle ?
   - On change les règles de l'arrondi pour que, après calculs, celui-ci s'effectue supérieurement. D'après vous, vers quoi va tendre la valeur de `f` si on augmente la taille de la boucle en gardant la ligne optimisée par le compilateur ?

Corrigé :

1. $8 = 1 \times 2^3 = \boxed{0\,1010\,00000}$
   $9 = 1{,}125 \times 2^3 = \boxed{0\,1010\,00100}$
   $72 = 1{,}125 \times 2^6 = \boxed{0\,1101\,00100}$
2. – L'opération `f-8.0` donne $0{,}00100 \times 2^3$ soit $1{,}00000 \times 2^0$ (qui vaut exactement 1.0). En multipliant par 9.0, on obtient donc exactement 9.0 et la valeur de `f` ne change pas.
   – Il faut maintenant calculer `9.0*f` soit $1{,}001 \times 1{,}001 \times 2^6$ c'est-à-dire $1{,}010001 \times 2^6$ qui s'arrondit inférieurement à $1{,}01000 \times 2^6 = 80$. On obtient donc la valeur $80 - 72 = 8$ après un passage de boucle puis $9 * 8 - 72 = 0$ après les deux passages.
   – Si on arrondit supérieurement, on obtient 82 après la première multiplication `9.0*f` et donc 10 après le premier passage. La valeur de `f` va ensuite tendre vers l'infini et au-delà...

IV. Circuits logiques

On cherche à faire une UAL simplifiée comme suit : on veut un circuit à deux entrées a et b, une ligne de commande F et trois sorties s_0, s_1 et s_2 tel que :

– si $F = 0$, le circuit se comporte comme un demi-additionneur/soustracteur : la sortie s_0 représente la somme des deux bits (identique pour les deux opérations), s_1 la retenue de l'addition et s_2 la retenue de la soustraction ;
– si $F = 1$, le circuit se comporte comme une unité logique : la sortie s_0 représente alors le XOR des deux entrées, la sortie s_1 le NAND des deux entrées et la sortie s_2 le NOR des deux entrées.

1. Construire les deux tables de vérité (pour $F = 0$ et $F = 1$).
2. Exprimer s_0, s_1 et s_2 en fonction de a, b et F.
3. En simplifiant l'écriture de s_1 et s_2, représenter le circuit à l'aide de 6 portes logiques (NOT, OR, AND, XOR sont autorisées).

Corrigé :

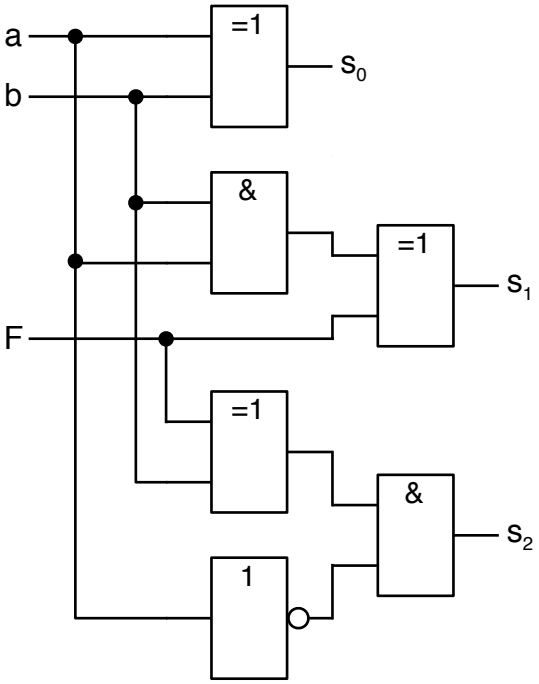
1. Pour $F = 0$

a	b	s_0	s_1	s_2
0	0	0	0	0
0	1	1	0	1
1	0	1	0	0
1	1	0	1	0

pour $F = 1$

a	b	s_0	s_1	s_2
0	0	0	1	1
0	1	1	1	0
1	0	1	1	0
1	1	0	0	0

2. $s_0 = (a \oplus b)$
   $s_1 = ab\overline{F} + \overline{ab}F = ab \oplus F$
   $s_2 = \overline{a}b\overline{F} + \overline{(a+b)}F$
3. On réécrit $s_2 = \overline{a}b\overline{F} + \overline{a}\overline{b}F = \overline{a}(b \oplus F)$
   Cela nous donne donc le circuit suivant :


a
b
F
=1
&
=1
=1
&
1
s_0
s_1
s_2