Architecture des ordinateurs (E. Lazard)
Examen du 23 janvier 2013

(durée 2 heures) – CORRECTION

I. 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 ($0 < e \leqslant 15$, 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 lorsque cela est indiqué.

1. Représenter 2,25 et 2,75 en virgule flottante.
2. On écrit le code suivant :

   Listing 1. Calculs

   ```
   float f = 2.25;
   float g = f * f - 2.75;
   ```

   (a) On suppose tout d'abord qu'après chacune des deux opérations, la valeur obtenue est remise en mémoire *et donc arrondie inférieurement à ce moment-là (il y a donc deux arrondis éventuels au total).* Quelle valeur obtient-on pour `g` ?

   (b) Le compilateur a optimisé le code et maintenant les valeurs intermédiaires restent dans les registres, l'arrondi n'intervenant qu'à la fin des calculs lors de la remise finale en mémoire. Quelle valeur obtient-on pour `g` ?

   (c) Quelle conclusion en tirez-vous ?

Corrigé :

1. $2{,}25 = 1{,}125 \times 2^1 = \boxed{0\ 1000\ 00100}$
   $2{,}75 = 1{,}375 \times 2^1 = \boxed{0\ 1000\ 01100}$
2. L'opération `f * f` se calcule par $1{,}001 \times 2^1 \times 1{,}001 \times 2^1$ ce qui donne exactement $1{,}010001 \times 2^2$, éventuellement arrondi à $1{,}01 \times 2^2$.

   (a) L'arrondi intervenant tout de suite, on doit calculer $1{,}01 \times 2^2 - 1{,}011 \times 2^1$ qui donne $0{,}1001 \times 2^2$ normalisé en $1{,}001 \times 2^1$ soit 2,25.

   (b) Puisqu'il n'y a pas d'arrondi après la multiplication, on doit calculer $1{,}010001 \times 2^2 - 1{,}011 \times 2^1$ qui donne $0{,}100101 \times 2^2$, normalisé en $1{,}00101 \times 2^1$. Il n'y a alors pas d'arrondi à effectuer et la valeur finale est 2,3125.

   (c) Le simple fait de rester dans les registres (paramètre sur lequel le programmeur n'a pas d'influence puisque cela est décidé par le compilateur) pour effectuer les calculs peut changer le résultat final si les arrondis n'interviennent pas au même endroit. C'est un des soucis de la norme de calcul en virgule flottante : ceux-ci ne sont pas forcément toujours reproductibles.

II. Assembleur

Une chaîne de caractères est stockée en mémoire. Chaque caractère est stocké sur un octet et l'octet qui suit le dernier caractère de la chaîne est nul. L'adresse du premier élément de la chaîne se trouve dans le registre `r0`.

On souhaite trier cette chaîne sur place par l'algorithme de tri par sélection suivant :

- on parcourt la chaîne du premier au dernier élément en mémorisant le plus petit ; ce caractère est ensuite échangé avec le premier ;
- on recommence en parcourant la chaîne du deuxième au dernier élément en mémorisant de nouveau le plus petit ; ce caractère est ensuite échangé avec le deuxième ;
- on procède de la même façon pour tous les éléments restants. L'algorithme est donc construit comme deux boucles imbriquées : une boucle externe qui parcourt tous les éléments et une boucle interne qui va de l'élément courant de la boucle externe au dernier pour chercher le plus petit.

Écrire une procédure assembleur qui, lorsqu'elle se termine, assure que la chaîne initialement pointée par `r0` est maintenant triée en place (c'est-à-dire directement à sa place et pas recopiée triée ailleurs). On ne vous demande pas de conserver la valeur de `r0` à la fin.

Corrigé :

Listing 2. Tri par sélection

```
lpExt:  LDB     r10,(r0)        ; a-t-on fini la chaîne ?
        JZ      r10,fin
        MOV     r1,r0           ; initialisation du pointeur interne
        MOV     r2,r0           ; init. du pointeur mémorisant le plus petit élément
lpInt:  ADD     r1,r1,#1
        LDB     r20,(r1)
        JZ      r20,suite       ; a-t-on fini la boucle interne ?
        SUB     r31,r10,r20     ; comparer l'élément avec le plus petit déjà vu
        JLE     r31,lpInt
        MOV     r2,r1           ; s'il est plus petit, on mémorise son pointeur
        LDB     r10,(r2)        ; et l'élément lui-même
        JMP     lpInt
suite:  LDB     r30,(r0)        ; on permute l'élément courant de la boucle externe
        LDB     r31,(r2)        ; et le plus petit trouvé dans la boucle interne
        STB     (r0),r31
        STB     (r2),r30
        ADD     r0,r0,#1        ; passer au suivant dans la boucle externe
        JMP     lpExt
fin:
```

III. Circuits logiques

On souhaite construire un incrémenteur/décrémenteur sur n bits, c'est-à-dire un circuit à n bits d'entrée $A = a_{n-1} \dots a_0$ représentant un nombre binaire **signé** et une ligne de commende F, qui génère n bits de sortie $S = s_{n-1} \dots s_0$ représentant la valeur binaire $A + 1$ (si $F = 0$) ou $A - 1$ (si $F = 1$), ainsi qu'un bit de débordement X valant 1 si la valeur souhaitée ($A + 1$ ou $A - 1$ suivant le cas) ne peut pas être représentée sur n bits.

1. On appelle r_i la retenue intermédiaire utilisée pour calculer $s_{i+1} = a_{i+1} + r_i$ ou $s_{i+1} = a_{i+1} - r_i$. Donner la table de vérité de s_0 et r_0 en fonction de a_0 et F. Donner les expressions logiques de s_0 et r_0.
2. Donner la table de vérité de s_i et r_i en fonction de F, a_i et r_{i-1}. Donner les expressions logiques de s_i et r_i.

3. La retenue finale se calculant en propageant toutes les retenues intermédiaires, son calcul est très long. Donner une expression des s_i et r_i permettant de les calculer beaucoup plus rapidement (sous réserve de disposer des bonnes portes logiques).
4. Quels sont les deux cas de débordement ? Y a-t-il débordement lorsqu'il y a une retenue finale r_{n-1} ? Lorsqu'il y a une retenue r_{n-2}, y a-t-il des cas de débordement ? Proposer une expression logique pour le bit de débordement faisant intervenir ces deux retenues.

Corrigé :

1.

F	a_0	s_0	r_0
0	0	1	0
0	1	0	1
1	0	1	1
1	1	0	0

$s_0 = \overline{a_0}$

$r_0 = F \oplus a_0$

2.

F	a_i	r_{i-1}	s_i	r_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	0	0
1	0	1	1	1
1	1	0	1	0
1	1	1	0	0

$s_i = a_i \oplus r_{i-1}$

$r_i = \overline{F}.a_i.r_{i-1} + F.\overline{a_i}.r_{i-1} = r_{i-1}(F \oplus a_i)$

3. Par récurrence, on peut écrire

$$r_i = \prod_{j=0}^{i} (F \oplus a_j)$$

$$s_i = a_i \oplus \prod_{j=0}^{i-1} (F \oplus a_j)$$

4. Les nombres étant signés, il y a débordement dans deux cas :
   - lors d'une addition ($F = 0$), lorsque l'on passe de $011\ldots11$ à $100\ldots00$;
   - lors d'une soustraction ($F = 1$), lorsque l'on passe de $100\ldots00$ à $011\ldots11$.

   Il y a une retenue finale r_{n-1} lorsque :
   - lors d'une addition ($F = 0$), on passe de $111\ldots11$ à $000\ldots00$;
   - lors d'une soustraction ($F = 1$), on passe de $000\ldots00$ à $111\ldots11$.

   Dans ces deux cas il n'y a pas de débordement puisque les nombres sont signés.
   Il y a une retenue r_{n-2} lorsque :
   - lors d'une addition ($F = 0$), on passe de $111\ldots11$ à $000\ldots00$ (sans débordement) ;
   - lors d'une addition ($F = 0$), on passe de $011\ldots11$ à $100\ldots00$ (avec débordement) ;

– lors d'une soustraction (F = 1), on passe de 000 . . . 00 à 111 . . . 11 (sans débordement) ;
– lors d'une soustraction (F = 1), on passe de 100 . . . 00 à 011 . . . 11 (avec débordement).

On a donc un débordement lorsqu'il y a une retenue r_{n-2} mais pas de retenue finale :
$X = r_{n-2}.\overline{r_{n-1}}$

IV. Mémoire cache

Un programme se compose d'une boucle de 24 instructions à exécuter 5 fois ; les instructions se trouvant, dans l'ordre, aux adresses mémoire 13 à 18, 1 à 6, 13 à 18, 7 à 12. Ce programme doit tourner sur une machine possédant un cache d'une taille de 12 instructions. Le temps de cycle de la mémoire principale est M et le temps de cycle du cache est C. Le cache est associatif (un bloc mémoire peut venir dans n'importe quel bloc du cache) et la stratégie de remplacement utilisée est LRU (on remplace le bloc le moins récemment utilisé).

1. Le cache possède 6 blocs de 2 instructions : les blocs que l'on peut transférer sont 1-2, 3-4, ..., 13-14, 15-16, 17-18... 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 cache possède 3 blocs de 4 instructions : les blocs que l'on peut transférer sont 1-4, 5-8, ..., 13-16, 17-20... 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. Le cache possède 2 blocs de 6 instructions : les blocs que l'on peut transférer sont 1-6, 7-12, 13-18... Quel est le temps total d'exécution du programme en ne tenant pas compte des temps de calcul ? Le cache est vide au départ.
4. Le cache possède 1 bloc de 12 instructions : les blocs que l'on peut transférer sont 1-12, 13-24... Quel est le temps total d'exécution du programme en ne tenant pas compte des temps de calcul ? Le cache est vide au départ.
5. Qu'en concluez-vous sur l'efficacité du cache ?

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.

$13 \to 14$	$M + C$	bloc 1	$13 \to 14$	$2C$	bloc 1	
$15 \to 16$	$M + C$	bloc 2	$15 \to 16$	$2C$	bloc 2	
$17 \to 18$	$M + C$	bloc 3	$17 \to 18$	$2C$	bloc 3	
$1 \to 2$	$M + C$	bloc 4	$1 \to 2$	$M + C$	bloc 4	
$3 \to 4$	$M + C$	bloc 5	$3 \to 4$	$M + C$	bloc 5	
$5 \to 6$	$M + C$	bloc 6	$5 \to 6$	$M + C$	bloc 6	Les trois dernières
$13 \to 14$	$2C$	bloc 1	$13 \to 14$	$2C$	bloc 1	itérations
$15 \to 16$	$2C$	bloc 2	$15 \to 16$	$2C$	bloc 2	sont identiques
$17 \to 18$	$2C$	bloc 3	$17 \to 18$	$2C$	bloc 3	
$7 \to 8$	$M + C$	bloc 4	$7 \to 8$	$M + C$	bloc 4	
$9 \to 10$	$M + C$	bloc 5	$9 \to 10$	$M + C$	bloc 5	
$11 \to 12$	$M + C$	bloc 6	$11 \to 12$	$M + C$	bloc 6	

Soit un total de $33M + 87C$.

2.

$13 \to 16$	$\mathrm{M} + 3\mathrm{C}$	bloc 1	$13 \to 16$	$\mathrm{M} + 3\mathrm{C}$	bloc 3	
$17 \to 18$	$\mathrm{M} + \mathrm{C}$	bloc 2	$17 \to 18$	$\mathrm{M} + \mathrm{C}$	bloc 1	
$1 \to 4$	$\mathrm{M} + 3\mathrm{C}$	bloc 3	$1 \to 4$	$\mathrm{M} + 3\mathrm{C}$	bloc 2	
$5 \to 6$	$\mathrm{M} + \mathrm{C}$	bloc 1	$5 \to 6$	$\mathrm{M} + \mathrm{C}$	bloc 3	Les trois dernières
$13 \to 16$	$\mathrm{M} + 3\mathrm{C}$	bloc 2	$13 \to 16$	$\mathrm{M} + 3\mathrm{C}$	bloc 1	itérations
$17 \to 18$	$\mathrm{M} + \mathrm{C}$	bloc 3	$17 \to 18$	$\mathrm{M} + \mathrm{C}$	bloc 2	sont identiques
$7 \to 8$	$2\mathrm{C}$	bloc 1	$7 \to 8$	$2\mathrm{C}$	bloc 3	
$9 \to 12$	$\mathrm{M} + 3\mathrm{C}$	bloc 2	$9 \to 12$	$\mathrm{M} + \mathrm{C}$	bloc 1	

Soit un total de $35\mathrm{M} + 85\mathrm{C}$.

3.

$13 \to 18$	$\mathrm{M} + 5\mathrm{C}$	bloc 1	$13 \to 18$	$6\mathrm{C}$	bloc 1	
$1 \to 6$	$\mathrm{M} + 5\mathrm{C}$	bloc 2	$1 \to 6$	$\mathrm{M} + 5\mathrm{C}$	bloc 2	Les trois dernières
$13 \to 18$	$6\mathrm{C}$	bloc 1	$13 \to 18$	$6\mathrm{C}$	bloc 1	itérations
$7 \to 12$	$\mathrm{M} + 5\mathrm{C}$	bloc 2	$7 \to 12$	$\mathrm{M} + 5\mathrm{C}$	bloc 2	sont identiques

Soit un total de $11\mathrm{M} + 109\mathrm{C}$.

4.

$13 \to 18$	$\mathrm{M} + 5\mathrm{C}$	bloc 1	$13 \to 18$	$\mathrm{M} + 5\mathrm{C}$	bloc 1	
$1 \to 6$	$\mathrm{M} + 5\mathrm{C}$	bloc 1	$1 \to 6$	$\mathrm{M} + 5\mathrm{C}$	bloc 1	Les trois dernières
$13 \to 18$	$\mathrm{M} + 5\mathrm{C}$	bloc 1	$13 \to 18$	$\mathrm{M} + 5\mathrm{C}$	bloc 1	itérations
$7 \to 12$	$\mathrm{M} + 5\mathrm{C}$	bloc 1	$7 \to 12$	$\mathrm{M} + 5\mathrm{C}$	bloc 1	sont identiques

Soit un total de $20\mathrm{M} + 100\mathrm{C}$.

5. L'efficacité du cache n'est pas proportionnelle à la taille de ses blocs. De gros blocs permettent de mémoriser plus d'information à chaque accès mémoire mais de petits blocs permettent de garder des parties du programme qui sont souvent réutilisées.