Architecture des ordinateurs (E. Lazard)
Examen du 7 février 2014

(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 : $2^{-1} = 0{,}5$; $2^{-2} = 0{,}25$; $2^{-3} = 0{,}125$; $2^{-4} = 0{,}0625$; $2^{-5} = 0{,}03125$). **Chaque calcul se fait sur tous les chiffres significatifs et l'arrondi s'effectue supérieurement.**

1. Représenter 6 et 31 en virgule flottante.
2. Calculer la multiplication virgule flottante de 6 et 31. Quelle est la valeur exacte obtenue ? Quelle est la valeur arrondie obtenue ?
3. Comparer les valeurs obtenues pour chacun des deux calculs suivants effectués en virgule flottante : $(31 + 1) \times 6$ et $31 \times 6 + 6$.

Corrigé :

1. $6 = 1{,}5 \times 2^2 = \boxed{0\,1001\,10000}$
   $31 = 1{,}9375 \times 2^4 = \boxed{0\,1011\,11110}$
2. La multiplication des deux mantisses $1{,}10000_2$ et $1{,}11110_2$ donne $10{,}111010_2$ qui se normalise en $1{,}0111010_2$ avec un exposant de $2 + 4 + 1$ soit : $1{,}0111010_2 \times 2^7 = 2^7 + 2^5 + 2^4 + 2^3 + 2^1 = 186$. Après arrondi supérieurement, la mantisse devient $1{,}01111_2$. Le résultat obtenu est alors $1{,}01111_2 \times 2^7 = 2^7 + 2^5 + 2^4 + 2^3 + 2^2 = 188$. (Notons qu'un arrondi inférieur aurait donné $1{,}01110_2 \times 2^7 = 184$.)
3. $31 + 1 = 32$ s'écrit exactement $1{,}00000_2 \times 2^5$ qui multiplié par $6 = 1{,}10000_2 \times 2^2$ donne $1{,}10000_2 \times 2^7 = 192$ qui est la valeur exacte. En revanche, 31×6 s'écrit $1{,}01111_2 \times 2^7$ qui additionné avec $6 = 1{,}10000_2 \times 2^2 = 0{,}000011_2 \times 2^7$ donne $1{,}100001_2 \times 2^7$ qui s'arrondit en $1{,}10001_2 \times 2^7$ et vaut $2^7 + 2^6 + 2^2 = 196$.

II. Circuits logiques

Soit une machine qui travaille sur des nombres binaires de 3 bits représentés en « signe et valeur absolue ». La valeur d'un mot $A = a_2a_1a_0$ est égale à $(-1)^{a_2} \times (2a_1 + a_0)$.

On désire construire un circuit qui donne en sortie un nombre binaire $B = b_2b_1b_0$ représentant le même nombre A mais cette fois écrit en représentation complément à 2.

1. Toutes les valeurs sont-elles autorisées en entrée ? Autrement dit, le circuit donne-t-il toujours une valeur correcte ?
2. Donner la table de vérité du circuit.
3. Donner les expressions logiques des 3 bits de sortie en fonction des 3 bits d'entrée en simplifiant au maximum les expressions.

CORRIGÉ :

1. Les valeurs représentables en « signe et valeur absolue » vont de -3 à 3, tout aussi représentables en complément à deux. Il n'y a donc pas de valeur interdite.
2.

A	a_2	a_1	a_0	b_2	b_1	b_0
0	0	0	0	0	0	0
1	0	0	1	0	0	1
2	0	1	0	0	1	0
3	0	1	1	0	1	1
-0	1	0	0	0	0	0
-1	1	0	1	1	1	1
-2	1	1	0	1	1	0
-3	1	1	1	1	0	1

3. On a :

$$b_0 = a_0$$

$$b_1 = \overline{a_2}a_1\overline{a_0} + \overline{a_2}a_1a_0 + a_2\overline{a_1}a_0 + a_2a_1\overline{a_0}$$

qui s'écrit aussi :

$$b_1 = \overline{a_2}a_1 + a_2(a_1 \oplus a_0)$$

Quant au bit de signe, il ne change pas, sauf pour la valeur -0 qui se transforme en $+0$:

$$b_2 = a_2(a_1 + a_0)$$

III. 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 deux sorties s_0 et s_1 tel que :

- si $F = 0$, le circuit se comporte comme un demi-soustracteur, la sortie s_0 représentant la différence des deux bits et s_1 la retenue ;
- si $F = 1$, le circuit se comporte comme une unité logique, la sortie s_0 représentant alors le NAND des deux entrées et la sortie s_1 le XOR des deux entrées.

1. Construire les deux tables de vérité (pour $F = 0$ et $F = 1$).
2. Exprimer s_0 et s_1 en fonction de a, b et F.

CORRIGÉ :

1. Pour $F = 0$

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

pour $F = 1$

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

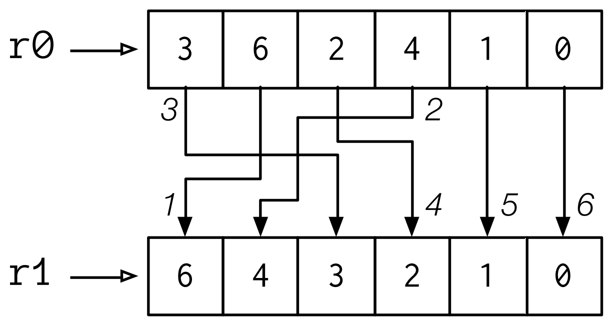
2. $s_0 = (a \oplus b)\overline{F} + \overline{ab}F = (a \oplus b) + (\overline{a + b})F$
$s_1 = \overline{a}b\overline{F} + (a \oplus b)F$

IV. Assembleur

On souhaite implémenter en assembleur l'algorithme de tri qui, à partir d'un tableau d'entiers pointé par `r0`, le réécrit trié en ordre décroissant dans un tableau pointé par `r1`. On utilise l'algorithme suivant :

- Le tableau d'entiers pointé par `r0` contient des entiers entre 1 et 127, chacun sur un octet. Le tableau se termine par un élément nul.
- On parcourt tout le tableau en mémorisant le plus grand élément.
- On l'enlève du tableau et on le réécrit à la première case du tableau pointé par `r1`.
- On recommence à chaque fois la boucle en cherchant à nouveau le plus grand élément et en le réécrivant à la suite dans le tableau pointé par `r1`.
- L'algorithme se termine lorsque tous les éléments du premier tableau ont été enlevés ; on complète alors le deuxième tableau par un octet nul.

Le dessin ci-dessous est un exemple d'exécution de la procédure, les numéros sur les flèches indiquant l'ordre de recopie des éléments du premier tableau.



Écrire une procédure assembleur qui, lorsqu'elle se termine, assure que le tableau initialement pointé par `r0` est recopié, trié en ordre décroissant, à partir de l'adresse pointée par `r1`. On ne vous demande pas de conserver les valeurs de `r0` et `r1` à la fin.

CORRIGÉ :

La seule difficulté est de remplacer les éléments du premier tableau lorsqu'on les recopie par une valeur qui ne peut être ni positive, ni zéro (qui indique la fin du tableau) ; c'est pourquoi on les remplace par −1.

Listing 1. Tri par récopie

```
loopE:  MOV    r10,r0          ; début premier tableau
        MVI    r30,#0          ; mémorisation plus grand
loopI:  LDB    r2,(r10)        ; élément courant
        JZ     r2,suite        ; fin du parcours du tableau
        SUB    r31,r2,r30      ; comparer courant et plus grand
        JLE    r31,next        ; passer au suivant si plus petit
        MOV    r30,r2          ; nouveau plus grand
        MOV    r11,r10         ; mémorisation de sa place
next:   ADD    r10,r10,#1      ; élément suivant
        JMP    loopI           ; on revient à la boucle interne
; À la fin de la boucle interne, on a le plus grand
; élément dans r30 et son adresse dans r11.
suite:  STB    (r1),r30        ; on recopie le plus grand
        JZ     r30,fin         ; si r30 = 0, c'est la fin
        ADD    r1,r1,#1        ; avancer le pointeur
        MVI    r30,#-1         ; on enlève le plus grand du tableau
        STB    (r11),r30       ; en le remplaçant par -1
        JMP    loopE           ; on recommence avec le plus grand suivant
fin:
```

V. Mémoire cache

Une machine possède un cache de 12 octets. Au cours d'un programme, elle accède successivement les octets situés aux adresses mémoire :

3, 2, 8, 6, 20, 18, 19, 68, 2, 10, 11, 40, 18, 6, 5, 10

1. On suppose le cache initialement vide et organisé en 12 blocs de 1 octet à accès direct (dans le bloc 1 du cache vont les octets mémoire 1, 13, 25...). Combien sont effectués d'accès cache et d'accès mémoire pour la suite des références mémoire ci-dessus ?
2. On suppose le cache initialement vide et organisé en 3 blocs de 4 octets à accès direct (dans le bloc 1 du cache vont les blocs mémoire 1-4, 13-16, 25-28...). Combien sont effectués d'accès cache et d'accès mémoire pour la suite des références mémoire ci-dessus ?
3. On suppose le cache initialement vide et organisé en 3 blocs de 4 octets à accès associatif (avec l'algorithme de remplacement LRU). Combien sont effectués d'accès cache et d'accès mémoire pour la suite des références mémoire ci-dessus ?

CORRIGÉ :

1. M : 3, 2, 8, 6, 20 (remp. 8), 18 (remp. 6), 19, 68 (remp. 20), 10, 11, 40, 6 (remp. 18), 5

   C : 2, 18, 10

   Soit 13 accès mémoire et 3 accès cache.
2.

3	M	bloc 1 (1-4)	20	M	bloc 2 (17-20)	2	C	bloc 1	18	M	bloc 2 (17-20)
2	C	bloc 1	18	C	bloc 2	10	M	bloc 3 (9-12)	6	M	bloc 2 (5-8)
8	M	bloc 2 (5-8)	19	C	bloc 2	11	C	bloc 3	5	C	bloc 2
6	C	bloc 2	68	M	bloc 2 (65-68)	40	M	bloc 1 (37-40)	10	C	bloc 3

   Soit 8 accès mémoire et 8 accès cache.
3.

3	M	bloc 1 (1-4)	20	M	bloc 3 (17-20)	2	M	bloc 2 (1-4)	18	M	bloc 2 (17-20)
2	C	bloc 1	18	C	bloc 3	10	M	bloc 3 (9-12)	6	M	bloc 3 (5-8)
8	M	bloc 2 (5-8)	19	C	bloc 3	11	C	bloc 3	5	C	bloc 3
6	C	bloc 2	68	M	bloc 1 (65-68)	40	M	bloc 1 (37-40)	10	M	bloc 1 (9-12)

   Soit 10 accès mémoire et 6 accès cache.