UNIVERSITÉ DE PARIS-DAUPHINE L2 MI2E

Architecture des ordinateurs (E. Lazard)
Examen du 15 janvier 2018

(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 6,5 ; 3,5 et 12 en virgule flottante.
2. On écrit le code suivant :

Listing 1. calculs

```
float a = 6.5;
float b = 3.5;
float x = a * b - 12;
```

   (a) Quelle valeur obtient-on pour `x` si on suppose que tous les calculs sont exacts (sans arrondi) ?

   (b) 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 `x` ?

   (c) 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 `x` ?

   (d) Quelle conclusion en tirez-vous ?

Corrigé :

1. $6{,}5 = 1{,}625 \times 2^2 = \boxed{0\ 1001\ 10100}$
   $3{,}5 = 1{,}75 \times 2^1 = \boxed{0\ 1000\ 11000}$
   $12 = 1{,}5 \times 2^3 = \boxed{0\ 1010\ 10000}$
2. (a) Si les calculs étaient parfaits, x vaudrait $6{,}5 \times 3{,}5 - 12 = 22{,}75 - 12 = 10{,}75$
   (b) L'opération `a * b` se calcule par $1{,}101 \times 2^2 \times 1{,}110 \times 2^1$ ce qui donne exactement $10{,}11011 \times 2^3$, normalisé en $1{,}011011 \times 2^4$, éventuellement arrondi à $1{,}01101 \times 2^4$. L'arrondi intervenant tout de suite, on doit calculer $1{,}01101 \times 2^4 - 1{,}1 \times 2^3$ qui donne $(1{,}01101 - 0{,}11) \times 2^4 = 0{,}10101 \times 2^4$ normalisé en $1{,}0101 \times 2^3$ soit 10,5.

(c) Puisqu'il n'y a pas d'arrondi après la multiplication, on doit calculer $1{,}011011 \times 2^4 - 1{,}1 \times 2^3$ qui donne $(1{,}011011 - 0{,}11) \times 2^4 = 0{,}101011 \times 2^4$ normalisé en $1{,}01011 \times 2^3$. Il n'y a alors pas d'arrondi à effectuer et la valeur finale est 10,75, soit la valeur correcte.

(d) 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. Circuits logiques

Soit une machine qui travaille en entrée sur des nombres binaires *non-signés* de 3 bits.

On désire construire un circuit qui donne en sortie le quotient et le reste de la division de l'entrée N par 3 suivant la règle $N = 3q + r$ avec $0 \leqslant r < 3$.

1. Quel est l'intervalle de valeurs possibles pour l'entrée N ? Quel est l'intervalle de valeurs obtenues pour le quotient q et le reste r ?
2. Donner la table de vérité du circuit.
3. Donner les expressions logiques des bits de sortie (ceux du quotient et ceux du reste) en fonction des 3 bits d'entrée en simplifiant au maximum les expressions.

Corrigé :

1. Sur 3 bits non-signés, on exprime les valeurs de 0 à 7. Le quotient va donc de 0 à 2 et le reste de 0 à 2, nécessitant chacun 2 bits.
2.

a_2	a_1	a_0	q_1	q_0	r_1	r_0
0	0	0	0	0	0	0
0	0	1	0	0	0	1
0	1	0	0	0	1	0
0	1	1	0	1	0	0
1	0	0	0	1	0	1
1	0	1	0	1	1	0
1	1	0	1	0	0	0
1	1	1	1	0	0	1

3. On voit donc que l'on a :

$q_1 = a_2 a_1$

$q_0 = \overline{a_2} a_1 a_0 + a_2 \overline{a_1}\,\overline{a_0} + a_2 \overline{a_1} a_0 = a_2 \overline{a_1} + \overline{a_2} a_1 a_0$

$r_1 = \overline{a_2} a_1 \overline{a_0} + a_2 \overline{a_1} a_0$

$r_0 = \overline{a_2}\,\overline{a_1} a_0 + a_2 \overline{a_1}\,\overline{a_0} + a_2 a_1 a_0 = \overline{a_1}(a_2 \oplus a_0) + a_2 a_1 a_0$

III. Assembleur

Un tableau d'entiers est stocké en mémoire. Chaque élément a une valeur de 1 à 127 et est donc stocké sur un octet. L'octet qui suit le dernier élément du tableau est nul. L'adresse du premier élément du tableau se trouve dans le registre `r0`.

On souhaite recopier ces éléments dans deux tableaux, le premier, dont l'adresse est dans `r1`, va contenir tous les éléments impairs et le second, dont l'adresse est dans `r2`, les éléments pairs. On terminera l'écriture de chacun des tableaux par un octet nul suivi, sur deux octets, de la somme des éléments correspondants.

Écrire une procédure assembleur qui, lorsqu'elle se termine, a recopié les éléments du tableau initial dans les deux tableaux suivis de la somme des éléments correspondants.

Exemple : si le tableau initial contient `1 2 10 3 14 0`, les deux tableaux finaux seront `1 3 0 4` et `2 10 14 0 26`.

Corrigé :

Listing 2. recopie pair-impair

```
        MVI     r11, #0         ; initialisation de la somme des impairs
        MVI     r12, #0         ; initialisation de la somme des pairs
loop:   LDB     r31, (r0)       ; charger le prochain nombre
        JZ      r31, fin        ; est-ce fini ?
        ADD     r0, r0, #1      ; passer au suivant
        AND     r30, r31, #1    ; tester la parité
        JZ      r30, pair       ; sauter au traitement pair
        ADD     r11, r11, r31   ; sinon additionner un nombre impair
        STB     (r1), r31       ; et le recopier
        ADD     r1, r1, #1      ; pointer la case suivante
        JMP     loop            ; retour à la boucle
pair:   ADD     r12, r12, r31   ; additionner un nombre pair
        STB     (r2), r31       ; et le recopier
        ADD     r2, r2, #1      ; pointer la case suivante
        JMP     loop            ; retour à la boucle
fin:    STB     (r1), r31       ; stockage du zéro final
        ADD     r1, r1, #1
        STB     (r2), r31       ; stockage du zéro final
        ADD     r2, r2, #1
        STH     (r1), r11       ; stockage de la somme des impairs
        STH     (r2), r12       ; stockage de la somme des pairs

; pour calculer la parité, on peut aussi récupérer le bit de poids faible.
        LLS     r30, r31, #-1
        CCR     r30
; ou calculer le modulo 2
        DIV     r30, r31, #2
        MUL     r30, r30, #2
        SUB     r30, r31, r30
```

IV. Mémoire cache

Un programme se compose d'une boucle de 42 instructions à exécuter 4 fois ; cette boucle se trouve aux adresses mémoire 1 à 42. Ce programme doit tourner sur une machine possédant un cache d'une taille de 28 instructions. Le temps de cycle de la mémoire principale est M et le temps de cycle du cache est C.

1. Le cache est à correspondance directe, formé de 7 blocs de 4 instructions. On rappelle que cela veut dire que les mots mémoire 1 à 4, 29 à 32... vont dans le premier bloc, que les mots 5 à 8, 33 et 36... vont dans le deuxième, etc. Quel est le temps total d'exécution du programme en ne tenant pas compte des temps de calcul ?
2. Le cache est maintenant 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). Quel est le temps d'exécution du programme ?
3. Refaire les deux premières questions en prenant un cache composé de 14 blocs de 2 mots.

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. Les quatre premiers blocs sont partagés
   ($1-4$ et $29-32$, $5-8$ et $33-36$, $9-12$ et $37-40$, $13-16$ et $41-44$), donc :

$$T = 10(M + 3C) + (M + C) + 3[7(M + 3C) + (M + C) + 12C] = 35M + 133C$$

2. On ne réutilise jamais un bloc, donc :

$$T = 4[10(M + 3C) + (M + C)] = 44M + 124C$$

3. (a) Les sept premiers blocs sont partagés, donc :

$$T = 14(M + C) + 7(M + C) + 3[14(M + C) + 14C] = 63M + 105C$$

   (b) On ne réutilise jamais un bloc, donc :

$$T = 4[21(M + C)] = 84M + 84C$$