Architecture des ordinateurs (E. Lazard)
Examen du 10 janvier 2022

(durée 2 heures) – CORRECTION

Documents autorisés : une feuille A4 manuscrite recto-verso
Calculatrice autorisée

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 supérieurement lorsque cela est indiqué.

1. Représenter 6,5 ; 7 et 24 en virgule flottante.
2. On écrit le code suivant :

Listing 1. calculs

```
float a = 6.5;
float b = 7;
float x = a * b - 24;
```

   (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 supérieurement 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. $6{,}5 = 1{,}625 \times 2^2 = \boxed{0\ 1001\ 10100}$
   $7 = 1{,}75 \times 2^2 = \boxed{0\ 1001\ 11000}$
   $24 = 1{,}5 \times 2^4 = \boxed{0\ 1011\ 10000}$
2. (a) Si les calculs étaient parfaits, x vaudrait $6{,}5 \times 7 - 24 = 45{,}5 - 24 = 21{,}5$

(b) L'opération `a * b` se calcule par $1{,}101 \times 2^2 \times 1{,}110 \times 2^2$ ce qui donne exactement $10{,}11011 \times 2^4$, normalisé en $1{,}011011 \times 2^5$, éventuellement arrondi à $1{,}01110 \times 2^5$.
L'arrondi intervenant tout de suite, on doit calculer $1{,}0111 \times 2^5 - 1{,}1 \times 2^4$ qui donne $(1{,}0111 - 0{,}11) \times 2^5 = 0{,}1011 \times 2^5$ normalisé en $1{,}011 \times 2^4$ soit 22.

(c) Puisqu'il n'y a pas d'arrondi après la multiplication, on doit calculer $1{,}011011 \times 2^5 - 1{,}1 \times 2^4$ qui donne $(1{,}011011 - 0{,}11) \times 2^5 = 0{,}101011 \times 2^5$ normalisé en $1{,}01011 \times 2^4$. Il n'y a alors pas d'arrondi à effectuer et la valeur finale est 21,5, 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

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 :

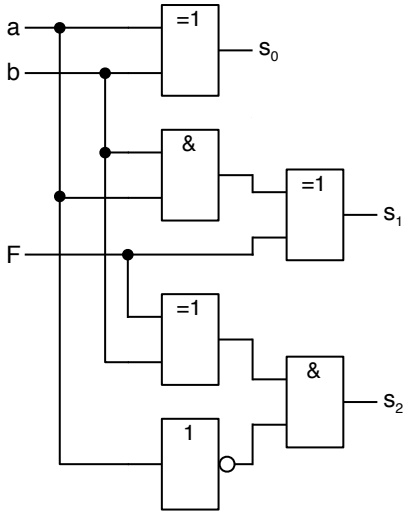
1. Pour $F = 0$

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 :



III. 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 savoir si cette chaîne est un palindrome, c'est-à-dire qu'elle est identique quel que soit le sens de lecture (de gauche à droite ou de droite à gauche). Par exemple, `"laval"`, `"abba"` et `"ici ici"` sont des palindromes. On ne distinguera pas les lettres des autres caractères ; autrement dit, il faut absolument que la chaîne soit symétrique caractère par caractère. Ainsi, `"la val"` n'est pas un palindrome car l'espace « ne colle pas » avec le `v`. Une chaîne vide ne sera pas considérée comme un palindrome valide.

Écrire une procédure assembleur qui, lorsqu'elle se termine, met la valeur 1 dans le registre `r1` si la chaîne pointée par `r0` est un palindrome et 0 sinon. On ne vous demande pas de conserver la valeur de `r0` à la fin.

Corrigé :

Listing 2. Validation palindrome

```
        MVI     r1,#0
        LDB     r2,(r0)         ; Test si la chaîne est vide
        JZ      r2,fin
        MOV     r10,r0
loop:   ADD     r10,r10,#1      ; parcours de la chaîne
        LDB     r2,(r10)        ; jusqu'au dernier caractère
        JNZ     r2,loop
        SUB     r10,r10,#1      ; sur lequel pointe maintenant r10
cmp:    LDB     r2,(r0)         ; chargement des deux caractères
        LDB     r3,(r10)        ; symétriquement placés
        SUB     r30,r2,r3       ; et comparaison
        JNZ     r30,fin         ; si différent, c'est fini
        ADD     r0,r0,#1        ; décalage des pointeurs
        SUB     r10,r10,#1
        SUB     r30,r0,r10      ; comparaison des deux pointeurs
        JLT     r30, cmp        ; et retour dans la boucle
        MVI     r1,#1           ; ok, le milieu est dépasé
fin:
```

IV. 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 3. 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:
```

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.