Architecture des ordinateurs (E. Lazard)
Examen du 10 janvier 2017

(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.

1. Représenter 1 et 10 en virgule flottante.
2. Quels sont les deux nombres représentables encadrants 0,1 (valeurs décimales et représentations) ?
3. En prenant pour 0,1 sa représentation arrondie inférieurement, calculer la valeur binaire exacte et la représentation arrondie inférieurement de l'opération de multiplication flottante $0{,}1 \times 10$. Quelle est la valeur décimale finale obtenue (après arrondi inférieur) ?
4. On écrit le code suivant :

Listing 1. Calculs

```
float f = 1.0;
while (f > 0)
    f = (0.1*10)*f;
```

   (a) Que vaut `f` après la première itération ? puis après une deuxième ?
   (b) Que va-t-il se passer ensuite ?

Corrigé :

1. $1 = 1 \times 2^0 = \boxed{0\,0111\,00000}$
   $10 = 1{,}25 \times 2^3 = \boxed{0\,1010\,01000}$
2. $0{,}1 = 1{,}6 \times 2^{-4}$ et 1,6 est encadré par $1 + 2^{-1} + 2^{-4} + 2^{-5} = 1{,}59375$ et $1 + 2^{-1} + 2^{-3} = 1{,}625$. On a donc $0{,}099609375 < 0{,}1 < 0{,}1015625$ respectivement écrits $\boxed{0\,0011\,10011}$ et $\boxed{0\,0011\,10100}$.
3. La multiplication flottante donne $1{,}10011_2 \times 2^{-4} \times 1{,}01_2 \times 2^3 = 1{,}1111111_2 \times 2^{-1}$ qui est arrondi à $1{,}11111_2 \times 2^{-1} = 0{,}984375$.
4. (a) `0.1*10` vaut en fait $0{,}984375 = 1 - 2^{-6}$. `f` va donc valoir cette valeur-là après une itération. Après une deuxième itération, on a $f = (1 - 2^{-6})^2 \approx 1 - 2^{-5}$ après arrondi.
   (b) La constante étant légèrement inférieure à 1, `f` va décroitre. On peut en fait être un peu plus précis. La formule $f = (1 - 2^{-6}) \times f$ se traduit par $f = (1{,}m - 2^{-6} - m \times 2^{-6}) \times 2^e$ qui, en raison de l'arrondi, devient $f = (1{,}m - 2^{-5}) \times 2^e$. Autrement dit, on retranche 2^{-5} à la pseudo-mantisse de f à chaque itération. On passe d'abord de $1{,}abcde_2 \times 2^e$ à $1{,}00000_2 \times 2^e$ puis à $0{,}111111_2 \times 2^e = 1{,}11111_2 \times 2^{e-1}$ pour finir à $1{,}abcde_2 \times 2^{e-1}$ au bout de 32 itérations.
   On démarre à $1{,}0 = 1{,}00000_2 \times 2^0$. Au bout de $6 \times 32 = 192$ itérations, on arrive à $1{,}00000_2 \times 2^{-6}$ qui est le plus petit nombre strictement positif représentable. La 193^e itération amène donc `f` à zéro et à la fin de la boucle.

II. Circuits logiques

Soit une machine qui travaille sur des nombres binaires signés de 4 bits. La valeur signée d'un mot $A = a_3a_2a_1a_0$ est égale à $a_0 + 2a_1 + 4a_2 - 8a_3$ (c'est la représentation classique en complément à 2).

On désire construire un circuit qui donne en sortie un nombre binaire $B = b_3b_2b_1b_0$ représentant le même nombre A mais cette fois écrit en représentation « signe et valeur absolue » (b_3 étant le signe et $b_2b_1b_0$ la valeur absolue).

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 pour toutes les valeurs autorisées.
3. Donner les expressions logiques des 4 bits de sorties en fonction des 4 bits d'entrée en **simplifiant au maximum les expressions** et en ne tenant pas compte des valeurs interdites (c'est-à-dire que les sorties peuvent être quelconques dans les cas interdits).
4. Donner l'expression logique d'un bit supplémentaire en sortie, e *(débordement)*, qui indiquerait qu'une valeur interdite est présente sur les entrées.

Corrigé :

1. La valeur $-8 = 1000_2$ ne peut pas se représenter en convention « signe et valeur absolue ».
2.

a_3	a_2	a_1	a_0	b_3	b_2	b_1	b_0	b_3	b_2	b_1	b_0	a_3	a_2	a_1	a_0
0	0	0	0	0	0	0	0	–	–	–	–	1	0	0	0
0	0	0	1	0	0	0	1	1	1	1	1	1	0	0	1
0	0	1	0	0	0	1	0	1	1	1	0	1	0	1	0
0	0	1	1	0	0	1	1	1	1	0	1	1	0	1	1
0	1	0	0	0	1	0	0	1	1	0	0	1	1	0	0
0	1	0	1	0	1	0	1	1	0	1	1	1	1	0	1
0	1	1	0	0	1	1	0	1	0	1	0	1	1	1	0
0	1	1	1	0	1	1	1	1	0	0	1	1	1	1	1

3. La parité est conservée ; on a donc :

$$b_0 = a_0$$

On a ensuite :

$$b_1 = \overline{a_3}a_1 + a_3(a_1 \oplus a_0)$$

$$b_2 = \overline{a_3}a_2 + a_3\overline{a_2} + a_3a_2\overline{a_1}\,\overline{a_0} = (a_3 \oplus a_2) + a_3a_2\overline{a_1}\,\overline{a_0}$$

On peut aussi trouver :

$$b_2 = \overline{a_3}a_2 + a_3(a_2 \oplus (a_1 + a_0))$$

Et comme le bit de poids fort représente toujours le signe :

$$b_3 = a_3$$

4. Le bit de débordement est mis à 1 pour la seule valeur interdite, -8 : $e = a_3\overline{a_2}\,\overline{a_1}\,\overline{a_0}$.

III. Assembleur

1. Une chaîne de caractères composée uniquement de lettres majuscules et d'espaces 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`.
   Afin de transmettre secrètement cette chaîne, on souhaite décaler chaque lettre (mais pas les espaces) d'une valeur positive contenue dans le registre `r1`, cette valeur étant inférieure à 26. Ainsi, si la chaîne est « `ARCHI TD` » et `r1` contient 4, on veut obtenir la chaîne « `EVGLM XH` ». Le décalage se fait circulairement : après `Z`, on retrouve `A`, puis `B`…
   Écrire une procédure assembleur qui, lorsqu'elle se termine, assure que la chaîne initialement pointée par `r0` (dont on n'a pas besoin de conserver la valeur initiale) contient la chaîne d'origine décalée.
   Remarque : on pourra écrire `#'A'` pour utiliser comme valeur immédiate le code ASCII du caractère `A`.
2. Notre chaîne de départ est toujours stockée en mémoire à l'adresse contenue dans le registre `r0`.
   Toujours afin de transmettre secrètement cette chaîne, une clé de codage est utilisée dans laquelle les 26 lettres sont rangées dans le désordre. Par exemple, une clé de codage possible serait « `CLUMNPHKOZYWVQBIFDJAXREGTS` ». L'adresse du premier caractère de cette clé de codage se trouve dans le registre `r1`. Cette clé de codage sert ensuite à chiffrer le message : on substitue à chaque lettre du message (mais pas aux espaces) celle située dans la clé à la même position alphabétique. Ainsi, avec notre exemple de clé, les A deviennent des C, les B des L, …, et les Z des S. Ainsi, avec la chaîne « `ARCHI TD` », on doit obtenir au final la chaîne « `CDUKO AM` » (avec cet exemple de clé).
   Écrire une procédure assembleur qui, lorsqu'elle se termine, assure que la chaîne initialement pointée par `r0` (dont on n'a pas besoin de conserver la valeur initiale) contient la chaîne d'origine chiffrée à l'aide de la clé pointée par `r1`.

Corrigé :

1.

Listing 2. Chiffrement par décalage

```
loop:   LDB     r2,(r0)         ; récupérer le caractère
        JZ      r2,fin          ; fini ?
        SUB     r31,r2,#' '     ;
        JZ      r31,espace      ; est-ce une espace ?
        ADD     r2,r2,r1        ; on décale
        SUB     r31,r2,#'Z'     ;
        JLE     r31,avantZ      ; r2 <= 'Z' ?
        SUB     r2,r2,#26       ; sinon on revient vers A
avantZ: STB     (r0),r2         ; on stocke le caractère décalé
espace: ADD     r0,r0,#1        ; on passe au caractère suivant
        JMP     loop            ; et on recommence
fin:
```

2.

Listing 3. Chiffrement par clé

```
loop:   LDB     r2,(r0)         ; récupérer le caractère
        JZ      r2,fin          ; fini ?
        SUB     r31,r2,#' '     ;
        JZ      r31,espace      ; est-ce une espace ?
        SUB     r2,r2,#'A'      ; calcul de l'indice
        ADD     r3,r1,r2        ;
        LDB     r2,(r3)         ; chargement du caractère
        STB     (r0),r2         ; on stocke le nouveau caractère
espace: ADD     r0,r0,#1        ; on passe au caractère suivant
        JMP     loop            ; et on recommence
fin:
```

IV. Mémoire cache

Une machine possède un cache de 16 octets. Au cours d'un programme, elle accède successivement les octets situés aux adresses mémoire : 1, 4, 8, 5, 20, 17, 19, 56, 4, 9, 11, 43, 5, 6, 9, 17.

1. On suppose le cache initialement vide et organisé en 16 blocs de 1 octet à accès direct (dans le bloc 1 du cache vont les octets mémoire 1, 17, 33...). 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 4 blocs de 4 octets à accès direct (dans le bloc 1 du cache vont les blocs mémoire 1-4, 17-20, 33-36...). 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 4 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 : 1, 4, 8, 5, 20 (remp. 4), 17 (remp. 1), 19, 56, 4, 9, 11, 43, 6
   C : 5, 9, 17
   Soit 13 accès mémoire et 3 accès cache.
2.

1	M	bloc 1 (1-4)	20	M	bloc 1 (17-20)	4	M	bloc 1 (1-4)	5	M	bloc 2 (5-8)
4	C	bloc 1	17	C	bloc 1	9	M	bloc 3 (9-12)	6	C	bloc 2
8	M	bloc 2 (5-8)	19	C	bloc 1	11	C	bloc 3	9	M	bloc 3 (9-12)
5	C	bloc 2	56	M	bloc 2 (53-56)	43	M	bloc 3 (41-44)	17	M	bloc 1 (17-20)

   Soit 10 accès mémoire et 6 accès cache.
3.

1	M	bloc 1 (1-4)	20	M	bloc 3 (17-20)	4	C	bloc 1	5	M	bloc 4 (5-8)
4	C	bloc 1	17	C	bloc 3	9	M	bloc 2 (9-12)	6	C	bloc 4
8	M	bloc 2 (5-8)	19	C	bloc 3	11	C	bloc 2	9	C	bloc 2
5	C	bloc 2	56	M	bloc 4 (53-56)	43	M	bloc 3 (41-44)	17	M	bloc 1 (17-20)

   Soit 8 accès mémoire et 8 accès cache.