UNIVERSITÉ DE PARIS-DAUPHINE L2 MI2E

Architecture des ordinateurs (E. Lazard)
Examen du 17 janvier 2020

(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 $2^{-5} = 0{,}03125$).

1. Représenter 0,3125, 2 et 4 en virgule flottante.
2. Quel est le nombre représentable le plus proche de $2/3$?
3. On écrit le code suivant :

Listing 1. Calculs (code C)

```c
float f = 2.0 / 3.0;
for (i = 0; i < 50; i++)
    f = 4.0 * f - 2.0;
```

Listing 2. Calculs (code Python)

```python
f = 2.0 / 3.0
for i in range(50):
    f = 4.0 * f - 2.0
```

   - Si les calculs étaient parfaits, que devrait-on obtenir comme valeur de `f` après la boucle ?
   - En prenant comme approximation représentable de $2/3$ la valeur calculée à la question 2, quelle est en fait la valeur de `f` après le premier passage dans la boucle?
   - D'après vous, que va-t-il se passer ensuite ? Vers quoi va tendre la valeur de `f` ?

Corrigé :

1. $0{,}3125 = 1{,}25 \times 2^{-2} = \boxed{0\,0101\,01000}$
   $2 = 1 \times 2^{1} = \boxed{0\,1000\,00000}$
   $4 = 1 \times 2^{2} = \boxed{0\,1001\,00000}$
2. $2/3 = 0{,}666666... = 1{,}33333... \times 2^{-1}$ est encadré par $1{,}3125 \times 2^{-1}$ et $1{,}34375 \times 2^{-1}$. C'est ce dernier nombre qui est le plus proche. Donc $2/3 \approx 1{,}34375 \times 2^{-1} = \boxed{0\,0110\,01011}$.
3. - Si les calculs étaient parfaits, `f` garderait indéfiniment la valeur $2/3$ qui est le point fixe de la fonction $4f - 2$.
   - `f` $= 1{,}01011_2 \times 2^{-1}$ donc `4f` $= 1{,}01011_2 \times 2^{1}$ et `4f` $- 2 = 0{,}01011_2 \times 2^{1} = 1{,}01100_2 \times 2^{-1} = 0{,}6875$.
   - À cause de l'arrondi supérieur, `f` est légèrement au-dessus de $2/3$ et cet écart ne fait que s'accroître à chaque itération car il est multiplié par 4. La valeur de `f` va donc tendre vers plus l'infini...

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 deux sorties s_0 et s_1 tel que :

- si $F = 0$, le circuit se comporte comme un demi-additionneur, la sortie s_0 représentant la somme 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 OU des deux entrées et la sortie s_1 le NON-ET 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.
3. En remarquant que le OU peut se décomposer en XOR et ET, simplifier les sorties et représenter le circuit à l'aide de 5 portes logiques.

CORRIGÉ :

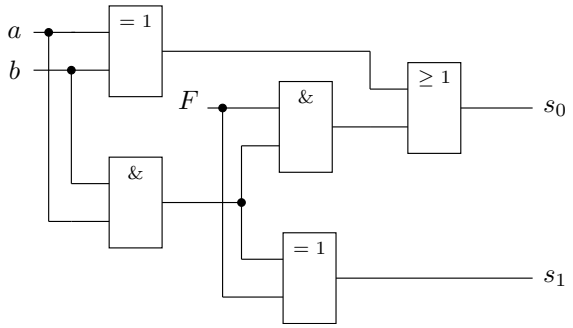
1. Pour $F = 0$

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

pour $F = 1$

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

2. $s_0 = (a \oplus b)\overline{F} + (a + b)F$
   $s_1 = ab\overline{F} + \overline{ab}F = ab \oplus F$
3. On remarque que $a + b = (a \oplus b) + ab$.
   Cela nous donne $s_0 = (a \oplus b) + abF$ et donc le circuit suivant :



III. Circuits logiques

On souhaite construire un 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 non signé, qui génère n bits de sortie $S = s_{n-1} \dots s_0$ représentant la valeur binaire $A - 1$, ainsi qu'un bit de débordement X valant 1 si la valeur $A - 1$ ne peut pas être représentée sur n bits.

1. On appelle r_i la retenue intermédiaire utilisée pour calculer $s_{i+1} = a_{i+1} - r_i$. Donner la table de vérité de s_0 et r_0 en fonction de a_0. 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 a_i et r_{i-1}. Donner les expressions logiques de s_i et r_i.
3. À quel bit est égal le bit de débordement X ?

4. Dessiner le circuit donnant $S = s_{n-1} \dots s_0$ et X, en fonction des $a_{n-1} \dots a_0$ (vous pouvez utiliser toutes les portes à deux entrées).
5. 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).

Corrigé :

1.

a_0	s_0	r_0
0	1	1
1	0	0

$s_0 = \overline{a_0}$

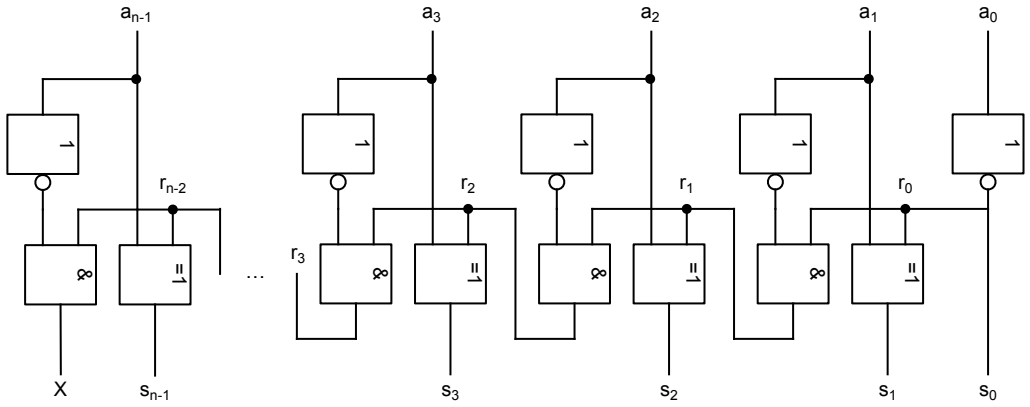
$r_0 = \overline{a_0}$

2.

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

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

$r_i = \overline{a_i}.r_{i-1}$

3. X est en fait équivalent à la retenue finale r_{n-1}.
4. 

5. Par récurrence, on peut écrire

$$r_i = \overline{a_i}.\overline{a_{i-1}} \dots \overline{a_1}.\overline{a_0} = \prod_{j=0}^{i} \overline{a_j}$$

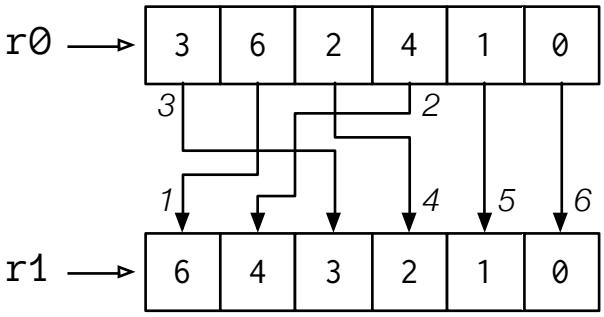
$$s_i = a_i \oplus (\overline{a_{i-1}}.\overline{a_{i-2}} \dots \overline{a_1}.\overline{a_0}) = a_i \oplus \prod_{j=0}^{i-1} \overline{a_j}$$

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 3. Tri par recopie

```
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. Assembleur

1. Un nombre entier compris entre 0 et 15 est stocké en mémoire sur un octet, son adresse étant dans le registre `r0`. On souhaite générer le caractère hexadécimal (de `0` à `9` ou de `a` à `f`) équivalent et le mettre dans `r2`. Écrire la procédure assembleur correspondante. *(La valeur constante `#'0'` représente le caractère `0`)*
2. Un nombre entier compris entre 0 et 32767 est stocké en mémoire sur deux octets, son adresse étant dans le registre `r0`. On souhaite générer une chaîne de quatre caractères hexadécimaux, se terminant par l'octet nul, représentant l'écriture hexadécimal de ce nombre. Cette chaîne sera donc composée de caractères hexadécimaux (chacun sur un octet) et sera stockée à partir de l'adresse contenue dans le registre `r1`. Pour simplifier, on écrira la chaîne « à l'envers » : le caractère représentant les 4 bits de poids faible sera mis à l'adresse pointée par `r1`, le caractère représentant les 4 bits suivant à l'adresse `r1+1`,... , le 4^{e} caractère à l'adresse `r1+3`. Écrire la procédure assembleur correspondante. *(On pourra récupérer les quatre bits de poids faible d'un registre en effectuant un ET logique entre ce registre et la valeur 15 ; on décalera pour les 4 bits suivants.)*

Corrigé :

1.

Listing 4. Hexadécimal

```
        LDB     r2,(r0)         ; récupérer le nombre
        SUB     r31,r2,#10
        JGE     r31,lettre      ; est-ce une lettre ?
        ADD     r2,r2,#'0'      ; sinon on construit le car. chiffre
        JMP     fin             ; et c'est tout...
lettre: ADD     r2,r2,#'a'-10   ; sinon c'est le car. lettre
fin:
```

ou avec une instruction de moins :

Listing 5. Hexadécimal v2

```
        LDB     r2,(r0)                 ; récupérer le nombre
        ADD     r2,r2,#'0'              ; on construit le car. chiffre
        SUB     r31,r2,#'9'
        JLE     r31,fin                 ; était-ce une lettre ?
        ADD     r2,r2,#'a'-'0'-10       ; on construit le car. lettre
fin:
```

2.

Listing 6. Décimal vers hexa

```
        MVI     r31,#4                  ; 4 symboles à traiter
        LDH     r2,(r0)                 ; récupérer le nombre
loop:   AND     r3,r2,#15               ; extraction des 4 bits poids faible
        LLS     r2,r2,#-4               ; décaler pour diviser par 16
        ADD     r3,r3,#'0'              ; on construit le car. chiffre
        SUB     r31,r3,#'9'
        JLE     r31,suite               ; était-ce une lettre ?
        ADD     r3,r3,#'a'-'0'-10       ; on construit le car. lettre
suite:  STB     (r1),r3                 ; stocker le caractère
        ADD     r1,r1,#1                ; avancer le pointeur
        SUB     r31,r31,#1              ; décrémenter compteur
        JNZ     r31,loop                ; et revenir si encore des bits
        STB     (r1),r31                ; le zéro final à mettre
```