UNIVERSITÉ DE PARIS-DAUPHINE – MIDO L2 MI2E

Architecture des ordinateurs (E. Lazard)
Examen du 12 janvier 2021

(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 inférieurement.**

1. Représenter 5 ; 24 et $5 \times 5 = 25$ en virgule flottante.
2. On écrit le code suivant :

Listing 1. Calculs (code C)

```c
float f = 5.0;
for (i = 0; i < 10; i++)
        f = f * (f*f - 24.0);
```

Listing 2. Calculs (code Python)

```python
f = 5.0
for i in range(10):
    f = f * (f*f - 24.0)
```

   (a) Si les calculs étaient parfaits, que devrait-on obtenir comme valeur de `f` après la boucle ?

   (b) En détaillant les calculs vraiment effectués en virgule flottante, indiquer ce qu'on obtient comme valeur de `f` après une itération de la boucle. Quelle sera la valeur après toute la boucle ?

   (c) Le compilateur « optimise » le code et remplace la ligne de la boucle par la ligne :

```
f = f*f*f - f*24.0;
```

   Quel est maintenant le résultat effectif après la première itération de la boucle ?
   Vers quoi pensez-vous que le calcul va converger après la boucle ?

Corrigé :

1. $5 = 1{,}25 \times 2^2 = \boxed{0\,1001\,01000}$
   $24 = 1{,}5 \times 2^4 = \boxed{0\,1011\,10000}$
   $5 \times 5 = 25 = 1{,}5625 \times 2^4 = \boxed{0\,1011\,10010}$
2. (a) Si les calculs étaient parfaits, `f` garderait indéfiniment la valeur 5.0 qui est un des points fixes de la fonction $f(f^2 - 24)$.

   (b) $\texttt{f} = 1{,}0100_2 \times 2^2$ et $\texttt{f} \times \texttt{f} = 1{,}1001_2 \times 2^4$ donc $\texttt{f} \times \texttt{f} - 24.0 = 1{,}1001_2 \times 2^4 - 1{,}1 \times 2^4 = 0{,}0001_2 \times 2^4 = 1 \times 2^0$.
   Pour finie, $\texttt{f} \times (1 \times 2^0) = \texttt{f}$ bien sûr, donc la valeur de `f` ne change pas après une itération, ni après toute la boucle.

(c) On calcule maintenant $\texttt{f} \times \texttt{f} \times \texttt{f} = 1{,}11110\not{1}_2 \times 2^6$, arrondi à $1{,}11110_2 \times 2^6 = 124$. Par ailleurs, $\texttt{f} \times 24.0 = (1{,}0100_2 \times 2^2) \times (1{,}1_2 \times 2^4) = 1{,}111_2 \times 2^6$.
Donc $\texttt{f} \times \texttt{f} \times \texttt{f} - \texttt{f} \times 24.0 = 1{,}11110_2 \times 2^6 - 1{,}111_2 \times 2^6 = 0{,}0001_2 \times 2^6 = 1 \times 2^2 = 4$.
À cause de l'arrondi, la valeur de `f` diminue légèrement après la première itération.
À la deuxième itération, on a $\texttt{f} \times \texttt{f} \times \texttt{f} - \texttt{f} \times 24.0 = 1 \times 2^6 - (1{,}1_2 \times 2^4) \times (1 \times 2^2) = -0{,}1_2 \times 2^6 = -1 \times 2^5 = -32$. Aux itérations suivantes, le calcul va rapidement diverger vers $-\infty$ (ou ce qui en tient lieu dans notre représentation).

II. Circuits logiques

On souhaite construire un incrémenteur/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 **signé** et une ligne de commande F, qui génère n bits de sortie $S = s_{n-1} \dots s_0$ représentant la valeur binaire $A + 1$ (si $F = 0$) ou $A - 1$ (si $F = 1$), ainsi qu'un bit de débordement X valant 1 si la valeur souhaitée ($A + 1$ ou $A - 1$ suivant le cas) ne peut pas être représentée sur n bits.

1. On appelle r_i la retenue intermédiaire utilisée pour calculer $s_{i+1} = a_{i+1} + r_i$ ou $s_{i+1} = a_{i+1} - r_i$. Donner la table de vérité de s_0 et r_0 en fonction de a_0 et F. 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 F, a_i et r_{i-1}. Donner les expressions logiques de s_i et r_i.
3. 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).
4. Quels sont les deux cas de débordement ? Y a-t-il débordement lorsqu'il y a une retenue finale r_{n-1} ? Lorsqu'il y a une retenue r_{n-2}, y a-t-il des cas de débordement ? Proposer une expression logique pour le bit de débordement faisant intervenir ces deux retenues.

Corrigé :

1.

F	a_0	s_0	r_0
0	0	1	0
0	1	0	1
1	0	1	1
1	1	0	0

$s_0 = \overline{a_0}$

$r_0 = F \oplus a_0$

2.

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

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

$r_i = \overline{F}.a_i.r_{i-1} + F.\overline{a_i}.r_{i-1} = r_{i-1}(F \oplus a_i)$

3. Par récurrence, on peut écrire

$$r_i = \prod_{j=0}^{i} (F \oplus a_j)$$

$$s_i = a_i \oplus \prod_{j=0}^{i-1} (F \oplus a_j)$$

4. Les nombres étant signés, il y a débordement dans deux cas :
   - lors d'une addition ($F = 0$), lorsque l'on passe de $011\ldots11$ à $100\ldots00$;
   - lors d'une soustraction ($F = 1$), lorsque l'on passe de $100\ldots00$ à $011\ldots11$.

   Il y a une retenue finale r_{n-1} lorsque :
   - lors d'une addition ($F = 0$), on passe de $111\ldots11$ à $000\ldots00$;
   - lors d'une soustraction ($F = 1$), on passe de $000\ldots00$ à $111\ldots11$.

   Dans ces deux cas il n'y a pas de débordement puisque les nombres sont signés.
   Il y a une retenue r_{n-2} lorsque :
   - lors d'une addition ($F = 0$), on passe de $111\ldots11$ à $000\ldots00$ (sans débordement) ;
   - lors d'une addition ($F = 0$), on passe de $011\ldots11$ à $100\ldots00$ (avec débordement) ;
   - lors d'une soustraction ($F = 1$), on passe de $000\ldots00$ à $111\ldots11$ (sans débordement) ;
   - lors d'une soustraction ($F = 1$), on passe de $100\ldots00$ à $011\ldots11$ (avec débordement).

   On a donc un débordement lorsqu'il y a une retenue r_{n-2} mais pas de retenue finale :
   $X = r_{n-2}.\overline{r_{n-1}}$

III. Assembleur

1. 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`. Un autre tableau de caractères est également stocké en mémoire, chaque caractère sur un octet, chacun à la suite des autres et l'octet qui suit le dernier caractère est nul. L'adresse du premier caractère du tableau se trouve dans le registre `r1`.

   On souhaite compter combien de fois on retrouve dans la première chaîne un des caractères du tableau et mettre ce résultat dans `r2`.

   Par exemple, si la chaîne est `"abcbaccdfdeG"` et que le tableau de caractères contient les quatre caractères `"acfg"`, on veut obtenir comme résultat la valeur 6.

   Écrire une procédure assembleur qui, lorsqu'elle se termine, assure que `r2` contient le nombre de fois qu'on retrouve un caractère du tableau dans la chaîne. On ne vous demande pas de conserver les valeurs des registres à la fin de la procédure.
2. Le tableau contient maintenant des paires de caractères et chaque fois qu'on trouve dans la chaîne un exemplaire du premier caractère de la paire, on veut le remplacer (dans la chaîne originale) par le deuxième caractère de la paire.

   Par exemple, si la chaîne est `"abcbaccdfdeG"` et le tableau de caractères contient `"aAcCfFgG"`, on veut obtenir comme résultat la chaîne `"AbCbACCdFdeG"` et toujours la valeur 6 comme nombre de caractères trouvés/remplacés. Modifier votre procédure assembleur pour que `r2` contienne toujours le nombre de fois qu'on retrouve un caractère du tableau dans la chaîne et que les remplacements aient été effectués.. On ne vous demande pas de conserver les valeurs des registres à la fin de la procédure.

Corrigé :

1.

Listing 3. Comptage

```
        MVI     r2,#0           ; init. compteur
loop:   LDB     r3,(r0)         ; caractère suivant
        JZ      r3,fin          ; fin de la chaîne ?
        MOV     r4,r1           ; début du tableau
loop_i: LDB     r5,(r4)         ; car. suivant du tableau
        JZ      r5,suite        ; fin tableau
        SUB     r31,r5,r3       ; comparer les deux caractères
        JNZ     r31,next        ; différent ?
        ADD     r2,r2,#1        ; si non, incrémenter compteur
next:   ADD     r4,r4,#1        ; avancer pointeur tableau
        JMP     loop_i          ; revenir boucle interne
suite:  ADD     r0,r0,#1        ; avancer pointeur chaîne
        JMP     loop            ; revenir boucle chaîne
fin:
```

2.

Listing 4. Remplacements

```
        MVI     r2,#0           ; init. compteur
loop:   LDB     r3,(r0)         ; caractère suivant
        JZ      r3, fin         ; fin de la chaîne ?
        MOV     r4,r1           ; début du tableau
loop_i: LDB     r5,(r4)         ; car. suivant du tableau
        ADD     r4,r4,#1        ; evnt. nv caractère
        JZ      r5,suite        ; fin tableau
        SUB     r31,r5,r3       ; comparer les deux caractères
        JNZ     r31,next        ; différent ?
        ADD     r2,r2,#1        ; si non, incrémenter compteur
        LDB     r6,(r4)         ; charger nv caractère
        STB     (r0),r6         ; remplacer dans chaîne
next:   ADD     r4,r4,#1        ; avancer pointeur tableau
        JMP     loop_i          ; revenir boucle interne
suite:  ADD     r0,r0,#1        ; avancer pointeur chaîne
        JMP     loop            ; revenir boucle chaîne
fin:
```

IV. Mémoire cache

Notre machine possède un cache d'une taille de 12 instructions regroupées en 4 blocs de 3 instructions. Le cache est à correspondance directe. On rappelle que cela veut dire que les mots mémoire 1-3, 13-15... vont dans le premier bloc du cache, que les mots 4-6, 16-18... vont dans le deuxième, etc. Le temps de cycle de la mémoire principale est M et le temps de cycle du cache est C.

1. Un programme se compose d'une boucle de 33 instructions à exécuter 3 fois ; les instructions se trouvant, dans l'ordre, aux adresses mémoire 1 à 15, 16 à 18, 1 à 3, 16 à 18, 1 à 3, 16 à 18, 1 à 3. Quel est le temps total d'exécution du programme en ne tenant pas compte des temps de calcul ? Le cache est vide au départ.
2. Le compilateur a réussi à optimiser votre programme et en a réduit la taille en gagnant quelques instructions. Il se compose maintenant d'une boucle de 30 instructions à exécuter 3 fois ; les instructions se trouvant, dans l'ordre, aux adresses mémoire 1 à 12, 13 à 15, 1 à 3, 13 à 15, 1 à 3, 13 à 15, 1 à 3. Quel est le temps total d'exécution du programme en ne tenant pas compte des temps de calcul ? Le cache est vide au départ.
3. Que concluez-vous des deux précédents résultats ?

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.

$1 \to 3$	$M + 2C$	bloc 1	$1 \to 3$	3C	bloc 1	$1 \to 3$	3C	bloc 1
$4 \to 6$	$M + 2C$	bloc 2	$4 \to 6$	$M + 2C$	bloc 2	$4 \to 6$	$M + 2C$	bloc 2
$7 \to 9$	$M + 2C$	bloc 3	$7 \to 9$	3C	bloc 3	$7 \to 9$	3C	bloc 3
$10 \to 12$	$M + 2C$	bloc 4	$10 \to 12$	3C	bloc 4	$10 \to 12$	3C	bloc 4
$13 \to 15$	$M + 2C$	bloc 1	$13 \to 15$	$M + 2C$	bloc 1	$13 \to 15$	$M + 2C$	bloc 1
$16 \to 18$	$M + 2C$	bloc 2	$16 \to 18$	$M + 2C$	bloc 2	$16 \to 18$	$M + 2C$	bloc 2
$1 \to 3$	$M + 2C$	bloc 1	$1 \to 3$	$M + 2C$	bloc 1	$1 \to 3$	$M + 2C$	bloc 1
$16 \to 18$	3C	bloc 2	$16 \to 18$	3C	bloc 2	$16 \to 18$	3C	bloc 2
$1 \to 3$	3C	bloc 1	$1 \to 3$	3C	bloc 1	$1 \to 3$	3C	bloc 1
$16 \to 18$	3C	bloc 2	$16 \to 18$	3C	bloc 2	$16 \to 18$	3C	bloc 2
$1 \to 3$	3C	bloc 1	$1 \to 3$	3C	bloc 1	$1 \to 3$	3C	bloc 1

Soit un total de $15M + 84C$.

2.

$1 \to 3$	$M + 2C$	bloc 1	$1 \to 3$	3C	bloc 1	$1 \to 3$	3C	bloc 1
$4 \to 6$	$M + 2C$	bloc 2	$4 \to 6$	3C	bloc 2	$4 \to 6$	3C	bloc 2
$7 \to 9$	$M + 2C$	bloc 3	$7 \to 9$	3C	bloc 3	$7 \to 9$	3C	bloc 3
$10 \to 12$	$M + 2C$	bloc 4	$10 \to 12$	3C	bloc 4	$10 \to 12$	3C	bloc 4
$13 \to 15$	$M + 2C$	bloc 1	$13 \to 15$	$M + 2C$	bloc 1	$13 \to 15$	$M + 2C$	bloc 1
$1 \to 3$	$M + 2C$	bloc 1	$1 \to 3$	$M + 2C$	bloc 1	$1 \to 3$	$M + 2C$	bloc 1
$13 \to 15$	$M + 2C$	bloc 1	$13 \to 15$	$M + 2C$	bloc 1	$13 \to 15$	$M + 2C$	bloc 1
$1 \to 3$	$M + 2C$	bloc 1	$1 \to 3$	$M + 2C$	bloc 1	$1 \to 3$	$M + 2C$	bloc 1
$13 \to 15$	$M + 2C$	bloc 1	$13 \to 15$	$M + 2C$	bloc 1	$13 \to 15$	$M + 2C$	bloc 1
$1 \to 3$	$M + 2C$	bloc 1	$1 \to 3$	$M + 2C$	bloc 1	$1 \to 3$	$M + 2C$	bloc 1

Soit un total de $22M + 68C$.

3. On voit que « l'amélioration » effectuée par le compilateur a augmenté le nombre d'accès mémoire par une mauvaise répartition des instructions dans les blocs, même si on a gagné en nombre d'instructions. On a $7M$ en plus et $16C$ en moins, ce qui fait que si $7M > 16C$, le programme « optimisé » s'exécute plus lentement !