Architecture des ordinateurs (E. Lazard)
Examen du 11 janvier 2023

(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 8, 9 et 72 en virgule flottante.
2. *Rappel : pour additionner deux nombres en virgule flottante, il faut décaler la mantisse d'un des deux nombres pour égaliser les exposants, additionner les mantisses (sans oublier le* 1 *débutant la mantisse), puis renormaliser le nombre en arrondissant éventuellement la pseudo-mantisse.*
   Ainsi, on a $7 = 1{,}11000_2 \times 2^2$ *et* $1{,}25 = 1{,}01000_2 \times 2^0 = 0{,}0101000_2 \times 2^2$ *dont l'addition donne* $10{,}0001_2 \times 2^2$ *qui se normalise en* $1{,}00001_2 \times 2^3 = 8{,}25$.
   On écrit le code suivant :

Listing 1. Calculs

```
float f = 9.0 ;
for (i = 0; i < 2; i++)
        f = 9.0 * (f - 8.0);
```

   – Qu'obtient-on comme valeur de `f` après la boucle ?
   – Le compilateur optimise le code et remplace la ligne de la boucle par `f = 9.0*f-72.0;` Quel est maintenant le résultat après la boucle ?
   – On change les règles de l'arrondi pour que, après calculs, celui-ci s'effectue supérieurement. D'après vous, vers quoi va tendre la valeur de `f` si on augmente la taille de la boucle en gardant la ligne optimisée par le compilateur ?

Corrigé :

1. $8 = 1 \times 2^3 = \boxed{0\ 1010\ 00000}$
   $9 = 1{,}125 \times 2^3 = \boxed{0\ 1010\ 00100}$
   $72 = 1{,}125 \times 2^6 = \boxed{0\ 1101\ 00100}$
2. – L'opération `f-8.0` donne $0{,}00100 \times 2^3$ soit $1{,}00000 \times 2^0$ (qui vaut exactement 1.0). En multipliant par 9.0, on obtient donc exactement 9.0 et la valeur de `f` ne change pas.

– Il faut maintenant calculer `9.0*f` soit $1{,}001 \times 1{,}001 \times 2^6$ c'est-à-dire $1{,}010001 \times 2^6$ qui s'arrondit inférieurement à $1{,}01000 \times 2^6 = 80$. On obtient donc la valeur $80 - 72 = 8$ après un passage de boucle puis $9 * 8 - 72 = 0$ après les deux passages.
– Si on arrondit supérieurement, on obtient 82 après la première multiplication `9.0*f` et donc 10 après le premier passage. La valeur de `f` va ensuite tendre vers l'infini et au-delà...

II. Circuits logiques

On souhaite construire un incrémenteur 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	0
1	0	1

$s_0 = \overline{a_0}$

$r_0 = a_0$

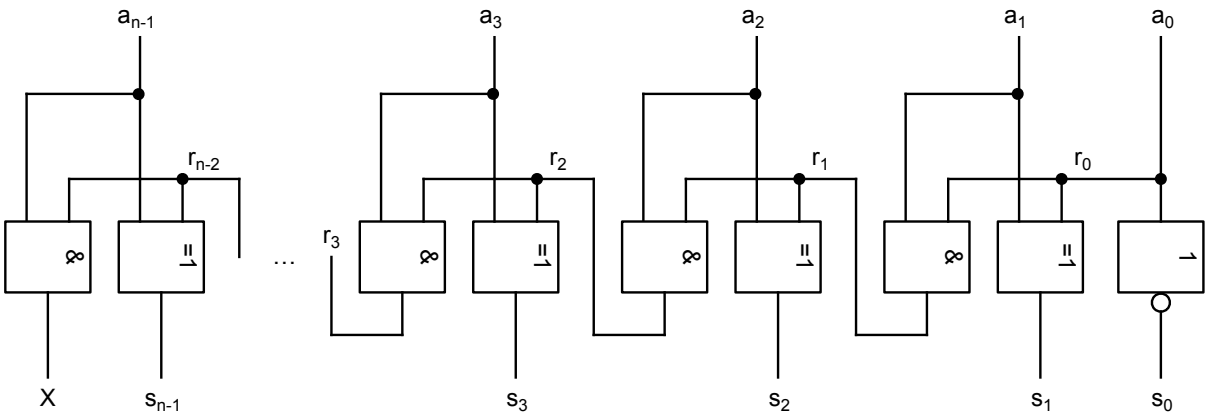
2.

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

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

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

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



5. Par récurrence, on peut écrire

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

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

III. Assembleur

On cherche à savoir si un entier positif stocké dans le registre `r0` est premier. Pour ce faire, on divisera ce nombre par tous les nombres inférieurs (sans s'arrêter à la racine carrée qu'on ne sait pas calculer) en calculant le reste ($a = qb + r$). À la fin de la procédure, le registre `r1` devra contenir 0 (le nombre n'est pas premier) ou 1 (le nombre initial est premier).

Écrire une procédure assembleur qui, lorsqu'elle se termine, a mis 1 ou 0 dans le registre `r1` suivant que le nombre initial dans `r0` est premier ou non.

Corrigé :

Listing 2. test nombre premier ?

```
        MVI r1, #0              ; initialisation
        MVI r2, #2              ; on commence par diviser par 2
        SUB r31, r0, r2         ;
        JLT r31, fin            ; (Il y a le cas r0=0 ou 1)
loop:   SUB r31, r0, r2         ; test d'arrêt ; arrivé jusqu'à r0 ?
        JZ r31, prem
        DIV r31, r0, r2         ; calcul du quotient
        MUL r31, r31, r2        ; puis du reste
        SUB r31, r0, r31
        JZ r31, fin             ; reste nul ?
        ADD r2, r2, #1          ; sinon on passe au "diviseur" suivant
        JMP loop
prem:   MVI r1, #1
fin:
```

IV. Assembleur

Un tableau de 20 entiers est stocké en mémoire. Chaque élément a une valeur de 0 à 9 et est stocké sur un octet. L'adresse du premier élément du tableau se trouve dans le registre `r0`.

On cherche à construire un histogramme de ces valeurs, c'est-à-dire à avoir un tableau de 10 valeurs, comptabilisant combien de fois chacune apparait dans le tableau initial. Par exemple, si le tableau initial est :

```
9 3 1 4 1 7 7 2 2 1 1 7 2 9 6 0 3 9 7 8
```

on souhaite obtenir l'histogramme de valeurs :

```
1 4 3 2 1 0 1 4 1 3
```

car 0 apparait une fois, 1 apparait quatre fois, etc.

Cet histogramme sera stocké en mémoire en commençant à l'adresse se trouvant dans `r1`, un octet par valeur. On supposera comme d'habitude que l'espace nécessaire est disponible.

Écrire une procédure assembleur qui, lorsqu'elle se termine, a rempli l'histogramme pointé par le registre `r1` avec le nombre d'occurrences de chaque valeur du tableau pointé par `r0`.

Corrigé : Première solution, on compte combien de fois apparait chaque valeur.

Listing 3. histogramme

```
        MOV r30, r0             ; sauvegarde début tableau
        MVI r10, #0             ; 10 valeurs, on commence à 0
loop1:  MOV r0, r30             ; on repart du début
        MVI r2, #0              ; compteur à 0
        MVI r20, #20            ; 20 valeurs à tester
loop2:  LDB r31, (r0)           ; chargement valeur
        ADD r0, r0, #1
        SUB r31, r31, r10       ; égale à celle recherchée ?
        JNZ r31, next
        ADD r2, r2, #1          ; on augmente le compteur
next:   SUB r20, r20, #1        ; on continue le parcours
        JNZ r20, loop2
        STB (r1), r2            ; on stocke le compteur
        ADD r1, r1, #1
        ADD r10, r10, #1
        SUB r31, r10, #10
        JNZ r31, loop1          ; valeur suivante
```

Autre solution : après avoir initialisé l'histogramme avec des 0, on incrémente la case correspondant à chaque valeur du tableau.

Listing 4. histogramme v2

```
        MOV r31, r1
        MVI r2, #0
        MVI r10, #10            ; initialisation de l'histo à 0
loop10: STB (r1), r2
        ADD r1, r1, #1
        SUB r10, r10, #1
        JNZ r10, loop10
        MOV r1, r31             ; retour au début
        MVI r20, #20            ; 20 valeurs à tester
loop20: LDB r2, (r0)
        ADD r31, r1, r2         ; on va sur la bonne case
        LDB r30, (r31)
        ADD r30, r30, #1        ; on incrémente le compteur
        STB (r31), r30
        ADD r0, r0, #1
        SUB r20, r20, #1
        JNZ r20, loop20
```

V. Mémoire cache

Un programme se compose d'une boucle de 24 instructions à exécuter 5 fois ; les instructions se trouvant, dans l'ordre, aux adresses mémoire 13 à 18, 1 à 6, 13 à 18, 7 à 12. Ce programme doit tourner sur une machine possédant un cache d'une taille de 12 instructions. Le temps de cycle de la mémoire principale est M et le temps de cycle du cache est C. Le cache est 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).

1. Le cache possède 6 blocs de 2 instructions : les blocs que l'on peut transférer sont 1-2, 3-4, ..., 13-14, 15-16, 17-18... 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 cache possède 3 blocs de 4 instructions : les blocs que l'on peut transférer sont 1-4, 5-8, ..., 13-16, 17-20... 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. Le cache possède 2 blocs de 6 instructions : les blocs que l'on peut transférer sont 1-6, 7-12, 13-18... Quel est le temps total d'exécution du programme en ne tenant pas compte des temps de calcul ? Le cache est vide au départ.
4. Le cache possède 1 bloc de 12 instructions : les blocs que l'on peut transférer sont 1-12, 13-24... Quel est le temps total d'exécution du programme en ne tenant pas compte des temps de calcul ? Le cache est vide au départ.
5. Qu'en concluez-vous sur l'efficacité du cache ?

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.

$13 \to 14$	$M + C$	bloc 1	$13 \to 14$	$2C$	bloc 1	
$15 \to 16$	$M + C$	bloc 2	$15 \to 16$	$2C$	bloc 2	
$17 \to 18$	$M + C$	bloc 3	$17 \to 18$	$2C$	bloc 3	
$1 \to 2$	$M + C$	bloc 4	$1 \to 2$	$M + C$	bloc 4	
$3 \to 4$	$M + C$	bloc 5	$3 \to 4$	$M + C$	bloc 5	
$5 \to 6$	$M + C$	bloc 6	$5 \to 6$	$M + C$	bloc 6	Les trois dernières
$13 \to 14$	$2C$	bloc 1	$13 \to 14$	$2C$	bloc 1	itérations
$15 \to 16$	$2C$	bloc 2	$15 \to 16$	$2C$	bloc 2	sont identiques
$17 \to 18$	$2C$	bloc 3	$17 \to 18$	$2C$	bloc 3	
$7 \to 8$	$M + C$	bloc 4	$7 \to 8$	$M + C$	bloc 4	
$9 \to 10$	$M + C$	bloc 5	$9 \to 10$	$M + C$	bloc 5	
$11 \to 12$	$M + C$	bloc 6	$11 \to 12$	$M + C$	bloc 6	

Soit un total de $33M + 87C$.

2.

$13 \to 16$	$M + 3C$	bloc 1	$13 \to 16$	$M + 3C$	bloc 3	
$17 \to 18$	$M + C$	bloc 2	$17 \to 18$	$M + C$	bloc 1	
$1 \to 4$	$M + 3C$	bloc 3	$1 \to 4$	$M + 3C$	bloc 2	
$5 \to 6$	$M + C$	bloc 1	$5 \to 6$	$M + C$	bloc 3	Les trois dernières
$13 \to 16$	$M + 3C$	bloc 2	$13 \to 16$	$M + 3C$	bloc 1	itérations
$17 \to 18$	$M + C$	bloc 3	$17 \to 18$	$M + C$	bloc 2	sont identiques
$7 \to 8$	$2C$	bloc 1	$7 \to 8$	$2C$	bloc 3	
$9 \to 12$	$M + 3C$	bloc 2	$9 \to 12$	$M + C$	bloc 1	

Soit un total de $35M + 85C$.

3.

$13 \to 18$	$M + 5C$	bloc 1	$13 \to 18$	$6C$	bloc 1	
$1 \to 6$	$M + 5C$	bloc 2	$1 \to 6$	$M + 5C$	bloc 2	Les trois dernières
$13 \to 18$	$6C$	bloc 1	$13 \to 18$	$6C$	bloc 1	itérations
$7 \to 12$	$M + 5C$	bloc 2	$7 \to 12$	$M + 5C$	bloc 2	sont identiques

Soit un total de $11M + 109C$.

4.

$13 \to 18$	$M + 5C$	bloc 1	$13 \to 18$	$M + 5C$	bloc 1	
$1 \to 6$	$M + 5C$	bloc 1	$1 \to 6$	$M + 5C$	bloc 1	Les trois dernières
$13 \to 18$	$M + 5C$	bloc 1	$13 \to 18$	$M + 5C$	bloc 1	itérations
$7 \to 12$	$M + 5C$	bloc 1	$7 \to 12$	$M + 5C$	bloc 1	sont identiques

Soit un total de $20M + 100C$.

5. L'efficacité du cache n'est pas proportionnelle à la taille de ses blocs. De gros blocs permettent de mémoriser plus d'information à chaque accès mémoire mais de petits blocs permettent de garder des parties du programme qui sont souvent réutilisées.