Architecture des ordinateurs

Département Informatique

Erwan LEBAILLY — Vilavane LY — Vincent TRÉLAT — Benjamin ZHU

21 février 2022

***

Table des matières

1 Chapitre 1

1.1 Exercice 1

Avec la convention $0 \leftrightarrow$ `faux` et $1 \leftrightarrow$ `vrai`, $0 \wedge 1 =$ `faux`.

1.2 Exercice 2

On donne la table de c_0 :

c_0 :

$a_0 \backslash b_0$	0	1
0	0	1
1	1	0

On peut interpréter cette table comme la table de vérité du "ou exclusif", le *xor*. Ainsi, c_0 coincide avec $a_0 \oplus b_0 = (a_0 \vee b_0) \wedge (\neg(a_0 \wedge b_0))$.

1.3 Exercice 3

a. Montrer que xor est associatif et commutatif puis recopier calcul

b. inclure schéma

1.4 Exercice 4

a. On écrit le code suivant :

```
int main()
    {
        printf("Sizeof int: %lu octets\n", sizeof(int));
        printf("Sizeof short: %lu octets\n", sizeof(short));
        printf("Sizeof char: %lu octets\n", sizeof(char));
        return 0;
    }
```

La sortie est la suivante :

```
    Sizeof int: 4 octets
    Sizeof short: 2 octets
    Sizeof char: 1 octets
```

b. On écrit le code suivant :

```
int main()
    {
        int a = pow(2, 31);
        int b = pow(2, 31);
        int c = a + b;
        printf("%d\n", c);
        return 0;
    }
```

La sortie affiche 0, ce qui correspond bien à $2^{32} \mod (2^{32})$

1.5 Exercice 5

On donne ci-dessous l'écriture binaire sur 4 et 8 bits de 0, 1, -1 et -2 :

x	4 bits	8 bits
0 :	0000	0000 0000
1 :	0001	0000 0001
-1 :	1111	1111 1111
-2 :	1110	1111 1110

1.6 Exercice 6

a. $m_1 = 0001$ et $m_{-1} = 1001$.
b. En abusant de la notation + pour des mots : $m_0 = m_1 + m_{-1} = 1010$.
c. En suivant la règle de signes, 1010 est l'encodage de -2.

1.7 Exercice 7

Soit b un nombre de bits. Soit x un entier relatif qu'on souhaite représenter sur b bits.
Si $x \geq 0$, alors l'encodage de x correspond à une écriture dans $[0, 2^{b-1} - 1]$, alors cette écriture commence par un zéro (de 00...0 à 01...1). Si $x < 0$, alors $2^b - x \in [2^{b-1}, 2^b - 1]$ (soit de 10...0 à 11...1), son écriture commence par un 1.

1.8 Exercice 8

Dans le premier code, on dispose de 2 cases mémoires différentes. Le résultat affiché est -106 pour la valeur de d, ce qui est normal puisque d est signé.
Dans le deuxième code, on utilise une seule case mémoire à travers l'utilisation de deux pointeurs, un signé et un non signé. Le résultat affiché est

identique au premier code.
Cela permet de montrer que la mémoire est "non typée", l'interprétation de la valeur mémoire dépend directement du type de l'objet qui lit cette valeur.

1.9 Exercice 9

a. 10 s'écrit 2×5 et toute puissance de 2 s'écrit 2^k où $k \in \mathbb{N}$.
Ainsi, si $x \in 2^{\mathbb{N}}$ (par abus de langage) est divisible par 10, alors x contient au moins 2 et 5 dans sa décomposition en facteurs premiers, ce qui donne une contradition avec la propriété précédemment énoncée.

b. Supposons que 0.1 soit représentable sur kl bits. Alors, d'après le résultat du cours, $2^l \times 0.1$ est un entier, autrement dit 2^l est divisible par 10. D'après la question précédente, c'est impossible.

1.10 Exercice 10

L'écriture binaire approchée de 0.1 est $0.0001\ 1001_2$, de valeur décimale 0.09765625.

1.11 Exercice 11

a. On écrit la fonction suivante en C :

```c
int main() {
    for (int i = 0; i<10; i++){
        for (int j = 0; j < 10; j++){
            float a = i/10.0, b = j/10.0, c = (i+j)/10.0;
            printf("(%d, %d) : %s\n", i, j, (a+b == c)?"true":"false");
        }
    }
    return 0;
}
```

La sortie affichée contient, entre autres, les résultats suivants :

- (1, 4) : true
- (1, 5) : true
- (1, 6) : false
- (1, 7) : true
- (1, 8) : false
- (3, 4) : false
- (3, 5) : true
- (3, 6) : false
- (3, 7) : true
- (3, 8) : true

b. On modifie seulement la ligne d'affichage dans le code[1] :

```
printf("%.16f + %.16f = %.16f\n", a, b, c);
```

On donne seulement les résultats pour les 5 premiers couples ci-dessus :

```
0.1000000014901161 + 0.4000000059604645 = 0.5000000000000000
0.1000000014901161 + 0.5000000000000000 = 0.6000000238418579
0.1000000014901161 + 0.6000000238418579 = 0.6999999880790710
0.1000000014901161 + 0.6999999880790710 = 0.8000000119209290
0.1000000014901161 + 0.8000000119209290 = 0.8999999761581421
```

c. On remarque que pour une addition, l'égalité `a+b==c` est vérifiée lorsque `a` et `b` sont représentables, ou quand l'un des deux seulement l'est. Dans le second cas, l'erreur de représentation n'a pas eu d'impact sur le résultat puisque c'est la seule erreur du calcul. Ainsi l'égalité reste vraie.
En revanche, dès que les deux flottants ne sont pas représentables, les erreurs s'accumulent et alors la représentation de `c` peut différer de la valeur de `a+b`.

Dans un cas général si on prend $x, y \in \mathbb{R}$ tels que $x = y$, on aura `x==y` quand x et y sont représentables sur un nombre de bits donné. Dans les autres cas, il est possible d'obtenir un résultat correct mais cela résulte plutôt du hasard.

d. On modifie la fonction précédente :

```
int main() {
    for (int i = 0; i<10; i++){
        for (int j = 0; j < 10; j++){
            float a = i/10.0, b = j/10.0, c = (i+j)/10.0;
            if(a+b!=c){
                int m1 = a+b>c;
                int m2 = a+b+.000001>c+.000001;
                printf("(%d, %d) : %d %d\n", i, j, m1, m2);
            }
        }
    }
    return 0;
}
```

On obtient la sortie suivante :

```
// a+b > c is tested before and after adding 1e-6
(1, 6) : 1 1
(1, 8) : 1 1
(3, 4) : 1 1
(3, 6) : 1 1
```

1. Comme un `int` est codé sur 4 octets, on donne 16 caractères à chaque affichage.

```
(4, 3) : 1 1
(6, 1) : 1 1
(6, 3) : 1 1
(6, 8) : 1 1
(7, 9) : 0 0
(8, 1) : 1 1
(8, 6) : 1 1
(9, 7) : 0 0
```

On constate que l'ordre est conservé à chaque fois. Comme 10^{-6} n'est pas représentable sur 16 bits ($10^{-6} \approx 2^{-20}$), on simule l'effet de la propagation d'une erreur.
On en déduit que des erreurs successives d'arrondi ne bousculent pas l'ordre sur des valeurs arrondies. Toutefois ici on effectue le même calcul des deux côtés. On peut donc toujours les comparer[2], même après plusieurs calculs, puisque l'ordre est conservé. On peut également penser que cela ne provoquera pas d'évolution chaotique ou aléatoire de ces valeurs dans les calculs. En revanche, pour une suite d'opérations inconnue, cela risque de devenir insignifiant de vouloir comparer deux flottants.

1.12 Exercice 12

On note $s_1 = \sum_{i=1}^{k} \frac{1}{i}$ et $s_2 = \sum_{i=k}^{1} \frac{1}{i}$.

On écrit la fonction suivante :

```
int main() {
    float s1 = 0.0, s2 = 0.0;
    long k = 1000000000;
    for (double i = 1; i<k+1; i++){
        s1 += 1/i;
        s2 += 1/(k+1-i);
    }
    printf("k=%ld\n    s1 = %.16f\n    s2 = %.16f\n", k, s1, s2)
    ;
    return 0;
}
```

Les résultats sont les suivants :

```
k=1000
    s1 = 7.4854784011840820
    s2 = 7.4854717254638672
k=1000000
    s1 = 14.3573579788208008
```

2. Pas l'égalité.

```
    s2 = 14.3926515579223633
k=1000000000
    s1 = 15.4036827087402344
    s2 = 18.8079185485839844
```

On constate que les résultats diffèrent [3] d'autant plus que le nombre d'erreurs successives est grand, ce qui n'est pas étonnant. Là où le résultat interpelle, c'est que l'ordre de parcours de la somme a un impact conséquent sur le résultat.

3. Même plus que ça, on dirait que la série converge ! On connait l'équivalent pour la série harmonique : $H_n \underset{n\to+\infty}{\sim} \gamma + \ln(n)$ où $\gamma \approx 0.5$ est la constante d'Euler. Pour $n = 10^9$, on devrait donc plutôt être autour de $H_n \approx 21$.