Architecture des ordinateurs

Département Informatique

Erwan LEBAILLY — Vilavane LY — Vincent TRÉLAT — Benjamin ZHU

3 mars 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. Montrons que l'opérateur xor $\oplus$ est associatif et commutatif :
Soient $a, b, c \in \{0, 1\}$.

Associativité : on donne ci-dessous la table de vérité de $(a \oplus b) \oplus c$ et $a \oplus (b \oplus c)$:

a	b	c	$a \oplus b$	$(a \oplus b) \oplus c$	$a \oplus (b \oplus c)$
0	0	0	0	0	0
0	0	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	0
0	1	0	1	1	1
0	1	1	1	0	0
1	1	0	0	0	0
1	1	1	0	1	1

Commutativité : on donne ci-dessous la table de vérité de $a \oplus b$ et $b \oplus a$:

a	b	$a \oplus b$	$b \oplus a$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

Enfin, $a \oplus a = 0$ et $a \oplus 0 = a$.

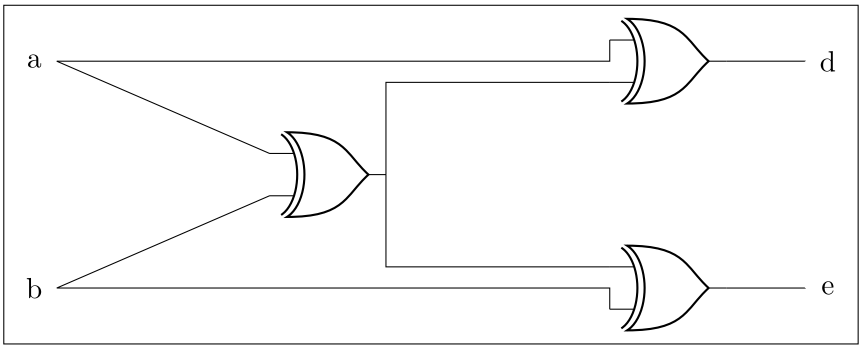
On peut maintenant montrer le résultat demandé :

$$d = a \oplus c = a \oplus (a \oplus b) = a \oplus a \oplus b = b$$

$$e = b \oplus c = b \oplus (a \oplus b) = b \oplus a \oplus b = a$$

$\blacksquare$

b. On donne ci-dessous le circuit correspondant :



1.4 Exercice 4

a. On écrit le code suivant :

```c
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 :

```c
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.

$\blacksquare$

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.

$\blacksquare$

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] :

```c
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.

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

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 [2]. Dans les autres cas, il est possible d'obtenir un résultat correct mais cela résulte plutôt du hasard.

d. On modifie la fonction précédente :

```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;
            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
(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 [3], 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.

2. Il faut tout de même faire attention à la précision. Augmenter la précision ne rendra pas les calculs exacts pour autant.

3. Pas l'égalité.

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 :

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

On constate que les résultats diffèrent [4] 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.
Cette erreur est due au fait que lorsque le parcours est décroissant, on finit pas les petits nombres. Ces derniers causent alors des erreurs d'arrondi car leur ordre de grandeur est faible par rapport aux premiers nombres.

1.13 Exercice 13

a. La taille d'un `float` en C est de 4 *bytes*, soit 32 bits :
- 1 bit de signe
- 8 bits d'exposant

4. 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 $H_n \approx 21$.

- 23 bits de mantisse

b. Convertissons d'abord `0x414BD000` en binaire :

$$414BD000_{16} = 01000001010010111101000000000000_2$$

On identifie ensuite les bits de signe, d'exposant et de mantisse :

$$\underbrace{0}_{S}\ \underbrace{10000010}_{E=130}\ \underbrace{10010111101000000000000}_{T}$$

Le biais b valant 127, notre exposant ici vaut $E - b = 3$. Donc, en "écriture binaire à virgule", on obtient :

$$1.10010111101 \times 2^3 = \underbrace{1100}_{=12}.\underbrace{10111101}_{=0.73828125}$$

Soit finalement :

$$\texttt{0x414BD000}_{16} \text{ encode } 12.73828125$$

c. En suivant le raisonnement inverse, on peut trouver l'exposant et la mantisse de l'encodage de 0.1. On commence par l'écrire en "binaire à virgule" :

$$0.1 = 0001100110011001100110011001101_2$$

On décale la virgule pour trouver l'exposant :

$$0.1 = 1.10011001100110011001101_2 \times 2^{-4}$$

Cela donne donc un exposant de $E = b - 4 = 123 = 01111011_2$ sur 8 bits. Enfin, $0.1 > 0$ donc on met un premier bit à 0. On obtient donc l'encodage suivant – dont on donne également la valeur décimale réelle – pour le nombre 0.1 :

$$\underbrace{0}_{\text{signe}}\ \underbrace{01111011}_{\text{exposant}}\underbrace{10011001100110011001101_2}_{\text{mantisse}} = 0.100000001490116119384765625$$

2 Chapitre 2

2.1 Exercice 1

a. Lorsqu'on crée par exemple un tableau de taille un milliard, on obtient une erreur similaire à la suivante :

```
[1]    54133 segmentation fault  ./a.out
```

b. En expérimentant à la main, on trouve qu'on peut créer un tableau de taille maximale 2 096 286.

2.2 Exercice 2

On vérifie par exemple qu'on peut créer un tableau de taille un milliard.

2.3 Exercice 3

La machine utilisée pour ce TD utilise la convention *little endian*.

2.4 Exercice 4

On écrit déjà le code suivant :

```c
#include <stdio.h>
#include <x86intrin.h>

unsigned long int squareSum(int n){
        unsigned long int tic,toc;
        unsigned int ui;
        int a = 0;
        tic = __rdtscp(&ui);
        for (int i=0; i < n; ++i){
            a = a*a+a*a;
        }
        toc = __rdtscp(&ui);
        return toc-tic;
    }

int main(){
    printf("n=%d: %lu tics\n", 1000, squareSum(1000));
    printf("n=%d: %lu tics\n", 10000, squareSum(10000));
    printf("n=%d: %lu tics\n", 1000000, squareSum(1000000));
    printf("n=%d: %lu tics\n", 10000000, squareSum(10000000));
    printf("n=%d: %lu tics\n", 100000000, squareSum(100000000));
    return 0;
}
```

On obtient les résultats suivants :

n	10^3	10^4	10^6	10^7	10^8
mesure	3310	42456	7134866	67719612	637584056

On note qu'on semble bien obtenir une relation qui a l'air linéaire, hormis la dernière mesure.

2.5 Exercice 5

On obtient les résultats suivants :

appel\\n	10^3	10^5	10^7	10^8	10^9
1er	9108	835910	52244848	532734614	5350923602
2ème	3238	685744	31633110	317468584	3161147144
3ème	3088	322012	32683670	341747932	3100426690
$\frac{\text{écart max.}}{\text{moyenne}}$ en %	117	83	53	53	58

2.6 Exercice 6

a. On teste la fonction avec la fonction `test` calculant la somme des premiers carrés. On obtient des résultats cohérents (croissance linéaire). La fonction `print_timing` comprend une amorce qui exécute 10 fois la fonction à tester et moyenne les mesures sur 100 appels.

b. Voici la fonction `print_timing` :

```c
void print_timing(int arg, void (*func)(int), int nb_boot, int
    nb_call)
{
    // boot up
    for (int i = 0; i < nb_boot; i++)
    {
        func(arg);
    }

    // measure
    unsigned long int tic, toc;
    unsigned int ui;
    tic = __rdtscp(&ui);
    for (int i = 0; i < nb_call; i++)
    {
        func(arg);
    }
    toc = __rdtscp(&ui);

    printf("average time : %lu\n", (toc - tic) / n);
}
```

c. Oui.

d. On obtient les résultats suivants :

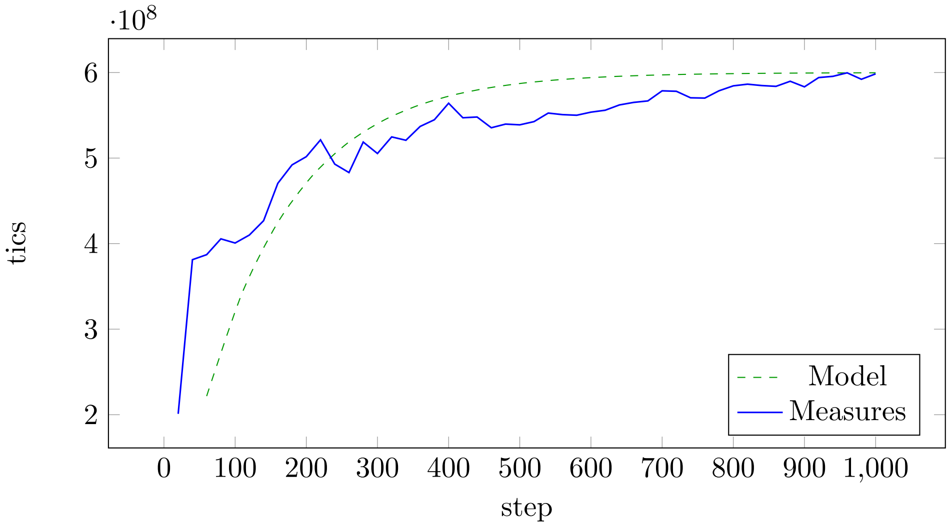
appel\n	10^3	10^5	10^7	10^8	10^9
mesure `print_timing`	4214	432308	31686176	308206789	3176004203

2.7 Exercice 7

On obtient les résultats suivants (arrondis) :

pas	1	2	3	4	8	16	32
mesure ($\times 10^6$ tics)	59	61	63	66	83	102	197

On peut ainsi tracer les résultats obtenus pour les pas situés entre 1 et 1000 :



On remarque que les temps d'accès semblent converger [5] ! Pour mieux l'illustrer, on trace par dessus les mesures la fonction suivante (en vert sur le graphique) :

$$x \mapsto 6 \times 10^8 (1 - e^{-\frac{x}{130}})$$

2.8 Exercice 8

a. On écrit d'abord les deux fonctions suivantes [6] :

```c
void access_seq(int *tab, int n)
{
    int tmp;
    for (int i = 0; i < n; i++)
    {
        tmp = tab[i];
    }
}
```

```c
void access_rand(int *tab, int n)
{
    int tmp;
    for (int i = 0; i < n; i++)
    {
        tmp = tab[(unsigned long int)rand() % 1000000000];
    }
}
```

On obtient les temps suivants :

5. Plus précisément – car on sait qu'il est impossible que cela converge – on semble observer une croissance logarithmique.

6. Il faut inclure les bibliothèques `stdlib` et `time` au début du code, et initialiser le *seed* avec `srand(unsigned int)time(NULL)`.

n	10^3	10^6	10^7	10^8	10^9
`access_seq`	5111	5316032	44992606	450441905	4609590108
`access_rand`	3855618	139739559	1386369206	14181946914	148526968620

b. On constate que la fonction `access_rand` est (très largement, d'un facteur 30 environ) plus lente que la fonction `access_seq` ! Est-ce réellement le cas ? Pour l'instant, on ne peut pas répondre à cette question, toutefois on peut facilement se rendre compte qu'on mesure bien trop de choses. En effet, à chaque accès aléatoire, on appelle – donc on mesure – la fonction `rand`, qui prend visiblement un temps non négligeable.

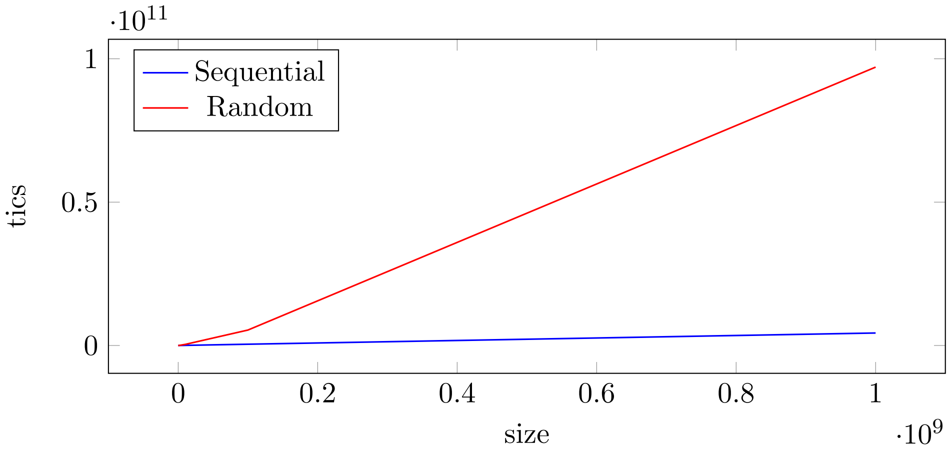
c. On modifie légèrement les fonctions de mesure [7] et on écrit la fonction suivante :

```c
void access_aux(int *tab, unsigned long int *aux, unsigned long
    int n)
{
    int tmp;
    for (register unsigned long int i = 0; i < n; i++)
    {
        tmp = tab[aux[i]];
    }
}
```

On obtient les temps suivants :

n	10^3	10^6	10^7	10^8	10^9
`access_seq`	4298	4263336	42850857	444757816	4356153500
`access_rand`	3463	8848371	377748557	5385496993	97091291039

On obtient les courbes suivantes :



7. Tous les codes sont disponibles sur GitHub. Attention toutefois, l'exécution du code correspondant à cette mesure peut prendre plusieurs dizaines de minutes et nécessite au moins 32Go voire 64 Go de RAM.

Passons rapidement sur l'étude de ces courbes.
Accès séquentiel : on peut définir la fontion

$$t_{\text{seq}} : n \mapsto \tau_{\text{seq}} n$$

donnant une estimation temps mis pour réaliser des accès séquentiels en mémoire, où $\tau_{\text{seq}} \approx 4.298$. Le temps d'un accès séquentiel en mémoire, *i.e.* sur des cases mémoires successives, est donc d'environ 4 *tics*[8]. C'est très rapide !

Accès aléatoire : on peut de même définir la fontion

$$t_{\text{rand}} : n \mapsto \tau_{\text{rand}} n$$

où $\tau_{\text{rand}} \approx 101.9$. Ainsi, un accès aléatoire dans la mémoire prendra environ 102 *tics*. C'est approximativement 20 fois plus long que pour un accès séquentiel, toutefois cela reste du temps constant ! La différence s'explique par le fait que pour "sauter" d'une position à une autre, le pointeur qui fait office de tête de lecture doit faire un calcul arithmétique pour déterminer l'adresse mémoire de la case sur laquelle il doit se rendre. Cela peut nécessiter une soustraction, par exemple, ou encore un modulo, qui est tout de même assez coûteux (quelques dizaines de *tics*).

Conclusion : On constate ainsi que le temps mis pour faire un accès mémoire est constant, peu importe que l'accès soit séquentiel ou aléatoire. On comprend donc l'intérêt du modèle RAM (*Random Acess Memory*) qui permet d'avoir cette propriété tout à fait indispensable aujourd'hui, puisqu'on fait en majorité des appels aléatoires.On peut donc accéder (quasi) instantanément à des données éparpillées en mémoire, ce qui se révèle notamment utile lorsqu'on sollicite plusieurs programmes en même temps.

8. Le temps de la mesure augmente linéairement avec le nombre d'accès mémoire. En moyennant, on obtient bien un temps constant par mesure.