PROG L3-MI
ET

2020-12-16

Consignes. *La durée de cet examen est de deux heures. Une feuille de notes manuscrite au format A4 recto-verso est autorisée. Dans tous les exercices, il est possible d'admettre le résultat d'une question pour répondre aux questions suivantes. Toute réponse doit être justifiée.*

Exercice 1 : Questions courtes

Q.1 : Pour chaque déclaration suivante, quel est le type de la variable déclarée ? Dans chaque cas, que pouvez-vous dire sur sa valeur après déclaration ?

1. `int a = 2.3;`
2. `float a = 2;`
3. `double a;`
4. `float a[1];`
5. `void **a;`
6. `char *a = (char*)malloc(5);`

Q.2 : Pour chacune des suites d'expressions suivantes, décrivez le résultat attendu et si celui-ci mène à une erreur ou un comportement non souhaité.

1. `int8_t t[256]; t[255] = 256;`
2. `int *t; t = 64;`
3. `int *t; *t = 64;`
4. `int *t = (int *)malloc(64); t[63] = 64;`
5. `int t[10]; t = (int*)malloc(10*sizeof(int));`
6. `int t[1000000000]; t[0] = 4;`

Q.3 : Quel problème pose l'extrait de code suivant ? Comment pouvez-vous y remédier ?

```
1 struct poly {int deg; uint32_t *coeffs;};
2 struct poly *alloc_poly(int deg)
3 {
4     struct poly p;
5     p.deg = deg;
6     p.coeffs = malloc(deg*sizeof(uint32_t));
7     return &p;
8 }
```

Q.4 : Quel problème pose l'extrait de code suivant ? Comment pouvez-vous y remédier ?

```
struct poly {int deg; uint32_t *coeffs;};
void delete_poly(struct poly*p)
{
    free(p);
}
```

Q.5 : Quel problème pose la fonction suivante ? Comment pouvez-vous y remédier *sans changer sa signature* ?

```
// renvoie l'élément d'indice x du tableau d'entier p
int getx(const void *p, unsigned x)
{
    return p[x];
}
```

Q.6 : Quel problème pose l'extrait de code suivant ? Comment pouvez-vous y remédier ?
w

```
for (uint8_t i = 0; i < 997; i++)
{
        ...
}
```

Exercice 2 : Choix de représentation

Q.1 : Nous avons étudié lors d'un TD une fonction `uint64_t tr8x8_r(uint64_t m)` qui calcule la transposée d'une matrice 8×8 à coefficients dans le corps à deux éléments $\mathbb{F}_2 \cong \mathbb{Z}/2\mathbb{Z}$ (c-à-d « les entiers modulo 2 »).

1. Décrivez explicitement la représentation choisie pour la matrice dans cette fonction.
2. Illustrez-la sur un exemple non trivial (c-à-d qui n'est pas la matrice dont tous les coefficients sont des 0 ou des 1).
3. Quels sont les intérêts de cette représentation, par exemple par rapport à `uint8_t m[8][8]` ?

Q.2 : On se donne une représentation des éléments du corps à trois éléments $\mathbb{F}_3 \cong \mathbb{Z}/3\mathbb{Z}$ (c-à-d « les entiers modulo 3 ») sous la forme d'un couple (x_0, x_1) de deux bits telle que : $0 = (0,0)$, $1 = (1,0)$, $-1 = (1,1)$. On note que sous cette représentation, la somme (z_0, z_1) de (x_0, x_1) et (y_0, y_1) peut se calculer de la façon suivante :

— $s := x_0 \oplus x_1 \oplus y_1$
— $t := y_0 \oplus y_1 \oplus x_1$
— $z_0 = (x_0 \oplus y_1) \wedge (y_0 \oplus x_1)$
— $z_1 = s \wedge t$

où $\oplus$ et $\wedge$ représentent le « ou exclusif » et le « et » logique respectivement.

1. Proposez un format de représentation en C d'un élément de $\mathbb{F}_3^{64}$ (c-à-d d'un vecteur de dimension 64 sur $\mathbb{F}_3$) exploitant la représentation décrite ci-dessus.
2. Définissez et implémentez une fonction calculant la somme de deux vecteurs de $\mathbb{F}_3^{64}$ représentés de la façon que vous avez définie.

Exercice 3 : Permutations *sheep & goat* et radix-sort en base 2

On s'intéresse au problème d'appliquer une permutation π arbitraire à un tableau de 64 bits représenté par une unique variable de type `uint64_t`.

Q.1 :

1. Combien de bits sont nécessaires pour pouvoir représenter toutes les permutations de 64 éléments possibles ? Le résultat doit être donné sous la forme d'une expression exacte utilisant les fonctions et notations mathématiques usuelles.
2. Combien de bits sont suffisants pour pouvoir représenter toutes les permutations de 64 éléments possibles ? Le résultat doit être donné sous la forme d'un entier dont vous montrerez qu'il est supérieur ou égal à l'expression obtenue dans la réponse précédente.

Q.2 :

1. Écrivez une fonction `uint64_t naive_shuffle64(uint64_t x, uint64_t p[64])` qui renvoie `x` auquel on a appliqué la permutation représentée par `p` t.q. `p[i]` contient $\pi(i)$, la position à laquelle le `i`$^{\text{ème}}$ bit (partant de zéro) de `x` se retrouve dans le résultat. On se contentera pour cette question d'un algorithme naïf.
2. Quel commentaire pouvez-vous faire sur la compacité de la représentation de la permutation utilisée ici ?

Exemple : Un appel à `naive_shuffle64(0x5, p)` avec `p[0] == 63` et `p[2] == 3` doit renvoyer `0x8000000000000008ULL`.

La fonction ci-dessus n'étant pas très efficace, on souhaite résoudre le problème de façon plus satisfaisante en utilisant un radix-sort en base 2. On commence par définir les fonctions :

— `uint64_t compress(uint64_t x, uint64_t m);`
— `uint64_t compress_left(uint64_t x, uint64_t m);`

La fonction `compress` renvoie un entier `uint64_t` contenant dans ses bits de poids faible et de façon contiguë et stable tous les bits de `x` pour lesquels le bit correspondant du masque `m` est à 1. Par exemple `compress(0xA, 0xA)` renvoie `0x3ULL` ; `compress(0xA, 0x5)` renvoie `0x0ULL` ; `compress(0xA, 0x9)` renvoie `0x2ULL` ; `compress(x, x)` renvoie un mot dont seuls les k bits de poids faible sont à 1, où k est le nombre de bits à 1 dans `x` (son *population count*).

La fonction `compress_left` fonctionne de façon identique mais en stockant le résultat dans les bits de poids fort. Par exemple `compress_left(0x5, 0x3)` renvoie `0x4000000000000000ULL`.

Q.3 : L'extension d'instruction BMI2 pour processeurs x86 contient une instruction `pext` accessible via l'intrinsèque `uint64_t _pext_u64(uint64_t x, uint64_t m)` qui implémente la fonction `compress` définie ci-dessus. On rappelle l'existence d'une instruction `popcnt` de calcul du population count accessible par exemple via l'intrinsèque `int64_t _mm_popcnt_u64(uint64_t x)`.

1. Écrivez une implémentation des fonctions `compress` et `compress_left` en utilisant les intrinsèques `_pext_u64` et `_mm_popcnt_u64`.

Q.4 : Une seconde étape dans la résolution du problème consiste à implémenter une fonction `uint64_t sag(uint64_t x, uint64_t m)` qui trie de façon stable son entrée `x` suivant les bits de `m` (indiquant chacun si le bit correspondant de `x` est une chèvre ou un mouton). Autrement dit, tous les bits de `x` aux positions où les bits de `m` sont à 1 (respectivement 0) doivent être stockés dans les bits de poids fort (respectivement faible) du résultat, en préservant leur ordre relatif au sein de chaque partition.

1. Écrivez une implémentation de `sag` faisant appel à `compress` et `compress_left`.

Q.5 : On souhaite maintenant utiliser `sag` pour implémenter un radix-sort en base 2, ce qui permettra au passage d'appliquer efficacement une permutation arbitraire sur 64 bits.

1. Écrivez une fonction `uint64_t sag_shuffle64(uint64_t x, uint64_t p[6])` qui renvoie `x` auquel on a appliqué la permutation représentée par `p` t.q. le $\texttt{i}^{\text{ème}}$ bit de `p[j]` contient le $\texttt{j}^{\text{ème}}$ bit de $\pi(i)$.[1]
2. Écrivez une fonction `void convert_perm(uint64_t p[64], uint64_t pbs[6])` qui convertit la représentation de la permutation utilisée dans **Q.2** en celle utilisée dans cette question.
3. Quel commentaire pouvez-vous faire sur la compacité de la représentation de la permutation utilisée ici ?

Q.6 : L'implémentation de la question précédente nécessite de réordonner à la volée les masques contenus dans `p`.

1. Proposez un format de stockage alternatif pour la permutation qui ne nécessite pas de réordonnancement lors de son application.
2. Estimez le gain de performance apporté par ce nouveau stockage.

Q.7 :

1. Proposez une stratégie complète permettant de tester l'ensemble de vos fonctions.

Références

[BB09] Tomas J. Boothby and Robert W. Bradshaw. Bitslicing and the Method of Four Russians Over Larger Finite Fields. *CoRR*, abs/0901.1413, 2009.

[Jr.13] Henry S. Warren Jr. *Hacker's Delight*. Pearson Education, 2nd edition, 2013.

1. Par ex. sur 4 éléments, la permutation $(0, 2, 3, 1)$ serait représentée de cette façon par `p[0] == 0xC`; `p[1] == 0xE`.