L3-MI
Programmation — Examen Terminal

2023-12-13

Consignes. *La durée de cet examen terminal est de deux heures. Aucun document n'est autorisé.* ***Dans tous les exercices, il est possible d'admettre le résultat d'une question pour répondre aux questions suivantes.*** *Sans mention du contraire, toute réponse doit être justifiée de façon « raisonnable ».*

Exercice unique : *18+*

Le thème de cet exercice est le *décodage majoritaire* pour deux *codes correcteurs binaires* : le *code par répétition de longueur 63* (ci-après « R63 »), et le *code de Reed-Muller en 6 variables et de degré 1* (ci-après « $\mathrm{RM}(1,6)$ »).

On rappelle l'existence en C des opérateurs et *intrinsèques* suivantes :

— `<<` : informellement, `a << b` renvoie l'entier obtenu en décalant les bits de `a` de `b` positions « vers la gauche » (vers les bits de poids fort), en introduisant `b` zéros dans les bits de droite (de poids faible).

— `>>` : informellement, `a >> b` renvoie l'entier obtenu en décalant les bits de `a` de `b` positions « vers la droite » (vers les bits de poids faible), en introduisant `b` zéros dans les bits de gauche (de poids fort) (en faisant ici l'hypothèse d'une opérande `a` de type non signé).

— `~` : `~a` renvoie le complément binaire de `a`.

— `&` : `a & b` renvoie le ET bit à bit de ses deux opérandes.

— `^` : `a ^ b` renvoie le XOR bit à bit de ses deux opérandes. On rappelle en outre que pour tout entier `a`, `a ^ a` vaut `0`.

— `int _popcnt64(int64_t a)` : renvoie le nombre de bits à 1 dans (l'écriture binaire de) `a`.

— `uint64_t _tzcnt_u64(uint64_t a)` : renvoie l'indice (entre 0 et 63) du premier bit à 1 dans (l'écriture binaire de) `a` (ou de façon équivalente, le nombre de zéros de queue dans celle-ci).

Q.1 ***(échauffement)*** **:** Sans justification :

1. Quelle est la valeur renvoyée par `_popcnt64(0xFF00)` ?
2. Quelle est la valeur renvoyée par `_popcnt64(0xBF4E)` ?
3. Quelle est la valeur renvoyée par `_tzcnt_u64(0x800000)` ?
4. Quelle est la valeur renvoyée par `_tzcnt_u64(0xABCDEF)` ?
5. Sachant que les intrinsèques `_popcnt64` et `_tzcnt_u64` font respectivement partie des jeux d'extension d'instruction `popcnt` et `bmi`, donnez la commande permettant de compiler un programme `toto` les utilisant, d'unique fichier source `toto.c`.

On commence par s'intéresser au code R63. Un *mot* de ce code est stocké sur un entier non signé de 64 bits, et peut seulement prendre les deux valeurs suivantes : `0x0ULL` ou `0x7FFFFFFFFFFFFFFFULL`.

Q.2 ***(décodage d'effacements)*** **:**

1. Écrivez une fonction `int8_t r63_dec_era(uint64_t c, uint64_t m)` qui prend en entrée :
   — un mot de code `c` dont certains bits ont possiblement été modifiés ;
   — un masque `m` dont les bits à `1` correspondent aux positions des bits de `c` qui n'ont ***pas*** été modifiés ; notamment, on a que `c & m` vaut `0` ou `m` ;
   et qui renvoie :
   — le code d'erreur `-12` si `m` vaut `0` ;
   — `1` si le mot de code originel (avant possible modifications) valait `0x7FFFFFFFFFFFFFFFULL`, et `0` s'il valait `0x0ULL`.

Q.3 ***(décodage d'erreurs)*** **:**

1. Écrivez une fonction `int8_t r63_dec_bf(uint64_t c)` qui prend en entrée :
   — un mot de code `c` dont certains bits ont possiblement été modifiés ;
   et qui renvoie :
   — `1` si `c` a une majorité de bits à `1`, et `0` sinon.

On s'intéresse maintenant au code $\mathrm{RM}(1,6)$. Un *mot* de ce code est stocké sur un entier non signé de 64 bits, et les 128 valeurs possibles qu'il peut prendre sont données par les 128 combinaisons linéaires binaires (c'est à dire à coefficients `0` ou `1`) des 7 mots contenus dans le tableau constant `g` suivant :

```
const uint64_t g[7] = {0xAAAAAAAAAAAAAAAAULL,
                       0xCCCCCCCCCCCCCCCCULL,
                       0xF0F0F0F0F0F0F0F0ULL,
                       0xFF00FF00FF00FF00ULL,
                       0xFFFF0000FFFF0000ULL,
                       0xFFFFFFFF00000000ULL,
                       0xFFFFFFFFFFFFFFFFULL};
```

et où les combinaisons sont calculées *par un XOR bit à bit*. On admettra que ces 128 valeurs sont bien distinctes.
De plus, on indexe les mots du code avec l'entier `ix` de type `uint8_t` de la façon suivante :

— Le bit de poids fort d'`ix` est nul (autrement dit, `ix` vaut au plus `127`)
— Le mot de code indexé par `ix` est celui qui correspond au XOR bit à bit des éléments de `g` dont les index sont les bits à `1` dans l'écriture binaire d'`ix`.

Exemples :

— `0xCCCCCCCCCCCCCCCCULL` est un mot de $\mathrm{RM}(1,6)$. Puisqu'il s'agit du mot `g[1]`, son index vaut `2`.
— `0xFFFFFFFFFFFFFFFFULL` est un mot de $\mathrm{RM}(1,6)$. Puisqu'il s'agit du mot `g[6]`, son index vaut `64`.
— `0x6666666666666666ULL` est un mot de $\mathrm{RM}(1,6)$. Puisqu'il s'agit du mot `g[0] ^ g[1]`, son index vaut `3`.
— Le mot d'index `13` s'obtient en calculant le XOR bit à bit de `g[0]`, `g[2]` et `g[3]`, et vaut donc `0xA55AA55AA55AA55AULL`.

Q.4 ***(échauffement bis)*** **:**

1. Montrez que `0x0ULL` est un mot de $\mathrm{RM}(1,6)$, dont l'index vaut `0`.

Q.5 :

1. Écrivez une fonction `uint64_t get_ix_rm16_cw(uint8_t ix)` qui renvoie le mot de $\mathrm{RM}(1,6)$ d'index `ix`. *Vous pouvez supposer si vous le souhaitez que le tableau constant `g` est accessible comme variable globale.*

Q.6 ***(décodage d'erreurs bis)*** **:**

1. Écrivez une fonction `uint64_t rm16_dec_bf_nv(uint64_t c)` qui prend en entrée :
   — un mot de code `c` dont certains bits ont possiblement été modifiés ;
   et qui énumère les 128 mots de code possibles de $\mathrm{RM}(1,6)$, et renvoie :
   — un mot de code `cw` de $\mathrm{RM}(1,6)$ tel que le nombre de bits à 1 de `c ^ cw` est minimal[1].

Exemples :
— Sur l'entrée `0xCCCCCCCCCCCCCCCCULL`, cette fonction renvoie `0xCCCCCCCCCCCCCCCCULL`.
— Sur l'entrée `0xCCCCCCCCCCCCCCCDULL`, cette fonction renvoie `0xCCCCCCCCCCCCCCCCULL`.
— Sur l'entrée `0xFEDCBA9889ABCDEFULL`, cette fonction peut renvoyer plusieurs mots possibles, par exemple `0xAAAAAAAAAAAAAAAAULL` ou `0xFFFFFFFFFFFFFFFFULL`.

Q.7 ***(itérateur efficace)*** **:** On rappelle que la suite (c_n) des mots du code binaire réfléchi peut s'obtenir en prenant $c_0 = 0$ et en calculant c_i à partir de c_{i-1} en changeant dans celui-ci la valeur du bit de position j, où j est la position du premier bit non nul dans l'écriture binaire de i.

1. Écrivez une fonction `uint64_t get_next_rm16_cw(uint64_t *cw, uint8_t ix)` qui prend en entrée :
   — un entier `ix` représentant i ;
   — un pointeur `cw` vers un entier contenant le mot de code de $\mathrm{RM}(1,6)$ d'index égal à l'entier correspondant au mot c_{i-1} du code binaire réfléchi ;
   qui modifie l'entier pointé afin qu'il représente le mot de code de $\mathrm{RM}(1,6)$ d'index égal à l'entier c_i du code binaire réfléchi, et qui renvoie :
   — le mot de code de $\mathrm{RM}(1,6)$ ainsi obtenu.
   Vous pouvez toujours supposer si vous le souhaitez que le tableau constant `g` est accessible comme variable globale.

Q.8 ***(décodage d'erreurs ter)*** **:**

1. Écrivez une fonction `uint64_t rm16_dec_bf_cc(uint64_t c)` de spécifications identiques à `rm16_dec_bf_nv`, mais qui utilise `get_next_rm16_cw` pour énumérer les mots de $\mathrm{RM}(1,6)$.

Une personne vous propose maintenant l'approche suivante pour obtenir une fonction de décodage de spécifications identiques à `rm16_dec_bf_nv` et `rm16_dec_bf_cc`, qui ne nécessite pas l'énumération des 128 mots de $\mathrm{RM}(1,6)$:

1. Convertir l'entrée `c` en un tableau `cc` de 64 octets, où l'octet d'index `i` vaut `-1` si le bit de position `i` de `c` vaut `1`, et `1` s'il vaut `0`.
2. Calculer la *transformée de Walsh-Hadamard* de `cc`, grâce à la fonction `fwht_b8` dont le code est donné dans la Figure 1. Cette fonction effectue le calcul de la transformée en place, et écrase donc son entrée avec le résultat.
3. Trouver dans la transformée `cc` un index `ix` pour lequel la valeur absolue de `cc[ix]` est maximale, et :
   (a) si `cc[ix]` est positif, renvoyer le mot de code de $\mathrm{RM}(1,6)$ d'indice `ix` ;
   (b) sinon, renvoyer le complément binaire du mot de code de $\mathrm{RM}(1,6)$ d'indice `ix`.

1. Ceci correspond en fait à une *distance* entre l'entrée et un mot du code.

```
1  void fwht_b8(int8_t cc[static 64]) {
2      int8_t ts;
3      int8_t ta;
4      for (int w = 32; w > 0; w /= 2) {
5          for (int ol = 0; ol < 64; ol += 2*w) {
6              for (int il = 0; il < w; il++) {
7                  int b = ol + il;
8                  int bw = b + w;
9                  ta = cc[b] + cc[bw];
10                 ts = cc[b] - cc[bw];
11                 cc[b] = ta;
12                 cc[bw] = ts;
13 } } } }
```

FIGURE 1 – La fonction `fwht_b8`.

Q.9 ***(décodage d'erreurs quater)*** **:**

1. Écrivez une fonction `uint64_t rm16_dec_bf_fwht(uint64_t c)` de spécifications identiques à `rm16_dec_bf_nv` et qui utilise l'approche décrite ci-dessus. *Vous pouvez supposer l'existence d'une fonction `void cv_u64_b8(uint64_t c, int8_t cc[static 64])` qui en implémente la première étape, en écrivant dans `cc` le résultat de la conversion sur l'entrée `c`*

Q.10 ***(test)***

1. Spécifiez de la façon la plus précise possible (mais sans les implémenter) ***deux*** fonctions de test permettant de tester une implémentation de `rm16_dec_bf_fwht`.

Q.11 ***(coût de `fwht_b8`)*** **:**

1. Montrez que lors d'une itération de la boucle externe (commençant à la ligne 4) de `fwht_b8`, le nombre d'itérations du bloc des lignes 7–12 est une constante (qui ne dépend notamment pas de `w`), que vous déterminerez.
2. Déduisez-en le nombre total d'itération du bloc des lignes 7–12 dans un appel à `fwht_b8`.

Q.12 ***(comparaison du coût des différents décodeurs)*** **:**

1. Comparez sans effectuer de calcul précis le coût des deux fonctions `rm16_dec_bf_nv` et `rm16_dec_bf_cc`.
2. Comparez le coût des deux fonctions `rm16_dec_bf_cc` et `rm16_dec_bf_fwht` en prenant comme métrique le nombre d'affectations, hors affectations de compteurs de boucles.
3. Sachant qu'il est possible pour tout $n > 0$ de définir un code $\mathrm{RM}(1, n)$ similaire au code $\mathrm{RM}(1, 6)$, que celui-ci possède 2^{n+1} mots de longueur 2^n, et que le décodage d'erreur peut être implémenté des mêmes façons que pour le code $\mathrm{RM}(1, 6)$, quel peut être à votre avis l'intérêt de l'utilisation de la transformée de Walsh-Hadamard pour ce décodage [2] ?

2. Pour des mots de longueur 2^n, la transformée de Walsh-Hadamard s'obtient en modifiant dans la Figure 1 l'initialisation de `w` à 2^{n-1} et la borne d'`ol` à 2^n.