

Codes correcteurs d'erreurs

Ce projet peut être fait seul ou en binôme. Il est à rendre sur Moodle le 17 Mai 2023 au plus tard.

Avant de commencer

Cette section comporte des consignes générales pour le projet. Le non-respect de ces consignes entraînera une pénalisation.

Ce projet comporte des questions théoriques, auxquelles vous devez répondre dans un rapport à fournir avec votre rendu. Le rapport devra entre autres comporter une page de garde, contenant vos noms, le nom de l'UE, du projet et le logo de l'Université. Le rapport doit être rendu au format PDF. La qualité du rapport (forme, mise en page, lisibilité, orthographe et grammaire) sera prise en compte dans la notation. Vous y commenterez, lorsque nécessaire, vos choix d'implémentation.

Votre code doit être commenté et indenté. Vous devez fournir un makefile et votre code doit compiler sans erreurs sur Turing. Vous devez suivre les consignes concernant le prototype des fonctions à implémenter. Le code doit être fait dans le langage C. Vous veillerez à citer vos sources éventuelles.

Le rendu sera fait sur Moodle, dans une archive .zip dont le nom contiendra les noms des membres du binôme.

1 Sujet

1.1 Matrice & Outils basiques

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Question 1. Quel est le rendement du code décrit par la matrice G ? Justifiez.

Dans la suite ce projet, nous compterons les bits de gauche à droite. Ainsi, le premier bit de 1000000000000000 est 1. Son indice est 0. Les 4 premiers bits de 1010111100011000 sont 1010. Nous travaillerons principalement sur des variables de 8 ou 16 bits.

Vous trouverez à la fin de ce sujet quelques rappels sur les opérations bit à bit en C. On rappelle également qu'il est possible d'écrire des entiers en binaire en C à l'aide de la notation 0b. Par exemple, il est possible de déclarer une variable en binaire via `short x = 0b1010101010101010`.

Question 2. Pour simplifier votre travail, sa lecture, et le debuggage, commencez par écrire ces quatre fonctions utilitaires `set_nth_bit`, `read_nth_bit`, `print_word` et `chg_nth_bit`. Pour implémenter ces fonctions, vous veillerez à n'utiliser que des opérateurs logiques bit à bit et/ou des shifts. L'utilisation de `if` ou autre structure de contrôle n'est pas autorisée (à l'exception de `for` pour la fonction `print_word`).

1. La fonction `uint16_t set_nth_bit(int n, uint16_t m)` prend en argument un entier n , un mot (encodé sur 16 bits) m , et mets le n -ième bit du mot à 1. Le mot modifié est retourné.
Exemple. `set_nth_bit(0, 0b0100000000000000)` renvoie `0b1100000000000000`.
2. La fonction `uint16_t get_nth_bit(int n, uint16_t m)` prend en argument un entier n , un mot (encodé sur 16 bits) m , et retourne la valeur du n -ième bit du mot.
Exemple. `get_nth_bit(3, 0b0001000000000000)` renvoie 1.
3. La fonction `uint16_t chg_nth_bit(int n, uint16_t m)` prend en argument un entier n , un mot (encodé sur 16 bits) m , et change la valeur du n -ième bit (de 0 à 1 ou de 1 à 0). Le mot modifié est renvoyé.
4. La fonction `void print_word(int k, uint16_t m)` prend en argument un mot (encodé sur 16 bits) m et un entier k , et imprime les k premiers bits de m .
Exemple. `print_word(7, 0b1010101000111111)` renvoie `0b1010101`.

1.2 Étude du code

Question 3. Implémentez en C le code décrit par la matrice G . Plus précisément, vous implémenterez une fonction `uint16_t encode_G(uint16_t m)` prenant en argument un mot à encoder, et renvoyant un mot du code sur 16 bits, rajoutant au mot m les bits de parité décrit par G . Le mot à encoder (sur 8 bits) sera placé sur les premiers bits de la variable m et complété par 8 bits de padding avant d'être fourni en argument à la fonction. Vous emploierez des opérateurs bit à bit et non des opérations arithmétiques. Vous expliquerez avec vos mots votre implémentation.

Exemple. On considère par exemple un mot à encoder `00111001`. Ici, `encode_G(0b0011100100000000)` renverra le mot du code `0b0011100110011010`.

Question 4.1. Implémentez une fonction comptant le nombre de bits à 1 dans un mot. Cette fonction doit avoir le prototype `uint8_t cnt_bits(uint16_t m)`.

Exemple. `cnt_bits(0b1111000011110000)` renverra 8.

Question 4.2. Écrivez une fonction `uint8_t dist_code_G()` qui vérifie que la distance de Hamming du code représenté par G est égale à 4. Expliquez votre algorithme.

Question 4.3. Sachant la distance, déduisez combien d'erreur(s) ce code peut détecter/corriger. Est-il pertinent d'avoir un code avec une distance paire ?

1.3 Polynôme générateur

On rappelle que les codes polynomiaux sont un sous-ensemble des codes linéaires pouvant être implémentés efficacement en hardware à l'aide d'un registre à décalage. Un code matriciel peut donc également être décrit par un polynôme générateur.

Le code généré par la matrice G peut être représenté de manière équivalente par le polynôme $P(X) = X^8 + X^7 + X^5 + X^4 + X^3 + 1$.

Question 5. Dessinez le registre à décalage implémentant le code généré par $P(X)$. Expliquez votre construction (aidez vous du corrigé du TD si besoin).

Question 6. Implémentez une fonction `uint8_t mod_poly(uint16_t m)` simulant le comportement du registre à décalage dessiné à la question précédente. La fonction prendra en argument le mot à encoder, et renverra les bits de contrôle (i.e., le modulo du message par le polynôme générateur). L'implémentation devra tenter d'imiter au plus proche le comportement d'un registre à décalage matériel. Notez que cette fonction ne modifie pas le mot m passé en paramètre, mais calcule simplement les bits de contrôle associés.

Note : Vous pouvez vérifier que votre code est correct en vérifiant que la distance du code est bien 4. Vous pouvez également utiliser cet outil : <http://www.ece.unb.ca/cgi-bin/tervo/polygen2.pl?p=110111001&d=8&c=3&s=1&h=1&g=1>.

Exemple : `mod_poly(0b0000101100000000)` ; calculera le résultat de la division du polynôme représenté par `0b0000101100000000` par $P(X)$, et renverra ici `0b00101100` dans une variable de type `uint8_t`.

Question 7. À l'aide de la fonction précédente, implémentez

- la fonction `uint16_t encode_poly(uint16_t m)` qui prend un mot `m` à encoder, et renvoie le mot du code associé (i.e., `m` suivi des bits de contrôle associés)
- la fonction `uint8_t decode_poly(uint16_t m)` qui prend un mot du code `m` en entrée, et renvoie 1 si le mot est correct, et 0 si une erreur est détectée.

Question 8. Choisissez un mot `m`, et encodez-le à l'aide de votre fonction `encode_poly`. Ajoutez une erreur sur le premier bit, et observez le résultat de `mod_poly` sur le mot erroné. Répétez l'opération sur un autre mot `m` différent. Enfin, observez le résultat de `mod_poly` lorsque le mot `0b1000000000000000` est passé en entrée.

Au vu de vos observations, peut-on (a priori) faire du décodage par syndrome avec un code polynomial ? Justifiez votre réponse.

Question 10. Implémentez du décodage par syndrome en codant la fonction `unsigned uint8_t correct(uint16_t m)` qui prend le message reçu en argument, et renvoie les données (et uniquement les données) corrigées. On considérera qu'une seule erreur peut survenir sur chaque mot. Expliquez votre implémentation.

Exemple : `correct(0b1011100110011010)` renverra `0b00111001`;

Question 11. Étudiez à présent les syndromes obtenus lorsque l'on considère 2 erreurs sur le mot. Répondez (en justifiant) aux questions suivantes :

1. Y-a-t-il une correspondance unique entre les syndromes et les vecteurs d'erreur de 2 bits ? Pourquoi ?
2. Peut-on faire du décodage par syndrome correct dans tous les cas si deux erreurs surviennent sur le medium ? Dans certains cas ?
3. Le code est-il parfait ?

Question 12. Nous allons néanmoins implémenter du décodage par syndrome supportant jusqu'à 2 erreurs. Plus précisément, vous implémenterez une fonction `unsigned char correct2errors(uint16_t m)`, qui :

1. corrige jusqu'à deux erreurs s'il n'y a pas d'ambiguïté pour le syndrome en question
2. S'il y a une ambiguïté (le syndrome peut résulter de deux erreurs différentes), le mot est corrigé aléatoirement suivant les différents vecteurs d'erreur possibles pour ce syndrome.

Vous expliquerez votre implémentation.

Question 14. Nous disposons à présent d'information sur le contenu qui sera transmis sur le medium. Le contenu

- Sera du texte, composé *exclusivement* de lettres en minuscule, d'espaces, de virgules, de points et de retour à la ligne.
- Sera en anglais

À l'aide de ces informations, améliorez votre fonction `correct2errors` en rendant la correction des données plus intelligente.

Question 15. Testez votre implémentation en décodant le message fourni dans le fichier `corrupted_file`, contenant du texte encodé à l'aide de $P(X)$ et ayant subi une ou deux erreur(s) sur chaque bloc de 16 bits.

BONUS Encodez *une image* à l'aide de votre fonction `encode_poly`. Rajoutez une unique erreur par bloc. L'image sera corrigée (à l'aide d'un décodage par syndrome suivant $P(X)$) et examinée par le correcteur. La qualité du meme envoyé sera pris en compte dans la notation (attention, vos intervenants ont un sens de l'humour pouvant être démodé).

2 Rappel : Shift et opérateurs binaires

2.1 Décalage bit à bit (shift)

On rappelle que pour décaler tous les bits d'un nombre n de p bits vers le côté de poids fort, on utilise l'opérateur «. Pour le décalage vers les bits de poids faible on utilise l'opérateur ». Par exemple si $n = 46$ est un nombre de type `char`, alors on peut le représenter en binaire sous la forme `00101110`.

```
n      : 00101110
n>>1  : 00010111
n<<1  : 01011100
```

En particulier, $1 \ll n$ représente le nombre 2^n et $n \gg 1$ le nombre $\frac{n}{2}$.

2.2 Opérateurs binaires

Les opérateurs de bits exécutent les opérations logiques ET, OU, OU exclusif (XOR) et NON sur tous les bits, pris un à un, de leurs opérandes entières. Le tableau ci-dessous résume le fonctionnement de ces opérateurs binaires.

$\&$ représente le ET logique et retourne 1 si les deux bits de même poids sont à 1. $01100000 \& 01000000$ renverra 01000000 .

$|$ représente le logique et retourne 1 si l'un ou l'autre des deux bits de même poids est à 1 (ou les deux). $01100000 \& 0000011$ renverra 01100011 .

\wedge représente le OU exclusif et retourne 1 si l'un des deux bits de même poids est à 1 (mais pas les deux). $01000000 \wedge 11000001$ renverra 10000001 .

\sim représente le NON logique et inverse tous les bits. ~ 00000000 renverra 11111111 .