Examen d'Architecture et Système

11 janvier 2023

Durée : 2 heures.

1 Arithmétique d'entiers

Dans cet question, on traite des entiers non-signés composés de 16 bits. On notera les valeurs hexadécimales avec le préfixe `0x` et les valeurs binaires avec le préfixe `0b`. Les opérations permises sont `+`,`-`,`*` ainsi que les "et" (`&`), "ou" (`|`), "xor" (`^`) et "non" (`~`) binaires et le décalage à gauche et droite (`<<`, `>>`).

On va étudier une méthode pour inverser l'ordre de bits dans un entier. P.ex, si l'entier est `0b` 0111 0101, on cherche à construire l'entier `0b` 1010 1110. Pour n bits, l'algorithme évident marche en temps $\mathcal{O}(n)$, mais il existe une méthode qui fonctionne avec quatre opérations arithmétiques. Dans l'intérêt de simplicité, on va étudier le cas $n = 4$, même le principe s'applique plus généralement. Soit donc $x =$ `0b` $pqrs$, où p, q, r, s sont les valeurs des quatre bits, et on cherche à construire le mot `0b` $srqp$.

(a) *Dupliquer une séquence de bits:* Si $x =$ `0b` $pqrs$, quelle seule opération produit le mot `0b` $0000\,pqrs\,00pq\,rs00$?

(b) *Additionner des sous-séquences:* Soient $a, b, c, d \in \{0, 1\}$ et $y =$ `0b` $ab00\,00cd$. Quelle seule opération produit un mot dont les quatre bits les plus significatifs sur 16 sont `0b` $abcd$, c'est à dire la somme de $ab00$ et $00cd$?

(c) *Masquer des bits:* Encore pour $x =$ `0b` $pqrs$, quelle seule opération produit `0b` $0q0s$?

(d) À partir de x, comment produire `0b` $srqp$ avec quatre opérations (dont un décalage) ?

Indication: Utiliser les opérations de (a)–(c) (pas nécessairement dans cet ordre, et avec de différentes constantes), puis un décalage. Il y a plus qu'une seule solution.

2 Circuits logiques

Pour rappel, un k-multiplexeur est un circuit qui prend en entree 2^k signaux $x_0, \ldots x_{2^k-1}$ et un entier s (codé sur k bits). La sortie du multiplexeur est un signal $y = x_s$, un multiplexeur sélectionne donc un bit parmi 2^k, selon s.

Un *k-codeur* est un circuit qui prend 2^k signaux $(x_0, \ldots, x_{2^k-1})$ et fournit un vecteur de k sorties $y_{k-1} \cdots y_0$ représentant un entier y. Si $x_i = 1$ (pour un i quelconque) et $x_j = 0$ pour tout $j \neq i$, alors y égale i. Si zéro ou plusieurs signaux sont 1, le comportement n'est pas spécifié.

(a) Construire un circuit réalisant un 2-codeur.

(b) Comment géneraliser la construction pour un k quelconque ? Quelle est la taille et la profondeur de votre construction par rapport à k ?

Un *codeur de priorité* (CP) est comme un codeur, mais il gère le cas où plusieurs des signaux en entrée sont 1. Dans ce cas, y prend la valeur du plus grand indice i tel que $x_i = 1$. Une autre sortie z indique si au moins un des x_i était 1. Si $z = 0$, la valeur de y n'est pas spécifiée. Un CP est utilisé, par exemple, au sein d'un processeur pour décider quelle parmi plusieurs interruptions est prioritaire.

(c) Construire un 2-CP (i.e., avec 4 signaux en entrée).

(d) Construire un $(k+1)$-CP à partir de deux k-CP et un multiplexeur (plus quelques simples portes binaires).

(e) Construire un $2k$-CP à l'aide des k-CP et k-multiplexeurs. (Vous pouvez utiliser un nombre illimité de k-CP.)

3 Détection et correction d'erreurs

Un problème d'ingénierie des ordinateurs sont les *fuites de mémoire* : un bit dans la mémoire peut changer sa valeur de façon imprévue en raison des fautes éléctriques. On s'intéresse aux méthods pour détecter et automatiquement corriger ces erreurs. On va supposer que les erreurs sont rares et que, dans un mot binaire, au plus deux bits sont erronés.

La *parité* d'un mot est 1 si dans sa représentation binaire le nombre d'uns est impair, sinon 0. Autrement dit, un mot avec son bit de parité possède un nombre pair d'uns. Si n'importe quel bit (en inclus celui de parité) change de valeur, cette dernière propriété ne tient plus ce qui permet de détecter la présence d'une erreur.

Un seul bit de parité ne permet pas de détecter si deux bits sont erronés. Pire, en cas d'une seule erreur on détecte sa présence mais sans savoir quel bit est fautif. Notre objectif est (i) d'identifier le bit fautif dans le cas d'une seule erreur, et (ii) de détecter la présence de deux erreurs (dans ce dernier cas, sans identifier de quels bits il s'agit). Pour cela, on étudie les *codes de Hamming*.

Dans un mot de Hamming, on mélange les bits de données et de parités. La position 0 est inutilisée, les positions 1,2,4,8,... sont utilisées pour des bits de parité, les autres pour des bits de données. Un bit de parité sur position p donne la parité des bits sur les positions x où p figure dans la représentation binaire de x (du coup, le bit sur position 1 compte la parité des positions impaires, celui de 2 la parité des positions 2,3,6,7,... etc.) Dans 15 bits, on code donc 11 bits de données et 4 bits de parité.

(a) Compléter les bits de parité dans le mot suivant:

position	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
bit	1	0	1	1	1	0	1		0	0	1		1		0

(b) Dans les deux mots suivants, existe-t-il une erreur ? Si c'est le cas, comment identifier la position du bit fautif ?

position	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
bit	0	1	1	1	0	0	0	1	1	1	0	1	1	0	1
bit	1	1	0	1	1	1	1	1	0	1	0	0	0	0	1

(c) Le schéma proposé ne permet pas encore de détecter la présence de deux erreurs. Pourquoi ? Proposer une amélioration qui le permettra.

4 Programmation concurrente et sémaphores

(a) On considère les trois thread ci-dessous, sous l'hypothèse qu'ils ont été lancés tous en même temps. Combien d'affichages différentes peut-on produire avec ce programme ?

Indications: C'est de la combinatoire... D'ailleurs, ne pas prendre compte des questions de tamponnage, on va supposer que tout `printf` est transmis à l'écran tout de suite.

```
void t1 () {
    printf("a");
    printf("b");
    printf("c");
}
```

```
void t2 () {
    printf("d");
    printf("e");
    printf("f");
}
```

```
void t3 () {
    printf("g");
}
```

(b) Dans le suivant, nous allons utiliser un syntaxe simplifié pour les opérations sur les sémaphores:

- `init(s,n)`: créer un sémaphore `s` avec `n` créneaux ;
- `wait(s)`: obtenir un créneau dans sémaphore `s` ;
- `post(s,k)`: libérer `k` créneaux dans sémaphore `s`.

On dispose de k sémaphores et de n threads, chacun avec un identifiant i. L'action principal d'un thread est d'afficher son identifiant sur l'écran une fois. La fonction initiale `main` initialise tous les sémaphores à 0, lance les thread et attend leur fin (avec une syntaxe légèrement simplifié par rapport à C) :

```
sem_t s[K];
pthread_t th[N];

void thread (int id) {
  // opérations sur les sémaphores, à remplir
  printf("mon id est %d\n",id);
  // opérations sur les sémaphores, à compléter
}

int main () {
  for (i = 0; i < N-1; i++) sem_init(s+i,0,0);
  for (i = 0; i < N; i++) pthread_create(th+i,thread,i);
  for (i = 0; i < N; i++) pthread_join(th+i);
}
```

On veut ajouter des opérations sur les sémaphores afin que les identifiants apparaissent dans l'ordre croissant. Si $k = n - 1$, ceci est facile (cas traité dans le cours). Trouvez donc une solution pour le cas $n = 2^k$, en modifiant la fonction `thread` et aucune autre partie du programme.