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` $0000\,pqrs\,00pq\,rs00$?

Solution: `x * 0x104` (équivalent à `x*256 + x*4`)

(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 $00cd$?

Solution: `y * 0x1100`

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

Solution: `x & 5`

(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.

Solution: En reprenant les idées des réponses précédentes, on peut faire:

```
(((x * 0x82) & 0x294) * 0x110) >> 12
```

Expression	Résultat en binaire
x	0000 0000 0000 pqrs
* 0x82	0000 0pqr s00p qrs0
& 0x294	0000 00q0 s00p 0r00
* 0x110	srqp 0000 0000 0000
>> 12	0000 0000 0000 srqp

D'autres jeux de constantes sont possibles. Remarque technique: Le code suivant échoue dans C si on l'exécute sur une machine à 32 ou 64 bits:

```
unsigned x,y;
x = ...;
y = (((x * 0x82) & 0x294) * 0x1110) >> 12;
```

La raison en est que le calcul entier se fait dans un registre de 32/64 bits avant de tronquer le résultat. Il faut donc mettre le décalage dans un instruction séparée.

2 Circuits logiques

Pour rappel, un k-multiplexeur est un circuit qui prend en entree 2^k signaux $x_0, \dots x_{2^k-1}$ 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 $(x_0, \dots, x_{2^k-1})$ 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.

Solution: Pour le 2-codeur, $y_0 = x_1 \vee x_3$ et $y_1 = x_2 \vee x_3$.

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

Solution: En général, y_i est la disjonction de tous les x_j tel que le i-ème bit dans la représentation binaire de j est 1 (donc la moitié des signaux dans tous les cas). Un "ou" entre 2^{k-1} signaux peut se construire en profondeur $\mathcal{O}(k)$. La taille est alors $\mathcal{O}(k \cdot 2^{k-1}) = 2^{\mathcal{O}(k)}$.

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).

Solution:

- $z = x_0 \vee x_1 \vee x_2 \vee x_3$
- $y_0 = x_3 \vee (x_1 \wedge \neg x_2)$
- $y_1 = x_3 \vee x_2$

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

Solution: (Dans le suivant, les superscripts note de différents vecteurs.) Supposons qu'un k-CP sur $x_0, \ldots, x_{2^k-1}$ donne z_0 et un k-vecteur $y^{(0)}$, et qu'un deuxième k-CP sur $x_0, \ldots, x_{2^k-1}$ donne z_1 et $y^{(1)}$.

Ensuite, $z = z_0 \vee z_1$. Il reste à produire un $(k+1)$-vector y. Pour le bit le plus significatif, $y_k = z_1$. Pour le reste, on choisit entre $y^{(0)}$ et $y^{(1)}$ à l'aide d'un multiplexeur qui sélectionne avec z_1.

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

Solution: Similaire à la partie précédente, cette fois on a 2^{2k} signaux en entrée, et on utilisera 2^k k-CP pour produire $z_0, \ldots, z_{2^k-1}$, ainsi que des k-vecteurs $y^{(0)}, \ldots, y^{(2^k-1)}$.

Pour finir, nous devons calculer le bit z et un $2k$-vecteur y. Appellons y' les k bits les plus significatifs de y, et y'' les k bits les moins significatifs.

Pour produire z et y', on utilisera encore un k-CP sur les z_i. Ensuite, y'' est sélectionné parmi les $y^{(i)}$ à l'aide des multiplexeurs avec y' comme sélecteur.

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

Solution: De gauche à droite, 1, 0 et 1.

(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

Solution: Dans le premier mots, tous les bits de parité sont corrects.
Dans le deuxième mot, les bits aux positions 8 et 2 sont fautifs et ceux aux positions 4 et 1 sont corrects. En supposant qu'il s'agit d'une seule erreur, on obtient la position fautive ainsi :

- Puisuqe le bit 8 est fautif, l'erreur doit être parmi les positions de 8 à 15.
- Puisque le bit 4 est correct, l'erreur ne peut être parmi les positions 12 â 15, il en reste de 8 à 11.
- En appliquant le même raisonnement sur les bits 2 et 1, on limite le champs de recherche à la position 10.

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

Solution: Le schéma proposé ne permet pas de distinguer le cas d'une seule erreur du cas de deux erreurs. Par exemple, dans le second mot de (c), il se pourrait que les deux bits 8 et 2 sont fautifs (au lieu du seul bit de 10).

Pour corriger le problème, il convient d'ajouter un bit de parité pour le mot entier sur la position numéro 0 (qui elle n'est pas pris en compte par les autres bits de parité). Dans ce cas :

- Si deux erreurs interviennent, le bit de parité 0 reste "correct". Par contre au moins une des erreurs concernera les bits 1 à 15, et du coup au moins un des autres bit de parité indiquera une erreur.
- Si une seule erreur intervient, le bit de parité 0 sera incorrect et les autres bits de parité en indiqueront la position (entre 0 et 15).
- Évidemment, sans aucune erreur, tous les bits de parité seront corrects.

Attention, il ne suffit pas d'introduire un bit de parité qui compterait uniquement les autres bits de parité, ou uniquement les données !

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");
}
```

Solution: Commençons dans un premier temps par les entrelacements entre `t1` et `t2`, sans prendre compte de `t3`. Ceux-ci produiseront 6 lignes, dont trois de `t2`. Le résultat est entièrement déterminé par le choix de ces trois lignes, il y a donc $\binom{6}{3} = 20$ possibilités.

Si on ajoute `t3`, sa seule ligne d'affichage intervient soit avant toutes les autres, soit après n'importe des six lignes de `t1` et `t2`. Au total, il y a donc $7 \cdot 20 = 140$ affichages possibles.

(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.

Solution:

L'idée de départ est bien sûr que chaque thread attend des threads en correspondance avec l'écriture binaire de son identifiant. Si les identifiants des threads sont de 0 à $n-1$, ceux des sémaphores de 0 à $k-1$, un thread attend les sémaphores qui correspond aux uns dans son écriture binaire, en commençant par le bit le plus significatif. Du coup le thread numéro $40 = (101000)_2$ attend le sémaphore numéro 5, puis numéro 3.

Suivant ce principe, si les sémaphores sont tous initialisés à 0, alors la moitié des threads est bloqué au sémaphore d'index maximal, la moitié des autres au prochain et de suite, et seulement le thread numéro 0 est libre de procéder.

Supposons qu'un thread vient d'afficher son identifiant et que son identifiant est de $id = (x \cdots x01 \cdots 1)_2$ (pour des x quelconque), avec m uns à la fin. Le prochain thread à libérer est $id + 1 = (x \cdots x10 \cdots 0)_2$ qui est bloqué sur le sémaphore numéro m. Or, il y a 2^m threads on total qui sont bloqués sur ce même sémaphore. Il convient de les libérer tous ; néanmoins seul le thread $id + 1$ sera complètement libre de continuer. Le code d'un thread devient ainsi comme suit:

```
void thread (int id)
{
  int i;
  for (i = K-1; i >= 0; i--)
    if (id & (1<<i)) wait(s+i);

  printf("mon id est %d\n",id);

  i = 0; while (id & (1<<i)) i++;
  if (i < K) post(s+i,1<<i);
}
```