

# L3 MI — Programmation

Pierre Karpman

`pierre.karpman@univ-grenoble-alpes.fr`

`https://www-ljk.imag.fr/membres/Pierre.Karpman/tea.html`

2019-10-23

## Opérateurs bit-à-bit

## Premier exemple

---

Le langage C définit des opérateurs permettant de manipuler individuellement les bits d'entiers (généralement non signés)

- ▶ L'opérateur `^` calcule le XOR bit-à-bit

Exemple :

```
uint8_t a = 0x55;  
uint8_t b = 0xAA;  
uint8_t c = a ^ b; // 0xFF
```

Si on interprète  $a$  et  $b$  comme des vecteurs de  $\mathbf{a}, \mathbf{b} \in \mathbb{F}_2^8$ ,  $a \wedge b$  calcule  $\mathbf{a} + \mathbf{b}$

*Remarque* : Il est pratique (et usuel) d'écrire les opérandes avec une base compatible avec une action bit-à-bit, c-à-d la base 16

On dispose aussi des opérateurs suivants

- ▶ L'opérateur  $\&$  ; calcule le AND bit-à-bit (le produit de Hadamard/Schur de vecteurs de  $\mathbb{F}_2^n$ )
- ▶ L'opérateur  $|$  ; calcule le OR bit-à-bit
- ▶ L'opérateur unaire  $\sim$  ; calcule le NOT bit-à-bit
  - ▶ Exercice : comment peut-on calculer  $\sim x$  grâce à  $\wedge$  ?
  - ▶ comment peut-on calculer  $a | b$  grâce à  $\sim$  et  $\&$  ?
  - ▶ comment peut-on calculer  $a \wedge b$  grâce à  $\sim$  et  $\&$  ?
  - ▶ proposition : l'opérateur NAND est universel

## Exercice : inclusion d'ensembles

---

Soit un ensemble  $\mathcal{S}$  d'au plus  $n$  éléments  $x_i$ ,  $i \in \llbracket 0, n-1 \rrbracket$ . On représente un sous-ensemble  $\mathcal{S}'$  de  $\mathcal{S}$  par l'entier  $s' := \sum_{i=0}^{n-1} [x_i \in \mathcal{S}'] \times 2^i$  (où  $[P]$  est un entier qui vaut 1 si le prédicat  $P$  est vrai, et 0 sinon)

**Q 1** : supposons que  $n \leq 64$  ; comment peut-on caculer efficacement l'inclusion de  $a$  dans  $b$ , où chacun est représenté par un `uint64_t` ?

**Q 2** : utilisez le résultat précédent pour définir une fonction d'évaluation d'un monôme de  $\mathbb{F}_2[X_1, X_2, \dots, X_n] / \langle X_i^2 - X_i \rangle_{1 \leq i \leq n}$  en un point de  $\mathbb{F}_2^n$  quand  $n \leq 64$

## Opérateurs de décalage

Il est également possible de décaler les bits d'un unique entier :

- ▶ L'opérateur `<<` agit sur un entier de  $n$  bits (à gauche) et un entier entre 0 et  $n$  à droite
- ▶ Le résultat de `a << b` est égal à `a` dont les bits sont décalés de `b` positions vers la « gauche » (c-à-d vers les bits de poids fort), et autant de zéros sont introduits à droite

▶ `1 << 1; // 2`

`1 << 2; // 4`

`uint64_t a = 1 << 40; // 0, car '1' est un`

*↪ littéral sur 32 bits...*

`uint64_t a = 1ULL << 40; // 0x100000000000`

`1 << 64; // 0`

`0x5 << 1; // 0xA`

`0xF << 1; // 0x1E`

## Opérateurs de décalage (cont.)

---

L'opérateur `>>` fonctionne de façon analogue, avec deux variantes :

- ▶ Décalage *logique* ; le résultat de `a >> b` est égal à `a` dont les bits sont décalés de `b` positions vers la « droite » (c-à-d vers les bits de poids faible), et autant de zéros sont introduits à gauche
- ▶ Décalage *arithmétique* pour entiers signés ; le résultat de `a >> b` est [...], et autant de zéros sont introduits à gauche, *excepté le bit de poids fort* dont la valeur conserve le signe de `a` avant le décalage
- ▶ En C, `>>` implémente *habituellement* le décalage logique (resp. arithmétique) pour les entiers non signés (resp. signés)
- ▶ `2 >> 1; // 1`  
`-4 >> 2; // -1 (en général)`  
`0xA >> 1; // 0x5`

## Exercice : décalage circulaire

---

Écrivez une fonction qui implémente le décalage circulaire (la « rotation ») d'un entier non signé de 64 bits par  $r$  positions



D'un point de vu logique, on peut interpréter :

- ▶  $a \ll b$  comme la multiplication de  $a$  par  $2^b$
- ▶  $a \gg b$  comme la division (entière) de  $a$  par  $2^b$
- ▶  $a \& ((1 \ll b) - 1)$  comme le reste de la division (entière ; non signée) de  $a$  par  $2^b$

~>

- ▶ Les divisions par des puissances de 2 sont beaucoup plus efficaces que celles par des nombres arbitraires
- ▶ Ditto les multiplications, ainsi que les multiplications par des nombres dont l'écriture en base 2 a un poids faible (par ex.  $(a \ll 3) + a$  ; //  $a*9$ )

- ▶ Un bon compilateur remplacera une expression  $a * / \% b$  par la variante appropriée quand  $b$  est une puissance de 2 *constante*
- ▶ Le programmeur doit lui-même faire les substitutions si  $b$  est variable (ou inconnu à la compilation)
- ▶ La programmeuse doit choisir avec sagesse la valeur des constantes pour lesquelles une puissance de 2 est admissible
- ▶ Ex. : la réduction de Barrett

Un nombre surprenant de tâches peuvent s'implémenter en utilisant (entre autres) des opérations bit-à-bit, cf. :

- ▶ <https://graphics.stanford.edu/~seander/bithacks.html>
- ▶ *Hacker's Delight*

Certaines ont une interprétation naturelle au niveau des bits (par ex. transposer une matrice binaire), mais d'autres moins (par ex. calculer une approximation de  $1/\sqrt{x}$  quand  $x$  est représenté par un `float`)

## Exemple : calcul du *population count*

---

Soit  $x$  un entier non signé, on définit `popcnt(x)` comme le nombre de 1 dans la représentation de  $x$  en base 2 (on dit aussi que c'est le poids de Hamming de  $x$ )

Exercice : écrire une fonction qui calcule `popcnt` pour un entier non signé de 32 bits

## Exemple : calcul du *population count* (cont.)

---

Une alternative possible à un calcul naïf de `popcnt` est la suivante :

```
uint32_t popcnt(uint32_t x)
{
    x = (x & 0x55555555) + ((x >> 1) & 0x55555555);
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
    x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F);
    x = (x & 0x00FF00FF) + ((x >> 8) & 0x00FF00FF);
    x = (x & 0x0000FFFF) + ((x >> 16) & 0x0000FFFF);

    return x;
}
```

- ▶ En pratique, la plupart des processeurs (modernes) possèdent une instruction permettant de calculer `popcnt`
- ▶ Celle-ci n'est pas directement accessible en C, mais les compilateurs fournissent généralement une fonction `__builtin_popcount` (et `__builtin_popcountl`) qui :
  - ▶ Utilise une instruction native si elle existe (et si la compilation se fait avec l'option `-mpopcnt`, en x86), ou une implémentation efficace sinon
- ▶ D'autres instructions avancées existante, par ex. les extensions BMI1 et BMI2 pour x86 (qui incluent par ex. une instruction comptant le nombre de zéros avant le premier bit non nul)