Architecture et Système

Stefan Schwoon

Cours L3, 2018/19, ENS Cachan

Entiers

Les entiers (tels qu'on les utilise dans les langages de programmation habituels) sont stockés dans un mot de taille fixe (typiquement 8, 16, 32 bits).

Types en C: `char`, `short int`, `int`, `long int`, `long long int`

En C, les tailles de ces types ne sont pas précisement définis par le langage de C, seulement leurs tailles minmales : 8, 16, 16, 32, 32, où `int` est un mot de registre.

On peut obtenir les valeurs concrètes avec `sizeof(char)` etc.

Big vs little endian

Un mot de taille > 8 s'étend sur plusieurs octets.

Big-endian: on stocke les bits les plus significatifs dans le premier octet
p.ex. $(12345678)_{16}$ dans l'ordre 12, 34, 56, 78

Little-endian: c'est l'inverse, on stocke les octets dans l'ordre 78, 56, 34, 12.

Le mode de stockage devient important lors des échanges des données binaires (fichiers, réseau). Dans ces cas, l'ordre doit être spécifié par le protocol / format de fichier.

P.ex., l'Internet protocol (IP) définit cet ordre comme big-endian.

Fonctions en C : `ntohl`, `ntohs`, `htonl`, `htons`

Entiers avec/sans signe

Un mot de n bits peut représenter les valeurs $0 \ldots 2^n - 1$.
On l'appele un entire sans signe (*unsigned*). Les opération arithmétiques travaillent implicitement modulo 2^n.

Un entier avec signe (*signed*) est typiquement stocké en format complément à deux où un mot prend les valeurs $-2^{n-1} \ldots 2^{n-1} - 1$.

Pour les valeurs non-negative, le bit le plus significatif (MSB) est de 0.

Pour les valeurs négatives, le MSB est de 1.

On obtient la représentation de $-i$ en prenant la négation (bit par bit) de i, puis en rajoutant 1.

P.ex., -1 représenté par $11 \cdots 1$, -2^{n-1} par $10 \cdots 0$.

Du coup, l'addition de i and $-i$ en utilisant l'addition *sans signe* donne `0`, le bon résultat.

Conclusion : sur le niveaux binaire, les opérations arithmétiques pour les naturels (p.ex. addition/multiplication) donnent aussi le bon résultat pour les entiers avec signe.

$\Longrightarrow$ la distinction sans/avec signe n'existe pas pour le processeur – c'est simplement une façon d'interpréter (lors de la saisie/affichage sur l'écran) !

En C

Les types entiers peuvent être déclaré comme `signed` ou `unsigned` ; par défaut ils sont `signed`.

Pour des `char` cette distinction n'est pas important lorsqu'on les interprète comme des caractères.

Attention aux opérateurs de décalage (<< et >>) :

pour les `unsigned`, l'opération est dite *logique*, le décalage se fait sur l'intégralité du mot, en inclus le bit le plus significatif;

pour les `signed`, l'opération est dite *arithmétique*, elle conserve le signe.

Valeurs réelles

Les valeurs réelles sont typiquement représentées dans un format virgule flottante, dans un mot de taille fixe, avec une précision limitée.

Idée en général : tuple $\langle s, m, e \rangle$ avec l'interprétation $\pm 2^e \cdot m$.

s est le signe (un seul bit, 0 non-négatif, 1 négatif);

m est la mantisse;

e est l'exposant.

$\rightarrow$ compromis entre taille et précision des valeurs représentables.

Besoin des standards

Problèmes:

- Comment repartir les bits entre taille et mantisse ?
- Représentations non uniques : $\langle s, m, e \rangle \equiv \langle s, 2m, e - 1 \rangle$
- Comment traiter des cas spéciaux (division par zéro), comment traiter les arrondis ?

Le standard le plus important pour règler ces questions s'appele IEEE 754.

En C : `float` en C = IEEE 754, 32 bit; `double` = IEEE 754 (64-bit).

Dans le suivant, on discutera la partie 32 bit, les autres parties étant similaires.

IEEE 754 (variante 32-bit)

IEEE 754 spécifie les conventions suivantes :

1 bit pour le signe, 8 pour l'exposant, 23 pour la mantisse;

Signe : 0 pour positif, 1 pour négatif

Exposant (domaine ± 127) : on stocke la valeur $e + 127$ dans les 8 bits (sans signe). Si tous les bits de l'exposant sont 1, voir ci-dessous.

Interprétation de la mantisse: $1 + (m/2^{23})$, ce qui donne $[1, 2)$.

Remarques :

Cette interprétation de la mantisse garantit une représentation unique.

Cas spéciaux, si exposant = 255 : $\pm\infty$ (avec $m = 0$)
ou NaN (not a number, avec $m \neq 0$)

Opérations arithmétiques dans IEEE

Le standard IEEE définit la procédure à suivre pour effectuer des opérations arithmétiques (comment arrondir, comment traiter des cas spéciaux, . . .).

Discutons le cas d'une addition:

Soient $x = 2^{e_x} \cdot m_x$ et $y = 2^{e_y} \cdot m_y$ et $x > y$
(par simplicité on suppose qu'ils sont tous les deux positifs).

P.ex. $x = 2.5$ et $y = 0.75$, du coup,
$e_x = 1, m_x = 1.25, e_y = -1, m_y = 1.5$.

D'abord, on représente les mantisses en tant qu'entier, en prenant compte de la "'une cachée"). Ceci donne deux entiers $i_x = 0\text{xa}00000$ et $i_y = 0\text{xc}00000$ (pour 23 bits de mantisse).

Ensuite on adapte i_y par rapport à la différence des exposants (décalage par $1-(-1)=2$ au droite), i_y devient $0x300000$.

L'addition entière entre i_x et i_y donne $i_z = 0xd00000$.

Comme le résultat n'excède pas les 24 bits (une cachée plus 23 bits), on garde l'exposant de x et on enlève simplement l'une cachée de i_z pour obtenir la mantisse du résultat.

Si i_z avait débordé les 24 bits, il aurait d'abord fallu décaler i_z à droit par une position (perdant un bit de précision).

Problèmes de virgule flottante

Précision limitée : certaines valeurs “rondes” comme 0.1 ou 2.3 ne sont pas représentables.

Erreurs d’arrondi; p.ex. $x + y$ donne x si x beaucoup plus grand qu’y.

Du coup, certaines lois comme distributivité ne sont plus valables.