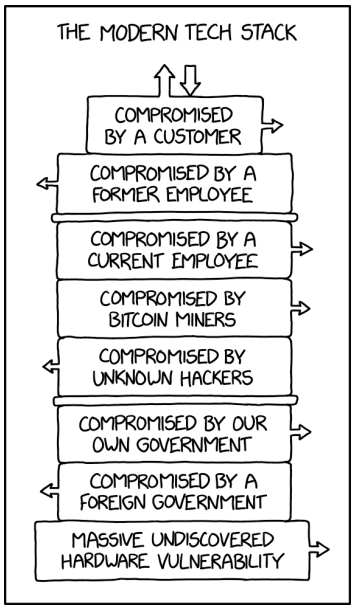
# Architecture des ordinateurs

Version électronique

Responsables du cours : Xavier Goaoc et Vincent Laporte




Source: https://xkcd.com/2166/

# À lire avant la première séance

Ce cours d'architecture est à aborder comme un cours d'informatique expérimentale. Nous allons mettre évidence des phénomènes surprenants par l'observation du comportement de programmes simples et courts. Pour expliquer ces phénomènes, nous introduirons des concepts d'architecture que nous mettrons ensuite à l'épreuve de nouvelles expériences.

Chaque séance correspond à 3-4 chapitres de ce polycopié. Une séance typique se décompose en quatre temps :

- **Avant** le cours, vous **devez** lire un chapitre désigné et essayer d'en faire les exercices ; il convient d'y consacrer de 15 à 30 minutes maximum.
- **Le début** du cours (15-30 minutes) est un temps de questions/réponses en classe complète sans machine. L'objectif est de vous donner l'occasion d'expliciter ce que vous n'avez pas (complètement) compris. Si le travail préparatoire s'avérait insuffisant pour que cette phase soit fructueuse, les enseignants pourront décider de la transformer en **évaluation écrite**.
- **Le reste** du cours consiste en un TP sur machine standart par binôme, en intéraction avec les enseignants. Il s'agit de progresser dans la lecture des autres chapitres de la séance, d'en faire les exercices et d'ébaucher la rédaction d'un compte-rendu de séance.
- **Après** le cours, vous devez terminer les exercices commencés et finaliser votre compte-rendu. Il convient d'y consacrer au plus 1h par séance et il est normal de ne pas réussir à traiter tous les exercices proposés. Les compte-rendus sont à déposer sur la page Arche du module au plus tard le vendredi suivant le cours, avant 23 :59.

**Les objectifs de ce cours** sont d'une part de faire le lien entre la couche logicielle (ici très rudimentaire) et la couche matérielle, et d'autre part d'introduire à de l'informatique bas-niveau. Ce cours se concentre sur la *micro-architecture* des processeurs de la famille Intel x86, c'est à dire la manière dont ces processeurs implémentent leur langage machine. Si certains termes vous sont inconnus en début de cours, c'est tout à fait normal.

Pour pouvoir interpréter le comportement du matériel sur un code, on va examiner des codes écrits en assembleur (en l'occurrence, x86). Pour limiter le temps passé en apprentissage de cet assembleur, on va ici faire produire de l'assembleur par compilation de C. Ce cours inclut donc une introduction aux langages C et assembleur.

Cette démarche nous amène occasionnellement à écrire des codes C invalides du point de vue de la *sémantique* du C afin d'induire la traduction en assembleur souhaitée. Autrement dit, les exemples de programmes C fournis dans ce cours peuvent constituer de très mauvais exemples de programmation C ; ceci est un cours d'architecture, pas de programmation.

**IA ou sécurité ?** Le cours illustre les notions d'architecture de deux manières. D'un côté, on propose des exemples d'**ingéniérie algorithmique**, où l'on cherche à *implémenter* un algorithme donné aussi efficacement que possible. De l'autre côté, on propose la réalisation, par étapes, d'une preuve de concept

de la faille de sécurité SPECTRE. Chaque binôme d'élève choisit l'un de ces fils conducteurs ; cela se materialise par des **jalons** différents aux séances 3, 4 et 6.

| # séance | Chapitre à lire avant le cours | Chapitres "ingéniérie algorithmique" | Chapitres "sécurité" |
|---|---|---|---|
| 1 | 1 | 2, 3 et 4 | 2, 3 et 4 |
| 2 | 5 | 6, 7 et 8 | 6, 7 et 8 |
| 3 | 9 | 10 | 11 |
| 4 | 12 | 13 et 14 | 13 et 15 |
| 5 | aucun | 10 et 14 (puis 16 et 17) | 11 et 15 (puis 16 et 17) |
| 6 | 18 (inclut vidéo) | 16, 17 et 20 | 19 |
| 7 | 1, 2, 3, 5, 6 7, 9, 12, 13 | examen écrit | examen écrit |

**L'évaluation** comportera une partie de contrôle continu (participation en TP et rendus de TP) et une partie d'examen individuel final écrit.

**Les modalités de rendu** sont les suivantes.

- Chaque binôme doit créer un projet sur le `gitlab` de l'UL (`gitlab.univ-lorraine.fr`) intitulé `Rendus_archi_XY` où `X` et `Y` sont les noms de famille du binôme, et déclarer les deux enseignants (`goaoc6@univ-lorraine.fr` et `laporte3@univ-lorraine.fr`) comme membres.
- Pour chaque séance, chaque binôme déposera sur Arche **un unique** compte-rendu au format pdf intitulé `CR_n_XY` où `n` est le numéro de la séance, et `X` et `Y` sont les noms de famille du binôme (toujours dans le même ordre). Ce compte-rendu doit présenter de manière synthétique les réponses aux exercices. Les extraits de code doivent être **minimes** et être commentés dans le texte de la réponse aux exercice. Lorsqu'un exercice demande d'écrire du code, le rapport se bornera à dire si cette écriture de code a posé des difficultés et si oui lesquelles.
- Pour chaque séance, chaque binôme déposera dans leur projet git le code source des programmes écrits pour répondre aux exercices, lorsque ce code apparait pertinent à rendre. Il conviendra d'utiliser **un seul fichier par chapitre**, que l'on intitulera `chapitre_n.c` (ou .s, ...) où `n` est le numéro du chapitre.

> Les enseignants se réservent le droit de pénaliser tout manquement à ces consignes, y compris dans le nommage des fichiers...

**Sources complémentaires.** Pour approfondir les sujets abordés dans ce cours, citons :

- Le cours de Florent de Dinechin à l'ENS Lyon

  `http://perso.citi-lab.fr/fdedinec/enseignement/2019/ASR1/polyENS.pdf`

- Les manuels d'optimisation d'Agner Fog :

  `https://www.agner.org/optimize/`

**Remerciements.** Ce cours est une révision d'un cours construit conjointement avec Carine Pivoteau, de l'Université Gustave Eiffel et ayant bénéficié des conseils, suggestions et explications de Jérémie Detrey. Les erreurs, coquilles et autres approximations sont bien entendu entièrement de notre fait.

# Table des matières

# Chapitre 1

# Représentation de l'information et circuits

Ce premier chapitre décrit les principes de traitement d'une information codée par un mot binaire par un processeur au moyen de *circuits logiques*. Nous en profitons pour expliciter la convention de codage des entiers relatifs.

**Objectifs.** À l'issue de cette séance, il est attendu que vous...

- Compreniez les principes de traitement d'une information numérique par un circuit booléen et sachiez concevoir un circuit réalisant une opération simple.
- Compreniez les méthode standard de codage de nombres entiers et sachiez expliquer le comportement d'un calcul numérique en nombre entiers.

L'Annexe A complète cela par une présentation des principes de codage de nombres à virgule flottante selon le standard IEEE 754.

## 1.1 Calcul booléen

Le calcul booléen définit un ensemble de règles de calcul sur des variables pouvant prendre deux valeurs, traditionnellement décrites comme `vrai` et `faux`. (De telles variables sont appelées *booléennes*.) Il définit des opérations sur ces variables, les plus classiques prenant un argument (opérateur `non`, noté $\neg$) ou deux arguments (opérateurs `et`, noté $\wedge$, et `ou`, noté $\vee$). Ces règles de calcul sont données sous la forme de tables de vérités. Cf

https://en.wikipedia.org/wiki/Boolean_algebra#Basic_operations

pour des exemples.

L'ensemble des valeurs possibles pour un bit, $\{0, 1\}$, est de taille 2, tout comme l'est l'ensemble des valeurs possibles pour une variable booléenne, {`vrai, faux`}. On peut donc fixer une bijection $\{0, 1\} \to$ {`vrai, faux`} et interpréter une formule de calcul booléen comme s'appliquant à des bits. Il existe deux bijections possibles, et on peut en théorie utiliser n'importe laquelle. La convention usuelle, que l'on utilise dans ce cours, identifie 0 à `faux` et 1 à `vrai`. Ainsi, $0 \vee 0 = 0$, puisque `faux ou faux = faux` en calcul booléen, et $0 \vee 1 = 1$ puisque `faux ou vrai = vrai` en calcul booléen.

**Exercice 1.1** ★ Toujours avec la convention $0 \leftrightarrow$ `faux` et $1 \leftrightarrow$ `vrai`...

a. Que vaut $0 \wedge 1$ ?

b. Donner une formule booléenne en trois variables $x, y, z$ qui s'interprète comme calculant la fonction suivante :

$$(0,0,0) \mapsto 0, \quad (0,1,0) \mapsto 1, \quad (1,0,0) \mapsto 1, \quad (1,1,0) \mapsto 0.$$

$$(0,0,1) \mapsto 0, \quad (0,1,1) \mapsto 0, \quad (1,0,1) \mapsto 0, \quad (1,1,1) \mapsto 1.$$

Nous allons appeler une « interprétation d'une fonction de l'algèbre booléenne au travers de la bijection $\{0,1\} \to \{\texttt{vrai, faux}\}$ » une *fonction booléenne*.

> Pour toute fonction booléenne, il existe un système physique qui l'évalue.

Cela peut par exemple se faire par un circuit électrique. On dispose autant de générateurs que l'on a de variables. On allume un générateur si et seulement si la variable correspondante prends la valeur `vrai`. On utilise ces générateurs pour commander des interrupteurs, que l'on dispose en série pour réaliser un `et`, en parallèle pour réaliser un `ou`. Ce n'est bien sûr pas le seul système physique (de la plomberie marcherait tout aussi bien).

## 1.2 Traduction de l'arithmétique en calcul booléen

Observons qu'il est facile de traduire l'arithmétique binaire en calcul booléen. Tout entier naturel $x$ peut s'écrire

$$x = \sum_{i=0}^{k-1} a_i 2^i \quad \text{avec} \quad a_i \in \{0,1\}$$

et cette écriture est unique si l'on impose $a_{k-1} \neq 0$ ou, pour $x = 0$, que $k = 1$. Le mot $a_{k-1}a_{k-2}\dots a_0$ sur l'alphabet $\{0,1\}$ est appelé la *représentation binaire* de $x$ ; on écrit cela $x = (a_{k-1}a_{k-2}\dots a_0)_2$. Chaque $a_i$ est un *bit* (contraction de *binary digit*).

Il s'avère que les fonctions booléennes contiennent le calcul arithmétique. Supposons que l'on additionne deux nombres 1-bit $a_0$ et $b_0$. Le résultat est un nombre 1- ou 2-bits, donc écrivons-le $c_1c_0$ (quitte à ce que $c_1$ vaille 0). On peut décrire la fonction $(a_0, b_0) \mapsto c_1$ par sa table de valeurs :

| $a_0 \setminus b_0$ | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

Remarquons que cette table de valeurs coïncide en tous points avec *l'interprétation* de la table de vérité de l'opération $\wedge$ du calcul booléen. Autrement dit, $c_1$ coïncide avec $a_0 \wedge b_0$.

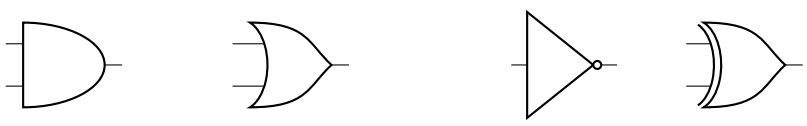
**Exercice 1.2 ★★**

a. Donnez de la même manière $c_0$ en comme une fonction booléenne de $a_0$ et $b_0$.

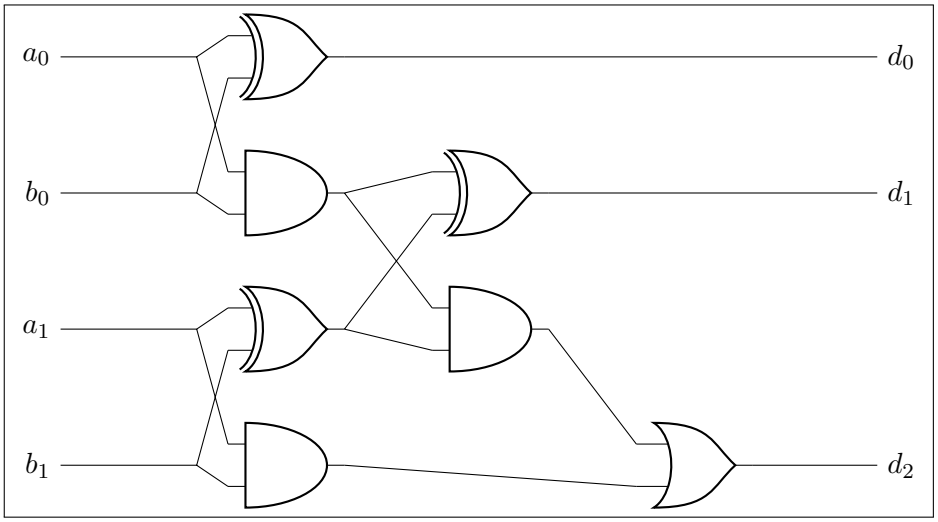
b. Notons maintenant $d_2d_1d_0$ le résultat de l'addition des nombres 2-bits $a_1a_0$ et $b_1b_0$. Donner les formules booléennes donnant $d_2$, $d_1$ et $d_0$ en fonction de $a_0, a_1, b_0$ et $b_1$. On autorise à introduire des variables booléennes intermédiaires.

## 1.3 Circuits combinatoires

Rappelons qu'en calcul booléen, l'opérateur `ou exclusif` est défini par $a \oplus b \stackrel{\text{def}}{=} (a \vee b) \wedge (\neg(a \wedge b))$. On utilise des portes logiques schématisées comme suit (dans l'ordre : `et, ou, non, ou exclusif`) :



On peut vérifier que les formules établies à l'exercice 1.2 donnent lieu au circuit suivant :




Tout comme il existe plus d'un algorithme résolvant un problème algorithmique donné, il existe plus d'un circuit calculant une fonction booléenne donnée. Plusieurs critères d'optimisation sont envisageables : minimiser le nombre de portes, minimiser la *profondeur* (c'est à dire le nombre de portes traversées par un même signal), assurer une *uniformité* (c'est à dire que toute traversée franchisse le même nombre de portes), etc. La conception de circuits booléens est un sujet en soi.

## 1.4 Arithmétique entière sur ordinateur

Les principes précédents guident la réalisation du circuit arithmétique d'un processeur classique. Sans entrer dans les nombreuses complications techniques [1], nous allons en tirer quelques contraintes générales qui s'imposent à *tout* [2] processeur calculant en entiers.

### L'arithmétique est finie

Tout d'abord, un circuit réalise le calcul d'une fonction booléenne et manipule donc l'information en binaire. On mesure donc la taille d'un objet informatique (par exemple un entier) au nombre de bits qu'il faut pour le coder.

Ensuite, un circuit opère en "une passe" sur un entier de taille fixée (par exemple 2 bits dans notre circuit additionneur ci-dessus). Un processeur travaille donc généralement en une ou plusieurs tailles prédéterminées : lorsque l'on dit d'un processeur qu'il est « 64 bits » c'est pour signifier que ses circuits arithmétiques (entre autres) travaillent sur des entiers de taille 64 bits.

Enfin, le résultat d'une opération arithmétique a généralement la même taille que ses entrées. Autrement dit, dans notre circuit additionneur ci-dessus, le bit $c_2$ serait "jeté". Un processeur $b$-bits tronque les bits de poids supérieur à $b$, et calcule donc modulo $2^b$. Comme nous le verrons, les bits tronqués sont *temporairement* disponibles pour qui souhaite les prendre en compte.

### Entiers signés et non signés

Au niveau électronique, un processeur $b$ bits manipule des séquences de $b$ bits, que l'on appelle *mots machines*. Un mot machine peut naturellement s'interpréter comme l'écriture binaire sur $b$ bits d'un entier naturel. Si l'on souhaite travailler sur des entiers *relatifs*, il nous faut fixer de même une *convention de codage*.

Avant tout, puisqu'il existe $2^b$ mots machines $b$-bits distincts, on ne peut coder que $2^b$ nombres distincts sur un mot machine. Pour les entiers naturels, ce sont bien évidemment les entiers 0, 1, ..., $2^b - 1$ qui sont codés. Pour les entiers relatifs, il semble raisonnable de « centrer le 0 », et donc de consacrer la moitié des mots machines au codage d'entiers positifs et l'autre moitié au codage d'entiers

1. Par exemple : quel algorithme de multiplication est le plus intéressant à « câbler » ?
2. Bon, presque tous : rien de ce que l'on décrit ici ne s'applique, par exemple, aux systèmes de calcul quantique.

négatifs. Ainsi, les mots machine de taille $b$ codent généralement les entiers entre $-2^{b-1}$ et $2^{b-1}$ (nous allons préciser sous peu si les bornes sont incluses).

> La règle de codage d'un entier relatif $x$ sur un mot machine $b$ bits consiste à l'encoder par le même mot machine que l'entier *naturel* $y \in \{0, 1, \ldots, 2^b - 1\}$ qui satisfait $x = y \mod 2^b$.

Ainsi, si $0 \leq x \leq 2^{b-1}$ alors l'encodage de $x$ comme entier relatif coïncide avec son encodage comme entier naturel. Les $0 > x > -2^{b-1}$ sont, quant à eux, encodés par le mot machine correspondant à l'écriture binaire de $x + 2^b$.

**Exercice 1.3 ★** Donnez le codage, par cette règle, des entiers relatifs 0, 1, $-1$ et $-2$ sur 4 bits puis sur 8 bits.

Fixons $0 \leq x \leq 2^{b-1}$ et notons $m = a_{b-1}a_{b-2}\ldots a_0$ le mot machine qui le code. Le nombre $-x$ est codé par le mot $m'$ qui représente l'écriture binaire de $2^b - x$. On peut facilement vérifier que $m'$ peut être obtenue depuis $m$ en (i) inversant chaque bit $a_i$ (on remplace 0 par 1 et 1 par 0), et (ii) ajoutant 1 au résultat. L'opération consistant à inverser chacun des bits d'un mot machine est appelée *complément à un*. Pour cette raison, cette règle d'encodage des entiers naturels est appelée **règle du complément à deux**.

Il est très pertinent de se demander pourquoi la règle du complément à deux a été préférée à une alternative plus simple. Considérons par exemple la **règle du bit de signe**, qui consacre un bit (par exemple $a_{b-1}$) au codage du signe (par exemple $0 \leftrightarrow +$ et $1 \leftrightarrow -$) et le reste, soit $a_{b-2}a_{b-3}\ldots a_0$, au codage de l'entier naturel $|x|$ (par sa simple écriture binaire sur $b-1$ bits). Voici une raison pour laquelle préférer la règle du complément à deux à la règle du bit de signe :

> On peut utiliser le même circuit additionneur pour traiter des entiers naturels et pour traiter des entiers relatifs lorsque ces derniers sont codés par complément à deux. Ce circuit ne permet en revanche pas d'additionner des entiers relatifs codés par la règle du bit de signe.

Expliquons cela. Pour faire additionner deux entiers (naturels ou relatifs) à un circuit, on procède en trois étapes : (i) on code ces entiers sur des mots machines, (ii) on fait traiter ces mots machines par un circuit (additionneur), et (iii) on décode le mot-machine retourné comme résultat par le circuit en un entier relatif. Naturellement, le circuit qu'il convient d'utiliser à l'étape (ii) dépends du codage choisi. L'observation ci-dessus exprime le fait que ce circuit est le même pour le codage des entiers naturels par leur écriture binaire, et par le codage des entiers relatifs par complément à deux. (Je vous laisse le soin de vérifier ce point : c'est une conséquence du fait que le circuit additionneur que l'on a décrit calcule modulo $2^b$.) Comme expliqué à l'Annexe A, le codage par bit de signe est utilisé pour les nombres à virgule, pour lesquels il faut de toute façon refaire les circuits.

### Le typage est une vue du compilateur, pas du processeur

Un *entier non-signé* (resp. *signé*) $b$-bits désigne un mot binaire de $b$ bits interprété comme le codage d'un entier naturel (resp. relatif). Le contenu d'une case mémoire (ou d'un groupe de cases mémoires) peut être librement interprété comme un entier signé ou non-signé : cette interprétation n'est attachée ni à la case mémoire, ni au mot binaire qui y est stockée. Elle n'existe, en fait, que lorsque l'on exécute une instruction qui *interprète* ce mot d'une manière ou d'une autre.

Précisons ce point. En C, déclarer une variable a deux effets. D'une part cela réserve une ou plusieurs cases mémoire (le nombre dépends du type de la variable). D'autre part, cela crée une manière d'accéder à ces cases mémoires avec une interprétation prescrite par le type de la variable déclarée. Ainsi, on peut accéder à une *même* case mémoire via des interprétations diverses.

Chaque type d'entier (signé ou non-signé) a un intervalle de représentation. Pour un entier non-signé $b$-bit cet intervalle est $[0, 2^b - 1]$, pour un entier signé $b$-bits c'est $[-2^{b-1}, 2^{b-1} - 1]$. Lorsqu'une opération arithmétique sur des entiers typés produit un résultat qui sort de l'intervalle de représentation de ce type, on parle de *dépassement de capacité*. Ainsi, pour le calcul entier, il y a deux dépassements de capacité distincts selon que le calcul est interprété comme signé ou non-signé.

# Chapitre 2

# C et langage machine

Ce chapitre introduit à l'usage du langage C pour écrire un programme simple en langage haut-niveau (afin que ce soit confortable) tout en en contrôlant la traduction en langage bas-niveau.

## 2.1 Vue d'ensemble : C, assembleur et langage machine

Chaque processeur dispose d'un langage natif, appelé *assembleur* de ce processeur. Il y a donc a priori autant de langages assembleurs qu'il y a de modèles de processeurs. En pratique, il existe des familles de processeurs de langages compatibles à rebours ; ainsi, chaque processeur intel de la famille `x86` peut exécuter tout code écrit pour ses prédécesseurs (mais pas l'inverse, car de nouvelles instructions ont pu apparaître).

Les programmes sont rarement conçus directement en assembleur. On préfère généralement les écrire dans un langage de plus haut niveau [1]. Dans ce cours, on se servira du langage C, et plus précisément du compilateur `gcc`, pour cela.

> Dans ce cours, on invoque `gcc` uniquement par ligne de commande. Autrement dit, on proscrit l'usage d'un IDE.

L'*entrée* d'un compilateur tel que `gcc` est un ou plusieurs fichiers sources C. Un *fichier source* C est un fichier texte [2] contenant la séquence des instructions composant le programme. Voici par exemple un code C et l'assembleur produit par `gcc` lors de sa compilation :

```c
int main(void){
  int a,b,c;

  a=1;
  b=3;
  c=12;

  a = a-b+c;

  return 0;
}
```

```
push   rbp
mov    rbp,rsp
mov    DWORD PTR [rbp-0xc],0x1
mov    DWORD PTR [rbp-0x8],0x3
mov    DWORD PTR [rbp-0x4],0xc
mov    eax,DWORD PTR [rbp-0xc]
sub    eax,DWORD PTR [rbp-0x8]
mov    edx,eax
mov    eax,DWORD PTR [rbp-0x4]
add    eax,edx
mov    DWORD PTR [rbp-0xc],eax
mov    eax,0x0
pop    rbp
ret
```

Les instructions assembleur sont relativement expressives. Le *langage machine* est une convention d'encodage des instructions assembleur en nombres. Ce sont ces nombres qui sont stockés en mémoire.

1. Cette notion de niveau d'un langage est informelle et fait référence à l'abstraction qu'il permet, entendue comme la mise à distance des contraintes imposées par le matériel sur lequel est exécuté le programme.
2. Produit par un outil comme `emacs`, `vi`, `gedit`, `notepad++`, ... mais *pas* par un traitement de texte (`libreoffice`.

Lorsque le processeur exécute un programme, il charge ces nombres depuis la mémoire les uns après les autres, les décode, et agit en conséquence. Voici par exemple, la traduction en langage machine du code assembleur ci-dessus :

```
55 48 89 e5 c7 45 f4 01 00 00 00 c7 45 f8 03 00 00
00 c7 45 fc 0c 00 00 00 8b 45 f4 2b 45 f8 89 c2 8b
45 fc 01 d0 89 45 f4 b8 00 00 00 00 5d c3
```

## 2.2 « Rappels » de C

Nous détaillons ici quelques éléments de syntaxe du langage C, en combinant une introduction rapide à destination d'élèves n'ayant jamais programmé en C et des précisions, spécifiques à ce cours, qui peuvent être nouvelles y compris pour des élèves habitué·es au C.

**Généralités.** Toute instruction termine par un " ;". Un *groupe d'instructions* est une séquence d'instructions délimitée par des accolades {}. Le contenu d'une ligne suivant les caractères // est ignoré par le compilateur, c'est à dire traité comme un commentaire. Un fichier source peut inclure un autre fichier source `toto.h` au moyen de la directive `#include <toto.h>` ; cela permet notamment d'inclure des *bibliothèques standard* de fonctions, en particulier :

- `stdio.h` pour les fonctions d'entrée-sortie (par exemple `printf()`, cf plus bas).
- `stdlib.h` pour des fonctions système (par exemple `malloc()`, cf plus bas).
- `stdint.h` pour des types explicites d'entiers (par exemple `uint64_t`, cf plus bas).
- `x86intrin.h` pour des commandes spécifiques aux processeurs x86 (par exemple `rdtscp`, au chapitre 7).

**Variables et types.** En C, les variables doivent être déclarées avant d'être utilisées, et cette déclaration doit expliciter le type de la variable. Par exemple :

```c
int a;  // déclare une variable a de type entier
int b = 10; // déclare une variable b de type entier
            // et l'initialise à la valeur 10
float c = 1.0; // déclare une variable c de type flottant
            // et l'initialise à la valeur 1
char d = 'a'; // déclare une variable d de type caractère
            // et l'initialise à la valeur a
```

La *taille* d'un type de variable désigne le nombre d'octets occupés en mémoire par une variable de ce type. Cette taille peut être fixe (par exemple un `char` est toujours de taille 1) ou dépendre de facteurs externes (par exemple la taille d'un `int` dépend de la version du langage C suivie par le compilateur). On peut obtenir la taille d'un type au moyen de l'instruction `sizeof()` :

```c
int a = sizeof(float); // déclare une variable entière et
                       //  l'initialise avec la taille du type float
```

Dans ce cours, on utilise beaucoup les types entiers. Il est utile de pouvoir expliciter la taille (16 bits, 32 bits, 64 bits, ...) et si les entiers sont signés ou pas. On utilise donc la bibliothèque `stdint.h` :

```c
uint8_t a;  // déclare une variable a de type entier 8 bits non signé
int16_t b;  // déclare une variable a de type entier 16 bits signé
int32_t c;  // déclare une variable a de type entier 32 bits signé
uint64_t d;  // déclare une variable a de type entier 64 bits non signé
```

**Fonctions.** La déclaration d'une fonction indique le type de donnée de sortie (transmise par l'instruction `return`) et les types des arguments d'entrée. Par exemple :

```
int somme(int a, int b)
  { return a+b;}
```

Tout programme C doit contenir une fonction appelée `main` et retournant un `int` ; c'est le code de cette fonction qui est exécuté lorsque l'on lance le programme. Toute fonction doit être déclarée avant d'être utilisée.

**Instructions de contrôles de flot.** Voici quelques instructions de contrôle de flot usuel illustrant leurs syntaxes :

```
if (i<50)
{
  // code à exécuter
  // si i<50
}
else if (i == 60)
{
  // code à exécuter
  // si i = 60
}
else
{
  // code à exécuter
  // si les autres
  // tests ont échoué
}
```

```
for (i = 0 ; i < 50; i++)
{
  // code à exécuter
  // pour chacune des
  // valeurs de i
}
```

```
while (i<50)
{
  // code à exécuter
  // tant que ``i<50''
  // est vrai
}
```

**Affichage.** L'instruction `printf()` permet d'afficher du texte dans la sortie courante, qui est par défaut le terminal d'où l'exécutable a été lancé. On peut insérer dans le texte à afficher des balises (commençant par %) où doivent être insérées des valeurs de variables. Par exemple

```
printf("a vaut %d et b vaut %f\n",a,b);
```

Cf par exemple la wikipedia[3] pour une description du format des balises.

## 2.3 Compilation et désassemblage

Il nous reste à voir comment compiler un programme et accéder au code assembleur ainsi produit.

### Compilation et exécution d'un programme

Depuis un terminal positionné dans le répertoire contenant le fichier `source.c`, la compilation peut se faire au moyen de la commande

```
gcc -Os source.c
```

Cette commande peut, selon les cas, produire un exécutable (dans le fichier `a.out`) ou un message d'erreur (si le code de `source.c` est incorrect). Dans le premier cas, le programme peut être lancé par la commande

3. https://en.wikipedia.org/wiki/Printf

```
./a.out
```

L'expression "-Os" indique une *option de compilation*, en l'occurrence demandant à `gcc` d'optimiser la compilation afin que le code produit soit compact. Les options -O0, -O1, -O2, -O3 indiquent à `gcc` les optimisations à mettre en œuvre lors de la compilation. En première approximation, ces options sont, de -O0 à -O3, de plus en plus sophistiquées et produisent un code supposé plus efficace (tout en demandant plus de travail de compilation). Dans le détail, ce n'est pas aussi simple... [4]

Dans ce cours, sauf mention du contraire, tous les codes doivent être compilés avec l'option -Os.

### Compilation produisant le source assembleur

On peut demander à `gcc` de sauvegarder dans un fichier texte le source assembleur produit par la compilation. Cela se fait avec l'option -S. Par défaut, le fichier produit a le même nom que le fichier source, mais l'extension ".s".

Au moins deux conventions de syntaxe assembleur `x86` sont couramment utilisées aujourd'hui : la syntaxe Intel et la syntaxe AT&T. Ces syntaxes diffèrent par exemple dans l'ordre des arguments. Ainsi, en syntaxe Intel, `mov rax,rbx` décrit l'opération de copie du registre `rbx` dans le registre `rax`. En syntaxe AT&T, cette même instruction s'écrit `movq %rbx, %rax`. Selon la syntaxe choisie, deux instructions très semblables peuvent correspondre à différents codes en langage machine.

La syntaxe que nous utiliserons au cours des TP est celle d'Intel.

Lorsque l'on demande à `gcc` de produire le source assembleur par l'option -S, on peut (et dans ce cours on doit !) indiquer d'utiliser la convention intel en ajoutant l'option -masm=intel. Ainsi, la commande

```
gcc -Os -S -masm=intel test.c
```

produit dans le fichier test.s le source assembleur, rédigé dans la syntaxe Intel, résultant de la compilation d'un fichier test.c.

### Désassemblage du binaire

On peut visualiser le code assembleur et le langage machine constituant un programme par la commande

```
objdump -d -M intel nom_de_l_executable_a_examiner
```

L'option `-M intel` indique que l'on souhaite utiliser la convention Intel.

Lorsque l'on invoque `gcc` avec l'option -c, le résultat de la compilation n'est pas lié aux bibliothèques qu'il utilise. Le résultat est un fichier .o qui n'est pas exécutable, mais qui contient simplement la traduction en langage machine du code C. Lorsque l'on souhaite visualiser par `objdump` le code d'un programme créé en compilant du code C, il est conseillé d'appliquer `objdump` (avec les mêmes options que ci-dessus) au fichier `.o` plutôt qu'à l'exécutable.

### Alternative occasionnelle : godbolt

Le site https://godbolt.org/ propose un *explorateur de compilateurs* : il permet de visualiser facilement le résultat de la compilation d'un code écrit dans une grande variété de langages (dont le C) par une grande variété de compilateurs (dont gcc) vers une grande variété d'assembleurs (dont `x86-64`).

4. Cf https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html pour l'effet détaillé de ces instructions.

### Exercice

**Exercice 2.1** ★ Récupérez le code assembleur du programme suivant par *chacune* des méthodes décrites ci-dessus.

```
int main(int argc, char **argv) {
  int count = 100 + argc;
  int s = 0;
  int i;
  for(i = 0; i < count; i++) {
    s += i;
  }
  return s;
}
```

# Chapitre 3

# Assembleur : premiers pas

Ce chapitre présente quelques éléments d'assembleur `x86`. L'objectif est ici d'être capable de comprendre un petit code. La documentation officielle complète est disponible à :

https://cdrdv2.intel.com/v1/dl/getContent/671200

Une version html, non officielle, de la liste d'instructions est disponible à :

https://www.felixcloutier.com/x86/

## Les bases

**Registres 64 bits.** Les registres sont des variables internes au processeur. Il y en a peu et, techniquement, ce sont des cases mémoire situées dans chaque cœur du processeur. Les principaux registres 64 bits sont les suivants :

- Les registres de travail `rax, rbx, rcx` et `rdx`.
- Les registres d'index `rsi, rdi` et les registres de pointeur `rbp, rsp`.

**Mnémonique.** Chaque instruction assembleur est caractérisée par une mnémonique, ou mot-clé, en lien avec la fonction de l'instruction. Voici quelques exemples (on donnera un peu plus loin une liste plus fournie) :

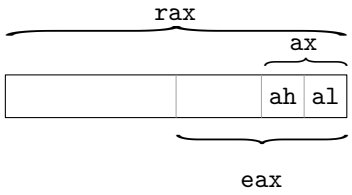
| | |
|---|---|
| `mov` | copie une donnée |
| `add` | effectue une addition |
| `sub` | effectue une soustraction |
| `inc` | incrémente |

**Arguments source/destination.** La majorité des instructions comporte un ou plusieurs arguments. Si une instruction doit retourner un "résultat", il est écrit dans un des arguments. On distingue ainsi les arguments *source*, qui ne sont pas modifié, et les arguments *destination*, qui peuvent être modifiés. Voici quelques exemples :

| | |
|---|---|
| `mov rax, rbx` | copie la valeur de `rbx` dans `rax` |
| `add rax, rbx` | ajoute la valeur de `rbx` à celle de `rax` |
| `sub rax, rbx` | soustrait la valeur de `rbx` à celle de `rax` |
| `inc rdx` | incrémente la valeur de `rdx` |

En convention Intel l'argument destination précède généralement l'argument source. Par ailleurs, la valeur initiale de l'argument destination est parfois utilisée par l'instruction (par exemple pour `add, sub` ou `inc`).

**Sous-registres.** Les registres `rax`, `rbx`, `rcx`, `rdx` sont des registres 64 bits d'usage générique. On peut accéder à différentes sous-parties du registre `rax` au moyen des registres `eax`, `ax`, `al` et `ah` selon le diagramme ci-dessous. Ainsi, `eax` est consitué des 32 bits de poids faible de `rax`, `ax` est constitué des 16 bits de poids faibles de `rax`, `ah` est constitué des 8 bits de poids fort de `ax`, et `al` de ses 8 bits de poids faible. On peut, de même, accéder à différentes sous-parties des registres `rbx`, `rcx` et `rdx` au moyen de `ebx, ecx, edx, bx, cx, dx, bh, bl, ch, cl, dh` et `dl`.




On peut accéder aux sous-parties de 32 ou 16 bits de poids faible des registres `rdi, rsi, rbp, rsp` par, respectivement, `edi, esi, ebp, esp` et `di, si, bp, sp`.

Cette possibilité d'accéder à des sous-parties d'un registre reflète en réalité l'extension progressive de la capacité des processeurs de la famille `x86`. Les registres `ax, ah, al` existaient sur les processeurs 16 bits. Le registre `eax` est apparu avec les processeurs 32 bits (le préfixe '`e`' signifie *extended*). Le registre `rax` est apparu avec les processeurs 64 bits.

## Les différents passages d'arguments

Les arguments d'une instructions peuvent être transmis autrement que par la donnée d'un registre.

**Arguments immédiats.** On peut donner une valeur constante (par exemple 42) comme argument source mais pas comme argument destination. Les nombres sont interprétés comme des décimaux s'ils ne sont pas préfixés, et comme des hexadécimaux s'ils sont préfixés par `0x`.

| | |
|---|---|
| `add rax, 12` | ajoute 12 à `rax` |
| `sub rax, 0x12` | soustrait 18 à `rax` |

**Arguments forcés.** Certaines instructions restreignent le choix du ou des registres permettant de transmettre les arguments. C'est le cas par exemple pour l'instruction `shl`, qui répète un nombre de fois donné (par le second argument) la multiplication du premier argurment par 2. Lorsque le second argument est donné par un registre, ce registre *doit* être `cl`. (Le second argument peut aussi être donné de manière immédiate.)

**Arguments implicites.** Certaines instructions utilisent *dans tous les cas* les données contenues dans certains registres, ce qui rend inutile la déclaration de ces arguments ; ils sont *implicites*. C'est par exemple le cas de l'instruction `mulx`, dont la déclaration demande trois arguments, mais dont l'exécution en utilise quatre ; l'argument implicite est le contenu du registre `rdx` (en 64 bits) ou `edx` (en 32 bits).

## Quelques instructions

On liste ci-dessous les principales instructions que l'on utilisera. Nous ne détaillons pas ici le fonctionnement de chacune, et renvoyons pour cela aux ressources listées en début de Section 3. Notons par ailleurs que toutes les mnémoniques n'admettent pas tous les types d'arguments (registre, ímmédiat, indirect, indirect avec décalage, ...). À nouveau, nous renvoyons aux ressources de référence pour la liste de ce qui est possible.

| | |
|---|---|
| `mov` | copie de donnée |
| `add, sub, inc, dec` | opérations arithmétiques |
| `and, or, xor, not` | opérations logiques (bit à bit) |
| `ror, rol, shr, shl`<br>`sar, sal` | opération de décalage et rotation (bit à bit) |
| `mul, imul, div, idiv` | multiplication et division |

## Exercices

**Exercice 3.1** ★ Indiquer ce que vaut le registre `rax` lors de l'exécution de l'instruction `nop` pour le code suivant.

```
mov     rax,12
add     rax,rax
add     rax,rax
dec     rax
nop
```

**Exercice 3.2** ★ À partir de la page `https://www.felixcloutier.com/x86/mulx` décrire ce que fait l'instruction `mulx`.

**Exercice 3.3** ★★

a. Que fait l'instruction `popcnt rax,rbx` ?

b. Que calcule l'instruction `lzcnt rax,rbx` ?

c. Pouvez-vous proposer un cas d'utilisation de `lzcnt` ?

# Chapitre 4

# Exercices complémentaires

**Exercice 4.1** ★★ Proposez un circuit combinatoire qui réalise l'instruction `lzcnt a,b` où `a` est 2-bits et `b` est 4-bits.

**Exercice 4.2** ★★ Prouver que la règle du complément à deux *induit* le fait que le bit le plus à gauche (dit « de poids fort ») révèle le signe du nombre codé. Autrement dit, le bit de poids fort du codage d'un entier relatif $x$ vaut 0 si $x \geq 0$, et 1 si $x < 0$.

**Exercice 4.3** ★★

a. Donner deux mots 8-bits dont le traitement par un circuit additionneur 8-bits déclenche un dépassement de capacité non-signé, mais pas de dépassement de capacité signé.

b. Donner deux mots 8-bits dont le traitement par un circuit additionneur 8-bits déclenche un dépassement de capacité signé, mais pas de dépassement de capacité non-signé.

c. Donner deux mots 8-bits dont le traitement par un circuit additionneur 8-bits déclenche un dépassement de capacité signé *et* un dépassement de capacité non-signé.

**Exercice 4.4** ★★ Le circuit additionneur dessiné page 11 comporte des croisements, ce qui peut s'avérer gênant pour certains modes de réalisation (par exemple par circuit imprimé monoface). On peut travailler à le dessiner sans croisement, mais que faire si on souhaite réaliser un circuit qui n'admet pas de tracé sans croisement ? Voyons cela...

a. Pour $a, b \in \{0, 1\}$ on définit

$$c \stackrel{\text{def}}{=} a \oplus b, \quad d \stackrel{\text{def}}{=} a \oplus c, \quad e \stackrel{\text{def}}{=} b \oplus c.$$

Donner des expressions aussi simples que possible pour $d$ et $e$ en fonction de $a$ et $b$.

b. En déduire un circuit « échangeur ». Votre circuit doit être dessiné dans un carré, les entrées étant aux deux coins gauches et les sorties aux deux coins droites. Chaque sortie doit être égale à l'entrée du coin opposée. Votre circuit doit être **sans** croisement.

**Exercice 4.5** ★★

a. Donnez les mots machines $m_1$ et $m_{-1}$ qui codent les entiers relatifs 1 et $-1$ par la règle du bit de signe sur 4 bits.

b. Donnez le mot machine $m_0$ produit par un circuit additionneur 4-bits « pour nombres naturels » lorsqu'il a en entrée $m_1$ et $m_{-1}$.

c. Quel est l'entier relatif codé par $m_0$ en règle du bit de signe sur 4 bits ?

**Exercice 4.6** ★★★ Considérons un nombre $a = (a_1 a_0)_2$ dans $\{0, 1, 2, 3\}$. On définit $e = (e_1 e_0)_2$ par $e \stackrel{\text{def}}{=} \min(3, a + 1)$. Donner un circuit booléen qui calcule $e$ à partir de $a$.

# Chapitre 5

# Modèle de mémoire virtuelle

Ce chapitre clarifie de quelle manière un programmeur C interagit avec la mémoire. L'objectif de ce premier modèle est de comprendre *ce que fait* un ensemble d'instructions qui mettent en jeu la mémoire. En revanche, comme nous le verrons, ce modèle ne rend aucunement compte des *performances* d'un tel ensemble d'instructions ; ce sera le rôle des modèles introduits aux chapitres suivants.

**Objectif.** À l'issue de cette séance, il est attendu que vous sachiez analyser une séquence d'accès mémoire par pointeurs.

## 5.1 Rappels de C : tableaux et pointeurs

Un tableau est une collection homogène (tous les éléments ont le même type) d'objets que l'on référence par leur indice dans ladite collection. Par exemple la définition `int32_t tab[3] = { 3, 14, 159 };` déclare un tableau `tab` de trois valeurs de type `int32_t`. La première (3) a pour indice zéro (0), la seconde (14) pour indice un (1) et la dernière (159) pour indice deux (2). Du point de vue du programmeur C, et plus généralement d'un processus exécuté sur un ordinateur classique, la mémoire RAM se comporte comme un grand tableau d'octets. L'indice d'un octet dans ce tableau est appelé son *adresse*. Un pointeur (ou référence) est l'adresse d'un objet en mémoire.

En C on l'obtient via l'opérateur d'adresse `&` et accède à la valeur pointée par l'opérateur de déréférencement `*`. On déclare une variable de type « pointeur vers X » en ajoutant une étoile avant le nom de la variable. Ainsi, déclarer `int *p;` déclare une variable `p` dont la valeur est interprétée comme l'adresse d'une valeur de type `int` ; on dit aussi que `p` est de type « `int*` ». L'instruction `p = &t[1];` lui affecte l'adresse de la deuxième case du tableau `t`, et l'expression `*p` représente la valeur contenue dans cette case. En C, une adresse est un type de donnée abstrait (en particulier, ce n'est pas un nombre). On peut l'afficher avec `%p` dans un appel à `printf`.

Un pointeur se comporte *exactement* comme une référence vers la valeur à l'indice 0 d'un tableau ; on peut notamment écrire indistinctement `*p` ou `p[0]` pour déréférencer un pointeur `p`. Les autres cases dudit tableau peuvent être référencées *via* ce que l'on nomme l'arithmétique de pointeurs. Sous ce nom pompeux, se cache le fait de pouvoir ajouter un nombre entier avec un pointeur : `p + n`, qui est exactement la même chose que `&(p[n])`, ou de faire la différence entre deux pointeurs vers un *même* tableau. Voici deux exemples de fonctions qui parcourent un tableau, à gauche en utilisant les notations « tableau » et à droite les notations « pointeur ».

```c
uint32_t sommeA(uint32_t tab[],
    uint64_t taille) {
  uint32_t resultat = 0;
  uint64_t i = 0;
  for (; i < taille; ++i) {
    resultat += tab[i];
  }
  return resultat;
}
```

```c
uint32_t sommeB(uint32_t *p,
    uint64_t taille) {
  uint32_t resultat = 0;
  uint32_t *end = p + taille;
  for (; p < end; ++p) {
    resultat += *p;
  }
  return resultat;
}
```

## 5.2 La mémoire est un tableau

Un programmeur peut avoir l'impression de disposer de l'intégralité de l'espace mémoire de la machine. C'est une impression trompeuse car cette mémoire est en réalité partagée (par le système) entre tous les processus exécutés par la machine. Ce partage se fait par un mécanisme de traduction des **adresses mémoire que manipule le processus** (dites *adresses virtuelles* de ce processus) en **adresses mémoire que manipule le composant électronique** (dites *adresses physiques*). Ainsi, deux processus peuvent avoir l'impression d'avoir chacun rangé une information à l'adresse `0x42 000` sans que ces informations ne se télescopent : cette adresse virtuelle sera traduite en deux adresses physiques distinctes.

Pour illustrer ce phénomène, considérons le programme suivant.

```c
#include <sys/mman.h>
#include <stdint.h>
#include <stdio.h>

int main(void) {
  uint8_t* p = mmap(
    0x0000000001000000,
    0x4000,
    PROT_READ | PROT_WRITE,
    MAP_ANONYMOUS | MAP_PRIVATE,
    -1,
    0);
  printf("%p\n", p);
  return getchar();
}
```

Ce programme demande au système d'exploitation (via l'appel système `mmap`) de lui attribuer une portion de mémoire RAM de taille 16 kiO (deuxième argument) débutant à l'adresse (virtuelle) `0x1000000` (premier argument). (On peut ignorer les arguments suivants dans le cadre de ce chapitre.) Si c'est possible, cet appel système renvoie un pointeur vers le bloc demandé. L'appel final à **getchar** permet de suspendre l'exécution du programme sans terminer le processus.

**Exercice 5.1 ★**

a. Exécuter le programme ci-dessus et s'assurer que l'appel système fonctionne. On pourra utiliser l'utilitaire en ligne de commande **pmap** pour observer quelles régions de la mémoire sont attribuées à un processus, étant donné son *pid*.

b. Modifier ce programme pour qu'il écrive dans le bloc de mémoire puis relise (et afficher) la valeur écrite. Un appel à **getchar** peut vous permettre de faire une pause entre la lecture et l'écriture.

c. Exécuter deux instances du programme modifié en parallèle et faire varier l'entrelacement des écritures et lectures des deux processus. Les deux appels à **mmap** renvoient-ils la même valeur ? Les deux processus partagent-ils le bloc de mémoire ?

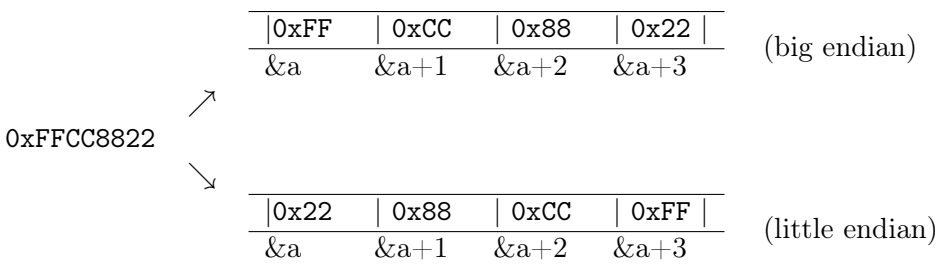
Dans ce cours, on ne travaillera qu'en terme d'adresses virtuelles. Ainsi, pour nous, la mémoire est un tableau d'octets qui commence à l'adresse 0.

## 5.3 Endianness (boutisme)

Une donnée qui fait plus d'un octet doit occuper plusieurs cases mémoires. Par exemple, un `uint32_t` occupe quatre octets (on peut le vérifier par exemple avec `sizeof()`). À l'issue de la déclaration

```
uint32_t a = 0xFFCC8822;
```

on a écrit les 4 octets `0xFF`, `0xCC`, `0x88` et `0x22`, qui composent notre entier, en mémoire. Cela peut se faire de deux manières :




L'*endianness* se manifeste notamment dans l'encodage des constantes numériques dans les programmes binaires. Dans l'exemple ci-dessous, nous avons à gauche un code source C et à droite le binaire résultant de sa compilation (`cc -Os -c boutisme.c`) puis désassemblé (`objdump -d -M intel boutisme.o`) :

```c
int main(void) {
  return 0xFFCC8822;
}
```

```
0000000000000000 <main>:
   0:   b8 22 88 cc ff          mov    eax,0xffcc8822
   5:   c3                      ret
```

**Exercice 5.2 ★** D'après le listing ci-dessus, quelle est le boutisme utilisé sur la machine/système ayant servi à le produire ?

## 5.4 La pile (« stack »)

Chaque processus dispose d'un bloc de mémoire appelé *pile* qui sert à stocker les valeurs des variables locales des fonctions. On peut connaître et modifier la taille par défaut de ce bloc via l'utilitaire système **ulimit**.

En C, on peut réserver de l'espace mémoire sur la pile par la *déclaration de variables* et en particulier de tableaux. Par exemple :

```c
// ... dans le corps d'une fonction
    int  tab[100];
```

Cette ligne de code réalise les opérations suivantes :

- elle réserve un espace mémoire, ici l'espace nécessaire au stockage de 100 `int`, c'est à dire de 100*`sizeof(int)` octets,
- elle crée une variable supplémentaire (ici `tab`) de type `int*` qu'elle initialise par l'adresse de l'espace mémoire réservé.

L'espace mémoire réservé n'est pas initialisé. Autrement dit, immédiatement après que le tableau a été déclaré, les variables `tab[i]` ont une valeur qui dépend du contenu de la mémoire à l'endroit qui leur a été dévolu au moment où la réservation à été faite. Soulignons quelques particularités de cette méthode de réservation mémoire :

- La taille de l'espace mémoire alloué (ici 100*`sizeof(int)`) doit être connu à la compilation : il ne peut donc pas dépendre d'un paramètre passé en ligne de commande ou d'un calcul effectué au cours du programme.
- L'espace mémoire ainsi alloué sera libéré dès la fin de l'exécution de la fonction au cours de laquelle la réservation a eu lieu.
- L'espace mémoire est réservé dans une partie de la mémoire allouée au processus que l'on appelle sa *pile*. La pile est généralement de taille limitée et ne représente qu'une petite partie de la mémoire disponible sur la machine.

**Exercice 5.3 ★** Vérifiez expérimentalement :

a. ce qui se passe sur votre machine/système lorsque la pile déborde : écrivez un programme avec des variables locales de grande taille ;

b. l'ordre de grandeur de la taille disponible sur la pile de votre programme : écrivez un programme qui stocke de plus en plus de données dans la pile jusqu'à ce qu'elle déborde ;

c. que la pile grandit vers le bas : écrivez un programme qui alloue de plus en plus de variable locales et affiche leurs adresses.

## 5.5 Tableaux multidimensionnels

La mémoire RAM n'est pas structurée à priori : c'est une longue séquence de *cases*. Lorsque l'on veut ranger une donnée dont la structure est plus complexe qu'une suite d'octets, il faut *choisir* comment *sérialiser* cette donnée. Considérons par exemple la matrice d'octets suivante, avec deux lignes et trois colonnes :

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}.$$

Il y a plusieurs manières répandues de stocker une telle matrice en mémoire, qui différent selon deux critères :

a. est-ce un tableau de deux lignes ou un tableau de trois colonnes ?

b. chacune des lignes (ou des colonnes) est-elle stockée à la suite de la précédente ou dans un bloc de mémoire qui lui est propre.

Ci-dessous deux programmes qui illustrent deux représentations possibles.

```
uint8_t matrice[2][3] =
  { 1, 2, 3, 4, 5, 6 };

void matriceEnLigne(void) {
  for (int i = 0; i < 2; ++i) {
    for (int j = 0; j < 3; ++j) {
      printf("%3d", matrice[i][j]);
    }
    printf("\n");
  }
}
```

```
uint8_t c0[2] = { 1, 4 };
uint8_t c1[2] = { 2, 5 };
uint8_t c2[2] = { 3, 6 };

uint8_t *mat[3] = { c0, c1, c2 };

void pointeursVersColonnes(void) {
  for (int i = 0; i < 2; ++i) {
    for (int j = 0; j < 3; ++j) {
      printf("%3d", mat[j][i]);
    }
    printf("\n");
  }
}
```

**Exercice 5.4 ★**

- Étudier et comparer l'assembleur correspondant aux deux programmes ci-dessus.
- Dans chaque cas, comment obtenir la valeur de la deuxième case de la troisième colonne (à partir de l'adresse de la matrice) ?
- Quels peuvent être les avantages de chacune des représentations ?

## 5.6 Le tas (« heap »)

Il est fréquent qu'un programme ait besoin de réserver un espace mémoire un peu conséquent, par exemple pour y stocker des données sur lesquelles il doit travailler. Le langage C offre deux manières classiques de faire cela.

On peut réserver de l'espace mémoire via le mécanisme d'allocation dynamique du C au moyen de la fonction `malloc` de la bibliothèque `<stdlib.h>`. Le code qui réalise la même tâche que l'allocation en pile ci-dessus est :

```c
#include <stdlib.h>

int*  tab;
tab = malloc(100 * sizeof(int));
```

L'espace mémoire réservé n'est pas non plus initialisé. Soulignons ici aussi quelques particularités de cette méthode de réservation mémoire :

- La taille de l'espace mémoire alloué (ici `100 * sizeof(int)`) n'a besoin d'être connu qu'au moment de l'exécution de cette commande. Autrement dit, ce peut être une variable dont la valeur résulte d'un calcul fait dans le programme ou d'un paramètre passé en ligne de commande.
- L'espace mémoire ainsi alloué n'est libéré qu'au moyen de la commande `free` (ou à la terminaison du processus).
- L'espace mémoire est réservé dans une partie de la mémoire allouée au processus que l'on appelle son *tas*. Le tas est généralement de taille comparable à la mémoire disponible sur la machine.

> Dorénavant, toute réservation mémoire d'un tableau « un peu grand » doit être faite sur le tas.

# Chapitre 6

# Assembleur : adressage mémoire

**Arguments indirects.** On peut donner comme argument "le contenu de l'adresse mémoire `adr`" au moyen de `[adr]`. Autrement dit, un argument entre crochets renvoie au contenu de l'adresse mémoire en question. Par exemple

| | |
|---|---|
| `mov [rax], rbx` | copie la valeur de `rbx` en mémoire à l'adresse `rax` |
| `add rax, [rbx]` | ajoute la valeur contenue en mémoire à l'adresse `rbx` à la valeur de `rax` |

Ainsi, un registre entre crochets [ ] est considéré comme un pointeur. On peut mettre entre crochets une adresse *immédiate*, c'est-à-dire constante. Par exemple :

| | |
|---|---|
| `mov rax, [0x4300]` | copie la valeur en mémoire à l'adresse `0x4300` dans `rax` |

L'adresse mémoire indiquée entre [ ] peut inclure certains calculs, par exemple :

| | |
|---|---|
| `mov rax, [rbx-8]` | copie dans `rax` la valeur en mémoire à l'adresse `rbx-8` |
| `mov rax, [rbx+rcx]` | copie dans `rax` la valeur en mémoire à l'adresse `rbx+rcx` |

Le résultat du calcul fait entre [ ] n'est *pas* gardé. Ainsi, dans les deux exemples ci-dessus, les registres `rbx` et `rcx` restent inchangés.

Comme les registres ne sont pas typés, le sens de `[rax]` peut être ambigu : s'agit-il de l'octet, du mot 16 bits, du mot 32 bits ou du mot 64 bits contenu à l'adresse `rax` ? Dans les deux exemples ci-dessus, on peut répondre à cette question par inférence : puisque l'autre opérande est un registre 64 bits, le contenu mémoire est considéré comme une valeur 64 bits. En revanche, l'instruction `mov [rax],[rbx]` pose problème : combien d'octets souhaite-t-on copier de l'adresse `rbx` vers l'adresse `rax` ? On peut toujours (et on doit parfois) expliciter la taille souhaitée au moyen de *directives de taille* `BYTE PTR, WORD PTR, DWORD PTR, QWORD PTR`, qui correspondent respectivement à un pointeur sur un octet (*byte*=octet), sur un mot 16 bits (*word*), sur un mot 32 bits (*double word*) et sur un mot 64 bits (*quadruple word*).

**Sauvegarde de registres sur la pile.** Le processeur dispose d'une *pile*. Elle fonctionne en LIFO (*last in, first out*). On peut y ajouter le contenu d'un registre avec l'instruction `push`. On peut en retirer la dernière valeur ajoutée avec l'instruction `pop`. En pratique, la pile est une zone de la mémoire centrale pointée par le registre `rsp`.

| | |
|---|---|
| push rbx | décrémente rsp puis copie le contenu du registre rbx au sommet de la pile |
| pop rbx | copie la valeur au sommet de la pile dans le registre rbx puis incrémente rsp |

**Modes d'adressage** Le jeu d'instructions x86 propose de nombreux *modes d'adressage*, c'est-à-dire de moyens de décrire une adresse. La table 6.1 en présente une partie. Une adresse peut être décrite par un décalage fixe par rapport à l'instruction courante (dont l'adresse est dans le registre rip) ou calculée à partir d'un registre de *base*, d'un registre d'*index* éventuellement multiplié par une petite puissance de deux (1, 2, 4 ou 8) et d'un décalage constant.

TABLE 6.1 – Diverses façons d'exprimer une adresse

| | **Syntaxe Intel** | **Syntaxe AT&T** |
|---|---|---|
| Relatif au point de programme | [rip + nom + 5] | nom + 5(%rip) |
| Base + offset | [rcx + 1] | 1(%rcx) |
| Base + index | [rcx + rdi]<br>[rcx + rdi * 8] | (%rcx,%rdi)<br>(%rcx,%rdi,8) |
| Base + index + offset | [rcx + rdi + 5]<br>[rcx + rdi * 2 + 5] | 5(%rcx,%rdi)<br>5(%rcx,%rdi,2) |

Un calcul d'adresse est une opération arithmétique non triviale (base + index × pas + décalage) qui peut être utilise indépendamment d'un accès mémoire.

L'instruction lea permet de réaliser un calcul d'adresse sans effectuer d'accès à la mémoire. On peut s'en servir par exemple pour additionner le contenu de deux registres.

**Exercice 6.1 ★★** Donner un cas d'usage pour chacun des modes d'adressage présentés ci-dessus.

# Chapitre 7

# Méthodologie de chronométrage

Nous donnons ici une méthodologie sommaire de mesure des performances d'un programme. Il ne s'agit pas d'apprendre à *profiler* un projet logiciel mais d'examiner les performances ou contre-performances de quelques lignes de code C. Cela nous permettra, au fil des séances suivantes, de chronométrer de petits programmes pour mettre en évidence des accélérations ou des ralentissements liés à l'architecture matérielle.

**Objectifs.** À l'issue de cette séance, il est attendu que vous sachiez chronométrer de manière précise une portion de code C.

## 7.1 Problématique

Lorsque l'on s'intéresse au temps d'exécution d'un programme, il convient de distinguer le *temps écoulé* du *temps de processeur consommé*. Le premier correspond au temps mesuré, par exemple avec notre montre, entre le début et la fin du programme. Le second correspond au temps de processeur consacré à son exécution. Sur une machine comportant un seul processeur qui exécute une seul tâche, comme par exemple une calculatrice programmable, on peut s'attendre à ces deux notions soient proches. Sur une machine multiprocesseur au système multi-tâches, ces deux notions peuvent être très différentes.

Notre but va être de mesurer le temps de processeur consommé par l'exécution d'un morceau de programme afin de tester une hypothèse ou de comparer les performances de deux codes a priori similaire. Par exemple, on peut vouloir tester l'hypothèse que *le temps de processeur pris par l'exécution du programme*

```
for (uint64_t i = 0; i < max; ++i)
    a = a * a;
```

*est proportionnel à la valeur de max*. On peut aussi vouloir comparer les temps pris pour parcourir un tableau $n \times n$ ligne par ligne et colonne par colonne :

```
int dum = 0;
for (int i=0; i<n; i++)
  for (int j=0; j<n; j++)
    dum += tab[i*n+j];
```

```
int dum = 0;
for (int i=0; i<n; i++)
  for (int j=0; j<n; j++)
    dum += tab[j*n+i];
```

## 7.2 L'outil : `rdtscp`

Les processeurs `x86` maintiennent un compteur appelé *time stamp counter* (`tsc`) qui compte le nombre de cycles d'horloge depuis la dernière remise à zéro, sous la forme d'un entier non-signé 64 bits. Il est possible de lire la valeur de ce compteur au moyen de la fonction C `__rdtscp()` de la bibliothèque `x86intrin.h`. [1]

1. Cette fonction exécute l'instruction assembleur `rdtscp` qui lit le `tsc`.

`__rdtscp()` s'appelle avec en argument un pointeur sur un `unsigned int`. La fonction (i) renvoie la valeur du `tsc` au format `uint64_t`, et (ii) écrit dans l'entier passé en argument l'identifiant du cœur qui exécute le programme.

On va chronométrer un code en mesurant la différence entre les valeurs du `tsc` avant et après son exécution. Par exemple, la fonction `test` suivante retourne le nombre de cycles d'horloge écoulés entre le début et la fin de la boucle `for`.

```
#include <x86intrin.h>
#include <stdint.h>

uint64_t test(uint64_t* result, uint64_t max){
    uint64_t tic, toc;
    uint64_t a = 2;
    unsigned int _pid;

    tic = __rdtscp(&_pid);
    for (uint64_t i = 0; i < max; ++i) {
        a = a * a;
    }
    toc = __rdtscp(&_pid);
    result[0] = a;
    return toc - tic;
}
```

Deux précisions :

a. Les compilateurs sont prompts à éliminer le code "mort", c'est-à-dire les calculs dont le résultat n'est pas utilisé. Aussi, la fonction `test` ci-dessus écrit le résultat du calcul dans le tableau `result` reçu en argument.

b. Le résultat de `__rdtscp()` est dans l'unité "nombre de cycles d'horloges". Il peut être tentant de vouloir convertir cela en quelque chose de plus familier, par exemple des millisecondes. C'est généralement inutile pour ce que l'on souhaite faire. Par exemple, pour tester l'hypothèse que le temps pris par la boucle ci-dessus est proportionnel à la valeur de `max`, l'unité "nombre de cycles d'horloge" convient tout à fait.

## 7.3 Limitations et biais

Bien que la résolution du `tsc` soit très fine, les mesures qu'il permet peuvent être bruitées pour plusieurs raisons

- Les conditions d'utilisation du processeur peuvent varier d'une exécution à l'autre, avec par exemple le réveil de certaines tâches de fond qui viennent occuper le cœur sur lequel se font les mesures.
- Il est courant que la fréquence du processeur s'adapte à la charge de travail de manière à réduire la consommation énergétique en l'absence de gros calculs. Les performances du processeur au fil de l'exécution d'une tâche peuvent donc varier. Pour plus d'information, regardez du côté des commandes linux `cpufreq` et `cpupower`.
- Certains éléments de l'architecture s'adaptent à l'exécution d'un programme et en accélèrent l'exécution au fil des répétitions.

Nous allons effectuer toutes nos mesures avec les conventions suivantes :

a. On appellera la fonction à chronométrer plusieurs fois *avant* de commencer à mesurer. Cet "amorçage" assure que les mesures ultérieures seront moins affectée par des changements tels que pré-chargement du cache ou changement de mode du CPU.

b. Une fois l'amorçage effectué, on chronométrera *de nombreuses exécutions* et estimera le temps d'exécution à partir de ces nombreuses estimations. Dans cette situation, le temps minimal peut être une bonne estimation. Le temps médian également (mais un peu plus complexe à calculer que le minimum). Le temps moyen est une mauvaise estimation.

## 7.4 Commentaires

Signalons que d'autres outils de chronométrage sont disponibles.

- Sous linux, une première option est la commande `time`. Exécuter `time prog` lance l'exécution de `prog` et affiche en fin d'exécution le temps écoulé et le temps CPU. C'est simple mais limité : on ne peut mesurer qu'un programme *complet*.
- La bibliothèque C `time.h` fournit diverses fonctions qui relèvent "l'horloge courante". Elles fonctionnent sur le même modèle que `__rdtscp()` mais sont bien moins précises.
- Sous GNU/Linux, l'outil `perf` (ou la suite LIKWID [2] d'outils de mesure de performances) opère de la même manière que la commande `time`. Il a l'inconvénient de mesurer l'ensemble du programme, mais fournit un diagnostic assez complet, avec notamment la lecture de plusieurs compteurs du processeur.

Les plus motivé·es pourront affiner leurs chronométrages en apprenant à se servir des commandes linux telles que `cpupower`, `taskset`, `setarch`, ... Le *white paper* écrit par Gabriele Paoloni (ingénieur Intel) *How to Benchmark Code Execution Times on Intel ® IA-32 and IA-64 Instruction Set Architectures*, disponible sur Arche, explique comment diminuer des erreurs de mesures liées au fonctionnement du processeur. Nous en recommandons la lecture attentive après la séance 4 (pipeline).

## 7.5 Exercices

**Exercice 7.1 ★★** Écrivez une fonction qui calcule la somme des carrés des entiers de 1 jusqu'à $n$, où $n$ est un paramètre. Chronométrez-la pour différentes valeurs de $n$ :

| $n$ | 1 000 | 10 000 | $10^6$ | $10^7$ | $10^8$ |
|---|---|---|---|---|---|
| mesure | | | | | |

**Exercice 7.2 ★★** On va maintenant écrire une fonction `print_timing` qui prend en argument une *autre fonction* et qui mesure son temps d'exécution. Supposons que notre fonction à tester soit

```
1 void test(int a){ /* ... */ }
```

On la transmet en argument à `print_timing` en utilisant un *pointeur de fonction* :

```
1 void print_timing(int mon_arg, void (*ma_fonction)(int)) {
2    // votre code peut appeler la fonction à mesurer
3    // par ma_fonction(mon_arg)
4    // il doit afficher (printf) le résultat de la mesure
5 }
```

2. https://hpc.fau.de/research/tools/likwid/

Ainsi, pour effectuer une mesure de la fonction appelée avec le paramètre $a = 12$ il suffira d'appeler la commande `print_timing(12,test)`. L'idée est d'écrire une fois pour toutes un code de chronométrage qui prend en compte les phénomènes de mise en route et qui moyenne sur plusieurs exécutions.

> Si le type de la fonction à mesurer change, il faudra adapter la fonction `print_timing` en conséquence, mais la modification sera légère.

a. Écrivez une fonction `print_timing` selon le modèle ci-dessus et qui prend en compte les deux sources d'erreurs identifiées en cours (amorçage et moyenne). Testez cette fonction.

b. Si ce n'est pas déjà fait, ajoutez en paramètre à `print_timing` le nombre d'appels souhaités à l'amorçage et le nombre d'appel sur lequel se fait le moyennage.

> À partir de maintenant, et pour toutes les séances à venir, tous les chronométrages doivent être faits par `print_timing`.

# Chapitre 8

# Exercices complémentaires

**Exercice 8.1** ★ Un programme C reçoit la ligne de commande qui a servi à l'invoquer via les deux premiers arguments de sa fonction principale : `argc` et `argv`. Chaque élément du tableau `argv` est un pointeur vers une séquence d'octets non nuls délimitée par un octet nul. La fonction **strlen** donne la longueur d'une telle séquence.

a. Écrivez un programme qui affiche l'adresse et la longueur des éléments de `argv`.

b. Exécutez ce programme avec trois arguments « a », « bc » et « d ».

c. Faites un dessin qui schématise l'organisation de ce tableau et de ses éléments lors de cette exécution.

**Exercice 8.2** ★★ Exécutez (séparément) chacun des code suivants (qui supposent chargées les bibliothèques `stdio.h` et `string.h`) et expliquez ce qu'ils font.

```c
unsigned char c = 150;
signed char d;

memcpy(&d, &c, 1);
printf("%hhi\n", d);
```

```c
unsigned char c = 150;
unsigned char* p;
signed char* q;

p = &c;
memcpy(&q, &p, sizeof(p));
printf("%hhu\n", *p);
printf("%hhi\n", *q);
```

Les deux derniers exercices ont pour objectifs de pratiquer le chronométrage de code et de mettre en évidence des phénomènes que l'on expliquera à la séance prochaine. Pour les exercices qui suivent, déclarez un tableau `TAB` de `TAILLE` entiers `int`. Lorsque la valeur de `TAILLE` n'est pas précisée, on la prendra $\approx 10^9$.

**Exercice 8.3** ★★ On va maintenant faire `n` accès à `TAB` et comparer les temps pris quand ces accès sont consécutifs et quand ils sont aléatoires.

a. Écrire deux fonctions qui prennent un entier `n` en paramètre et accèdent en écriture à `n` cases de `TAB` : `acces_seq` qui parcoure les cases dans l'ordre (accès séquentiel) et `acces_alea` qui parcoure les cases dans un ordre aléatoire. Mesurer les temps pris par ces deux fonctions :

| `n` = | 1000 | $10^6$ | $10^7$ | $10^8$ | $10^9$ |
|---|---|---|---|---|---|
| Séquentiel | | | | | |
| Aléatoire | | | | | |

b. Qu'observe-t-on ? Proposez une explication argumentée.

c. On va reproduire cette expérience de manière à ne chronométrer que les accès mémoire. Pour cela, on prépare un tableau auxiliaire `aux` de `n` cases, que l'on initialise aux indices des cases auxquelles on souhaite accéder. On effectue ensuite une série de lecture aux cases `TAB[aux[0]]`, `TAB[aux[1]]`, ... Dans une première expérience, on remplit `aux` par des valeurs qui se suivent. Dans une seconde expérience, on le remplit par des valeurs aléatoires. On ne chronomètre que les accès en lecture, en excluant les temps de préparation de `aux`.

| `n` = | 1000 | $10^6$ | $10^7$ | $10^8$ | $10^9$ |
|---|---|---|---|---|---|
| Séquentiel | | | | | |
| Aléatoire | | | | | |

Qu'observe-t-on ?

**Exercice 8.4 ★★** On se propose de mesurer l'effet de *l'espacement* entre accès mémoires. Écrivez une fonction prenant en argument un entier `pas` et réalisant `N` $\approx 10^7$ accès dans `TAB` en espaçant les accès d'un pas fixe : on commence à la case d'indice 0 et l'accès à la case `i` est suivi de l'accès à la case `i+pas`. Dans le cas où vous devriez sortir du tableau, repartez à partir du début (ie. l'indice est calculé modulo `TAILLE`). Mesurez les temps d'exécution de cette fonction pour les valeurs de `pas` suivantes.

| `pas` | 1 | 2 | 3 | 4 | 8 | 16 | 32 |
|---|---|---|---|---|---|---|---|
| mesure | | | | | | | |

Qu'observe-t-on ?

Faites des mesures plus systématiques et tracez la courbe du temps pris en fonction du `pas`, pour $1 \leq$ `pas` $\leq 1000$.

# Chapitre 9

# Hiérarchie mémoire

La quantité de mémoire disponible et la puissance de calcul des processeurs augmente régulièrement, mais la capacité de transfert des données entre mémoire et processeur évolue plus lentement. Ce point est identifié depuis **plusieurs décennies** comme un des facteurs critiques en architecture des ordinateurs. Les systèmes actuels tentent de limiter cela au moyen de *systèmes mémoires hiérarchiques*, l'objet de cette séance.

Soulignons qu'il ne s'agit pas ici de jeter aux orties le modèle de mémoire virtuelle, qui décrit correctement la manière dont la mémoire se présente au programmeur (un grand tableau d'octet, où l'on peut accéder à un octet simplement à partir de son adresse). Les chronométrages réalisés en fin de 2ème séance ont cependant mis en évidence que ce modèle de mémoire virtuelle, abstraction simplifiée de la manière dont le matériel réalise le stockage mémoire, ne permet ni de prédire, ni d'analyser le coût d'un accès mémoire. Cette séance vise donc à compléter cette vision, en examinant comment cette "interface utilisateur" est implémentée matériellement et l'impact de cette implémentation sur les performances de différents types d'accès mémoire.

**Objectifs.** À l'issue de cette séance, il est attendu que vous...

- Sachiez analyser une séquence d'accès mémoire sur un système dont on connaît les spécifications.
- Sachiez expliquer une contre-performance causée par une mauvaise utilisation de la hiérarchie mémoire.

## 9.1 Problématique

Il existe de nombreux supports mémoire, dont la fonction est de stocker puis restituer de l'information : RAM, disque dur (HDD, SSD), USB, CD ou DVD, mémoire cache, mais aussi bande magnétique, carte perforée, papier, ... Le tableau suivant donne les ordres de grandeurs de trois caractéristiques importantes pour quelques types de mémoire :

| | débit maximum | temps d'accès | taille typique |
|---|---|---|---|
| disque dur SSD | 25-450 Moct/s | 0.1 ms | $\sim$ To |
| disque dur HDD | 25 Moct/s | 3-12 ms | qq To |
| RAM | 10 Goct/s | 10-100 ns | $\sim$ Go |
| L2 cache | 200 Goct/s | $\sim$ 5 ns | $\sim$ Mo |
| L1 cache | 700 Goct/s | $\sim$ 1 ns | $\sim$ 100 Ko |
| registre | - | $\sim$ 0.1 ns | $\sim$ qq centaines d'octets |

Ces caractéristiques dépendent naturellement des technologies sous-jacentes. Cependant, il faut aussi noter que même à technologie fixée, il est en principe possible de garantir de meilleures performances pour une petite capacité mémoire que pour une grosse capacité. En effet, dans une petite mémoire,

l'information à moins de distance à parcourir (rien n'est instantané) et l'adressage est plus petit (donc consomme moins de transistors, etc.).

Les hiérarchies mémoire sont des systèmes combinant différents types de mémoire afin de garantir *à la fois* une grande capacité mémoire et un temps d'accès rapide.

## 9.2 Principe d'un cache

Une mémoire *cache*, ou *antémémoire*, est un système qui accélère les accès d'un utilisateur à un stockage de données selon le principe suivant :

- Le cache est constitué d'une mémoire plus rapide mais plus petite que le stockage, et il garde une copie d'une petite quantité des données du stockage.
- L'utilisateur adresse ses requêtes (en lecture ou en écriture) au cache et non pas au stockage.
- Si le cache dispose d'une copie des données concernées par la requête de l'utilisateur, il répond directement ; l'accès à la donnée par l'utilisateur est alors dite *en cache* (*cache-hit*).
- Sinon, le cache adresse une requête au stockage pour la donnée demandée par l'utilisateur. Une fois qu'il l'a reçue, il répond à la requête de l'utilisateur ; l'accès est dit *hors-cache* (*cache-miss*). Une fois cela fait, le cache garde une copie de la donnée. Si la mémoire du cache est pleine, le cache fait de la place en se débarrassant d'une donnée plus ancienne.

On trouve ce principe dans les navigateurs internet ou dans les bases de données distribuées. Soulignons deux points :

- Le fonctionnement du cache est *transparent* du point de vue de l'utilisateur, qui adresse simplement une requête au système {cache+stockage}. Le fait que la requête soit traitée par un intermédiaire (le cache) et non par le stockage n'affecte que le temps de réponse. Autrement dit, « l'interface utilisateur » reste celle du modèle de mémoire virtuelle.
- L'accès à une donnée n'est accélérée que pour les accès en cache, c'est à dire si la donnée est *déjà* en cache. Il est vraisemblable que l'ajout d'un intermédiaire ralentisse les accès hors-cache par rapport à un système sans cache.

## 9.3 Cache dans un ordinateur

Examinons maintenant les systèmes de cache dans un ordinateur. On se contente pour l'instant d'un système à deux niveaux : une mémoire rapide (petite) et une mémoire lente (grande). L'adresse d'une donnée correspond à son adresse *en mémoire lente*.

### 9.3.1 Blocs et ligne de cache

Les systèmes de cache pour les mémoires d'ordinateur ont une spécificité importante : les transferts entre mémoire lente et mémoire rapide ne se font pas octet par octet, mais par *blocs*. Tous les blocs ont la même taille ; ce paramètre, appelé *taille de ligne de cache*, est déterminé au niveau électronique par l'architecture. On le notera $b$.

La division par bloc est *fixe*. Ainsi, les octets d'adresses 0 à $b-1$ forment un premier bloc, ceux d'adresses de $b$ à $2b-1$ forment un second bloc, etc. Lorsque la mémoire rapide a besoin d'*un* octet, il récupère de la mémoire lente la totalité du bloc contenant cet octet. Ainsi, lors d'un accès hors-cache à un octet en début de bloc, les octets qui le suivent l'accompagnent en cache ; lorsque l'accès hors-cache porte sur un octet en fin de bloc, ce sont les octets qui le précèdent qui l'accompagnent en cache.

> Le transfert des données par blocs amortit le *coût de latence* d'un accès en mémoire lente sur plusieurs accès en mémoire rapide dès lors que des octets consécutifs en mémoire lente sont utilisés à peu de temps d'écart.

### 9.3.2 Adresse et numéro de bloc.

La mémoire lente est divisée en blocs, que l'on peut numéroter de 0 à $M/b - 1$, où $M$ est la taille mémoire totale et $b$ la taille d'un bloc. En pratique, $M$ et $b$ sont des puissances de 2.

Puisque les blocs sont formés d'octets consécutifs, et que le premier bloc commence à l'adresse 0, le numéro du bloc qui contient l'octet d'adresse `adr` est `adr`$/b$ (ici / est la division entière). Ainsi, deux octets d'adresses `adr1` et `adr2` sont dans un même bloc si et seulement si `adr1`$/b =$`adr2`$/b$. Ainsi, si $b = 2^k$ on obtient le numéro de bloc d'un octet en oubliant les $k$ derniers bits de l'écriture binaire de son adresse. Quand $k$ est un multiple de 4, cela revient à oublier les $k/4$ derniers chiffres de l'écriture hexadécimale de cette adresse.

Lorsqu'une donnée est composée de plusieurs octets, comme par exemple un `int` ou un `float` il est envisageable que ces octets se trouvent distribués dans plusieurs blocs. Dans ce cas, l'accès à la donnée déclenche un accès à *chacun* de ces blocs.

**Exercice 9.1** ★ Considérons deux variables qui ont pour adresses `0xF03E AAFD` et `0xF03E AAB4`.

a. Quelle taille mémoire occupe chacune de ces variables si elles sont de type `char` ?

b. Ces variables sont elles contenues dans un même bloc si la taille de bloc vaut $b = 64$ ?

c. Même question qu'au (b) si la taille de bloc vaut $b = 256$.

d. Refaire les questions (b) et (c) dans le cas où les variables sont des nombres flottant simple précision, c'est à dire occupant 4 octets chacune.

**Exercice 9.2** ★★ On a observé à l'exercice 8.4 du Chapitre 8 que le temps pris par des accès mémoire régulièrement espacés augmente lorsque ces accès ne sont pas consécutifs. Expliquons pourquoi ce phénomène *doit* se produire dans un système à un niveau de cache. Pour simplifier, on suppose que le cache est initialement vide.

On note $b$ la taille de bloc du système mémoire ; on suppose que $b$ est une puissance de 2 et que $b \geq 16$. On accède à $N$ éléments d'un tableau d'`int`, les indices de deux éléments accédés consécutivement étant séparés d'une valeur `pas`.

a. Combien d'`int` est-ce qu'un bloc de taille $b$ contient ?

b. Associons à cette séquence d'accès la séquence $s \in \{E, H\}^N$ telle que $s_i = E$ si le $i$ème accès est en cache, et $s_i = H$ sinon. Décrire la séquence $s$ en fonction de $b$ et `pas`.

## 9.4 Modèle de mémoire hiérarchique à associativité totale

Examinons un premier modèle de mémoire hiérarchique. Bien que simplifié, ce modèle permet de rendre compte de plusieurs phénomènes facilement observables dans les temps d'accès mémoire.

### 9.4.1 Stratégie de pagination

La mémoire rapide ne peut stocker qu'une petite partie des blocs que contient la mémoire lente. Aussi, assez vite, le chargement d'un nouveau bloc doit provoquer le déchargement d'un autre bloc. La manière de choisir quel bloc sera écrasé est spécifiée par la *stratégie de pagination*.

Une stratégie de pagination est dite à *associativité totale* si *tout* bloc de la mémoire lente peut être chargé dans tout emplacement de la mémoire rapide. C'est le cas que l'on examine ici, les stratégies à *associativité partielle* étant quant à elles abordées au Chapitre 16.

Dans le cadre de la pagination à associativité totale, un choix naturel consiste à décharger le bloc pour lequel le dernier accès est le moins récent parmi les blocs actuellement en cache. Ce choix est

désignée par l'acronyme LRU, pour *least recently used*. Il s'agit d'une heuristique naturelle si l'on fait l'hypothèse que les accès passés sont une bonne indication des accès futurs. Il s'agit aussi d'une solution qui offre des garanties théoriques intéressantes mais que l'on ne détaillera pas ici. [1]

### 9.4.2 Hiérarchie mémoire

Le système mémoire d'un ordinateur est formé d'une *hiérarchie* comportant plusieurs niveaux de cache. La hiérarchie d'une machine typique comporte 4 niveaux : 3 niveaux de mémoire cache (L1, L2, L3) et la mémoire RAM.

Le programme, et donc le programmeur, ne connaît une donnée mémorisée que par son adresse en RAM. Lorsqu'il souhaite y accéder, en lecture ou en écriture, il adresse une demande au cache L1 (le plus rapide et le plus petit). Si le cache L1 a cette donnée en mémoire, il répond directement (accès en cache L1), sinon, il la demande au cache L2 puis répond (accès hors-cache L1). Quand le cache L1 adresse une demande au cache L2 (un peu moins rapide mais un peu plus gros), celui ci répond directement s'il a la donnée en mémoire (accès en cache L2), sinon il la demande au cache L3 et répond (accès hors-cache L2). Idem pour L3 : il répond aux demandes adressées par L2 directement (accès en cache L3) ou après demande à la RAM (accès hors cache L3). Ainsi, l'accès par le programme à *un* octet de donné peut être :

- un accès en cache L1, ou
- un accès hors cache L1 et en cache L2, ou
- un accès hors caches L1 et L2 et en cache L3, ou
- un accès hors caches L1, L2 et L3.

Ces situations occasionnent naturellement des temps d'attente de plus en plus long.

Chaque paire de niveaux consécutifs de la hiérarchie (L1-L2, L2-L3 et L3-RAM) fonctionne sur le modèle à deux niveaux décrits précédemment. Les transferts entre la mémoire lente et la mémoire rapide se font par blocs, et la mémoire rapide (plus petite) doit gérer son ensemble de blocs mémorisés au moyen d'une stratégie de pagination.

Notons que sur les architectures récentes :

- Le niveau L1 comporte en général deux cache indépendants, un pour les instructions et l'autre pour les données.
- Le cache L1 d'instructions est utilisé par le CPU pour des accès mémoire correspondant à la lecture du programme à exécuter.
- Les caches L1 et L2 sont internes à chaque cœur du processeur, comme on peut le voir par exemple sur le diagramme de l'architecture skylake (figure 9.1).
- Le niveau L3 est partagé entre plusieurs cœurs.

**Exercice 9.3 ★** Déterminez les paramètres du système de cache de votre machine en examinant les informations qui se trouvent dans le répertoire

`/sys/devices/system/cpu/cpu0/cache`

et renseignez le tableau suivant :

| | L1 instruction | L1 données | L2 | L3 |
|---|---|---|---|---|
| Taille mémoire | | | | |
| Ligne de cache | | | | |

1. Le théorème de Sleator-Tarjan sur l'analyse amortie de la stratégie MOVE-TO-FRONT dans les listes auto-organisatrices a pour conséquence que pour toute séquence d'accès, LRU avec $m$ blocs produit au plus 2 fois plus d'accès hors-cache que n'en produirait la meilleure stratégie *pour cette séquence* avec $m/2$ blocs. En particulier, cela signifie que connaître la séquence d'accès en avance et consacrer d'importantes ressources à optimiser la stratégie de pagination pour *cette* séquence ne permet pas de gagner mieux qu'un "facteur 4" (i.e. diviser par 2 le nombre d'accès hors-cache et diviser par 2 la mémoire cache utilisée).

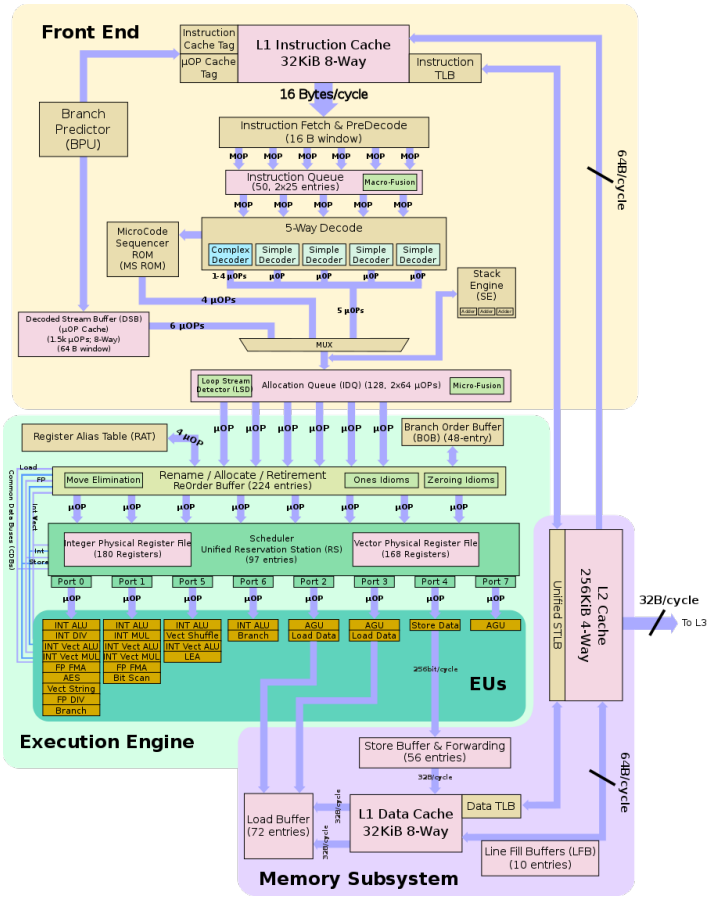



FIGURE 9.1 – Diagramme d'un cœur dans l'architecture Intel Skylake. Source : https://en.wikichip.org/wiki/WikiChip.

# Chapitre 10

# Premier jalon (ingéniérie algorithmique) : manipulation de grandes matrices

Ce premier jalon étudie l'implantation efficace d'algorithmes pour réaliser la transposition et la multiplication de grandes matrices. Dans tout ce jalon, les entrées des matrices considérées sont des nombres à virgule flottante de type `double`.

## 10.1 Échauffement

**Exercice 10.1** ★★ Comparons expérimentalement le temps mis pour parcourir un tableau 2D par ligne au temps mis pour le parcourir par colonnes. On "simule" ce tableau 2D par un tableau 1D déclaré par

```
uint64_t *tab = malloc(N*N*sizeof(uint64_t));
```

et l'élément à la $i$ème ligne et $j$ème colonne est `tab[i*N+j]`. Il s'agit par exemple de comparer les temps pris par les codes suivants :

```
uint64_t dum = 0;
for (int i=0; i<N; i++)
  for (int j=0; j<N; j++)
    dum += tab[i*N+j];
return dum;
```

```
uint64_t dum = 0;
for (int i=0; i<N; i++)
  for (int j=0; j<N; j++)
    dum += tab[j*N+i];
return dum;
```

Mesurez les temps d'exécution de ces deux fonctions pour $N$ allant de 100 à 2500 par palliers de 50. Tracez-en les graphes.

## 10.2 Structure : bloc d'une matrice

Un *bloc* $S$ d'une matrice $M$ est une matrice que l'on peut obtenir de $M$ en supprimant tout sauf certaines lignes consecutives et certaines colonnes consécutives. Ainsi

$$\begin{pmatrix} 1 & 2 & 3 \\ 5 & 6 & 7 \end{pmatrix} \quad \text{et} \quad \begin{pmatrix} 2 & 3 \\ 6 & 7 \\ 10 & 11 \\ 14 & 15 \end{pmatrix}, \quad \text{sont des blocs de} \quad \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}, \quad \text{mais pas} \quad \begin{pmatrix} 1 & 3 & 4 \\ 5 & 7 & 8 \\ 9 & 11 & 12 \\ 13 & 15 & 16 \end{pmatrix}.$$

Il va être utile de manipuler les blocs d'une matrice. On crée donc une structure dédiée :

```
struct bloc{
  double* adr;
  uint64_t haut,larg,saut;};
```

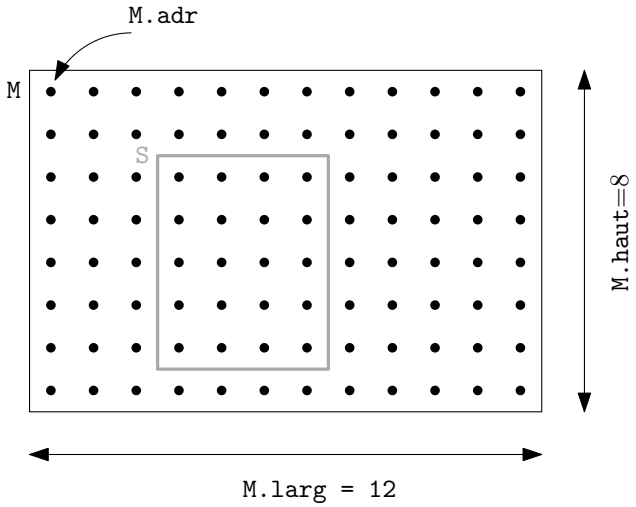
Le champs `adr` contient l'adresse de la première entrée du bloc. Les champs `haut` et `larg` contiennent, respectivement, le nombre de lignes et le nombre de colonnes du bloc. Le champs `saut` indique le nombre d'entrées à sauter entre la fin d'une ligne d'un bloc et le début de la ligne suivante. Dans ce jalon, les matrices seront traités comme des blocs : on déclare une matrice $M$ de `L` lignes et `C` colonnes de doubles par

```
double* mem = malloc(L*C*sizeof(double));
struct bloc M = {mem,L,C,0};
```

Pour extraire le bloc $S$ de $M$ de $a$ lignes et $b$ colonnes formé par l'intersection des lignes $y, y+1, \ldots, y+a-1$ et des colonnes $x, x+1, \ldots, x+b-1$ on procède comme suit :

```
struct bloc S = {mem+y*L+x,a,b,L-a};
```

Dans l'exemple ci-contre, `S.adr = M.adr+27`, `S.larg = 4`, `S.haut = 5` et `S.saut = 8`.




**Exercice 10.2 ★** Comment calculer, étant donné un `struct bloc S`, l'adresse à laquelle se trouve l'entrée en $i$ème ligne et $j$ème colonne du bloc $S$ ? Faites impérativement vérifier votre réponse par votre chargé de TD (une erreur ici compromet les réponses à tous les exercices qui suivent).

**Exercice 10.3 ★** Écrivez des fonctions réalisant les tâches suivantes :

a. `void reset(struct bloc B)` qui met à 0.0 toutes les entrées du bloc $B$.

b. `void init(struct bloc B)` qui fixe les entrées du bloc $B$ à 1.0, 2.0, 3.0, ...

c. `void affiche(struct bloc B)` qui visualise le bloc $B$ en affichant chaque entrée $x$ au moyen d'un `printf("%6.1f",x)`.

Vérifiez que le code ci-dessous produit bien le résultat annoncé.

```
double* m = malloc(40*40*sizeof(double));
struct bloc M = {m,40,40,0};
struct bloc S1 = {M.adr+208,4,4,36};
struct bloc S2 = {M.adr+167,5,5,35};
init(M);
reset(S1);
affiche(S2);
```

| | | | | |
|---|---|---|---|---|
| 167.0 | 168.0 | 169.0 | 170.0 | 171.0 |
| 207.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 247.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 287.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 327.0 | 0.0 | 0.0 | 0.0 | 0.0 |

## 10.3 Transposition

Commençons par examiner l'opération de transposition. Tout au long de ce jalon, lorsque l'on applique une transposition à une matrice (ou a un bloc), on écrit systématiquement le résultat dans une autre matrice (ou un autre bloc).

### Algorithme naïf

Considérons tout d'abord l'algorithme de transposition naïf suivant.

```
void transpose(struct bloc S, struct bloc D)
{
  uint64_t i, j;
  double* pS = S.adr;
  double* pD;

  for (i = 0; i < S.haut; ++i)
    {
      pD = D.adr+i;
      for (j = 0; j < S.larg; ++j)
        {
          *pD = *pS;
          pS += 1;
          pD += D.larg + D.saut;
        }
      pS += S.saut;
    }
}
```

**Exercice 10.4 ★★**

a. Analyser théoriquement les accès hors-cache lors de l'exécution de `transpose` sur un bloc de paramètres `{adr, n, n, 0}` dans le modèle à 2 niveaux de mémoire où lequel la mémoire rapide est à associativité totale et comporte $\ell$ lignes de cache, chacune de taille $w$.

b. Choisissez une machine pour vos expériences et mesurez le temps d'exécution de `transpose` pour une matrice carrée $n \times n$ pour $n$ allant de 100 à 2500 par palliers de 50. Tracez-en le graphe.

c. Rappelez les paramètres des caches L1 données, L2 et L3 de cette machine, et comparez vos résultats expérimentaux du (b) à la prédiction théorique du (a).

### Algorithme de transposition tuilée

On va maintenant décomposer la matrice à transposer en petits blocs carrés, que l'on appelle *tuiles*, et on réalise la transposition tuile par tuile.

**Exercice 10.5 ★** Écrivez une fonction `void transpose_tuile(struct bloc S, struct bloc D, uint64_t T)` qui écrit dans $D$ la transposée du bloc $S$, en procédant par tuiles de taille $T \times T$. Autrement dit

- la tuile `{S.adr,T,T,S.larg-T}` est transposé en la tuile`{D.adr,T,T,D.larg-T}`,
- la tuile `{S.adr+T,T,T,S.larg-T}` est transposé en la tuile `{D.adr+T*(D.larg+D.saut),T,T,D.larg-T}`,
- ...

On suppose que les dimensions de $S$ et $D$ sont compatibles et multiples de $T$. La transposition d'une tuile se fera par appel de la fonction `transpose`.

**Exercice 10.6 ★★**

a. Analyser théoriquement les accès hors-cache lors de l'exécution de `transpose_tuile` sur un bloc de paramètres `{adr, n, n, 0}` dans le modèle à 2 niveaux de mémoire où lequel la mémoire rapide est à associativité totale et comporte $\ell$ lignes de cache, chacune de taille $w$.

b. Choisissez une machine pour vos expériences et mesurez le temps d'exécution de `transpose_tuile` pour des tailles de tuile $T \in \{5, 25, 50\}$ et une matrice carrée $n \times n$, pour $n$ allant de 100 à 2500 par palliers de 50. Tracez-en le graphe.

c. Comparez vos résultats expérimentaux du (b) à la prédiction théorique du (a).

### Algorithme de transposition récursif

Prolonger l'idée du tuilage amène à un algorithme récursif de transposition... Plusieurs approches sont possibles, par exemple couper en deux la plus grande dimension ou couper en 4 blocs (en faisant attention à ce que les blocs diagonaux soient carrés !).

**Exercice 10.7 ★★★** Écrivez une fonction `void transpose_rec(struct bloc S, struct bloc D)` qui implémente un algorithme de transposition récursif et qui soit plus sensiblement rapide que la fonction `transpose_tuile` .

## 10.4 Multiplication de matrice

L'étude menée ci-dessus pour la transposition de matrice peut s'adapter à la multiplication de matrices, avec les ingrédients suivants :

- Quelque soit l'algorithme choisi, avant de calculer le produit d'une matrice `S1` par une matrice `S2`, il est souhaitable de transposer `S2` afin que les deux matrices soient parcourues "en lignes".
- On peut subdiviser les matrices en petites tuiles de tailles compatibles et multiplier les tuiles entre elles par l'algorithme naïf (triple boucle).
- Dans le cas de matrices rectangulaires, on peut subdiviser récursivement la plus grande des dimensions, et ce jusqu'à ce que toutes les dimensions soient suffisamment petites pour que l'algorithme naïf (triple boucle) tire bien parti du cache.
- On peut aussi tester expérimentalement les performances de l'algorithme de Strassen [1]...

1. https://en.wikipedia.org/wiki/Strassen_algorithm

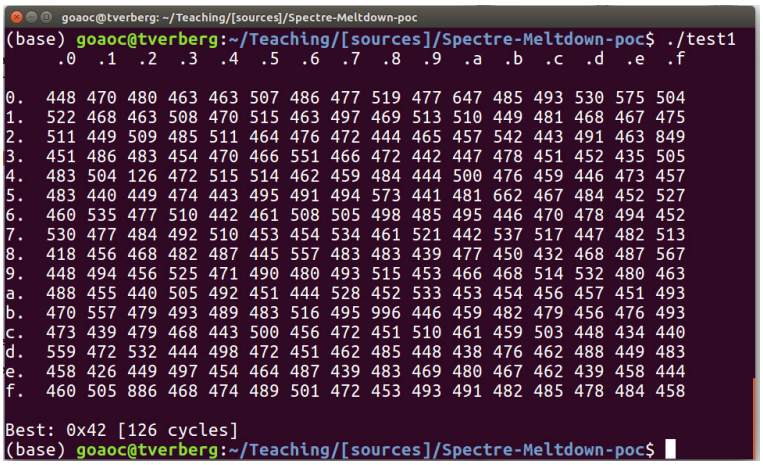
# Chapitre 11

# Premier jalon (sécurité) : chronométrage fin d'*un* accès mémoire

La première étape dans la réalisation d'un prototype de la faille de sécurité MELTDOWN consiste à pouvoir déterminer par chronométrage si *un* accès à une zone mémoire donnée— que nous appellerons `T` — a été réalisé en cache ou hors cache.

## 11.1 L'objectif

Nous allons mettre ici en place `T` et nous assurer d'une part que l'on sait chronométrer une lecture à une de ses cases et distinguer un accès en cache d'un accès hors cache et d'autre part que l'on arrive à distinguer 256 lignes de cache. L'objectif est d'aboutir à quelque chose ressemblant à ceci :




Les numéros de lignes de cache sont représentés en hexadécimal de `00` à `ff`), le premier chiffre correspondant à la ligne et le second à la colonne. Dans chaque case on indique le temps d'accès, normalisé en un entier entre 100 et 1000.

> **Exercice 11.1** ★ Dans l'exemple ci-dessus, exactement une ligne de cache a été activée. Laquelle ?

## 11.2 FLUSH+RELOAD

Nous ne nous intéresserons jamais au contenu des cases de `T` seulement à leur présence ou absence de la mémoire cache. Elles seront donc aussi petites que possible : un *byte*. Cependant, pour pouvoir distinguer les différentes cases, leurs adresses doivent être suffisamment éloignées de sorte qu'elles soient placées dans des lignes de cache différentes. On se donne donc une constante `PAS` telle que deux cases consécutives auront des adresses distantes de `PAS` bytes, et que la première case est en position `PAS`.

Pour s'assurer qu'une case de `T` n'est pas en cache, on peut utiliser l'instruction `CLFLUSH` qui admet comme opérande une adresse et, à l'exécution, vide du cache toute ligne qui contiendrait cette adresse. Dans un programme C, on pourra utiliser la fonction suivante :

```c
#include <emmintrin.h>
void _mm_clflush (void const* p);
```

Invalidate and flush the cache line that contains p from all levels of the cache hierarchy.

Pour laisser le temps à la machine de terminer le nettoyage du cache après l'exécution d'instructions `CLFLUSH`, on peut faire une petite pause ou plus simplement exécuter l'instruction `MFENCE`, accessible dans un programme C *via* la fonction :

```c
#include <emmintrin.h>
void _mm_mfence (void);
```

Mesurer le temps nécessaire pour lire une case de `T` requiert de lire cette case, ce qui a pour conséquence de la placer dans le cache. Si l'on commence à lire toutes les cases de `T` dans l'ordre, le microprocesseur peut spéculer que l'on va lire les autres et les placer en cache de manière optimiste. Pour prévenir cet excès de zèle, on accédera aux cases de `T` dans le désordre, par exemple en utilisant la permutation suivante des entiers modulo 256 : $i \mapsto i \times 167 + 13$.

**Exercice 11.2 ★★**

a. Définir une zone de mémoire `T` suffisamment grande pour contenir 256 cases bien espacées entre elles et bien séparées des autres données du programme, c'est-à-dire au moins $257 \times$ `PAS` bytes.

b. Initialiser tous les *bytes* de `T`.

c. Écrire une fonction `timings` qui mesure le temps d'accès à chacune des 256 cases de `T` et stocke les résultats dans un autre tableau. Pour garantir qu'une lecture mémoire soit bien exécutée, il peut être judicieux d'utiliser la valeur lue.

d. Écrire une fonction `flush` qui sort du cache toutes les cases de `T`.

e. Étudier la distribution des temps d'accès à toutes les cases de `T` après un appel à `flush` ; après un appel à `flush` suivi d'une lecture à une case de `T`.

f. Proposer un seuil qui permette de distinguer une lecture en cache d'une lecture hors cache.

# Chapitre 12

# Assembleur : contrôle de flot

Les langages auxquels vous êtes habitués (C, python, java) ont des instructions permettant de contrôler le *flot* du calcul, par exemple `if, for, while`... Ces instructions **n'ont pas d'équivalent** en assembleur, et sont réalisées par les moyens suivants.

**Adresses d'une instruction.** Pour qu'un code assembleur puisse être exécuté, il est d'abord chargé en mémoire sous la forme d'un code en langage machine. Chaque instruction est traduite par un ou plusieurs octets ; l'adresse du premier de ces octets est l'adresse de l'instruction. À tout moment, le CPU maintient un registre interne, `ip` (*instruction pointeur*) qui pointe sur l'instruction en cours d'exécution. Certaines instructions assembleur permettent de modifier la valeur d'`ip`, et d'agir ainsi sur le flot du programme.

**Sauts simple.** L'instruction `jmp` prends en argument une adresse, et provoque un saut (*jump*) dans le programme : l'exécution se poursuit par l'instruction située à l'adresse en question. Cela correspond à l'instruction `GOTO` que l'on trouve dans certains langages impératifs.

Lorsqu'un programme est exécuté, il est d'abord chargé par le système en mémoire. L'adresse à partir de laquelle ce chargement est effectué (et donc l'adresse de chaque instruction !) peut varier d'une exécution à l'autre. L'encodage de l'adresse d'un saut doit naturellement prendre cela en compte. Cela peut se faire par un encodage de l'adresse du saut *relativement* à l'adresse de l'instruction de saut ou par un mécanisme de relocation. Le code assembleur fourni par `gcc -S` comporte en outre des *labels* (colonne de gauche, suivis de ' :') qui repèrent l'adresse de l'instruction immédiatement suivante.

**Appel de sous-fonction.** L'instruction `call` prends en argument une adresse. Elle a deux effets : d'une part, elle sauvegarde sur la pile l'adresse de l'instruction qui la suit, puis elle effectue un `jmp` à l'adresse donnée en argument. Elle est associée à l'instruction `ret`, qui ne prends pas d'argument mais effectue un saut à l'adresse donnée par la valeur en tête de pile.

**Drapeaux et sauts conditionnels.** Le processeur maintient un registre spécial, `flags`, contenant différents "bits-drapeaux" : ZF (*zero flag*), CF (*carry flag*), OF (*overflow flag*), SF (*sign flag*), PF (*parity flag*)...Chaque instruction modifie tout ou partie des drapeaux. Par exemple,

| | |
|---|---|
| `add rax, rbx` | mets ZF à 1 si à la fin, `rax` vaut 0, et à 0 sinon<br>mets CF à 1 si un dépassement de capacité s'est produit en non-signé<br>mets OF à 1 si un dépassement de capacité s'est produit en signé<br>... |

Les drapeaux sont utilisés indirectement au travers de *sauts conditionnels*, par exemple :

| | |
|---|---|
| `jz 0x4300` | effectue un `jmp 0x4300` si ZF=1, ne fait rien si ZF=0 |

Il existe différentes instructions de saut conditionnel, chacune associée à une condition sur un ou plusieurs drapeaux.

**Exercice 12.1** ★★ Examinons le code assembleur produit par la compilation sans optimisation (option -O0 au lieu de -Os) du programme suivant :

```
1  uint32_t somme(uint32_t taille) {
2    uint32_t resultat = 0;
3    uint32_t i;
4    for (i=0 ; i < taille; ++i) {
5      resultat += i;
6    }
7    return resultat;
8  }
9
10 int main(int argc, char **argv) {
11   uint32_t count = 100 + argc;
12   uint32_t s = 0;
13   s = somme(count);
14   return s;
15 }
```

a. Donnez les instructions assembleur qui correspondent à la boucle `for` (lignes 4-5), en distinguant l'initialisation, le test et l'incrément.
b. Comment se fait l'appel à la fonction `somme` ?
c. Comment le paramètre (`count`) est-il transmis ?
d. Comment `somme` transmet-elle sa valeur de retour ?

**Quelques instructions de plus.** On liste ci-dessous les principales instructions que l'on utilisera. Nous ne détaillons pas ici le fonctionnement de chacune, et renvoyons pour cela aux ressources listées en début de Section 3. Notons par ailleurs que toutes les mnémoniques n'admettent pas tous les types d'arguments (registre, ímmédiat, indirect, indirect avec décalage, ...). À nouveau, nous renvoyons aux ressources de référence pour la liste de ce qui est possible.

| | |
|---|---|
| `call, ret, jmp` | saut/retour |
| `jz, jnz, jc, jnc`<br>`jg, jng, jge, jnge`<br>`...` | sauts conditionnels, voir https://www.felixcloutier.com/x86/jcc |

**Autres registres.** D'autres registres existent : certains servent au fonctionnement du CPU (`ip`, `flags`, voir plus loin), d'autre servent au FPU, d'autres aux instructions vectorielles, ...

À ce stade, pour pratiquer, il est conseillé d'écrire des petits morceaux de code en C et d'examiner de quelle manière ils sont traduits par le compilateur.

**Exercice 12.2** ★ Indiquer ce que vaut le registre `rax` lors de l'exécution de l'instruction `nop` pour chacun des codes suivants.

```
mov     rax,12
add     rax,rax
add     rax,rax
dec     rax
nop
```

```
        mov     rax,12
        mov     rcx,0
bcl:    add     rax,10
        inc     rcx
        cmp     rcx,10
        jle     bcl
        inc     rax
        nop
```

```
        mov     rax,12
        mov     rcx,13
lab:    add     rax,10
        dec     rcx
        jnz     lab
        inc     rax
        nop
```

**Exercice 12.3 ★** Indiquer ce que vaut le registre `rax` lors de l'exécution de l'instruction `nop` pour chacun des codes suivants.

```
        mov     rax,12
        call    fonc
        nop
        jmp     loin
fonc:   mov     rax,15
        ret
```

```
        mov     rax,12
        call    fonc
        nop
        jmp     loin
fonc:   push    rax
        mov     rax,15
        pop     rax
        ret
```

```
        mov     rax,12
        call    fonc
        nop
        jmp     loin
fonc:   push    rbx
        mov     rax,15
        pop     rax
        ret
```

# Chapitre 13

# Pipeline et prédiction de branchement

Cette 5ème séance introduit à la notion de pipeline dans les processeurs et le problème de la prédiction de branchement.

**Objectifs.** À l'issue de cette séance, il est attendu que vous sachiez...

- identifier les limites à l'accélération réalisée par un pipeline (les "bulles"),
- expliquer des comportements d'un code simple qui sont imputables au mécanisme de prédiction de branchement au moyen d'un modèle simple.

## 13.1 Problématique : un peu d'algorithmique

Supposons que l'on ait à calculer le minimum et le maximum d'un tableau de taille $n$. Sans hypothèse particulière sur le tableau, il est naturel de le parcourir séquentiellement en maintenant les petits et plus grands éléments vus. Comparons deux approches pour cette mise à jour :

```
1  void a(int* tab, int* l, int* u)
2  {
3    register int i,lo,up,v;
4    lo = tab[0];
5    up = tab[0];
6    for(i=1; i<taille; i++)
7      {
8        v=tab[i];
9        if (v<lo)
10                 lo=v;
11       if (v>up)
12                 up=v;
13     }
14   *l = lo;
15   *u = up;
16   return;
17 }
```

```
1  void b(int* tab, int* l, int* u)
2  {
3    register int i,lo,up,v1,v2;
4    lo = tab[0];
5    up = tab[0];
6    for(i=1; i<taille; i+=2)
7      {
8        v1=tab[i];
9        v2=tab[i+1];
10       if (v1 < v2)
11         { if (v1 < lo)
12             lo=v1;
13           if (v2 > up)
14             up=v2; }
15       else
16         { if (v2 < lo)
17             lo=v2;
18           if (v1 > up)
19             up=v1; }
20     }
21   *l = lo;
22   *u = up;
23   return;}
```

Remarquons que `a` fait $2n$ comparaisons là où `b` en fait $\frac{3}{2}n$. Cependant, sur certains systèmes et compilateurs, `a` est jusqu'à 5 fois plus rapide que `b`. Pour en comprendre la raison, il nous faut examiner le fonctionnement du *pipeline* du processeur.

## 13.2 Principe d'un pipeline

**Une nanoseconde dans la vie d'un CPU.** Une fois compilé, notre programme prends la forme d'une séquence d'octets en mémoire, à interpréter comme du langage machine. Lors de l'exécution, le registre `ip` contient l'adresse du premier octet de l'instruction courante. Supposons que la zone mémoire pointée par `ip` contienne

`01 c2 83 c0 01 3d 41 42 0f 00...`

Le traitement de l'instruction suivante nécessite plusieurs opérations distinctes :

- Il faut *décoder l'instruction*. Ici, `01 c2` code `add edx, eax` et correspond donc à la prochaine instruction à exécuter.
- Il faut *lire les opérandes*. Ici, il faut transmettre en entrée du circuit additionneur les valeurs $v_d$ et $v_a$ des registres `edx` et `eax`.
- Il faut *exécuter l'instruction*. Ici, il s'agit de calculer, via un circuit additionneur la valeur $v'_d = v_d + v_a \mod 2^{32}$ ainsi que les retenues signée et non signée.
- Il faut *écrire les résultats*. Ici, il faut mettre à jour le registre `edx` avec le résultat $v'_d$ et positionner les flags modifiés par l'instruction `add`.

Une fois cela fait, on peut ajouter `2` à `ip` pour le faire pointer sur l'instruction suivante. La zone mémoire pointée par `ip` contient donc

`83 c0 01 3d 41 42 0f 00...`

et on peut recommencer : décodage, lecture d'opérandes, exécution, écriture ...

**Principe d'un pipeline.** Les différentes étapes ci-dessus sont en général réalisées par des parties distinctes du processeur. Il n'est donc pas nécessaire d'attendre d'avoir terminé d'écrire les résultats de l'instruction en cours de traitement pour commencer à décoder l'instruction suivante.[1] C'est l'idée du *pipeline d'instruction* : le traitement d'une instruction est décomposé en une séquence d'étapes élémentaires réalisées par des parties indépendantes du processeur et ces parties sont mises à travailler à la chaîne. C'est une forme de « *Taylorisation* » du traitement des instructions par le processeur.

**Parallélisme et dépendance.** Un pipeline à $k$ niveaux peut, en régime permanent, traiter $k$ instructions à la fois. C'est une forme de *parallélisme*. Cette perspective optimiste est à tempérer car il est possible que des instructions qui se suivent soient dépendantes. Par exemple, dans

```
xor         eax,ebx
xor         ebx,eax
xor         eax,ebx
```

On ne peut pas lire les opérandes de la seconde instruction tant que le résultat de la première instruction n'a pas été écrit. Cette séquence d'instruction peut donc provoquer une *attente* dans le pipeline, que l'on appelle parfois aussi une « bulle ». De même, toute instruction de saut (`jmp, call, ret, jnz, jge...`) provoque une bulle puisque la lecture de l'instruction suivante doit attendre que soit connue l'adresse à laquelle se continue le programme.

## 13.3 Exemple de pipeline à 5 niveaux

Un modèle classique de pipeline simple (qui a existé dans le processeur `80 486`) comporte 5 niveaux :

- Instruction prefetch charge les instructions suivantes à exécuter dans un tampon interne au processeur. Sur une architecture ayant une ligne de cache $\ell$, il est naturel que le tampon soit de taille $2\ell$ et que le prefetch se fasse par tranche de $\ell$ octets.

1. De la même manière que dans le traitement du linge sale par lavage - séchage - repassage, on attend rarement d'avoir fini de repasser un premier paquet de linge pour lancer le lavage du second.

- STAGE 1 DECODE examine les premiers octets de l'instruction suivante (appelés *opcode* et *mod r/m*) pour déterminer l'instruction, les types des opérandes (registre, mémoire, immédiat) et leur taille (8, 16, 32 ou 64 bits).
- STAGE 2 DECODE récupère les opérandes immédiates et effectue les calculs d'adresses mémoires en cas de déplacement (ex : `[rbp - 8]`).
- EXECUTION réalise effectivement l'exécution de l'instruction : addition, opération logique, lecture en mémoire,
- REGISTER WRITE-BACK met à jour les registres suite à l'exécution de l'instruction (registre destination s'il y en a, `rsp` en cas de `push` ou `pop`, etc.), de la mémoire (si la destination est en mémoire) et le registre `flags` s'il est affectés.

Le nombre de cycles pris pour le traitement d'une instruction dans un niveau peut varier pour différentes raisons : lecture en/hors cache lors de l'instruction prefetch, le nombre d'octets d'opcode varie selon les instructions (chacun occupe le Stage 1 decode pour 1 cycle), etc.

## 13.4 Bulles

Les différents niveaux peuvent travailler en parallèle. Il arrive cependant qu'un niveau doive, pour traiter son instruction courante, attendre le résultat du traitement par un niveau *postérieur* d'une instruction *précédente*. Reprenons l'exemple :

```
1  0      31 D8          xor      eax,ebx
2  2      31 C3          xor      ebx,eax
3  4      31 D8          xor      eax,ebx
```

Ici, l'EXÉCUTION du second `xor` nécessite de connaître la valeur prise par `eax` suite au premier `xor`. Cette valeur n'est connue qu'après le WRITE BACK du premier `xor`. Il y a donc un temps d'attente, c'est à dire une *bulle* : le niveau EXECUTION attend sans rien faire que le niveau WRITE BACK termine sa tâche, et cette attente se répercute dans tout le pipeline en amont d'EXECUTION. Les instructions de saut, quant à elles, peuvent produire deux types de bulles :

- l'adresse de saut peut être déterminée après DECODE 2 pour les instructions comme `jmp` ou `call` sautant à des adresses immédiates.
- l'adresse de saut n'est déterminée qu'après EXECUTION pour des instructions comme `ret` pour lesquelles l'adresse de saut est lue en mémoire, ou les sauts conditionnels pour lesquels il faut évaluer la condition avant de savoir si on prends ou pas le saut.

**Réduction des bulles.** Il est parfois possible de permuter/modifier les instructions de manière à éliminer les bulles sans changer le résultat du calcul. Voici un exemple :

```
1  mov     eax, [rbp-4]
2  imul    eax, 6
3  mov     [rbp-4], eax
4  mov     ebx, [rbp-8]
5  add     ebx, 2
6  mov     [rbp-8], ebx
```

→

```
1  mov     eax, [rbp-4]
2  mov     ebx, [rbp-8]
3  imul    eax, 6
4  add     ebx, 2
5  mov     [rbp-4], eax
6  mov     [rbp-8], ebx
```

Cette idée est réalisée en cours d'exécution par le processeur et peut se faire à assez grande échelle : dans l'architecture `Skylake` (cf Figure 9.1), le *scheduler* et le *reorder buffer* peuvent examiner et réordonner jusqu'à 224 instructions. Un tel réordonnancement suppose que le processeur ait une assez bonne visibilité sur la séquence d'instructions à traiter. Cette visibilité est difficile à assurer en présence de sauts conditionnels.

## 13.5 Gestion des sauts conditionnels

Examinons maintenant le traitement des sauts conditionnels dans des pipelines généraux.

**Exécution spéculative.** Lors d'un saut conditionnel, on peut déterminer dès le décodage l'adresse à laquelle se fait le saut et on n'attends la fin de l'exécution que pour savoir *si il se fait*. La complexification des pipelines accroît la différence entre attendre la fin du décodage et attendre la fin de l'exécution. Dès 1993, les processeurs `intel` traitent chaque saut conditionnel comme suit :

- Lors du décodage, un *pari* est fait sur la valeur (vraie ou fausse) que prendra la condition de saut. Ce pari est calculé par un *prédicteur de branchement*.
- En attendant de connaître la valeur effectivement prise par cette condition, le saut est considéré comme *pris* ou *non pris* selon le pari. Cela détermine donc quelles instructions sont lues et décodées. Le traitement de ces exécutions est donc lancé *spéculativement*.
- Une fois la condition effectivement évaluée, on la compare au pari. En cas de pari gagné, l'exécution continue normalement. On a réussi à réduire la bulle. En cas de pari perdu, on vide le pipeline car il faut recommencer (lecture, décodage, etc.) à partir de l'instruction qui suit le saut conditionnel (dans la branche qu'il faut effectivement suivre).

Ainsi, en cas de pari réussi un saut conditionnel est géré comme un saut inconditionnel. La rapidité d'exécution du code dépends donc de la proportion de paris gagnés. On parle de *pénalité de prédiction erronée* (de l'ordre de 15-20 cycles sur les architectures de type Skylake).

**Prédicteur élémentaire.** Les spécifications effectives des prédicteurs de branchement sont peu documentées. Pour illustrer le type de méthodes mises en œuvre, décrivons le principe des *compteurs saturants*. Il s'agit d'associer à *chaque instruction de saut conditionnel* un compteur qui mémorise l'historique récent des branchements :

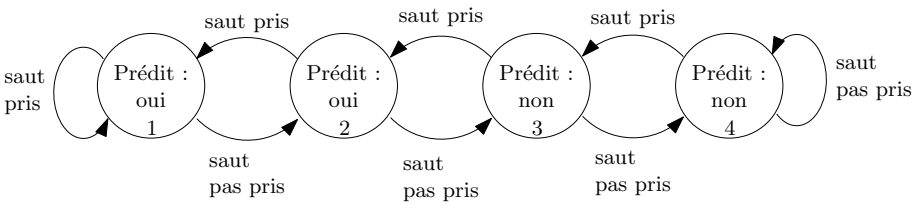
- À chaque fois que saut est pris, le compteur est incrémenté mais avec *saturation*, c'est à dire que s'il atteint sa valeur maximale, il garde cette valeur. Par exemple, incrémenter un compteur saturant sur 2 bits valant $(11)_2$, le laisse à $(11)_2$.
- À chaque fois que saut n'est pas pris, le compteur est décrémenté avec *saturation* : si le compteur valait zéro, la décrémentation le laisse à zéro.

Lorsque le processeur atteint à nouveau cette instruction de saut conditionnel, il peut utiliser le bit de poids fort de ce compteur comme un prédicteur du comportement : s'il est à 1 il parie que le saut sera pris, s'il est à 0 il parie que le saut ne sera pas pris.

**Le biais est une bonne nouvelle.** La prédiction de branchement est en fait de la détection de motifs dans l'exécution du programme. Des approches existent à base de compteurs saturants, de tables d'historiques, de réseaux de neurones, ...En première approximation, il est raisonnable d'escompter que leur efficacité sera d'autant plus grande que le branchement à prédire est *biaisé*. Réexaminer l'exemple initial par ce prisme devrait être éclairant.

### 13.5.1 Exercices

**Exercice 13.1 ★★** On s'intéresse dans cet exercice au comportement d'un processeur dont le pipeline utilise un prédicteur de branchement à 4 états.



a. On considère le programme suivant et sa traduction en assembleur :

```
1  int compte(int n, int* tab){
2    int s;
3
4    for (int i = 0; i < n; ++i)
5        if (tab[i]<50)
6            s += 1;
7    return s;
8  }
```

```
1  compte: mov     rdx, rsi
2          lea     ecx, [rdi-1]
3          lea     rsi, [rsi+4+rcx*4]
4  .L1:    cmp     DWORD PTR [rdx], 50
5          jge     .L2
6          inc     eax
7  .L2:    add     rdx, 4
8          cmp     rdx, rsi
9          jne     .L1
10         ret
```

Indiquez les numéros de ligne des instructions assembleur de saut conditionnel et, pour chacune d'entre elle, le numéro de ligne de l'instruction C correspondante.

b. On suppose que le processeur consacre un prédicteur de branchement 4 états uniquement à l'instruction assembleur de la ligne 5 (`jge .L2`). On appelle la fonction sur un tableau de 20 cases dont les valeurs provoquent le résultats suivants :

| indice dans `tab` | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| saut pris ? | oui | oui | oui | non | non | non | non | non | oui | oui |

| indice dans `tab` | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|
| saut pris ? | oui | non | oui | non | oui | non | non | oui | non | oui |

(a) On suppose que le prédicteur est initialement dans l'état 1. Indiquez les indices du tableau pour lesquels le prédicteur de branchement fait une erreur de prédiction.

(b) Même question en supposant que le prédicteur est initialement dans l'état 4.

# Chapitre 14

# Second jalon (ingéniérie algorithmique) :

Ce second jalon compare des implantations de fonctions de recherche dans un conteneur. Nous allons mesurer le temps nécessaire pour effectuer $K$ recherches dans un tableau de taille $N$. Lorsque $K$ est suffisamment grand, il est judicieux de préparer le tableau : le coût de la préparation se retrouve amorti par le gain obtenu sur un grand nombre de requêtes. Comme premier exemple, nous considérerons la recherche dichotomique dans un tableau trié.

Vous intégrerez dans vos réflexions : la hiérarchie des dispositifs de mémoire, la prédiction de branchement, le préchargement des données dans le cache, le rôle du choix des structures de données et algorithmes.

Dans ce jalon, les entrées des tableaux considérés seront des entiers 32-bit signés. Le nombre de requêtes $K$ sera fixe et pourra être ajusté pour trouver un juste équilibre entre la durée d'acquisition (le temps pour effectuer une mesure) et la qualité des mesures.

```
typedef int32_t T;
enum { K = 1000000 };
```

Une procédure de recherche dans un tableau est définie par deux opérations : une fonction de « préparation » et une fonction de recherche dans un tableau préparé. La première modifie le tableau en place, la seconde renvoie si l'élément est présent dans le tableau. Nous pouvons les regrouper dans une structure comme suit.

```
typedef struct {
    void (*prepare)(T* array, size_t size);
    bool (*search)(T* array, size_t size, T value);
} algorithme;
```

Nous allons par la suite donner plusieurs instances de cette structure de données et en comparer les performances selon la taille du tableau.

## 14.1 Mesures

Il s'agit dans un premier temps de mettre en place un dispositif de mesure qui puisse être utilisé pour différents algorithmes. Nous appellerons « mesure », pour un algorithme donné et une taille de tableau donnée, une estimation du temps nécessaire à l'exécution de $K$ recherches consécutives.

**Exercice 14.1 ★★**

a. Programmer une fonction `mesure` qui, étant donnés un algorithme et une taille $N$ réalise les étapes suivantes :

   (a) génère un tableau de longueur $N$ rempli de données aléatoires ;

   (b) *prépare* ce tableau conformément à l'algorithme ;

   (c) génère une liste aléatoire de requêtes de longueur $K$ (une requête étant une valeur de type `T` qui sera recherchée dans le tableau) ;

   (d) exécute ces requêtes conformément à l'algorithme ;

(e) renvoie une estimation du temps nécessaire à cette exécution (l'étape d'exécution devra être répétée pour obtenir une estimation de qualité).

b. Pour s'assurer que tout va bien, donner une instance factice d'un algorithme (la préparation ne fait rien, la recherche renvoie une constante) et en mesurer les performances sur un échantillon varié de tailles (suggestion : $\{\lfloor 10^{\frac{i}{3}} \rfloor \mid i \in [0; 22] \cap \mathbf{N}\}$).

## 14.2 Recherche dichotomique

La première implantation est la recherche dichotomique naïve dont on trouve une description sur wikipédia. Celle-ci requiert comme préparation que le tableau soit trié. À cette fin, on pourra utiliser la fonction `qsort` de la bibliothèque standard C.

**Exercice 14.2 ★★**

a. Implémenter la recherche dichotomique.

b. Mesurer ses performances pour diverses tailles de tableaux.

c. Observez-vous un lien entre les performances, la taille des tableaux et la taille des divers niveaux de cache ?

Les branchements conditionnels dans le corps de la boucle de l'implantation ci-dessus sont difficiles à prédire. Il est possible de les éviter.

```
bool dicho_branchfree(T* array, size_t size, T value) {
  T* head = array;
  while (size > 0) {
    uint64_t half = size >> 1;
    head = head[half] < value ? head + (size - half) : head;
    size = half;
  }
  return (head < array + size) && (head[0] == value);
}
```

**Exercice 14.3 ★★★**

a. Implanter la recherche dichotomique présentée ci-dessus.

b. Vérifier que le code assembleur correspondant n'a pas de branchement dans le corps de la boucle, contrairement à celui produit dans l'exercice précédent.

c. Mesurer les performances de cette recherche pour des tailles de tableaux variées.

d. Commenter.

## 14.3 Prefetch

Dans l'exercice précédent, les gains en performances ne sont pas toujours au rendez-vous. Cela peut s'expliquer par la difficulté à prédire les adresses mémoires qui seront lues : les accès à la mémoire non prédits commencent trop tard et les donnés ne sont pas dans le cache quand le programme en a besoin.

Le jeu d'instruction fournit des moyens d'aider le mécanisme de prédiction des lectures en mémoire : l'instruction `PREFETCHT0` appliquée à l'adresse d'une donnée suggère de placer cette donnée dans le cache de premier niveau.

```
#include <immintrin.h>
void _mm_prefetch (char const* p, int i)

  Fetch the line of data from memory that contains
address p to a location in the cache hierarchy specified by
```

*the locality hint i, which can be [...] `_MM_HINT_T0`.*

**Exercice 14.4 ★★★**

a. Insérer dans le programme de l'exercice précédent deux instructions de prefetch à chaque tour de boucle pour s'assurer que la case du tableau accédée à l'itération suivante sera disponible à temps.

b. Effectuer une série de mesures et comparer aux résultats précédents.

## 14.4 Agencement d'Eytzinger

Des données triées peuvent être organisées sous la forme d'un arbre binaire de recherche, que l'on peut agencer dans un tableau de la manière suivante[1] :

a. la racine est placée en première position ;

b. pour un nœud à la position $k$, son fils gauche est stocké à la position $2k + 1$ et son fils droit en $2k + 2$.

Cette transformation peut être programmée comme suit :

```
size_t toEytzinger(T* dst, T* src, size_t size,
  size_t i = 0, size_t k = 0) {
  if (k < size) {
    i = toEytzinger(dst, src, size, i, 2*k + 1);
    dst[k] = src[i];
    i = toEytzinger(dst, src, size, i + 1, 2*k + 2);
  }
  return i;
}
```

**Exercice 14.5 ★★★★**

a. Implémenter la fonction de préparation qui après avoir trié un tableau, réorganise ses éléments.

b. Implémenter la fonction de recherche qui parcourt l'arbre binaire de recherche.

c. L'implémenter sans branchement.

d. Mesurer, comparer, discuter.

1. Cette manière de ranger un arbre dans un tableau est parfois dite d'Eytzinger par analogie avec la numérotation de Sosa-Stradonitz, aussi attribuée à un certain Michael von Aitzing.

# Chapitre 15

# Second jalon (sécurité) : un déréférencement de trop

Le but de ce second jalon est de mettre en évidence que l'exécution spéculative peut laisser des traces observables grâce à un canal caché similaire à celui mis en œuvre précédemment.

Considérons un tableau `pointeurs` de $N+1$ pointeurs, les $N$ premiers pointant vers une adresse arbitraire et le dernier pointant vers la case $s$ de `T` (pour un certaine valeur *secrète* $s$). Si ce dernier pointeur a été déréférencé, on peut le repérer *via* un chronométrage analogue à la fonction réalisée au 1er jalon (exercice 11.2 du Chapitre 11).

Examinons la boucle suivante, qui déréférence les $(*k)$ premiers pointeurs du tableau :

```
do {
  p = pointeurs[i];
  v |= p[0];
  _mm_clflush(k);
} while (i++ < k[0]);
```

Dans cette boucle, il y a un saut conditionnel : faut-il continuer à itérer ou sortir de la boucle ? Les premières itérations entraînent le prédicteur de branchement : la boucle doit continuer. Aussi, quand la garde deviendra fausse (lorsque la valeur $i$ du compteur atteint $k$), la prédiction sera erronée et le corps de boucle sera exécuté une ou plusieurs fois, spéculativement, à tort. Pour faire durer cette période de spéculation erronée, on veillera à placer `k` en mémoire et à purger le cache de sa valeur à chaque itération.

**Exercice 15.1 ★★** Écrire une fonction qui :

- initialise `T` et appelle `flush` ;
- exécute une boucle du type ci-dessus, en mettant en œuvre les préconisations données ;
- puis recherche une case de `T` dans le cache.

En jouant sur les valeurs de $N$ et de $k$, estimer le nombre d'itérations exécutées spéculativement.

# Chapitre 16

# Approfondissement : Modèle de mémoire hiérarchique à associativité partielle

Le modèle de cache complètement associatif n'est qu'une approximation des réalisations matérielles. Cette approximation permet déjà de comprendre certains phénomènes, mais n'en explique pas d'autres. Par exemple, l'exercice 16.2 ci-dessous va vous faire constater que lorsque l'on effectue des recherche dichotomique sur des tableaux de taille

$$N_1 = 1\ 000\ 000, \quad N_2 = 1\ 048\ 576, \quad \text{et} N_1 = 1\ 300\ 000,$$

le cas $N_2$ est **substantiellement** plus lent. Pour expliquer ce phénomène, il convient de raffiner notre modèle de mémoire cache.

Un cache est généralement divisé en sous-cache indépendants. Le nombre de blocs que peut stocker un de ces sous-caches est appelé l'*associativité* du cache. Ainsi, un cache de taille totale $C$, de taille de bloc $b$ et d'associativité $a$ comporte $s = \frac{C}{ab}$ sous-caches. Numérotons les sous-caches de 0 à $s-1$. Dans le cas d'associativité complète, il y a un unique sous-cache.

Chaque bloc de mémoire centrale est pré-affecté à un sous-cache. Ainsi, le bloc 0 est affecté au sous-cache 0, le bloc 1 au sous-cache 1, ..., le bloc $s-1$ au sous-cache $s-1$, le bloc $s$ au sous-cache 0, le bloc $s+1$ au sous-cache 1, ... Lorsque le cache accède à un nouveau bloc $B$, ce dernier est transmis à son sous-cache d'affectation, disons $S$. La gestion de page est faite au niveau du sous-cache $S$ : ainsi, sous l'hypothèse d'une gestion de page LRU, le bloc qui sera déchargé pour faire de la place à $B$ est *le bloc accédé le moins récemment parmi les blocs stockés dans* $S$. Si un programme n'accède qu'à des données se trouvant dans des blocs affectés à un même sous-cache, seul ce sous-cache travaille.

Considérons par exemple un cache de taille 32 Mo ($C = 32 \cdot 2^{20}$), d'associativité 8 pour lequel les blocs sont de $b = 64$ octets. Chacun de ses sous-cache peut stocker 8 blocs, soit $8 * 64 = 512$ octets. Si on ne se sert que d'un sous-cache, la taille effective a été divisée par 64 000. Ces paramètres sont réalistes...

Il est utile de pouvoir déterminer si deux adresses mémoire données, disons `&a` et `&b`, sont associées au même sous-cache. On commence par calculer leurs numéros de blocs, puis on prend ce numéro modulo le nombre de sous-caches. Autrement dit, on oublie les $k$ bits de poids faible (où $2^k$ est la taille de bloc), puis on oublie les bits de poids fort pour ne garder que $\ell$ bits (où $2^\ell$ est le nombre de sous-caches).

**Exercice 16.1** ★ Complétez la description des paramètres du système de cache de votre machine en ajoutant l'associativité de chaque niveau (vous trouverez cette information au même endroit que les informations de taille et ligne de cache). Complétez avec cela la dernière ligne du tableau de l'exercice 9.3 (page 42).

**Exercice 16.2 ★★★** On pose $R = 10\ 000$ et $T = 100\ 000\ 000$. Réalisez l'expérience suivante pour $N = 1\ 000\ 000$, puis pour $N = 1\ 048\ 576 (= 2^{20})$, puis pour $N = 1\ 300\ 000$ et comparez les résultats :

- Réservez un tableau `tab` de $N$ `int`, initialisez le par des entiers aléatoires entre 1 et $T$, puis triez le.
- Réservez un second tableau `req` de $R$ `int` et initialisez le par des entiers aléatoires entre 1 et $T$.
- Programmez ou importez une fonction qui réalise une recherche dichotomique d'un `int` $x$ donné en argument dans le tableau trié `tab`.
- Chronométrez le temps que cela prend d'appeler votre fonction de recherche dichotomique pour chacune des $R$ valeurs du tableau `req`.

a. Que constatez-vous ?

b. Pouvez-vous proposer une explication argumentée au moyen de la notion d'associativité ?

# Chapitre 17

# Approfondissement : Sauts calculés et prédiction de branchements

Un saut calculé est un branchement *inconditionnel* dont la destination n'est connue qu'à l'exécution. Elle est le résultat d'un calcul, De tels sauts se rencontrent notamment dans l'implémentation des fonctions d'ordre supérieur (c'est-à-dire paramétrées par d'autres fonctions) ou des mécanismes de résolution dynamique (comme l'envoi de messages dans les langages à objet).

Les microprocesseurs tentent de prédire la destination de ces sauts. Le but de ce chapitre est d'explorer les facultés et limites d'une unité de prédiction de branchement.

## 17.1 Trois façons d'appeler une fonction

On considère le programme C suivant qui recherche dans les limite premiers éléments d'une suite celui qui optimise un critère. La suite est donnée par les appels successifs à la fonction suivant. Le critère est évalué en appelant la fonction compare.

```
uint32_t optimal(uint64_t limite) {
  uint32_t meilleur = suivant();
  while (limite-- != 0) {
    uint32_t candidat = suivant();
    meilleur = compare(meilleur, candidat) ? candidat : meilleur;
  }
  return meilleur;
}
```

Nous allons étudier trois façons d'appeler la fonction `compare` :

**en ligne** sans branchement, le code de la fonction est inséré dans le corps de la boucle ;

**appel direct** saut inconditionnel à une destination connue statiquement ;

**appel indirect** saut inconditionnel à une adresse qui n'est connue que lors de l'exécution.

On peut réaliser ces trois variantes en jouant sur la déclaration de compare : en faire une fonction marquée inline ou non, ou en faire un pointeur, initialisé lors de l'exécution pour pointer vers la fonction.

**Exercice 17.1 ★★★**

a. Écrire les trois variantes de ce programme. Combien y a-t-il de branchements ?

b. Fixer un nombre d'itération et remplir le tableau de mesures ci-dessous (en utilisant perf stat ou likwid-perfctr -C 1 -g BRANCH, donner le nombre de cycles, d'instructions, de branches et la proportion de branches mal prédites).

c. Commenter. Quel est le coût d'un appel de fonction ?

| | Cycles | Instructions | Branches | mal prédites |
|---|---|---|---|---|
| En ligne | | | | |
| Appel direct | | | | |
| Appel indirect | | | | |

## 17.2 Mise en échec de la prédiction

L'exemple ci-dessus illustre qu'un appel de fonction peut être prédit avec une grande précision. Nous allons maintenant essayer d'explorer les limites de ce mécanisme de prédiction.

Considérons un tableau `f` de (pointeurs de) fonctions, chacune de ces fonctions ayant le même corps et un programme (voir ci-dessous) appelant dans une boucle une de ces fonctions. La fonction `suivant` produit une suite d'indices de cases de notre tableau de fonctions.

```
while (n-- > 0) {
    r = f[suivant()](r);
}
```

**Exercice 17.2 ★★★** Écrire et compléter le programme ci-dessus, en commençant par un tableau à deux éléments.

a. Instancier `suivant` avec une suite constante et vérifier que les appels *via* `f` sont correctement prédits.

b. Instancier `suivant` avec une suite pseudo-aléatoire [1] et vérifier que les appels *via* `f` ne sont pas correctement prédits.

c. Expérimenter avec d'autres suites : 0, 1, . . . ; 0, 0, 1, 1, . . . Lesquelles sont bien prédites ?

d. Augmenter la taille du tableau : combien de destinations différentes peuvent être correctement prédites ?

## 17.3 Prédiction des retours de fonctions

Les retours de fonction sont aussi des sauts calculés. Ils peuvent disposer d'un dispositif de prédiction dédié qui contient une pile d'adresses de retour : on peut ainsi prédire qu'un `ret` retourne après le `call` le plus récent.

**Exercice 17.3 ★★**

a. Écrire un programme avec une fonction appelée depuis de très nombreux sites d'appel différents.

b. Mettre en évidence que les retours sont correctement prédits.

c. Peut-on trouver une limite au nombres d'adresses de retour bien prédites ?

**Exercice 17.4 ★★★★**

a. Écrire une fonction qui modifie son adresse de retour de sorte à utiliser l'instruction `ret` pour effectuer un saut.

b. Écrire un programme appelant cette fonction de sorte que la destination du saut soit prévisible mais systématiquement mal prédite.

1. On pourra utiliser `xoroshiro64**`, par exemple ; cf. https://prng.di.unimi.it/

# Chapitre 18

# Principes de Spectre et Meltdown

Cette 6e séance examine les failles de sécurité Spectre et meltdown. Ces deux failles exploitent certains comportements des mécanismes de hiérarchie mémoire et d'exécution spéculative. L'examen théorique de leurs principes va nous permettre de mettre en application les notions vues jusqu'à présent. La mise en œuvre pratique de ces principes, au travers d'une ébauche de preuve de concept, va nous amener à revisiter de manière plus pointue le chronométrage de code et l'analyse de fonctionnement du processeur.

Pour un premier aperçu de Spectre, regarder l'exposé suivant, de Paul Kocher, jusqu'à 10'30 :

https://www.youtube.com/watch?v=zOvBHxMjNls

**Objectifs.** Cette séance présente une étude de cas qui mobilise toutes les notions vues jusqu'ici et introduit à des problèmes de sécurité récents et encore mal résolus par l'industrie informatique.

## 18.1 Quelques notions de sécurité

Précisons tout d'abord quelques notions de sécurité. Il ne s'agit pas de faire un cours d'introduction à cette thématique mais simplement de définir quelques notions dont nous auront besoin.

Une *attaque* sur un système informatique est la réalisation d'une action non autorisée. Cette action peut viser à modifier le système (par exemple chiffrer un fichier) mais peut aussi simplement viser à l'observer (lecture de données, de clefs cryptographiques, etc.). Une *attaque par canal auxiliaire* (« side-channel attack ») est une attaque qui tire parti de défauts dans non pas la *conception*, mais dans l'*implantation* d'un système informatique. Une attaque par canal auxiliaire peut notamment s'appuyer sur la mesure d'effets secondaires du fonctionnement d'un système informatique (par exemple la consommation électrique ou les bruits émis, ...). Parmi ces attaques par canaux auxiliaires, les *attaques temporelles* (« timing attacks ») déduisent des informations sur un système informatique à partir de mesures de ses temps de réponse.

Les attaques par canaux auxiliaires exploitent un écart entre les *spécifications* d'un système et son *implantation*. Soulignons qu'un tel hiatus est inévitable. Les spécifications sont faites dans un modèle formel qui doit permettre de raisonner abstraitement sur le comportement global du système (par exemple de *prouver* qu'il accomplit correctement les tâches pour lesquelles il a été conçu et ne peut se retrouver dans un état problématique). Ce modèle abstrait ne traduit qu'une partie des caractéristiques du monde physique dans lequel se situe l'implantation du système. Les attaques par canaux auxiliaires exploitent précisément les « impensés » du modèle abstrait, c'est à dire les effets physiques qu'il ne modélise pas.

## 18.2 Principes de Spectre et Meltdown

« Spectre » et « Meltdown » sont les noms de deux failles de sécurité qui permettent essentiellement à un programme de lire des zones mémoires auxquelles il ne devrait pas avoir accès. On restera ici à un

niveau d'analyse qui ne distingue pas ces deux failles. Spectre et Meltdown réalisent impunément un accès mémoire interdit en combinant 2 idées.

### 18.2.1 Idée 1 : l'impunité de la spéculation

La première idée est qu'il existe **une situation** dans laquelle un **accès mémoire interdit** reste **impuni** : lorsque cet accès est exécuté dans le cadre d'une **exécution spéculative mal prédite**. Examinons par exemple le code suivant (et une traduction possible en assembleur) :

```
int tab[500];
...
if (i < 500)
   a += tab[i];
```

```
    ⋮
    cmp     rsi, 499
    ja      .L24
    add     eax, DWORD PTR [rcx+rsi*4]
.L24:
    ret
```

Comme on l'a vu, ce `if` se traduit en code assembleur par un saut conditionnel. Le fonctionnement en pipeline du processeur devrait amener à un temps d'attente (« bulle ») important car le simple chargement de l'instruction suivant ce saut conditionnel devrait attendre l'évaluation de la condition présidant au saut. Le mécanisme d'exécution spéculative contourne cela en choisissant (via le prédicteur de branchement) un des résultats comme le plus probable, et en l'exécutant sans attendre. En cas d'erreur, on jette le travail effectué et on reprend l'exécution à partir du saut, en choisissant cette fois la bonne alternative.

Imaginons que dans le code ci-dessus, on se présente à l'instruction `if` avec une valeur $i = 10\ 000$. Il est possible que le prédicteur de branchement prédise, à tort, que la comparaison donnera un résultat `vrai`. Dans ce cas, l'instruction `a += tab[i]`, ou plus précisément sa traduction en assembleur, est exécutée spéculativement ; en particulier, le processeur accède en lecture à l'adresse `tab + 40 000`. Une fois la condition du `if` correctement évaluée, le processeur fera machine arrière et « oubliera » la valeur lue à l'adresse `tab + 40 000`.

Que se passe-t-il si l'adresse `tab + 40 000` se trouve dans une zone mémoire à laquelle le programme n'a pas le droit d'accéder ? Si un tel accès était fait directement, il produirait une erreur au niveau du système qui se traduirait par une interruption brutale du programme et l'affichage d'un familier `segmentation fault`. Dans le cas où un tel accès est effectué lors d'une exécution spéculative erronée, il est raisonnable qu'il ne soit pas sanctionné car il ne correspond pas à une instruction qu'aurait dû effectuer le programme.

### 18.2.2 Idée 2 : la spéculation erronée laisse des traces

L'exécution spéculative est un mécanisme d'accélération qui doit s'avérer transparent. Ainsi, ses spécifications indiquent qu'une spéculation erronée ne doit laisser aucune trace dans le processeur : les registres, drapeaux, etc. doivent être remis dans l'état qu'ils auraient eu si l'exécution spéculative erronée n'avait pas eu lieu.

Ce spécifications ne couvrent cependant pas le reste de la micro-architecture en général, et la hiérarchie mémoire en particulier. Il s'avère que si un accès mémoire exécuté lors d'une exécution spéculative erronée a un effet (chargement/déchargement) sur un niveau de cache, cet effet **n'est pas annulé** et laisse donc des traces. Ainsi, dans notre exemple ci-dessus, si la lecture du contenu de l'adresse `tab + 40 000` lors d'une exécution spéculative erronée a provoqué le chargement d'un bloc en cache, l'annulation de cette exécution spéculative laisse ce bloc en cache.

Comme nous avons pu l'observer dans un chapitre précédent consacré aux hiérarchies de mémoires, l'état du cache peut être indirectement observé en mesurant la durée de certaines opérations.

# Chapitre 19

# Jalon 3 (sécurité) : mise en œuvre de Spectre

Chaque binôme peut :

- s'il a réalisé les jalons 1 et 2 en sécurité, entreprendre la réalisation de spectre proposée dans ce jalon 3 sécurité,
- s'il a réalisé les jalons 1 et 2 en ingéniérie algorithmique, entreprendre les jalons 1 puis 2 en sécurité,
- ou opter traiter les approfondissements 16 et 17 puis le complément 20.

## 19.1 En pratique

Nous allons réaliser un ébauche de preuve de concept illustrant la vulnérabilité Spectre : l'exécution spéculative erronée permet de manière transitoire de lire à des adresses interdites et d'utiliser la valeur lue pour modifier l'état du cache ; un canal auxiliaire nous permet d'observer ces modifications même après la fin de l'exécution transitoire et d'en déduire la valeur lue.

Le contrôle de la micro-architecture (prédiction de branchement, cache) étant plutôt délicat, nous ferons en sorte de nous placer dans des conditions aussi favorables que possible.

### 19.1.1 Canal caché

Dans un premier temps, nous allons construire un canal caché exploitant la hiérarchie de la mémoire en utilisant une zone de mémoire — que nous appellerons `T` — partagée entre l'émetteur et le récepteur. Cette zone contient 256 *cases*. Leur contenu ne nous intéresse pas : ce qui compte est de savoir si elles sont en cache ou pas. Avant toute communication, on s'assure qu'aucune portion de `T` n'est en cache. Pour envoyer un octet, l'émetteur *lit* la case correspondant à la valeur de l'octet à envoyer. Puis, pour recevoir un octet, il suffit de mesurer le temps d'accès à chacune des cases : l'une d'entre elles se trouve en cache, y accéder est donc significativement plus rapide.

#### FLUSH+RELOAD

À l'exercice 11.2 du Chapitre 11, vous avez mis en place un tableau `T` correspondant à 256 lignes de cache distinctes et vous avez réussi à y distinguer un accès en cache d'un accès hors cache. Nous allons réutiliser cela.

#### Envoi de message

Pour transmettre un octet $i$, le tableau `T` est au préalable intégralement sorti du cache, puis l'émetteur lit la case $i$ de `T`, puis le récepteur mesure le temps d'accès à toutes les cases et recherche *la* durée inférieure au seuil. Mais cette méthode, élaborée précédemment, n'est pas très fiable : deux appels à la fonction `timings` peuvent produire des résultats très différents ; et il peut y avoir dans une série de mesures, plusieurs durées inférieures au seuil. Pour améliorer la qualité du signal, on va répéter la transmission et compter combien de fois chaque octet a été potentiellement *vu* en cache.

Pour chaque octet à recevoir, on va donc tenir un tableau de scores comptabilisant, pour chacune des valeurs possibles de l'octet, les apparitions en cache de cette valeur.

Nous allons employer cette méthode pour transmettre un court message contenu dans un tableau `secret` composé de caractères ASCII.

**Exercice 19.1 ★★**

a. Écrire une fonction `init_scores` qui remet à zéro un tableau de 256 entiers.

b. Écrire une fonction `reception` qui appelle `timings` puis met à jour un tableau de scores.

c. Écrire une fonction `meilleur_score` qui renvoie la position de la valeur maximale dans un tableau de 256 entiers.

d. Écrire une fonction `emission(i, T)` qui émet l'octet `secret[i]` en *lisant* dans `T`.

e. Écrire une fonction `exfiltre(dst)` qui copie dans le tableau `dst` tout le message `secret` en utilisant le procédé décrit ci-dessus. Combien de fois faut-il répéter l'émission pour obtenir un résultat sans erreur ?

### 19.1.2 Exécution spéculative

Nous allons exploiter la vulnérabilité Spectre pour exfiltrer un message (comme dans l'exercice 19.1 ci-dessus), mais à l'insu de la victime (contrairement à l'exercice 19.1 dans lequel l'émetteur est actif). Le code vulnérable de la victime se présente comme suit (où `C(n)` calcule la position dans `T` de sa case $n$) :

```
char victime(char* T, uint64_t* k, uint64_t i) {
  uint8_t n;
  c = 0;
  if (i < k[0]) {
    n = trampoline[i];
    c = T[C(n)];
  }
  return c;
}
```

Dans cet exemple, `trampoline` est un tableau de longueur $k$ ; pour simplifier, considérons qu'il contient les entiers de 1 à $k$. Cette fonction émet un octet de ce tableau *via* notre canal auxiliaire, à la manière de la fonction `emission` évoquée ci-dessus. Cependant, appelée avec une valeur $i$ hors des bornes du tableau, cette fonction nous permet d'émettre quasiment n'importe quel octet présent en mémoire. En effet, une exécution spéculative peut exécuter à tort le corps du `if`, même si le test échoue.

Afin d'exécuter spéculativement le code vulnérable, il faut au préalable entraîner la prédiction de branchement, c'est-à-dire appeler la fonction `victime` avec une valeur $i$ dans les bornes du tableau `trampoline`. Ces exécutions d'entraînement auront un effet sur le cache : lequel ? Cela impacte-t-il la transmission *via* le canal auxiliaire ? Pour simplifier, nous ferons l'hypothèse que les messages à transmettre sont des textes ASCII.

Pour leurrer la prédiction de branchement, il est préférable que les deux sortes d'appels à la fonction `victime` (entraînement ou pas) ne puissent pas être facilement distingués par le flot de contrôle qui y mènent. On veillera en particulier à calculer la valeur de l'argument $i$ sans branchement conditionnel.

**Exercice 19.2 ★★★**

a. Adapter la fonction `reception` pour ignorer les octets émis lors de l'entraînement.

b. Adapter la fonction `exfiltre` pour utiliser `victime` à la place de la fonction `emission`.

c. En quoi ce programme est-il une *attaque* ?

## 19.2 Pour aller plus loin

Pour une source très complète sur ces attaques et certaines de leurs ramifications, voir :

- https://meltdownattack.com/

Pour une preuve de concept de Spectre en javascript, voir :

- https://leaky.page/

Pour un article de vulgarisation qui décrit un peu plus en détail les principes de Spectre et Meltdown et des possibilités de leur mise en œuvre, voir :

- *Spectre Attacks : Exploiting Speculative Execution*. Kocher et al. *Communication of the ACM*, juillet 2020.
https://cacm.acm.org/magazines/2020/7/245682-spectre-attacks/fulltext

Pour un article de vulgarisation qui dresse un panorama de conséquences de Spectre et Meltdown sur l'industrie informatique, voir :

- *How to Live in a Post-Meltdown and -Spectre World*. Bennett et al. *Communication of the ACM*, décembre 2018.
https://cacm.acm.org/magazines/2018/12/232898-how-to-live-in-a-post-meltdown-and-spectre-world/fulltext

# Chapitre 20

# Complément : exécution dans le désordre et superscalaire

Ce chapitre revient sur le modèle de pipeline présenté au chapitre 13, dont la simplicité n'est hélas qu'une approximation de la réalité. En effet, comme on peut l'apercevoir sur la figure 9.1, outre le parallélisme entre les différents étages du pipeline, il y a du parallélisme dans la communication entre les différents étages et du parallélisme au sein de chaque étage.

## 20.1 En attendant un load

Reprenons le grand tableau de 256 *cases* du premier jalon (page 50). Pour calculer avec une donnée contenue dans ce tableau, il faut *attendre* que celle-ci fasse le long chemin depuis la mémoire où elle est stockée. Nous allons mettre en évidence qu'il est possible de faire d'autres calculs pendant ce temps d'attente.

**Exercice 20.1 ★★**

a. Écrire un programme qui agrège (par exemple en calcule la somme) les 256 valeurs stockées dans le tableau en attendant que chaque termine avant de commencer le suivant (en utilisant une barrière telle que l'instruction LFENCE). Donner une estimation en cycles du temps nécessaires à une lecture.

b. À la place de la barrière, exécuter une séquence d'instructions qui ne peuvent pas être exécutées en parallèle et qui dépendent de la valeur lue (par exemple répéter N fois inc r, où r contient la valeur lue). Choisir une séquence dont la longueur correspond au temps d'attente.

c. Modifier la séquence d'instructions exécutées après la lecture mémoire pour qu'elle ne dépende plus de cette valeur mais réalise le même calcul.

d. Comparer les temps d'exécution des trois variantes ci-dessus et conclure.

## 20.2 Exécution superscalaire

Les (micro-) instructions prêtes à être exécutées sont réparties sur différents *ports* ; les unités de calcul desservies par différents ports peuvent s'exécuter en parallèle. L'exercice suivant a pour but de mettre en évidence ce phénomène.

Nous allons considérer plusieurs séquences d'instructions dont l'exécution sera entrelacée : dans une boucle, à chaque itération, une étape de chaque séquence est exécutée. Nous mesurerons l'influence du nombre de séquences d'instructions sur le temps total de leur exécution.

**Exercice 20.2 ★★★**

a. Écrire une boucle qui décrémente un registre et termine quand il atteint zéro. Quelles unités d'exécution sont utilisées ? Par quels ports sont-elles desservies ?

b. Ajouter une séquence de `xor` (une opération par itération, chaque itération dépend du résultat du `xor` de l'itération précédente mais est indépendante du reste de la boucle). Quel port est sollicité ? Quel effet cela a-t-il sur le temps total d'exécution de la boucle ?

c. Ajouter d'autres séquences similaires (chaque séquence est indépendante des autres, chaque étape d'une séquence dépend de l'étape précédente au sein de la même séquence). Mesurer la contribution de chaque séquence au temps total d'exécution. Représenter schématiquement l'ordonnancement des instructions.

d. Faire la même expérience avec des séquences de multiplications plutôt que des séquences d'opérations logiques. Combien y a-t-il d'unités de multiplications dans la microarchitecture ? Qu'observe-t-on ? Comment l'expliquer ?

# Chapitre 21

# Introduction à la vectorisation

Cette dernière séance introduit à la notion de *vectorisation* de code. L'objectif est de pouvoir appréhender le type de problèmes traitables efficacement sur GPU.

> **Objectifs.** À l'issu de cette dernière séance, il est attendu que vous compreniez les principes de la vectorisation de programme et que vous sachiez déterminer si un code simple bénéficierait d'une vectorisation.

## 21.1 Des parallélismes

Commençons par préciser le type de parallélisme opéré par le calcul vectoriel.

**SISD.** Dans la gamme `intel`, les premiers processeurs ont été conçus pour exécuter *séquentiellement* des instructions ne traitant qu'*une seule* donnée. On parle de processeurs SISD (*single instruction single data*). L'introduction de pipelines (cf séance 5) semble permettre d'exécuter plusieurs instructions en parallèle, mais il s'agit seulement d'utiliser plus efficacement chaque partie du processeur ; comme chaque partie continue à traiter les instructions séquentiellement, un processeur à pipeline reste SISD. Ce modèle a évolué de deux manières.

**MIMD.** Une première forme de réel parallélisme est le MIMD (*multiple instruction multiple data*). Il s'agit d'exécuter plusieurs instructions distinctes simultanément. En pratique, cela peut se réaliser en construisant plusieurs pipelines, éventuellement avec un pipeline principal capable de traiter toutes les instructions et un pipeline secondaire, allégé, capable de traiter certaines instructions simples et fréquentes.

Par exemple, le `pentium` (1993) comporte deux pipelines appelés U et V. Lors du décodage, certaines instructions sont appariées pour être traitées en parallèle, l'une par U et l'autre par V. Ces appariements doivent respecter certaines contraintes :

- Les instructions qui peuvent s'apparier dans U ou dans V sont `mov, push, pop, inc, dec, add, sub, cmp, and, or, xor`.
- Les instructions qui ne peuvent s'apparier que dans U sont `adc`, {`shl, shr, sal, sar`} avec un compteur immédiat, et {`ror, rol, rcr, rcl`} avec un compteur de 1.

Les autres instructions ne peuvent pas être appariées. Les appariements doivent par ailleurs respecter des règles d'indépendance (qu'on ne détaille pas ici), comme par exemple de travailler sur des registres différents.

Cette évolution s'est prolongée au travers du développement du *multithreading* et de *multicœurs*.

**SIMD.** Une seconde forme de réel parallélisme est le SIMD (*single instruction multiple data*) qui exécute les instructions une par une mais où chaque instruction opère sur plusieurs données à la fois. De manière équivalente, il s'agit de travailler sur des *vecteurs* en appliquant la même opération composante par composante. Le reste de la séance se concentre sur l'architecture de type SIMD. On la retrouve notamment sur les processeurs graphiques (GPU).

**Vocabulaire.** Les processeurs SIMD travaillant sur des vecteurs, on les appelle parfois *processeurs vectoriels*. Par analogie, les processeurs SISD et MIMD travaillent sur des scalaires. On appelle les SISD des processeurs *scalaires* et les MIMD des processeurs *superscalaires*. En pratique, ces distinctions ne sont pas forcément pertinentes au niveau des processeurs, les processeurs `intel` actuels comportant par exemple des parties superscalaires et des parties vectorielles.

## 21.2 Vectorisation d'algorithme : un exemple

Illustrons l'idée de la vectorisation sur un premier exemple. Supposons que l'on souhaite modifier une chaîne de caractères en ajoutant 1 au code ASCII de chaque caractère (modulo 256). [1]

**Solution "naturelle".** Voici une première solution et le code produit par `gcc -O2` :

```
1 void incr(char* t)
2 {
3   while ((*t) != '\0'){
4     (*t) += 1;
5     ++t;
6   }
7 }
```

```
1 incr:   jmp     .L7
2 .L5:    add     eax, 1
3         add     rdi, 1
4         mov     BYTE PTR [rdi-1], al
5 .L7:    movzx   eax, BYTE PTR [rdi]
6         test    al, al
7         jne     .L5
8         rep ret
```

**Une quasi-solution vectorisée.** Dans la solution "naturelle", les données sont lues et écrites en mémoire octet par octet (cf lignes 4 et 5 du code asm). Il peut être tentant d'utiliser le fait que les processeurs 64 bits peuvent travailler sur 8 octets simultanément comme ceci :

```
1  void pincr(char* t,int n)
2  {
3    unsigned long int* p;
4    p= (unsigned long int*) t;
5
6    for (int i=0; i<n; ++i){
7      (*p)+=0x0101010101010101;
8      ++p;
9    }
10 }
```

```
1  pincr:  test    esi, esi
2          jle     .L9
3          lea     eax, [rsi-1]
4          lea     rdx, [rdi+8+rax*8]
5          movabs  rax, 72340172838076673
6  .L11:   add     QWORD PTR [rdi], rax
7          add     rdi, 8
8          cmp     rdi, rdx
9          jne     .L11
10 .L9:    rep ret
```

Passons pour l'instant sur le fait que cette fonction `pincr` ne traite que des chaîne de caractères dont la longueur est un multiple de 8 (on pourrait gérer les caractères restants un par un).

**Pourquoi vectorisée ?** L'instruction `add` de la ligne 6 simule du parallélisme SIMD. En effet, ajouter `72340172838076673 = 0x0101010101010101`, à la valeur `QWORD PTR [rdi]` revient à ajouter 1 à chacun des 8 entiers 8-bits stockés en mémoire à l'adresse `[rdi]`. Au niveau du circuit additionneur, les différentes composantes 8-bits de `QWORD PTR [rdi]` sont traitées *en parallèle*. Ces opérations sont indépendantes.

**Pourquoi quasi ?** Remarquons que le traitement d'un des octets peut *déborder* sur un autre octet à cause des retenues. Par exemple, si `*p` vaut `...05 FF` avant traitement, il vaudra `...07 00` après traitement et non pas `...06 00`. L'idée de la vectorisation des processeurs est d'ajouter des registres et des instructions qui permettent de travailler sur des vecteurs de nombres, comme ci-dessus, en évitant les problèmes de débordement.

1. Si la chaîne contenait un caractère de code 255 elle se retrouvera donc tronquée. On ne s'en souciera pas.

**Calcul vectoriel.** Le calcul sur des vecteurs de nombres, ou calcul vectoriel, est facilement parallélisable puisque les différentes composantes sont indépendantes. Pour tirer parti de ces outils vectoriels, et par exemple "faire tourner un programme sur GPU", il faut (ré)écrire l'algorithme en terme de vecteurs.

## 21.3 Vectorisation de boucles par `gcc`

Pour qu'il soit intéressant de vectoriser un code, il faut que le gain apporté par la vectorisation compense le coût des changements de registres occasionnés par les aller-retours entre scalaire et vectoriel. Cela correspond donc à des traitements coûteux, les boucles répétées suffisamment de fois étant un exemple typique.

**Difficultés.** Voici quelques sources de problèmes dans la vectorisation de boucles :

- Un *flot* non séquentiel (branchements `if`, points d'entrée ou de sortie multiples, ...) qui pourrait se comporter différemment pour les différentes composantes d'un même vecteur.
- Un *nombre d'itérations* non prévisible au moment d'entrer dans la boucle (par exemple la fonction `incr` ci-dessus), et qui rends délicate la gestion des itération restant à effectuer en scalaire.
- L'appel de *fonctions*, qui même inliné peuvent cacher un code à flot compliqué. Notons, cependant, que certaines fonctions courantes disposent de versions vectorielles (comme `pow` ou `cos` par exemple).
- Les *accès en mémoire* à des adresses non contiguës (par exemple accéder à `tab[index[i]]`) poseront des problèmes d'efficacité de lecture/écriture en mémoire.
- Les *dépendances de données* provoquant des temps d'attente. Ainsi, sur une même variable, une lecture/écriture suivant une écriture posera problème, tandis qu'une lecture suivant une lecture ne posera jamais de problème, et une écriture suivant une lecture ne posera que parfois des problèmes.

Ajoutons qu'en cas de boucles imbriquées, il peut être difficile de vectoriser plus d'une boucle.

**Préconisations.** L'option `-O3` de `gcc` tente de vectoriser le code. Le sujet de TP illustre à quel point `gcc` est sensible à la rédaction du code lorsqu'il tente de vectoriser une boucle. Au vu des critères ci-dessus, on ne peut que conseiller de préférer des boucles simples (type `for` avec incrément régulier), de limiter les dépendances entre itérations de la boucle, d'utiliser le compteur de boucle aussi souvent que possible comme indice de tableau, d'utiliser autant que possible un espacement régulier des données en mémoire et d'éviter autant que possible l'adressage indirect.

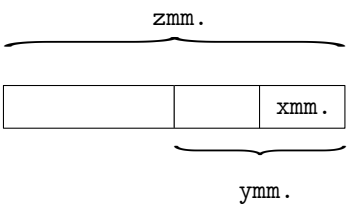
**Un premier exemple.** Revenons à notre fonction `incr` ci-dessus. Si on la compile en `-O3`, on obtient un code assembleur sans instruction vectorielle. Cela s'explique par exemple par le fait que le nombre d'itérations n'est pas facilement prévisible. En revanche, `gcc` réussit à vectoriser :

```
1 void sincr(char* t, int n)
2 {
3   for (int i=0; i<n; ++i){
4     (*t) += 1;
5     ++t;
6   }
```

## 21.4 Instructions vectorielles en assembleur

Depuis 1999, les processeurs `intel` comportent des registres et instructions vectoriels.

**registres vectoriels.** Les processeurs `intel` disposent de *registres vectoriel* qui peuvent se fractionner en blocs, de sorte que les calculs de débordent pas d'un bloc à l'autre. Les registres vectoriels ont été créés initialement en 128 bits (1999), puis ont été étendus à 256 (2011) puis à 512 bits (2013). Ils sont organisés comme indiqué ci-contre, le "." étant un index allant de `0` à `31`.



Par exemple, les registres `xmm0` à `xmm31` sont 128 bits et peuvent être utilisés comme des vecteurs d'entiers 64 bits (2 coordonnées), 32 bits (4 coordonnées), 16 bits (8 coordonnées) ou 8 bits (16 coordonnées) ; ils peuvent aussi être utilisés comme des vecteurs de nombres flottants 64 bits (2 coordonnées) ou 32 bits (4 coordonnées).

**Instructions vectorielles.** Les registres vectoriels sont utilisés par des instructions spécifiques, et souvent assez spécialisées pour le traitement de grandes quantité de données. Un descriptif complet des instructions x86 (instructions vectorielles comprises) est disponible à

https://www.felixcloutier.com/x86/

Par exemple, l'instruction `pavgw xmm0,xmm1` calcule la moyenne par composantes de 16 bits, avec arrondi supérieur. Cela revient à additionner ces deux vecteurs puis à décaler chaque composante (`shr` de 1). L'instruction `pavgb xmm0,xmm1` fait de même, mais en considérant `xmm0` et `xmm1` comme des vecteurs dont les composantes sont 8-bits.
On retrouve ce niveau de spécialisation des circuit dans les GPU, *cf* par exemple les *tensor cores* de la micro-architecture `volta` des cartes `Nvidia`.[2]

En calcul vectoriel, il reste possible d'appliquer à chaque scalaire un flot non-linéaire mais les outils ont changé. En assembleur scalaire, on a vu que cela pouvait se faire au travers des `flags` et de sauts conditionnels, voire d'instructions de transfert conditionnel du type `movc`. En assembleur vectoriel, on peut n'appliquer une instruction qu'à certaines composantes par du *masquage*.

Par exemple, l'instruction `pcmpgtp` compare un vecteur source à un vecteur destination, composante par composante. Chaque composante de la destination est remplacée par $-1$ ou par 0, selon qu'elle était supérieure ou inférieure à la composante correspondante dans le vecteur source. Ce vecteur de 0 et $-1$ permet ensuite, par masquage via `pand`, de sélectionner l'une ou l'autre famille de composantes.

## 21.5 Exercices

Les exercices 1 et 4 demandent d'écrire des algorithmes vectorisés. On utilisera pour cela les conventions suivantes. On utilise des vecteurs pouvant contenir 4 entiers (qu'on appelle des *composantes*). Chaque composante est traitée comme un `int`. Il est inutile de déclarer les variables vecteurs. On autorise les opérations suivantes sur les vecteurs.

- initialiser un vecteur (ex : `u = 1,1,1,1` et `v = 0,-33,42,806`)
- copier un vecteur dans un autre (ex : `w = v`)
- charger un vecteur depuis 4 cases consécutives de la mémoire à une adresse donnée en précisant la taille des cases (ex : `w = (char) TAB[i:i+3]` indique que l'on charge les composantes de `w` par un octet chacune, lus à partir depuis l'adresse `TAB[i]`).
- charger un vecteur dans 4 cases consécutives de la mémoire à une adresse donnée en précisant la taille des cases (ex : `TAB[0:3] = (char) w` indique que l'on place les 4 composantes de `w` dans 4 cases mémoire consécutives de la taille d'un `char` à l'adresse `TAB`)
- additionner (ou soustraire, multiplier, diviser) deux vecteurs composante par composante (ex : `w = u + v`. Avec les valeurs précédentes, `w` vaut `1,-32,43,807`)
- faire la somme des 4 composantes d'un vecteur (ex : `int i = sum_comp(w)` ; avec les valeurs précédentes, on obtient que `i` vaut `819`)

**Exercice 21.1** ★★ On souhaite écrire un pseudo-code qui vectorise le code suivant :

2. https://devblogs.nvidia.com/cuda-9-features-revealed/

```
sum:
    test    edi, edi
    jle     .L9
    lea     eax, [rdi-4]
    lea     ecx, [rdi-1]
    shr     eax, 2
    add     eax, 1
    cmp     ecx, 8
    lea     edx, [0+rax*4]
    jbe     .L10
    pxor    xmm0, xmm0
    movdqa  xmm2, ... .LC1
    xor     ecx, ecx
    movdqa  xmm1, ... .LC0
.L4:
    add     ecx, 1
    paddd   xmm0, xmm1
    paddd   xmm1, xmm2
    cmp     eax, ecx
    ja      .L4
    movdqa  xmm1, xmm0
    cmp     edi, edx
    psrldq  xmm1, 8
    paddd   xmm0, xmm1
    movdqa  xmm1, xmm0
    psrldq  xmm1, 4
    paddd   xmm0, xmm1
    movd    eax, xmm0
    je      .L13
.L3:
    lea     ecx, [rdx+1]
    add     eax, edx
    cmp     edi, ecx
    jle     .L2
    add     eax, ecx
    lea     ecx, [rdx+2]
    cmp     edi, ecx
    jle     .L2
    add     eax, ecx
    lea     ecx, [rdx+3]
    cmp     edi, ecx
    jle     .L2
    add     eax, ecx
    lea     ecx, [rdx+4]
    cmp     edi, ecx
    jle     .L2
    add     eax, ecx
    lea     ecx, [rdx+5]
    cmp     edi, ecx
    jle     .L2
    add     eax, ecx
    lea     ecx, [rdx+6]
    cmp     edi, ecx
    jle     .L2
    add     eax, ecx
    lea     ecx, [rdx+7]
    cmp     edi, ecx
    jle     .L2
    add     eax, ecx
    add     edx, 8
    lea     ecx, [rax+rdx]
    cmp     edi, edx
    cmovg   eax, ecx
    ret
    .p2align 4,,10
    .p2align 3
.L9:
    xor     eax, eax
    .p2align 4,,10
    .p2align 3
.L2:
    rep     ret
    .p2align 4,,10
    .p2align 3
.L13:
    rep ret
    .p2align 4,,10
    .p2align 3
.L10:
    xor     edx, edx
    xor     eax, eax
    jmp     .L3
.LFE21:
    .size   sum, .-sum
...
.LC0:
    .long   0
    .long   1
    .long   2
    .long   3
    .align 16
.LC1:
    .long   4
    .long   4
    .long   4
    .long   4
    .align 16
```

FIGURE 21.1 – Code assembleur obtenu en `-O3` pour l'exercice 21.1.

```
1  int sum (int n){
2      int s = 0;
3      for(int i = 0; i < n; i++)
4        s+=i;
5      return s;
6  }
```

a. Supposons que `n` est un multiple de 4. Écrire l'algorithme vectorisé correspondant. Votre algorithme ne doit exécuter l'instruction `sum_comp` qu'un nombre constant de fois.

b. Comment fait-on si `n` n'est pas un multiple de 4 ?

c. La Figure 21.1 présente un exemple de code assembleur produit en compilant avec `gcc -O3` (certaines portions non-essentielles ont été abrégées pour une meilleure lisibilité). Retrouvez dans ce code les étapes de l'algorithme vectorisé que vous avez proposé aux questions précédentes.

Note : un descriptif complet des instructions x86 (instructions vectorielles comprises) est donné à https://www.felixcloutier.com/x86/

**Exercice 21.2 ★** Voici quelques boucles dont on se demande si elles sont vectorisables. Commencez par indiquer pour chacune d'entre elle si vous réussissez à la vectoriser à la main (comme à l'exercice 21.1). Ensuite, **et ensuite seulement**, vérifiez si `gcc` y parvient. Pour ces vérifications, compilez le code en `-O3`, prenez soin d'écrire les boucles dans des fonctions.

```
for(i = 0; i < n-1; i++)
  A[i]=B[i+1];
```

Vectorisable ? ☐ Oui ☐ Non
Vectorisée par `gcc` ? ☐ Oui ☐ Non

```
for(i = 1; i < n-1; i++){
  A[i]=B[i];
  A[i+1]=B[i+1];
}
```

Vectorisable ? ☐ Oui ☐ Non
Vectorisée par `gcc` ? ☐ Oui ☐ Non

```
for(i = 1; i < n-2; i+=2){
  A[i]=B[i];
  A[i+2]=B[i+2];
}
```

Vectorisable ? ☐ Oui ☐ Non
Vectorisée par `gcc` ? ☐ Oui ☐ Non

```
for(i = 1; i < n-1; i++){
  A[i]=B[i+1];
  B[i]=C[i];
}
```

Vectorisable ? ☐ Oui ☐ Non
Vectorisée par `gcc` ? ☐ Oui ☐ Non

Proposez une analyse de ces résultats.

**Exercice 21.3 ★★★**

a. Écrivez une fonction C qui calcule le minimum d'un tableau d'entiers donné en argument. Compilez cette fonction par `gcc -O3` et examinez l'assembleur obtenu.

Le code a-t-il été vectorisé ? ☐ Oui ☐ Non

b. Examinez ce que font les instructions vectorisées `pcmpgtd`, `pand`, `pandn`. Proposez un code assembleur qui prend en entrée deux registres `xmm0` et `xmm1`, considérés comme des vecteurs de doubles, et calcule leur minimum composante par composante.

c. Si le code obtenu à la question (a) est vectorisé, décortiquez-le et résumez-en les principes.

**Exercice 21.4 ★★** On s'intéresse dans cet exercice à la manière dont la vectorisation et la hiérarchie mémoire influencent le choix d'une méthode de conversion d'une image couleur en niveaux de gris.

**Codage des images.** Il est courant de décrire la couleur d'un pixel[3] par trois valeurs : une composante rouge, une composante verte et une composante bleue. Une norme largement utilisée actuellement, le *truecolor*, décrit chacune de ces trois composantes par un entier 8 bits (`char`). Ainsi, le triplet $(0, 0, 0)$ désigne le noir, le triplet $(255, 255, 255)$ désigne le blanc, et $(x, 0, 0)$ désigne un rouge qui est vif si $x$ est proche de 255 et sombre si $x$ est proche de 0.

**Niveaux de gris.** En *truecolor*, une couleur est un niveau de gris si les trois composantes sont égales. Pour convertir une image en niveau de gris, une méthode consiste à remplacer, pour chaque pixel, les trois composantes par leur moyenne : ainsi, on remplace le triplet $(50, 70, 180)$ par $(100, 100, 100)$ puisque $\frac{50+70+180}{3} = 100$.

3. Un pixel est un point d'une image. Une image en résolution 2000×1000 est composée de 2 millions de pixels.

**Structure de données.** On numérote les pixels de l'image ligne par ligne, en commençant en haut à gauche. On veut stocker une image de taille `LARGEUR`×`HAUTEUR`, donc on réserve le tableau suivant :

```
char* image = malloc(LARGEUR*HAUTEUR*3);
```

On envisage deux solutions pour organiser les informations dans le tableau :

▷ **La solution A** écrit les trois composantes de chaque pixel à la suite. Les composantes du $i^{\text{ème}}$ pixel sont donc `(image[3*i],image[3*i+1],image[3*i+2])`.

▷ **La solution B** divise `image` en trois sous-tableaux

```
char* rouge=image, vert=rouge+LARGEUR*HAUTEUR, bleu=vert+LARGEUR*HAUTEUR;
```

Les composantes du $i^{\text{ème}}$ pixel sont donc `(rouge[i],vert[i],bleu[i])`, c'est à dire `(image[i],image[i+LARGEUR*HAUTEUR],image[i+2*LARGEUR*HAUTEUR])`.

**Méthode de conversion.** Le code de conversion en niveaux de gris est :

```
char* dest = malloc(LARGEUR*HAUTEUR*3);
int i,moy;
char gris;
// U et V sont deux constantes définies ci-dessous

for (i=0; i<LARGEUR*HAUTEUR; i++){
  moy = (int)image[V*i] + (int)image[V*i+U] + (int)image[V*i+2*U]; // cast en int
  moy = moy/3;                                                     // pour éviter le % 256
  gris = (char)moy;                                                   // du calcul en char
  dest[V*i] = gris;
  dest[V*i+U]= gris;
  dest[V*i+2*U] = gris;
}
```

Les constantes `U` et `V` sont définies ainsi :

Solution A

```
#define U 1
#define V 3
```

Solution B

```
#define U LARGEUR*HAUTEUR
#define V 1
```

Maintenant, c'est à vous de jouer.

a. On exécute la conversion sur un système doté de deux niveaux de mémoire, la RAM et une mémoire cache de paramètres :

$$\text{taille} = 64 \text{ Ko}, \quad \text{lignes} = 64 \text{ octets}, \quad \text{associativité} = 1.$$

(a) On suppose que `LARGEUR` = 1600 et `HAUTEUR` = 1000. Combien de cache-miss est-ce que la conversion en niveau de gris déclenche (expliquer le calcul) ?

(b) On suppose que `LARGEUR` = 1024 et `HAUTEUR` = 1024. Combien de cache-miss est-ce que la conversion en niveau de gris déclenche (expliquer le calcul) ?

b. On modifie un seul paramètre de la mémoire cache : on suppose que l'associativité vaut 8, chaque sous-cache utilisant la stratégie de remplacement LRU (*least recently used*). On suppose que `LARGEUR` = 1024 et `HAUTEUR` = 1024. Combien de cache-miss est-ce que la conversion en niveau de gris déclenche (expliquer le calcul) ?

c. Pour les 2 solutions (A et B), proposez soit une version vectorisée en pseudo-code de la fonction de conversion en niveaux de gris, soit une explication de ce qui rend cela difficile.

# Annexe A

# Encodage de nombres à virgule

Cette annexe clarifie quelques notions liés aux nombres à virgule. Pour simplifier la présentation on ne s'intéresse qu'aux nombres positifs.

### A.0.1 Nombres représentables

Quelle que soit la convention de codage, comme il y a $2^b$ mots binaires distincts sur $b$ bits, on ne peut coder qu'au plus $2^b$ nombres à virgule distincts. Ces nombres sont dits *représentables* (par le codage en question). Le premier enjeu du choix de la représentation des nombres à virgule est de fixer l'ensemble des nombres représentables.

Les langages de programmation permettent généralement d'utiliser des nombres non-représentables dans le code source. À la compilation, ces nombres sont *arrondis* à un nombre représentable proche. Ainsi,

```
float a = 0.1;
```

crée une variable de type flottant et lui affecte la valeur représentable la plus proche de 0.1 (c'est à dire 0.1 si ce nombre est représentable et une approximation sinon).

### A.0.2 Virgule fixe

L'écriture décimale usuelle s'étend naturellement en binaire. Ainsi, comme on écrit $\pi \approx 3.14$ on peut écrire $\pi \approx (0011.0010\ 0100)_2$. Formellement,

$$(a_{k-1}a_{k-2}\dots a_0.a_{-1}a_{-2}\dots a_{-\ell})_2 \quad \text{code le nombre} \quad \sum_{i=-\ell}^{k-1} a_i 2^i.$$

On peut donc décider de représenter des nombres à virgule en consacrant par exemple $k$ bits à la partie entière et $\ell$ bits à la partie fractionnaire ; on parlera ici de format $k.\ell$ bits.

> Un nombre $x$ est représentable sur $k.\ell$ bits si et seulement si $2^\ell x$ est un entier de $\{0, 1, \dots, 2^{k+\ell} - 1\}$.

**Exercice A.1 ★★**

a. Prouver qu'aucun entier puissance de 2 n'est divisible par 10.

b. En déduire que le nombre décimal 0.1 n'est *pas* représentable sur $k.\ell$ bits, quels que soient $k$ et $\ell$.

Remarquons que l'on a là une alternative intéressante à l'algorithme de *division par puissances décroissantes* pour calculer la représentation *tronquée* d'un nombre $x$ quelconque sur $k.\ell$ bits : calculer $x' = \lfloor 2^\ell x \rfloor$, convertir $x'$ en binaire sur $k + \ell$ bits, puis décaler la virgule de $\ell$ positions vers la gauche.

Cela ne fonctionne que si l'on connaît à l'avance le nombre $\ell$ de bits dévolus à la partie fractionnaire. Avec cette restriction, cette méthode présente l'avantage sur la division par puissances décroissantes que la partie "résiduelle" est éliminée en début de conversion.

**Exercice A.2 ★★** Calculer l'approximation par tronquage de 0.1 en 8.8 bits.

### A.0.3 Notation scientifique

La *notation scientifique de base b* représente un nombre $x$ par une paire $(e, m)$ telle que $x = m * b^e$, avec $0 \leq m \leq b$. Vous êtes déjà familiers avec la notation scientifique de base 10, couramment utilisée en physique[1] ; les processeurs utilisent la notation scientifique de base 2 pour coder les nombres à virgule. On appelle $e$ l'*exposant* et $m$ la *mantisse*. Ici, la virgule est *flottante* car $e$ n'est pas constant (contrairement à ce que l'on a vu au paragraphe précédent).

La notation scientifique permet d'écrire de manière *compacte certains* grands nombres, mais pas tous. Prenons deux exemples d'écriture scientifique à base 2 de nombres 16 bits :

$$\begin{array}{lcll} (1000\ 0000\ 0000\ 0000)_2 & = & 1 * 2^{(1111)_2} & \rightarrow \quad ((1111)_2, (1)_2) \\ (1000\ 0100\ 0010\ 0001)_2 & = & 1.000\ 0100\ 0010 * 2^{(1111)_2} & \rightarrow \quad ((1111)_2, (1.000\ 0100\ 0010)_2) \end{array}$$

Dans les cas favorables, on peut représenter un nombre $x$ en utilisant de l'ordre de $\log \log x$ bits (contre $\approx \log x$ bits en codage binaire standard).

**Exercice A.3 ★★** Fixons deux entiers $k$ et $\ell$. On s'intéresse à la notation scientifique de base 2 utilisant $k$ bits d'exposant et $\ell$ bits de mantisse.

a. Quel est le plus grand entier représentable ?

b. Fixons un entier $x > 0$ inférieur au plus grand entier représentable. Notons $\hat{x}$ l'entier représentable le plus proche de $x$. Majorer l'*erreur relative* $\frac{|x-\hat{x}|}{\max(x,\hat{x})}$.

## A.1 Norme IEEE 754

Le déploiement de nombreuses convention de codage de nombres à virgule, et les problèmes de non-portabilité associés, ont conduit l'*Institute of Electrical and Electronics Engineers* à mettre au point une norme. La première version, de 1985, est suivie d'une révision substantielle en 2008 et d'une révision plus restreinte en 2019. Nous allons donner ici un aperçu de ce que spécifie cette norme (en nous basant principalement sur la version de 2008).

### A.1.1 Des formats

La norme IEEE 754 définit plusieurs formats standard d'encodage de nombres à virgule flottante. Ces formats contiennent des règles d'encodage détaillées et soigneusement optimisées (comme nous le verrons). La norme définit non seulement le format des nombres flottants, mais aussi les règles d'arrondi (*round to nearest, ties to even*), la représentation du zéro signé et des infinis, la gestion des exceptions (division par zéro, overflow...),

Les formats s'appuient sur une notation scientifique, certains formats étant à base binaire et d'autre à base décimale. On ne s'intéresse ici qu'aux formats à base binaire, les plus courants. La base décimale permet de rendre représentable tous les nombres décimaux de petite taille (quelques chiffres après la virgule, par exemple 0.1). Cela s'avère important pour certains domaine d'application tels que la finance ou la comptabilité, où l'on ne peut pas se permettre d'effectuer des arrondis en série.

1. $c \approx 3 * 10^8$ et $N_A \approx 6.022 * 10^{23}$.

Chaque format défini par la norme IEEE 754 suit la même règle : le mot binaire codant le flottant est divisé en trois zones stockant respectivement l'information de *signe* $S$, d'*exposant* $E$ et de *mantisse* $M$. Ces informations apparaissent dans cet ordre du bit le plus signifiant au moins signifiant :

| S (signe) | E (exposant biaisé) | M (mantisse) |
|---|---|---|

Quatre tailles de formats à base binaire sont définies, utilisant respectivement 16, 32, 64, 128 et 256 bits pour représenter un nombre (on parle de *semi / simple / double / quadruple / octuple précision*). La répartition des bits entre signe, exposant et mantisse est la suivante :

| | 16 bits | 32 bits | 64 bits | 128 bits | 256 bits |
|---|---|---|---|---|---|
| $S$ | 1 | 1 | 1 | 1 | 1 |
| $E$ | 5 | 8 | 11 | 15 | 19 |
| $M$ | 10 | 23 | 52 | 112 | 236 |

Ainsi, le flottant codé par $(1101\ 1000\ 0010\ 0000)_2$ a $S = 1$, $E = 1\ 0110$ et $M = 000\ 0010\ 0000$.

### A.1.2 Décodage (cas général)

Soit $w$ un mot binaire de champs $S$, $E$ et $M = (m_k m_{k-1} \dots m_0)_2$. Si $E$ vaut $(00\dots0)_2$ ou $(11\dots1)_2$ le décodage suit des règles spéciales (cf ci-dessous). Sinon, $w$ encode la valeur

$$(-1)^S 2^{E-\text{biais}} (1.m_k m_{k-1} \dots m_0)_2$$

avec

| | 16 bits | 32 bits | 64 bits | 128 bits | 256 bits |
|---|---|---|---|---|---|
| taille d'exposant | 5 | 8 | 11 | 15 | 19 |
| biais | 15 | 127 | 1023 | 16383 | 262143 |

Autrement dit, quand $E$ est codé sur $k$ bits la valeur de biais est $2^{k-1} - 1$. Soulignons que le codage des exposants négatifs ne se fait pas ici par complément à deux mais par un simple décalage.

Ce choix de biais assure que l'exposant est compris entre $-2^{k-1} + 1$ et $2^{k-1}$. Les valeurs $-2^{k-1} + 1$ et $2^{k-1}$ sont atteintes par les valeurs $E = (00\dots0)_2$ et $E = (11\dots1)_2$, mais dans certains cas ce sont des règles spéciales de décodage qui s'appliquent (cf ci-dessous).

Pour tous les nombres sauf 0, le premier bit significatif de la mantisse est 1. La norme choisit de rendre ce 1 implicite afin de gagner un bit d'encodage. Cela a pour conséquence que le 0 doit être codé par une règle spéciale. Les règles particulières de décodage sont les suivantes :

- $E = (00\dots0)_2$ et $M = 0 \rightarrow$ code $+0$ ou $-0$ selon $S$.
- $E = (11\dots1)_2$ et $M = 0 \rightarrow$ code $+\infty$ ou $-\infty$ selon $S$.
- $E = (00\dots0)_2$ et $M \neq 0 \rightarrow$ comme la règle générale mais remplacer $1.M$ par $0.M$.
- $E = (11\dots1)_2$ et $M \neq 0 \rightarrow$ code NaN (*not a number*).

Cela va sans dire que le calcul sur flottants demande la conception de circuits additionneurs (et multiplicateurs, diviseurs, etc.) spécifiques. La partie du processeur contenant ces circuits est parfois appelée FPU (*floating point unit*).

### A.1.3 Exercices

**Exercice A.4 ★** On s'intéresse ici au test d'égalité sur les flottants.

a. Écrire un programme C qui, pour chaque paire d'entiers $(i, j)$ ci dessous, initialise les trois variables `float` (en C) à

```
a = i/10.0,   b = j/10.0,   c = (i+j)/10.0
```

et affiche la valeur (`true` ou `false`) du test "`a+b == c`". Reportez vos observations ci-dessous

- (1,4) : _____
- (1,5) : _____
- (1,6) : _____
- (1,7) : _____
- (1,8) : _____
- (3,4) : _____
- (3,5) : _____
- (3,6) : _____
- (3,7) : _____
- (3,8) : _____

b. Pour comprendre, faire afficher les valeur de `a+b` et `c`. (Attention à ce que l'affichage ne limite pas le nombre de chiffres significatifs.)

c. Peut-on en déduire dans quel(s) cas on peut utiliser l'égalité entre flottants ?

d. Pour chaque paire $(i, j)$ qui échoue lors du test de la première question :

- déterminer lequel de `a+b` et de `c` est le plus grand,
- puis ajouter $10^{-6}$ à la fois à `a+b` et à `c`,
- et vérifier si l'ordre entre `a+b` et `c` a changé.

D'après vos observations, que peut-on espérer ?

**Exercice A.5 ★** Calculer les sommes $\sum_{i=1}^{k} \frac{1}{i}$ et $\sum_{i=k}^{1} \frac{1}{i}$ pour $k$ valant $10^3$, $10^6$, $10^9$, ...et comparer les résultats. (La première somme est calculée par indices croissants, la seconde par indices décroissants.)

**Exercice A.6 ★★** Cet exercice consiste à utiliser l'extrait de la documentation *754-2008 - IEEE Standard for Floating-Point Arithmetic*, disponible sur le ARCHE, pour décoder et encoder des nombres dans le format binaire double précision (qui correspond au type `float` en C).

a. Quelle est la taille mémoire d'un `float`, et quelles tailles font sa mantisse et son exposant ?

b. Donner en notation scientifique décimale la valeur du `float` codé par 0x 414B D000.

c. Donner la mantisse et l'exposant de l'encodage en `float` du nombre 0.1.