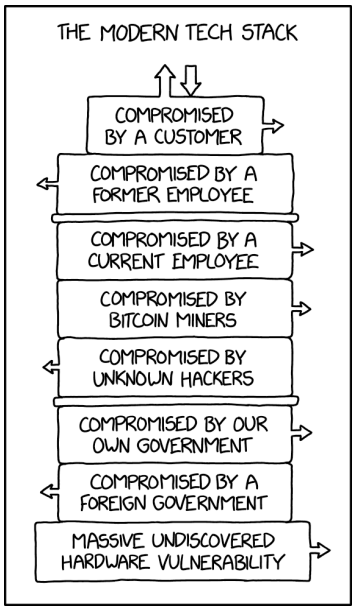
MINESnancy

Architecture des ordinateurs

Responsable du cours : Xavier Goaoc



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

FICM 2A Cours de département informatique 2022–2023

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.

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.

À chaque séance, vous travaillerez par groupes de 3 ou 4, en faisant régulièrement valider votre progression par l'enseignant.

Le programme du cours est le suivant.

- La séance 1 fait un rapide rattrapage de notions de base en architecture : circuits arithmétique, arithmétique entière naturelle et relative, arithmétique flottante et norme IEEE 754. Cette séance introduit plus de notions que les autres, mais ces notions sont plus élémentaires.
- La séance 2 introduit d'une part le modèle de *mémoire virtuelle* et d'autre part une méthodologie de chronométrage de code. On y met en évidence quelques contre-performances qui peuvent se produire lors de lectures/écritures en mémoire.
- La séance 3 présente deux modèles de *mémoire hiérarchique*. Le premier, à *associativité totale*, est rudimentaire mais permet d'expliquer les phénomènes observés à la 2ème séance. Le second, à *associativité partielle*, est plus précis et permet de rendre compte de certains phénomènes plus subtils mais néanmoins importants.
- La séance 4 est consacré aux principes de l'*assembleur x86*. L'objectif est d'en assimiler suffisamment pour pouvoir analyser le résultat de la compilation de petits codes par `gcc`.
- La séance 5 présente le principe de *pipeline* (au travers d'un modèle 5-niveaux) et aborde l'exécution spéculative au travers de la prédiction de branchement.
- La séance 6 examine de quelle manière les failles de sécurité Spectre et Meltdown tirent parti des comportement des hiérarchies mémoire et de la prédiction de branchement.
- La dernière séance introduit à la vectorisation d'algorithme et l'examen de codes vectorisés (toujours produits par `gcc`, sur intel 64 bits).

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

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 a été construit conjointement avec Carine Pivoteau, de l'Université Gustave Eiffel, et, pour le chapitre 6, avec Vincent Laporte, de l'INRIA Nancy Grand Est. Je tiens par ailleurs à remercier Jérémie Detrey dont les conseils, suggestions et explications m'ont été très précieux. Les erreurs, coquilles et autres approximations sont bien entendu entièrement de mon fait.

Table des matières

Chapitre 1

Du circuit à l'arithmétique

Commençons par une petite expérience (en langage C) :

- Déclarez `float a = 0.1, b = 0.2, c = 0.3` et examinez ce que vaut le booléen `a+b == c` (vrai ou faux ?).
- Réitérez l'expérience en déclarant les variables en `double` .
- Réitérez ces deux expériences avec `a = 1.1, b = 1.2, c = 2.3;`

Cela devrait mettre en évidence que le calcul en nombres à virgule flottante (on dira simplement *flottants*) est imprécis et, peut-être, surprenant. Cela peut avoir des conséquences spectaculaires :

- En 1991, naufrage de la plate-forme pétrolière Sleipner A (210m de fond) à cause d'approximation dans des calculs lors de la conception (coût : 700 M$). [1]
- En 1996, Ariane 5 est détruite peu après le décollage à cause d'une conversion `double` (64 bits) → `int` (16 bits) qui conduit à interpréter comme négative une correction voulue positive (coût : 500 M$). [2]

Pour expliquer, prévoir ou éviter ce phénomène, il convient de comprendre de quelle manière le processeur calcule au niveau des *circuits logiques*. Nous verrons ainsi comment les choix et limitations techniques bas-niveau se répercutent à un plus haut niveau, par exemple dans les langages de programmation C ou python.

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 à virgule et sachiez expliquer le comportement d'un calcul numérique en nombre entiers ou en nombres flottants.
- Compreniez la manière dont les nombres entiers et flottants sont stockés en mémoire et sachiez y accéder.

1.1 Circuits

Commençons par examiner les principes qui sous-tendent l'automatisation du calcul, et donc la manière dont un processeur calcule.

1. https://en.wikipedia.org/wiki/Sleipner_A
2. https://en.wikipedia.org/wiki/Ariane_5

1.1.1 Binaire et booléen

La première idée sur laquelle les processeurs sont basés est la suivante :

> Il est possible de traduire l'arithmétique binaire en calcul booléen.

Explicitons cela.

D'une part, 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*).

D'autre part, 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 ★ 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 $\{0,1\} \to$ {`vrai, faux`} » une *fonction booléenne*.

À ce stade, on a simplement défini des règles de calcul sur $\{0,1\}$ par interprétation (via une bijection $\{0,1\} \to$ {`vrai, faux`}) des règles de calcul booléen. Il s'avère que les fonctions booléennes ainsi obtenues 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 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.

La deuxième idée pour mécaniser le calcul est la suivante :

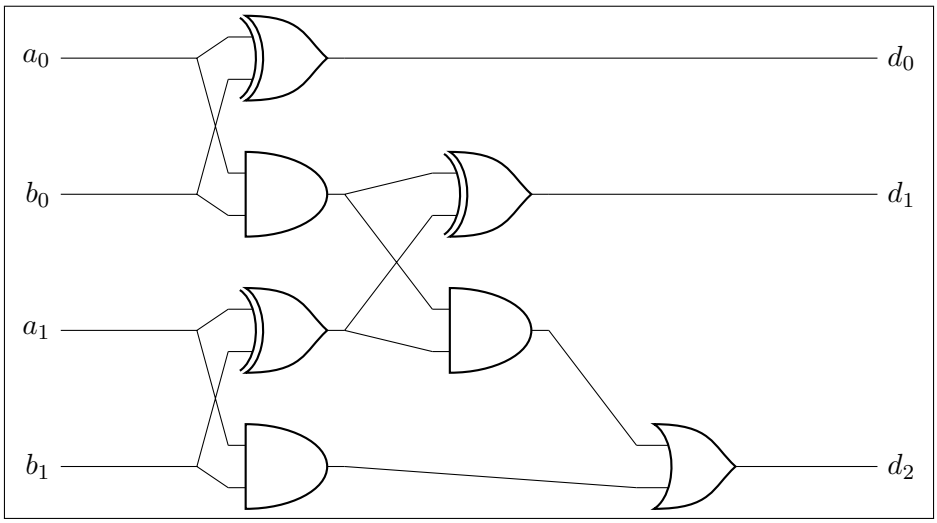
Pour toute expression de calcul booléen, il existe un système physique qui l'évalue.

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).

1.1.2 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 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 soit.

Exercice 3 ★★ Le circuit additionneur dessiné ci-dessus comporte des croisements, ce qui peut s'avérer gênant pour certains modes de réalisation (par exemple par circuit imprimé). 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 ★★★ 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.

Nous n'en dirons pas plus dans ce cours sur les circuits. Pour les volontaires souhaitant prolonger, un exercice intéressant consiste à concevoir, de la même manière, un *multiplexeur*, qui prends en entrée des signaux numérotés $1, 2, \ldots$ et k *bits de sélection* $a_0, a_1, \ldots a_{k-1}$, et produit en sortie le signal $N^o\ (a_{k-1} a_{k-2} \ldots a_0)_2$. Le nombre de signaux d'entrée est au plus 2^k ; s'il est inférieur à 2^k, on n'utilise qu'une partie de l'adressage offert par les k bits de sélection. À l'inverse, le *dé-multiplexeur* prends en entrée un signal et des bits de sélection, est connecté à plusieurs sorties, et dirige le signal vers la sortie indiquée par les bits de sélection.

1.2 Calcul entier

Les principes vus en Section 1.1 guident la réalisation du circuit arithmétique d'un processeur classique. Sans entrer dans les nombreuses complications techniques [3], nous allons en tirer quelques contraintes générales qui s'imposent à *tout* [4] processeur calculant en entiers.

1.2.1 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.

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

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.

Exercice 5 ★

a. Quelle est la taille, en bits, des types `int`, `short` et `char` en langage C sur votre machine ? *Indication : on pourra utiliser l'instruction* `sizeof`.

b. Proposez et réalisez, pour un de ces types, une expérience testant que l'addition est bien faite modulo 2^T où T est sa taille.

1.2.2 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 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 6 ★ 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 $a_{b-2}a_{b-3}\dots a_0$, 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.)

Exercice 7 ★★

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 ?

Nous verrons que le codage par bit de signe est utilisé pour les nombres à virgule, pour lesquels il faut de toute façon refaire les circuits.

Concluons en observant 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.

Exercice 8 ★★ Prouver que dans la règle du complément à deux, le bit de poids fort du codage d'un entier relatif x vaut 0 si $x \geq 0$, et 1 si $x < 0$.

1.2.3 Le typage est une vue de l'esprit (ou du compilateur)

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.

Exercice 9 ★★ Exécutez (séparément) chacun des code suivants (qui supposent chargées les bibliothèques `stdio.h` et `string.h`) et expliquez ce qu'ils font.

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

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

```
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);
```

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.

Exercice 10 ★★

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.

1.3 Encodage de nombres à virgule

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

1.3.1 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).

1.3.2 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 ℓ 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 11 ★★

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 ℓ.

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 ℓ positions vers la gauche. Cela ne fonctionne que si l'on connaît à l'avance le nombre ℓ 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 12 ★★ Calculer l'approximation par tronquage de 0.1 en 8.8 bits.

1.3.3 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[5] ; 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 :

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

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).

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

Exercice 13 ★★ Fixons deux entiers k et ℓ. On s'intéresse à la notation scientifique de base 2 utilisant k bits d'exposant et ℓ 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* $\frac{|x-\hat{x}|}{\max(x,\hat{x})}$.

1.4 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).

1.4.1 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.

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	E	M
(signe)	(exposant biaisé)	(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 $M = 000\ 0010\ 0000$.

1.4.2 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 $M = 0 \to$ code $+0$ ou -0 selon S.
- $E = (11\dots1)_2$ et $M = 0 \to$ code $+\infty$ ou $-\infty$ selon S.
- $E = (00\dots0)_2$ et $M \neq 0 \to$ comme la règle générale mais remplacer $1.M$ par $0.M$.
- $E = (11\dots1)_2$ et $M \neq 0 \to$ 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*).

1.4.3 Exercices

Exercice 14 ★ 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 15 ★ 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 16 ★★ 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.

Chapitre 2

Modèle de mémoire virtuelle et chronométrage

Ce chapitre met en place deux éléments fondamentaux, et indépendants, pour la suite du cours.

D'une part, nous clarifions 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 au chapitre suivant.

D'autre part, nous donnons 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 analyser une séquence d'accès mémoire par pointeurs.
- Sachiez chronométrer de manière précise une portion de code C.

2.1 Modèle de mémoire virtuelle

Commençons par donner un modèle assez simpliste du fonctionnement de la mémoire.

2.1.1 La mémoire est un tableau

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*.

En C, un pointeur est une variable interprétée comme une adresse. Ainsi, déclarer `int* p` crée une variable `p` dont la valeur est interprétée comme l'adresse d'une variable de type `int`.

Il est conseillé pour ce cours d'avoir les idées claires sur l'arithmétique des pointeurs. Par exemple, comment la valeur de `p` change-t-elle lorsque l'on fait `p++` ?

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.

> 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.

2.1.2 Allocation mémoire : deux mécanismes

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.

Sur la pile (« stack »). On peut réserver de l'espace mémoire par *déclaration de variable*, et en particulier de tableaux. Par exemple :

```
int  tab[100];
```

Cette commande 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.

Sur le tas (« heap »). 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 la commande ci-dessus est :

```
#include <stdlib.h>

int*  tab;
t = 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.

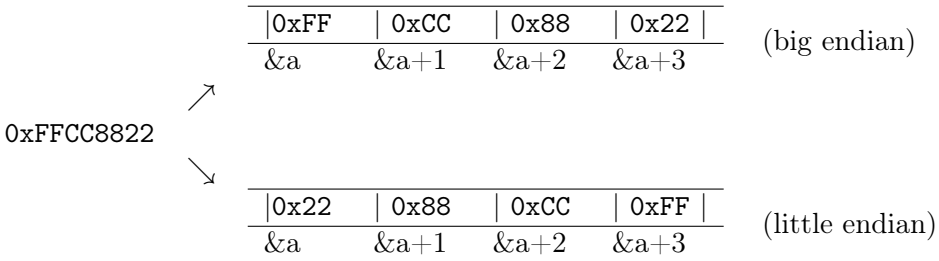
> Dorénavant, toute réservation mémoire d'un tableau « un peu grand » doit être faite sur le tas.

2.1.3 Endianness

Une donnée qui fait plus d'un octet doit occuper plusieurs cases mémoires. Supposons que sur notre machine un `int` occupe 32 bits (on peut le vérifier par exemple avec `sizeof()`). À l'issue de la déclaration

```
int  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 :



Une manière simple de déterminer l'*endianness* courante consiste à vérifier quel est le premier octet :

```
int  a=0xFFCC8822;
char* p = (char*)&a;

if ((*p) == 0xFF) printf("big endian\n");
else if ((*p) == 0x22) printf("little endian\n");
else printf("???\n");
```

2.1.4 Exercice

Exercice 1 ★ Vérifiez expérimentalement :

a. ce qui se passe sur votre machine/système lorsque l'on essaye de réserver par `int tab[...]` un tableau dont la taille excède l'espace disponible sur la pile,

b. l'ordre de grandeur de la taille disponible sur la pile de votre programme,

c. que vous arrivez à réserver sur le tas un tableau plus grand que ce que vous pouvez réserver sur la pile,

d. l'endianness utilisée par votre machine/système,

e. la manière dont sont implémentés les tableaux multidimensionnels du type `int tab[10][10][10]`.

2.2 Chronométrage

Passons maintenant à la méthodologie de chronométrage utilisée dans ce cours.

2.2.1 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 (int 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];
```

2.2.2 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]

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

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

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>

unsigned long int test(int MAX){
  unsigned long int tic,toc;
  unsigned int ui;

  tic = __rdtscp(&ui);
  for (int i=0; i < MAX; ++i)
    a = a*a;
  toc = __rdtscp(&ui);

  return toc-tic;
}
```

Deux précisions :

a. On ne fera généralement rien du résultat écrit dans la variable `ui`. Il faut néanmoins la fournir à la fonction `__rdtscp()`.

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.

2.2.3 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 le *temps moyen* d'exécution sur *de nombreuses exécutions*.

2.2.4 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 linux, l'outil `perf` 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*, en accès libre, explique comment diminuer des erreurs de mesures liées au fonctionnement du processeur. Nous en recommandons la lecture attentive après les séances 4 (assembleur) et 5 (pipeline).

2.3 Exercices

Exercice 2 ★★ Écrivez une fonction qui calcule la somme des carrés des entiers de 1 jusqu'à n, où n est un paramètre. Chronométrez cette fonction pour différentes valeurs de n :

n	1 000	10 000	10^6	10^7	10^8
mesure					

Exercice 3 ★★ 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){
2   // ...
3 }
```

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
3    // votre code peut appeler la fonction à mesurer
4    // par ma_fonction(mon_arg)
5    // il doit afficher (printf) le résultat de la mesure
6
7 }
```

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`.

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 4 ★★ 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 5 ★★ 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` ≤ 1000.

Chapitre 3

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 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.

3.1 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.

3.2 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 quelques 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 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.

3.3 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*.

3.3.1 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.

3.3.2 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.

3.3.3 Exercices

Exercice 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 2 ★★ Proposez une explication du phénomène observé à l'exercice 7 du chapitre précédent.

3.4 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.

3.4.1 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 présentées en Section 3.5.

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]

3.4.2 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 :

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).

- 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 3.1).
- Le niveau L3 est partagé entre plusieurs cœurs.

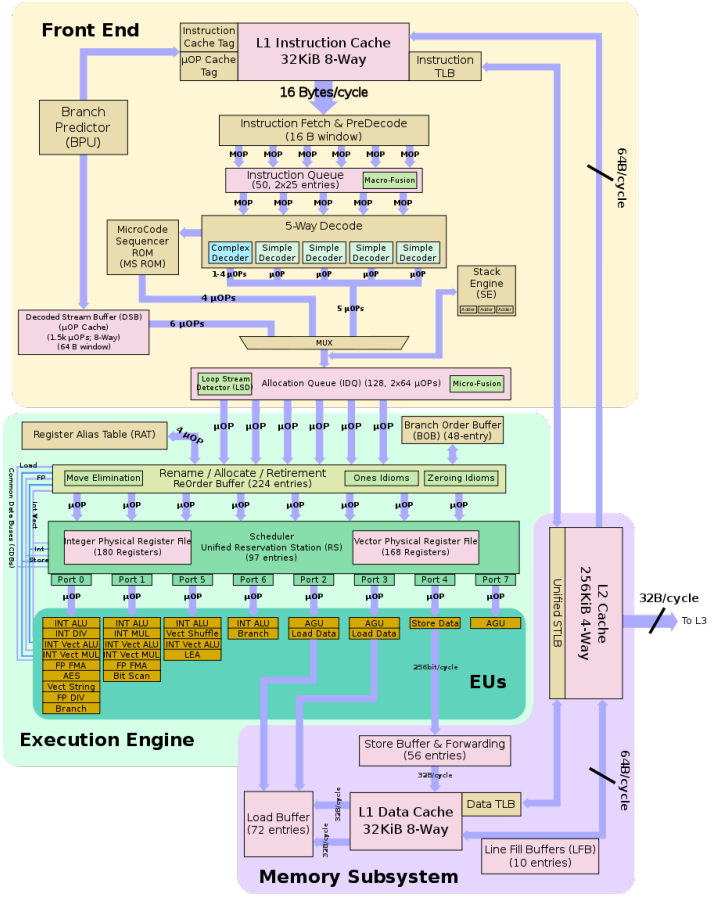


FIGURE 3.1 – Diagramme d'un cœur dans l'architecture Intel Skylake. Source : https://en.wikichip.org/wiki/WikiChip.

3.4.3 Exercices

Exercice 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 (laisser la 3ème ligne vide pour l'instant) :

	L1 instruction	L1 données	L2	L3
Taille mémoire				
Ligne de cache				

Exercice 4 ★★ On construit une liste chaînée dont les éléments sont définis comme suit :

```
struct noeud {
   int  valeur;
   struct noeud* prec;
   struct noeud* suiv;
};
```

On travaille sur une machine sur laquelle un `int` occupe 4 octets et un pointeur occupe 8 octets. Une structure `noeud` occupe donc 20 octets.

Supposons que l'on construise notre liste incrémentalement, en réservant l'espace mémoire par paquets de 10 cellules et en créant les nœuds dans un ordre qui ne correspond pas à leur ordre final. La flexibilité des listes doublement chaînée rend tout cela transparent pour le programmeur, puisque la place d'une structure dans la liste ne dépend pas de sa position en mémoire, mais simplement des pointeurs définissant le « chaînage ». Après insertion de la 30ème cellule on se retrouve avec l'état mémoire suivant (p est la position du nœud dans la liste, `adresse` est l'adresse du premier octet de ce nœud) :

adresse	p	adresse	p	adresse	p
0xAE01D500	2	0xAE02D500	5	0xAE03D500	4
0xAE01D514	7	0xAE02D514	1	0xAE03D514	10
0xAE01D528	3	0xAE02D528	15	0xAE03D528	8
0xAE01D53C	9	0xAE02D53C	16	0xAE03D53C	20
0xAE01D550	13	0xAE02D550	6	0xAE03D550	14
0xAE01D564	11	0xAE02D564	12	0xAE03D564	17
0xAE01D578	21	0xAE02D578	26	0xAE03D578	23
0xAE01D58C	25	0xAE02D58C	22	0xAE03D58C	30
0xAE01D5A0	28	0xAE02D5A0	19	0xAE03D5A0	24
0xAE01D5B4	18	0xAE02D5B4	27	0xAE03D5B4	29

a. La taille de ligne de cache sur le système est de $b = 64$ octets. Donner la position d'une cellule qui est stockée à cheval sur deux blocs.

b. Supposons que l'on travaille sur un système à 2 niveaux de mémoire pour lequel la mémoire rapide est de 32Ko, à associativité complète et à pagination LRU. On parcours la liste de la cellule 1 à la cellule 30 par positions croissantes (1, puis 2, puis 3, ...), en accédant à chaque fois à la valeur et au pointeur sur le nœud suivant. Quelles sont les cellules dont la traversée provoque un accès hors-cache ?

Exercice 5 ★★ Maintenant, 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

```
1  int *tab = malloc(N*N*sizeof(int));
```

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 :

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

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

Au vu des paramètres de la hiérarchie mémoire de votre machine, à quelles valeurs de `N` vous attendez-vous à voir un effet de seuil ?
Vérifiez cela expérimentalement. Si ce n'est pas concluant, cherchez des valeurs seuil expérimentalement et mesurez les temps pris au voisinage de ces seuils. **Ne remplissez le tableau ci-dessous qu'avec des mesures concluantes.**

`N`						
parcours en ligne						
parcours en colonne						

3.5 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 7 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 ℓ bits (où 2^ℓ est le nombre de sous-caches).

3.5.1 Exercices

Exercice 6 ★ 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 3.

Exercice 7 ★★ Reprendre la question (b) de l'exercice 4 en supposant que le cache est d'associativité 2.

Exercice 8 ★★★ 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 ?

3.6 Exercice optionnel : un peu d'ingénierie algorithmique

Le modèle complètement associatif de hiérarchie mémoire suffit déjà à guider la conception d'algorithmes bien plus efficace pour l'architecture considérée. L'exercice qui suit propose d'illustrer cela sur l'exemple de la *transposition de matrice*. Remarquons que d'un point de vue algorithmique, il n'y a rien de bien intéressant à dire : il faut accéder à chaque case mémoire au moins une fois, et une fois suffit. Il s'avère que l'*ordre* dans lequel on accède à ces cases mémoire a des conséquences importantes sur les performances... et qu'il est facile de décrire *récursivement* un bon ordre.

Exercice 9 ★★★ Cet exercice s'intéresse à un algorithme efficace de transposition de matrice. Le point important est que l'on va concevoir l'algorithme de sorte à ce qu'il tire au mieux parti de la hiérarchie mémoire.

Dans cet exercice, une matrice $n \times m$ est interprétée comme un tableau 2D à n lignes et m colonnes et est stockée en mémoire dans un tableau (1D) de `n*m int`, avec la convention que l'élément en $i$ème ligne et $j$ème colonne est stocké à l'indice $i * n + j$ (comme à l'exercice 4). On décrira systématiquement une matrice par la donnée de ses dimensions (`int n` et `int m`) et du tableau 1D (`int *`).

On rappelle que *transposer* une matrice `(n,m,mat)` revient à échanger les entrées `mat(i,j)` et `mat(j,i)` pour toutes les paires d'indices i, j où $i < j$.

a. Écrire une fonction qui initialise une matrice par des entiers aléatoires entre 0 et 99.

b. Écrire une fonction qui transpose une matrice $n*n$ par une simple double-boucle sur les paires (i, j) avec $1 \leq i < j \leq n$. Mesurez ses temps d'exécution pour différentes tailles de matrice. [2]

`n`	1 024	4 096	8 192	16 384	65 536
temps					

c. Soit $M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$ une matrice $m \times n$, où A, B, C et D sont des matrices (ce sont des *sous-matrices* de M). On rappelle que $M^t = \begin{pmatrix} A^t & C^t \\ B^t & D^t \end{pmatrix}$, ce qui permet de réduire la transposition d'une grande matrice à celle de petites matrices. Écrire une nouvelle fonction de transposition qui se base sur cette idée et mesurez ses temps d'exécution pour différentes tailles de matrice.

`n`	1 024	4 096	8 192	16 384	65 536
temps					

2. Comme on ne s'intéresse qu'aux variations relatives de ces temps, normalisez-les de manière à rendre l'ensemble le plus lisible possible.

d. Étendez cette idée récursivement. Mesurez les temps d'exécution de votre fonction pour différentes tailles de matrice.

`n`	1 024	4 096	8 192	16 384	65 536
temps					

Chapitre 4

Notions d'assembleur x86

Lors de cette 4ème séance, nous allons examiner le code produit par `gcc` et analyser certaines des optimisations qu'il effectue. Ce code produit est de l'assembleur (`x86` dans notre cas), aussi il nous faut nous familiariser avec ce langage, suffisamment pour pouvoir *lire* un code assembleur. Ce chapitre propose une introduction rapide à l'assembleur et est à considérer comme un « kit de survie » pour la 4ème séance.

Travail personnel à faire **avant la séance** : lire les sections 4.1, 4.2 et 4.3 (soit 6 pages), faire les exercices 1, 2 et 3 (prévoir $\sim$ 15 minutes) et saisir les réponses à ces exercices dans le questionnaire prévu à cet effet sur la page Arche du cours.

Objectifs. À l'issue de cette séance, il est attendu que vous sachiez lire et analyser un code assembleur simple.

4.1 Généralités

Assembleur. 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).

Voici par exemple un code C et l'assembleur produit par `gcc` à sa compilation :

```c
int main(){
  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
```

Langage machine. 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. Lorsque le CPU exécute un programme, il charge ces nombres depuis la mémoire les uns après les autres, les décode, et agit en conséquence.

Par exemple, la traduction en langage machine du code assembleur ci-dessus est :

```
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
```

Convention Intel VS AT&T. L'assembleur permet d'écrire du code facilement lisible par un humain et directement traduisible en langage machine : chaque instruction assembleur correspond de manière non-ambigüe à un code du langage machine. 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 décrit l'opération de copie du registre `rax` dans le registre `rbx`. Selon la syntaxe choisie, l'instruction `mov rax,rbx` correspond à différents codes en langage machine.

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

4.2 Obtention du code assembleur

On se contentera en TP d'étudier l'assembleur produit par `gcc` à la compilation de code C. Voici deux méthodes.

Par gcc. On peut demander à gcc de produire un fichier texte contenant le code assembleur qu'il produit. Cela se fait avec l'option `-S`. Par exemple

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

produit un fichier `test.s` contenant le code assembleur produit par gcc. L'option `-masm=intel` indique que l'on souhaite utiliser la convention Intel.

Par objdump. 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.

Pour examiner un programme créé en compilant du code C par `gcc`, il est conseillé d'appliquer `objdump` (avec les mêmes options que ci-dessus) au fichier `.o` plutôt qu'à l'exécutable. (Rappel : pour faire créer à `gcc` un fichier objet (`.o`), il faut compiler avec l'option `-c`.)

> **Exercice 1 ★** Récupérez le code assembleur du programme suivant par *chacune* des méthodes décrites ci-dessus.

```c
int sum (int count){
  int s = 0;
  int i;
  for(i = 0; i < count; i++)
    s+=i;
  return s;
}

int main() {
  int count = 100000000;
  sum(count);
}
```

4.3 Quelques notions d'assembleur

Les quelques pages qui suivent font un tour d'horizon succinct de quelques notions de base d'assembleur `x86`. L'objectif est ici d'être capable d'analyser 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/

4.3.1 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`).

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 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`.

> **Exercice 2 ★★** Reprenons le code assembleur que vous avez obtenu à l'exercice 1. Comment sont stockées les variables (registre, RAM, autre) ?

4.3.2 Gestion 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 mets CF à 1 si un dépassement de capacité s'est produit en non-signé mets OF à 1 si un dépassement de capacité s'est produit en signé ...

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 3 ★★ Reprenons le code assembleur que vous avez obtenu à l'exercice 1.

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 `sum` ?

c. Comment le paramètre (`count`) est-il transmis ?

d. Comment `sum` transmet-elle sa valeur de retour ?

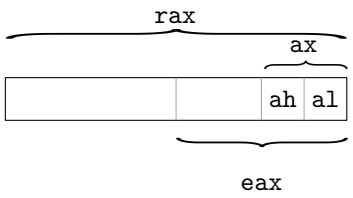
4.3.3 Plus en détail

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.

`mov`	copie de donnée
`add, sub, inc, dec`	opérations arithmétiques
`and, or, xor, not`	opérations logiques (bit à bit)
`ror, rol, shr, shl` `sar, sal`	opération de décalage et rotation (bit à bit)
`mul, imul, div, idiv`	multiplication et division
`call, ret, jmp`	saut/retour

`jz, jnz, jc, jnc`
`jg, jng, jge, jnge`
`...`
	sauts conditionnels, voir https://www.felixcloutier.com/x86/jcc

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.

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.

4.4 Exercices

Exercice 4 ★ 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 5 ★ 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
```

Exercice 6 ★ Dans cet exercice, nous allons examiner la traduction en langage machine du programme suivant :

```
1 int main() {
2   int i,j=0;
3   for (i=0; i<100; i++)
4     j += i;
5 }
```

a. Compilez le programme afin d'en faire un objet (option -c). Visualisez le contenu du fichier `.o` obtenu grâce à la commande `objdump` (options `-d` pour désassembler et `-M intel` pour utiliser la convention intel).

b. Déterminez le nombre d'instructions assembleur et le nombre d'octets occupés en mémoire par le programme.

c. Donnez le code en langage machine de l'instruction `mov rbp,rsp`

d. Quelle(s) instructions occupent le plus d'espace mémoire et pourquoi ?

Exercice 7 ★★ Reprenez le code C de l'exercice 1 et générez le code assembleur, mais cette fois-ci, en compilant avec l'option d'optimisation `-O1` (attention, c'est un 'O' majuscule, pas un zéro).

a. Dans la fonction `sum`, comment sont stockées les variables (registre, RAM, autre) ? Quel est l'intérêt de ce changement ?

b. Observez la partie du code qui correspond au `main`... Que s'est-il passé ?

c. Modifiez le code en C pour que la fonction `sum` soit effectivement exécutée.

d. Comparez le reste du code avec ce que l'on avait obtenu sans optimisation, notez les différences principales (au moins 3) et essayer de les expliquer.

e. Même question en utilisant l'optimisation `-O2` (au moins deux différences).

f. Changez la valeur de `count` pour une valeur (beaucoup) plus petite et compiler avec `-O2`. Qu'observe-t-on dans le code assembleur ?

g. Modifiez la fonction `sum` pour faire une somme de carrés (ou cubes, puissances quatrièmes, ...) et compiler avec `-O2`. Qu'observe-t-on dans le code assembleur ?

Exercice 8 ★★ Écrivez un code assembleur `x86` réalisant les tâches suivantes. Dans chaque cas, on prendra soin de ne modifier à terme que les registres qu'il est demandé de mettre à jour.

a. mettre dans `rax` le minimum de 4∗`rax`+5 et de 8∗`rbx`+7,

b. ajouter `rbx` à `rax` tant que le résultat est inférieur à `rcx`,

c. mettre `rcx` à 1 si `rax` est un multiple de 4 et à 0 sinon.

Exercice 9 ★★★ (Optionnel) Écrivez un code assembleur qui prend en entrée un entier a, qui calcule la suite de Collatz à partir de a,

$$u_0 = a \quad \text{et} \quad u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}$$

et qui termine quand u_n vaut 1.

Exercice 10 ★★★ (Optionnel) Examinons maintenant ce que devient un code récursif en assembleur.

a. On considère le code suivant :

```
int fac(int n){ return (n<=1) ? 1 : n * fac(n-1); }
```

b. Générez le code assembleur de cette fonction, en compilant avec l'option `-O1` puis avec l'option `-O2` et comparer.

   (a) Quelle est la principale différence entre les deux optimisations ? Quel est l'intérêt ?

c. Refaire les même tests avec la fonction qui calcule les nombres de Fibonacci :

```
int fib(int n){ return (n<=2) ? 1 : fib(n-1) + fib(n-2); }
```

d. Refaire les même tests avec la fonction suivante, qui calcule *aussi* les nombres de Fibonacci :

```
int fib2_rec(int n, int acc1, int acc2){
  return (n<=2) ? acc2 : fib2_rec(n-1, acc2, acc1 + acc2);
}

int fib2(int n){ return fib2_rec(n,1,1); }
```

e. Quelle est la différence entre les fonctions `fib` et `fib2` du point de vue de la récursivité ?

Chapitre 5

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.

5.1 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.

5.2 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
```

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.

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.

5.3 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 ℓ, il est naturel que le tampon soit de taille 2ℓ et que le prefetch se fasse par tranche de ℓ octets.
- 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.

5.4 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'execution 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
```

5.4.1 Exercices

Exercice 1 ★ On considère le code suivant :

```
1            mov   ecx, 1000
2            mov   eax, 0
3 .boucle:   mov   ebx,DWORD PTR [edx+4*ecx]
4            cmp   ebx,0
5            gle   .negat
6            add   eax,ebx
7 .negat:    dec   ecx
8            jnz .boucle
```

a. Que fait ce code ?
b. Identifiez les instructions qui occasionnent une bulle sur un pipeline 5 niveaux.
c. Réordonnez ces instructions pour éliminer les bulles sans changer le résultat.

Exercice 2 ★★ Dans cet exercice, on s'intéresse à la manière dont le compilateur `gcc` optimise le code pour réduire les bulles dans le pipeline. On prendra comme exemple le code C suivant :

```c
#include <stdio.h>
int main() {
  int i, j=0, k=0, l=0, res=0;
  for (i=1; i<10; i++){
    j+=i*i*i*i;
    k+=j*j*j*j;
    l+=j*j*k*k;
    res+=j/k;
    res+=l;
  }
  printf("%d\n",res); return 0;
}
```

Compilez ce code en `-O1` et `-O2` et visualisez, côte à côte, les fichiers .s après les avoir nettoyés pour ne garder que les instructions. Identifiez trois différences dans le code que vous savez expliquer par la réduction de bulle.

5.5 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.

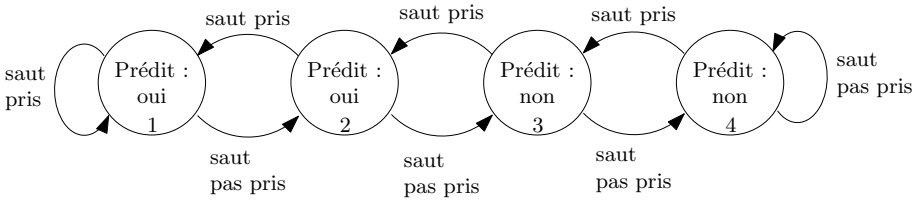
Lorsque le processeur atteint à nouveau cette instruction de saut conditionnel, il peut utiliser le bit de poids fort de ce compteur comme un prédicteur du comportement : s'il est à 1 il parie que le saut sera pris, s'il est à 0 il parie que le saut ne sera pas pris.

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.

5.5.1 Exercices

Exercice 3 ★★ 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.

Exercice 4 ★★ Cet exercice vise à mettre en évidence l'influence du pipeline sur l'exécution d'une boucle comportant un saut conditionnel.

Expérience. Initialisons un tableau de N$\approx 10^7$ entiers aléatoires choisis uniformément entre 0 et 99. Écrivons une fonction C qui compte les nombres inférieurs à un seuil, donné en paramètre, dans un tableau d'entiers. Mesurons le temps pris par notre fonction sur ce tableau lorsque le seuil varie de 0 à 100.

Observation. On devrait constater que le temps pris par la boucle augmente linéairement quand le seuil varie de 0 à 50, puis diminue linéairement quand le seuil varie de 50 à 100.

a. Réalisez cette expérience (en pensant à compiler sans optimisation, *i.e.* en `-O0`). Si les résultats observés dévient substantiellement de ceux annoncés, vérifiez le code machine produit par `gcc`.
   (a) Votre expérience est-elle concluante ?
   (b) Quel facteur multiplicatif mesurez-vous entre les temps minimum et maximum pris par la fonction ?

b. Comment expliquez-vous le phénomène observé ?

c. Ce phénomène persiste-t-il si on compile en `-O1` ? Examinez le code assembleur produit par `gcc`. Comment expliquez-vous cela ?

d. Modifiez votre code pour compter 5 fois chaque entier inférieur au seuil en augmentant le compteur de "+5" au lieu de "+1" à chaque test réussi. Réexaminez le résultat de la compilation en `-O1`.

Exercice 5 ★★ On s'intéresse maintenant aux sauts conditionnels imbriqués.

a. Expliquez comment on peut réaliser, en assembleur, un test composite de la forme

```
if ((tab[i]>24) && (tab[i]<51)) ...
```

Réalisons maintenant l'expérience suivante :

Expérience. Initialisons un tableau de N$\approx 10^7$ entiers aléatoires choisis uniformément entre 1 et 100. Écrivons deux fonctions C qui comptent les entrées de ce tableau dans l'intervalle $[25, 50]$ au moyen de tests de la forme :

```
if (v>24)
  if (v<51)
    j++;
```

et

```
if (v<51)
  if (v>24)
    j++;
```

Mesurons le temps pris par ces fonctions.

Observation. L'une devrait être sensiblement plus rapide que l'autre.

b. Réalisez cette expérience (en compilant en `-O0 -falign-functions`). Votre expérience est-elle concluante ? Si les résultats observés dévient substantiellement de ceux annoncés, vérifiez le code machine produit par `gcc`.

c. Quel facteur multiplicatif mesurez-vous entre les temps de ces deux fonctions ?

d. Comment expliquez-vous cette différence de rapidité ?

e. Ce phénomène disparaît-il si on compile en `-O1` ou `-O2` ? Comment cela s'explique-t-il ?

5.6 Exercice optionnel : un peu d'ingénierie algorithmique

Tout comme le modèle de mémoire hiérarchique, les modèles simples de prédiction de branchement permettent de guider la conception d'algorithmes adaptés à une architecture donnée. L'article de recherche *Good Predictions Are Worth a Few Comparisons* d'Auger, Nicaud et Pivoteau, disponible librement à

https://drops.dagstuhl.de/opus/volltexte/2016/5713/

propose notamment l'affirmation suivante : une recherche dichotomique légèrement *biaisée* est plus rapide qu'une recherche dichotomique équilibrée.

Exercice 6 ★★★ Pour commencer, **ne pas** lire l'article en question.

a. Vérifiez expérimentalement la validité de l'affirmation ci-dessus sur votre machine. Pour cela,
   - reprenez le protocole de chronométrage de recherches dichotomique utilisé à l'exercice 8 du chapitre 3,
   - fixez une valeur de N grande, mais ne donnant pas lieu à des collision pour cause d'associativité partielle,
   - mesurez le temps pris pour une même séquence de recherches au moyen d'une recherche dichotomique classique et de la BiasedBinarySearch proposée par l'article (page 9).

b. Proposez une analyse de ce phénomène.

c. Parcourez maintenant la section 5 de l'article et comparez votre analyse à celle des auteurs.

Les élèves souhaitant approfondir le contenu de ce chapitre trouveront dans l'article d'Auger, Nicaud et Pivoteau :

- en section 2, des modèles plus précis de prédicteurs de branchement,
- en section 3, une analyse quantitative fine de l'exemple introductif de ce chapitre (calcul simultané de minimum et maximum d'un tableau),
- en section 4, un autre exemple d'algorithme « prediction-aware », cette fois pour le calcul d'exponentiation rapide.

Tout cela devrait être abordable avec vos connaissances actuelles (et du temps !).

Chapitre 6

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.

6.1 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.

6.2 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.

6.2.1 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.

6.2.2 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.

6.3 En pratique

À suivre...

6.4 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 7

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.

7.1 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 *multi-cœurs*.

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.

7.2 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
```

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

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.

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.

7.3 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.

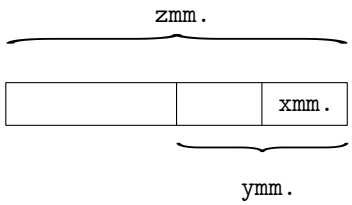
Un premier exemple. Revenons à notre fonction `incr` ci-dessus. Si on la compile en `-O3`, on obtient un code assembleur sans instruction vectorielle. Cela s'explique par exemple par le fait que le nombre d'itérations n'est pas facilement prévisible. En revanche, `gcc` réussit à vectoriser :

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

7.4 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]

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

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.

7.5 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 1 ★★ On souhaite écrire un pseudo-code qui vectorise le code suivant :

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

```
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 7.1 – Code assembleur obtenu en `-O3` pour l'exercice 1.

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 7.1 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 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 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 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 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.

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

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 $\frac{50+70+180}{3} = 100$.

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.