PROGRAMMATION

ASSEMBLEUR x86

32 et 64 bits sous Linux Ubuntu

Jean-Michel RICHER

Programmation Assembleur x86 32 et 64 bits sous Linux Ubuntu

Jean-Michel RICHER

4ème Edition

version 2023.09

Destinée au cours à partir de
la rentrée de Septembre 2023

© Copyright 2020 par Jean-Michel RICHER

Avertissements

Cet ouvrage peut être reproduit et utilisé uniquement à des fins non commerciales, notamment dans le cadre de l'enseignement de l'assembleur. Il ne peut, en aucune manière, être modifié ou commercialisé sans l'accord de son auteur.

Toute demande de modification, de rectification ou de correction peut être adressée par courrier électronique à l'auteur.

L'ensemble du code des études de cas et des différents chapitres est disponible sur le site web de l'auteur à l'adresse suivante :

`http://leria-info.univ-angers.fr/~jeanmichel.richer/assembleur.php`

ISBN-13 : 978-2-9573160-0-7

Contact : jean-michel.richer@univ-angers.fr
Adresse : Faculté des Sciences
Université d'Angers
2 Boulevard Lavoisier
49045 ANGERS Cedex 01
France

Ce livre est dédié à ceux qui ont contribué à sa réalisation
en premier lieu à mes parents
pour m'avoir donné la vie
pour m'avoir éduqué
et avoir financé mes études
ensuite, aux enseignants qui m'ont donné l'envie
d'apprendre et de transmettre mon savoir

Juillet 2020, Avrillé

Eduquer c'est chercher à concilier deux mouvements contraires :
celui qui porte à aider chaque enfant à trouver sa propre voie
et celui qui pousse à lui inculquer
ce que soi-même on croit
juste, beau et vrai

Nicolas Sarkozy, Lettre aux Educateurs
4 septembre 2007

Bien écrire, c'est déjà presque bien penser
et il n'y a pas loin de la jusqu'à bien agir

Thomas Mann

REMERCIEMENTS

J'adresse mes remerciements à l'équipe technique du Département Informatique de l'Université d'Angers dont notamment Eric Girardeau, Jean-Mathieu Chantrein, Benjamin Jeanneau et Frantz de Germain pour m'avoir facilité l'accès à certains matériels afin de réaliser de nombreux tests de performance.

Codes sources

Table des matières

Avant propos

Cet ouvrage s'adresse aux étudiants en informatique, automatique et électronique qui désirent s'initier à la programmation en assembleur x86 que ce soit en architecture 32 ou 64 bits ou qui désirent parfaire leurs connaissances dans ce domaine. Si j'ai voulu écrire ce livre c'est afin de partager l'expérience que j'ai pu acquérir au cours des vingts années passées à enseigner ce sujet à l'université. J'ai pu constater que les étudiants en informatique sont généralement rebutés par l'architecture des ordinateurs qui est pourtant un sujet fondamental dans leur cursus. La programmation assembleur qui en découle apparaît comme un sujet peu attrayant, difficile à maîtriser, et ce, généralement en raison de lacunes concernant des notions de base en informatique. On pensera bien évidemment aux notions relatives au codage de l'information, aux opérations de manipulation des bits (*and*, *or*, *not*), mais également aux pointeurs qui font partie des notions élémentaires fondamentales et utilisées de manière intensive en assembleur. Revenir aux sources de la programmation, c'est à dire à l'assembleur, permet de comprendre ce qui se passe réellement lorsque l'on code dans des langages structurés tels C, Pascal, Fortran ou des langages objet comme C++. Les notions liées à l'assembleur permettent également de comprendre comment rendre son code plus performant en ayant à l'esprit quelques règles élémentaires.

Ma génération, celle de la fin des années 60 et du début des années 70, fut la première à découvrir et utiliser les micro-ordinateurs. La révolution micro-informatique a consisté à mettre dans les mains de chacun un ordinateur de petite taille à un prix abordable alors que la plupart des ordinateurs de l'époque étaient des systèmes volumineux qui occupaient une pièce entière et dont le coût était prohibitif : de l'ordre de la centaine de milliers ou du million de Francs. Pour donner un ordre d'idée, au début des années 80, un IBM PC coûtait en fonction de sa configuration entre 30000 et 50000 Francs ce qui représentait une somme énorme pour la plupart des ménages alors qu'un Commodore 64 ne coûtait que 6000 Francs.

Bien que les premiers micro-informaticiens furent considérés comme des non scientifiques, des bidouilleurs, des personnes qui cherchaient mais sans véritable but si ce n'est celui de se faire plaisir en triturant des machines électroniques, c'est qu'à l'époque beaucoup de choses restaient à créer, imaginer, développer et il était nécessaire de tester, d'essayer, d'expérimenter afin de comprendre ce que ce nouvel appareil qui débarquait dans notre quotidien avait dans ses entrailles. C'est cet

esprit épris de curiosité qui a forgé notre engouement pour le matériel (hardware) et bien évidemment le logiciel qui permet de faire fonctionner le matériel.

J'ai très tôt été confronté à l'assembleur. D'une part mon premier ordinateur fut un Commodore 64 [25], ordinateur à succès, vendu à plus de 17 millions d'exemplaires à travers le monde. Ce *monstre de puissance* était affublé d'un microprocesseur MOS Technology 6510 tournant à la vitesse de 1 Mhz et était doté de 64 ko de RAM, dont 48 ko utilisables pour stocker les programmes. A l'époque cela était suffisant. Le langage BASIC (*Beginner's All-purpose Symbolic Instruction Code*) du C64 était sobre. Notamment, il ne disposait pas d'instruction de type `clrscr` ou `clearscreen` chargée d'effacer l'écran. Il fallait utiliser l'instruction PRINT avec un symbole particulier en forme de coeur afin de vider l'écran. Il était également nécessaire pour réaliser nombres d'opérations graphiques, d'exécuter des instructions PEEK ou POKE qui consistent respectivement en une lecture et une écriture de donnée en mémoire. C'est typiquement une action de bas niveau liée au matériel et donc proche de l'assembleur.

Je suis également redevable de mon engouement pour l'assembleur à *Benoît Michel* dont le "*Livre du 64*" [21] fut le livre de chevet de mon adolescence. On découvrait dans cet ouvrage, qui traite des arcanes du C64, que le BASIC n'était qu'une surcouche qui était orchestrée et exécutée par l'assembleur ou plutôt le langage machine du microprocesseur. Grâce au désassembleur dont le code était donné dans le livre, j'ai désassemblé la ROM et j'ai pu comprendre comment fonctionnait l'interpréteur BASIC et comment on pouvait le modifier de manière à intégrer de nouvelles instructions.

J'ai toujours trouvé plaisant de pouvoir programmer au plus bas niveau car on est au plus près de la machine et les problèmes à solutionner demandent une certaine ingéniosité. Il est également nécessaire de faire preuve de rigueur car on ne dispose pas de structures de contrôle. On utilise adresses et pointeurs à outrance et, dans le cas de l'architecture 32 bits de machines de type x86, la limitation imposée par le nombre de registres disponibles pour stocker données et adresses est handicapante. Il faut donc faire preuve d'inventivité.

L'ensemble de cet ouvrage se base sur la programmation dans un environnement Linux de type Ubuntu/Debian et utilise de nombreux logiciels inhérents à ce système d'exploitation comme **make** pour la compilation automatique, **g++** pour le compilateur C++ et **nasm** en ce qui concerne la partie assembleur. Linux, de par ses caractéristiques, offre au développeur un large panel d'outils puissants qui permettent de traiter tous les aspects du processus de développement logiciel en passant par le profilage et les tests. On pourra bien entendu transposer ce qui a été vu à d'autres environnements comme MacOS ou Windows.

J'ai conçu cet ouvrage comme un cours académique, c'est à dire que les premiers chapitres introduisent les notions fondamentales (Chapitres 1 à 9) et les suivants se révèlent plus pratiques.

Le **chapitre 1** traite de notions générales et fondamentales en informatique et

aborde succintement des notions liées au Génie Logiciel et au travail de l'informaticien.

Le **chapitre 2** concerne le codage de l'information notamment des nombres entiers, des réels que l'on qualifie de nombres à virgule flottante et des chaînes de caractères.

Le **chapitre 3** aborde les notions liées à la mémoire (comme l'alignement, l'adressage mémoire et le *dual channel*) et les notions relatives au fonctionnement du microprocesseur (chargement des instructions, décodage et exécution, pipeline, etc).

Le **chapitre 4** traite des logiciels utilisés dans le cadre de cet ouvrage c'est à dire les éditeurs, l'assembleur, le compilateur et le débogueur.

Le **chapitre 5** reprend les notions vues sur les registres dans le chapitre 3 et introduit les instructions assembleur de base qui travaillent sur les entiers. On montre par la suite comment traduire les structures de contrôle du langage C comme le *if*, le *while*, le *for* et le *switch* en assembleur x86.

Le **chapitre 6** apprend comment passer des paramètres à un sous-programme et comment récupérer ces paramètres dans le sous-programme appelé, que ce soit en architecture 32 bits ou en architecture 64 bits.

Le **chapitre 7** aborde le traitement des *nombres à virgule flottante* par *le coprocesseur* (FPU). On montre comment traduire simplement une expression arithmétique en utilisant les mnémoniques de la FPU.

Le **chapitre 8** traite des unités vectorielles et des instructions liées à ces unités que sont le *SSE*, l'*AVX* et l'*AVX512*. On introduit également les *intrinsics* qui sont des fonctions du C qui seront remplacées lors de leur traduction par des instructions vectorielles. Ces fameuses fonctions intrinsèques permettent d'écrire du code vectoriel qui sera optimisé par le compilateur C tout en restant au niveau du langage C.

Le dernier chapitre académique (**chapitre 9**) traite de l'*algèbre de Boole*. Même s'il n'est pas vraiment lié à l'assembleur, il permet de comprendre le rôle central que joue l'algèbre de Boole en informatique, allant du codage des circuits jusqu'à la logique. Ce chapitre quelque peu disgressif peut néanmoins faire partie de ce livre et permet une ouverture à la logique.

Plusieurs études de cas (chapitres 11 à 15) permettent de mettre en oeuvre les connaissances vues lors des premiers chapitres et j'explique comment traduire en assembleur des fonctions écrites en langage C afin d'obtenir le code le plus performant possible. Ce code est ensuite testé sur différents matériels et l'analyse des résultats permet de démontrer au lecteur que le matériel (processeur, carte mère et mémoire) peut avoir une influence sur un choix particulier de traduction en assembleur.

Le **chapitre 11** aborde l'optimisation du produit de deux matrices carrées d'entiers. On montre en particulier l'importance de l'accès mémoire et de la mémoire

cache.

Le **chapitre 12** se focalise sur l'implantation de l'instruction `popcnt` qui compte le nombre de bits positionnés à 1 dans un registre. On montre au travers de différentes implantations comment cette instruction peut être améliorée si on doit la coder en C lorsqu'elle n'est pas disponible nativement sur un microprocesseur.

Le **chapitre 13** traite de l'implantation de la fonction saxpy et permet d'introduire plusieurs instructions assembleur liées au coprocesseur arithmétique et au calcul vectoriel avec unités SSE sur les flottants.

Le **chapitre 14** aborde des notions liées à la bioinformatique. On montre comment améliorer très fortement l'implantation d'une fonction en utilisant les instructions vectorielles sur les entiers et notamment en traitant les données par groupe de 16 ou 32 octets en utilisant respectivement les registres SSE et AVX.

Le **chapitre 15** se focalise sur la programmation en architecture 64 bits et montre comment le fait de pouvoir disposer du double de registres par rapport à l'architecture 32 bits permet de simplifier et implanter efficacement une fonction qui compte des voyelles dans une chaîne de caractères.

Le **chapitre 16** sintéresse à l'implantation d'une fonction qui calcule le n-ème terme de la suite de Fibonacci. Plusieurs versions sont proposées allant du dépliage de boucle à la vectorisation en passant par la formule de calcul directe basée sur le nombre d'or.

Enfin, le dernier chapitre des études de cas 17 cherche à améliorer une fonction qui indique si un nombre entier est un nombre auto-descriptif ou non. Un nombre auto-descriptif se définit comme un entier naturel ayant pour propriété que chacun de ses chiffres, repéré par son rang, indique combien de fois ce rang apparaît en tant que chiffre dans l'écriture de ce nombre. On montre ici l'intérêt de remplacer la division par un invariant par une multiplication, comme évoqué section 2.4.7.

Se sont ajoutées au cours du temps quelques annexes, en fin d'ouvrage :

- un rappel des conventions d'appel Linux en 32 et 64 bits
- une démonstration de l'utilisation du débogueur GDB
- une mini étude de cas concernant l'utilisation de l'instruction assembleur `bsr`
- une mini étude de cas liée à l'implantation de la fonction signe qui donne le signe de son opérande
- une table ASCII descriptive pour les codes de 0 à 127
- un glossaire des instructions assembleur

J'espère, au travers de cet ouvrage, donner au lecteur une meilleure compréhension du fonctionnement du microprocesseur et réconcilier les développeurs avec l'assembleur en montrant comment les traitements de haut niveau peuvent être traduits de manière efficace dans un langage très limité. Il est certain que les compilateurs ont fait de gros progrès et sont capables de traduire efficacement bon nombre d'algorithmes, mais certains traitements, en raison de leur complexité,

peuvent encore être améliorés en les codant à la main. Passer à l'assembleur permet par exemple de paralléliser le code en utilisant les instructions vectorielles et gagner ainsi un facteur important en terme de performance.

Que la force de l'assembleur soit avec vous !

Chapitre 1

Informatique, informaticien et assembleur

There's an art to all this madness
Tho' it seems insane to you
There's a rhyme to all the reason
In everything I do
Have you any imagination
Of what I'm goin' through
The Jacksons, Art of Madness

1.1 Pourquoi apprendre l'assembleur

Le langage FORTRAN (*FORmula TRANslator*) mis au point par John Backus et son équipe chez IBM en 1956 représente une avancée majeure pour l'informatique car ce langage de haut niveau permet alors de s'affranchir des contraintes et spécificités propres à chaque microprocesseur. Dans ce type de langage la déclaration d'une variable permet de faire abstraction de sa localisation en mémoire, c'est le compilateur qui se chargera de placer la variable à une adresse fixe et l'identifiant de la variable permet de manipuler à la fois sa valeur et son adresse de manière transparente, alors qu'en assembleur, une variable est identifiée par son adresse.

L'ajout de structures de contrôle (*if then else*, *for*, *while*, etc) apporte en outre un confort notable pour l'écriture de traitements complexes et l'utilisation de l'*indentation* permet visuellement de comprendre la structure du programme. A contrario, le langage assembleur est un langage limité, sans structures de contrôle.

Alors pourquoi revenir en arrière ? Cela ne constitue t-il pas une régression que de coder en assembleur ?

Tout dépend du point de vue. Si vous devez conduire une voiture pour aller d'un point A à un point B vous vous fichez sans doute de savoir quelles sont les

différentes pièces qui constituent un moteur. Mais, si vous devez réparer une voiture ou si vous ne voulez pas rester en rade au beau milieu de nulle part à attendre une dépanneuse, alors, cela devient essentiel. Connaître l'assembleur c'est, en partie, être en mesure de comprendre comment fonctionne un ordinateur et comment trouver et corriger les bogues d'un programme.

La raison principale qui conduit généralement à programmer en assembleur tient au fait que l'assembleur est le langage le plus proche du microprocesseur et, en étant proche de celui-ci, on tente d'en extraire la *substantifique moëlle* [1], ou en d'autres termes, on tente d'extraire le maximum de son efficacité. Un exemple très concret est l'utilisation d'une fonction qui compte le nombre de bits positionnés à un dans un entier 16, 32 ou 64 bits. Nous verrons que cette fonction est très utile par la suite dans les études de cas de cet ouvrage. Sur les processeurs récents cette fonction est disponible sous forme d'une instruction assembleur appelée `popcnt` pour *Population Count* et est très efficace (voir Chapitre 12) comparativement à une fonction C que l'on devrait écrire pour obtenir le même résultat.

Dans le même esprit, on peut également penser aux instructions assembleur **`bsr`** et **`bsf`** pour *Bit ScanReverse / Forward* qui déterminent la position du bit le plus ou le moins significatif d'un entier. Ces deux instructions ne sont généralement pas disponibles dans la plupart des langages informatiques et il faut les implanter avec les instructions du langage.

```
1  // fonction qui implante bsr (bit scan reverse)
2  u32 function_bsr(u32 a) {
3      for (int i = 31; i >= 0; --i) {
4          if ((a & (1 << i)) != 0) return (u32) i;
5      }
6      return 0xFF;
7  }
8
9  // retourne la somme des bsr(t[i]) pour i dans [0..n-1]
10 u32 method_1(u32 *t, u32 n) {
11     u32 sum = 0;
12     for (u32 i = 0; i < n; ++i) {
13         sum += function_bsr(t[i]);
14     }
15     return sum;
16 }
```

Listing 1.1.1 – Fonction bsr, version 1

Pour rentrer abruptement dans le vif du sujet, regardons quel gain on peut obtenir en implantant la fonction **`function_bsr`** en C ou en utilisant directement l'instruction assembleur. Le but de ce test est d'évaluer l'efficacité de chaque méthode. L'instruction **`bsr`** détermine la position du bit de poids fort d'un entier. Ainsi,

1. Expression rendue célèbre par Rabelais dans Gargantua (1534) et qui désigne ce qu'il y a de plus précieux.

pour la valeur décimale 123 qui, en binaire, s'écrit 1111011, c'est le bit 6 qui est le bit de poids fort (ou bit le plus à gauche). Le bit de poids faible, c'est à dire le bit le plus à droite, a pour indice 0. Il est positionné à 1 dans 123.

Le Listing 1.1.1, fonction `method_1`, applique la fonction `function_bsr` sur un tableau `t` de `n` entiers 32 bits. On en profite pour réaliser la somme des valeurs obtenues afin de produire une somme de contrôle (*checksum*, variable `sum`) ce qui permet de vérifier que l'on obtient bien le même résultat pour chaque fonction testée.

Le type `u32` représente un entier non signé sur 32 bits et correspond au type `size_t` du langage C. La fonction `function_bsr` cherche le bit le plus significatif[2] en utilisant la variable `i` qui sera décrémentée progressivement. Initialement `i` est égale à 31 ce qui correspond au bit le plus à gauche dans un entier 32 bits (cf. Chapitre suivant).

La même fonction peut être implantée (cf. Listing 1.1.2) en utilisant la possibilité offerte par le compilateur C++ d'introduire l'appel à l'instruction assembleur `bsr`. Le codage est difficile à comprendre et il faut se référer à la documentation du compilateur pour avoir une idée de la syntaxe utilisée, mais laissons cela de côté pour le moment.

```
1  u32 method_3(u32 *t, u32 n) {
2      u32 sum = 0;
3      for (u32 i = 0; i < n; ++i) {
4          u32 input = t[i];
5          u32 output;
6          asm("bsr %0, %1" : "=r" (output) : "r" (input) : );
7          sum += output;
8      }
9      return sum;
10 }
```

Listing 1.1.2 – Fonction bsr, version 3

On notera cependant qu'en langage C, ces instructions (`bsr`, `bsf`, ...) ont été ajoutées sous forme de fonctions *built-in*, c'est à dire des extensions qui permettent au programmeur d'utiliser la syntaxe d'un appel de fonction pour faire appel à une instruction du processeur. L'implantation dépend alors du jeux d'instructions dont dispose le microprocesseur. Si l'instruction assembleur est présente elle sera utilisée, sinon elle sera remplacée par une fonction écrite en C.

La Table 1.1 résume les temps d'exécution en secondes des fonctions évoquées précédemment, appliquées sur un tableau d'un million d'entiers. On réalise le calcul 100 fois afin d'obtenir des temps significatifs. Trois méthodes ont été évaluées :

- la *méthode 1* correspond à l'appel d'une fonction écrite en C (Listing 1.1.1)

2. Le bit à 1 d'indice le plus grand.

- la *méthode 2*, écrite en C, utilise la fonction `__builtin_clz` qui retourne le nombre de bit à 0 avant de trouver un bit à 1 en partant du bit le plus significatif (bit 31), on retourne donc `31 - __builtin_clz(t[i])`
- la *méthode 3* utilise directement l'instruction assembleur `bsr` (Listing 1.1.2)

Méthode	Temps (s)	Amélioration
méthode 1 / fonction C	8,68	-
méthode 2 / `__builtin_clz`	1,19	× 7,3
méthode 3 / asm + `bsr`	0,96	× 9,0

TABLE 1.1 – Temps d'exécution en secondes pour le calcul répété 100 fois du bit le plus significatif sur un tableau d'un million d'entiers non signés sur AMD Ryzen 5 3600

Un rapide examen du code assembleur généré pour la *méthode 2* montre que le compilateur remplace la fonction `__builtin_clz` par l'instruction `bsr`. On obtient donc des temps très proches pour les méthodes 2 et 3. On constate que l'utilisation de l'instruction assembleur permet d'obtenir une méthode dont le temps d'exécution est environ 9 fois plus rapide que la fonction C que l'on devrait implanter pour réaliser le calcul. On trouvera en annexe (cf. Annexe C) de plus amples résultats.

Voilà donc un exemple très explicite de ce que permet l'assembleur en terme d'efficacité.

Cependant, un public peu averti pourrait considérer que dans la grande majorité des cas, savoir programmer en assembleur n'est d'aucune utilité au moins pour deux raisons :

- d'une part, les langages destinés au web (PHP, Javascript, Python [3], Ruby) sont des langages interprétés où l'assembleur n'est pas utilisé ou pas directement utilisable et, de plus, le développeur qui consacrera son temps à créer des interfaces graphiques, optimiser des requêtes SQL ou concevoir des sites web n'aura jamais d'interaction avec l'assembleur,
- d'autre part, pour les langages compilés tels que C, C++, Fortran le compilateur est généralement capable de produire un code assembleur bien plus optimisé que celui écrit à la main en faisant appel à différentes techniques (vectorisation, parallélisation, dépliage de boucle, optimisation guidée par profilage) qui sont accessibles via les options en ligne de commande des compilateurs.

En outre, la principale difficulté de la programmation en assembleur réside en partie dans l'absence de structures de contrôle que l'ontrouve dans les langages de haut niveau (*if*, *for*, *while*, etc). On est donc contraint d'écrire dans un langage

3. Concernant Python, il s'agit d'un cas particulier puisque l'on peut optimiser le code Python en le compilant et que les librairies Python sont écrites en C/C++.

bas niveau, ce qui rend la relecture et la compréhension du code difficile, il est absolument **nécessaire de bien commenter son code** !

Comme nous le verrons plus tard et comme nous l'avons déjà évoqué dans l'*Avant Propos*, le fait de ne disposer finalement que de 6 registres généraux en architecture 32 bits (`eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`) pour réaliser les traitements (`esp` et `ebp` étant utilisés pour gérer la pile), est très contraignant et nous oblige à constamment jongler avec les registres : tel registre va contenir telle donnée au début d'un sous-programme, puis telle autre donnée au milieu et finalement un autre résultat à la sortie du sous-programme.

Donc finalement, savoir programmer en assembleur c'est difficile et cela ne sera pas très utile ? Pourquoi alors écrire cet ouvrage ? La réponse est bien évidemment non, car apprendre à programmer en assembleur nous apporte plusieurs compétences qui, de mon point de vue, sont essentielles de possèder pour tout informaticien qui se respecte. L'apprentissage du langage assembleur nous apporte :

- la connaissance de notre outil de travail : programmer en assembleur nous amène à savoir comment fonctionne le microprocesseur, ce qu'il est capable de réaliser, comment il traite les données, comment il interagit avec la mémoire. On pensera également au fait que le microprocesseur est un système complexe qui combine plusieurs technologies et c'est de la synergie de ces technologies que provient l'efficacité de l'exécution du code (voir le Chapitre 3),

- la possibilité d'optimiser du code : vous apprendrez des notions liées à l'optimisation du code (dépliage de boucle, vectorisation) car tout informaticien se doit de produire du code valide (c'est à dire qui réalise le traitement demandé) et efficace (qui le fait de la manière la plus rapide possible), cela peut avoir une influence non négligeable sur votre carrière,

- la possibilité de supplanter le compilateur : certains traitements sont difficilement traduisibles de manière optimale par le compilateur, cela arrive rarement, mais dans certains cas, coder ces traitements en assembleur se révèle un atout primordial et permet de faire la différence,

- la rigueur : programmer en assembleur demande d'être rigoureux car il est nécessaire avant toute chose de spécifier ce que stockeront les registres, comment on va manipuler les données, comment on va les traiter. On retrouve la même necessité de rigueur lorsque l'on programme avec des langages de haut niveau dès lors que l'on manipule plusieurs concepts simultanément.

Notons enfin que l'utilisation de l'assembleur est parfois obligatoire lorsque liée au matériel : la programmation des *drivers* de périphériques passe généralement par une partie assembleur qui réalise l'interface entre le périphérique et le système d'exploitation.

1.1.1 Matériel et logiciel

Un autre point essentiel lié à la programmation en assembleur est le fait qu'un système informatique se compose de deux parties :

- une partie matérielle (*hardware*) qui représente l'ensemble des composants de la machine,
- une parite logicielle (*software*) constituée des logiciels s'exécutant en utilisant ou tirant partie du matériel.

Un informaticien se doit de comprendre le fonctionnement du système dans sa globalité car les caractéristiques du matériel influent sur les performances des programmes. Par exemple, il y a de cela quelques années j'ai réalisé un test sur l'implantation d'une version de la fonction `popcnt`, que nous avons évoqué en début de chapitre. Ce test s'exécutait en 8 secondes sur un microprocesseur Intel Pentium-M. Afin d'optimiser le code avec `gcc` j'ai utilisé l'option `-mtune=pentium-m` sensée prendre en considération les caractéristiques d'un Pentium-M afin de produire du code assembleur plus performant. Le résultat fut un temps d'exécution pour le même test de 23 secondes, soit près de trois fois plus lent ! Les deux codes ne différaient que par quelques instructions. Pour être en mesure de comprendre pourquoi le code est plus lent dans la version sensée être plus rapide il est nécessaire de comprendre le code assembleur ainsi que les caractéristiques du microprocesseur qui exécute le code.

Une bonne connaissance du fonctionnement interne de l'ordinateur permet de comprendre pourquoi certains algorithmes se révèlent efficaces et pourquoi d'autres sont mal adaptés par rapport à une architecture donnée ou par rapport au problème à traiter et nous permet alors d'en améliorer l'efficacté. Ainsi, les microprocesseurs AMD ont des unités de traitement des nombres flottants très lente par rapport aux microprocesseurs Intel. Si on veut gagner en efficacité sur le traitement des flottants il faut alors coder les traitements en utilisant la partie basse des registres vectoriels SSE (cf. Chapitres 6, 7 et 8).

Les traitements informatiques possèdent, au regard de ceux qui en sont les utilisateurs et donc les tributaires, une exigence de qualité (robustesse et performance) et l'informatique s'attache à résoudre des problèmes complexes par leur structure ou par le volume de données à gérer. L'informaticien doit donc être capable de trouver le traitement (algorithme) le plus adapté aux données à analyser et savoir coder correctement des algorithmes dans un langage donné. Par exemple, déterminer si un entier est pair peut être réalisé en effectuant une division par 2 et en vérifiant que le reste de la division est égal à 0. Malheureusement, la division, de part sa nature complexe, est pénalisante et demande plus de temps de traitement pour s'exécuter que les autres opérations comme l'addition, la soustraction ou la multiplication. Etant donné que l'on travaille avec des nombres codés en binaire, une autre méthode consiste à vérifier que le premier bit du nombre n'est pas à 1 ce qui prend beaucoup moins de temps.

1.2 Le métier d'informaticien

1.2.1 Qu'est ce qu'un ordinateur ?

Définition : Ordinateur

Un ordinateur est une machine électronique conçue pour effectuer des calculs et traiter des informations de manière automatique.

Le terme ordinateur fut inventé par Jacques Perret, professeur de philologie latine à la Sorbonne, à la demande d'IBM France en 1955. IBM cherchait en effet à cette époque un nom pour commercialiser son nouveau calculateur qui fut alors baptisé *ordinateur IBM 650*.

Un ordinateur est composé de plusieurs parties appelées :

- composants (carte mère, microprocesseur, barrette de mémoire, carte graphique)
- et périphériques (disque dur, lecteur de DVD, clavier, souris, moniteur, ...).

La distinction entre composant et périphérique est parfois ténue et repose généralement sur le fait qu'un périphérique se trouve éloigné de la carte mère alors qu'un composant est en contact direct avec celle-ci. Cependant le terme composant peut être utilisé pour englober les périphériques. Pour certains, le terme périphériques fait uniquement référence à tout ce qui est externe au boîtier : clavier, souris, moniteur, imprimante, en d'autres termes, ce qui se trouve à la périphérie du boîtier.

1.2.2 Qu'est ce que l'informatique ?

Définition : Informatique

Science du traitement de l'information effectué par un ordinateur. Elle comprend l'ensemble des activités consistant à collecter, organiser et traiter de manière automatique les données par un ordinateur.

Le terme informatique a été créé en mars 1962 par Philippe Dreyfus (Directeur du centre national de calcul électronique de la société Bull dans les années 1950, un des pionniers de l'informatique en France) à partir des mots information et automatique.

En anglais on trouve parfois le terme *Informatics*, mais plus généralement on emploie le terme *Computer Science*, voire *Computer Engineering* pour désigner l'informatique. On notera la différence établie entre le mot anglais *computer* (calculateur), c'est à dire la tâche première pour laquelle les ordinateurs furent conçus et

utilisés, et le mot informatique, c'est à dire, leur utilisation au quotidien : le traitement automatique de l'information. On peut alors soulever une question d'ordre philosophique et se demander si calculer c'est traiter l'information, et inversement traiter l'information n'est-ce que réaliser un calcul ?

Il faut insister ici sur le mot science, car à ses débuts l'informatique n'était pas considérée par les autres disciplines exactes (mathématiques, physique et chimie) comme une science. Ces dernières ont plus de deux mille ans d'histoire. L'informatique a atteint le rang de science en quelques décennies depuis les années cinquante même si on peut faire remonter les premiers travaux sur les calculateurs mécaniques au XVII^{e} siècle, voire même avant si on pense à la machine d'Anticythère.

Historiquement, l'informatique a commencé à entrer en tant qu'outil pédagogique dans l'enseignement secondaire français à partir de la fin des années 1970. En 1980 [17] lors d'une table ronde sur le sujet de *l'enseignement français face à l'informatique*, Jacques Tebeka [4], pose la question suivante : *Faudrait-il enseigner l'informatique comme une discipline indépendante, au même titre que les mathématiques ? Ou la considérer seulement comme une aide à l'enseignement dans les différentes disciplines ?*

En fait ces propos soulignent le dilemme auquel fait face l'informatique depuis plusieurs décennies : faut-il l'enseigner comme une science au même titre que les mathématiques ou comme outil technique ? Car finalement, savoir utiliser un traitement de texte, un tableur ou rechercher de l'information sur internet relèvent de compétences techniques. Savoir programmer un ordinateur (cf. ci-après) relève de la science informatique.

Le même intervenant, Jacques Tebeka, fit également part lors de cette conférence de son désir que l'informatique soit enseignée comme discipline : *Je viens de faire un petit calcul. À la vitesse actuelle, on pourra généraliser l'enseignement de l'informatique dans 430 ans... En tant qu'industriel je demande qu'on ne s'étende pas sur le problème philosophique de savoir s'il faut enseigner l'informatique ou l'informatique à travers les disciplines. Je demande qu'on aille vite, beaucoup plus vite...*

L'informatique, depuis des années, n'est enseignée au collège et lycée dans le système éducatif français que comme outil et non comme discipline. C'est seulement au niveau de l'enseignement supérieur (Universités, IUT, Ecoles d'Ingénieurs) que l'informatique devient une discipline à part entière.

En ce qui concerne ma formation, j'ai eu quelques cours d'informatique en 3ème (1984-1985) réalisés par l'enseignant de mathématiques. Il nous a appris à faire de la programmation en Logo, puis en Pascal sur Apple 2. Plus tard, lorsque je suis entré à l'université de Bourgogne à Dijon en 1988 comme étudiant, l'informatique en première année traitait de l'apprentissage de la programmation avec le langage BASIC et n'était pas enseignée uniquement par des informaticiens mais également par des physiciens. Lorsque j'ai été recruté à l'Université d'Angers comme maître

4. Responsable du centre informatique de la société ESSO en France et aux États-Unis, conseiller informatique du groupe BSN Gervais Danone, directeur général du groupe Datsun.

de conférence, en Octobre 2000, le langage C n'était pas enseigné en licence (L3). J'ai donc incorporé au cours d'Architecture des Ordinateurs que j'enseignais un volet langage C, car ce dernier est central en informatique. On l'utilise pour la programmation système sous Linux, et, de ce langage découlent d'autres langages comme le C++, le Java, le Javascript, le PHP.

Il aura fallu batailler très dur pour enraciner l'informatique comme discipline et science car c'est seulement dès la rentrée 2019 que l'enseignement de l'informatique fut proposé au lycée à tous les élèves de seconde générale et technologique (soit 1h30 par semaine), et en tant que discipline de spécialité de 1ère puis Terminale (4h puis 6h par semaine).

1.2.3 Qu'est ce qu'un informaticien ?

Définition : Informaticien

Un informaticien est un scientifique qui met en place des procédures de traitement automatique de l'information grâce à un ordinateur tout en concevant des algorithmes efficaces et en exploitant au mieux les capacités de la machine.

J'insiste ici sur le fait qu'en tant que scientifique l'informaticien se doit de réfléchir du point de vue de la complexité de ses algorithmes mais également du point de vue de leur implantation dans un langage informatique.

L'ordinateur est l'outil qu'utilise l'informaticien pour réaliser son travail et nous nous devons de connaître et maitriser notre outil de travail afin de solutionner les problèmes qui nous sont posés.

On peut dresser un parallèle avec l'automobile et le mécanicien. Imaginez que votre voiture vous pose des problèmes récurrents et qu'elle ait du mal à démarrer tous les matins. Pour régler le problème vous vous rendez chez un garagiste et confiez votre véhicule à un mécanicien qui, pour vous, représente un expert qui saura trouver une solution adéquate à votre problème. Il établira un diagnostique et vous indiquera la cause du problème (batterie, bougies, carburateur, etc) puis vous proposera une solution (remplacement de la pièce défectueuse) qui est sensée être la moins onéreuse pour vous.

Que penser si le mécanicien n'y entend rien en mécanique ? Vous avez des problèmes au démarrage, vous perdez de la puissance quand vous montez une côte, votre feux arrière ne fonctionne plus. Il saura vous proposez néanmoins une solution et elle sera toujours la même : changez de voiture ! Effectivement changer de voiture solutionnera le problème, mais à quel prix ! L'incompétence du prétendu spécialiste vous sera alors préjudiciable.

Il en va de même en informatique. Si vous n'avez pas su coder efficacement un algorithme ou si vous n'avez pas su choisir le bon algorithme, vous pouvez

toujours proposer à celui qui utilise votre programme et qui trouve que celui-ci prend trop de temps à s'exécuter, d'acheter une machine plus puissante, mais cela ne solutionnera le problème qu'en partie. Si un autre informaticien est capable de proposer un algorithme plus efficace ou de détecter dans votre algorithme ou votre codage un verrou, vous risquez de passer pour une personne peu compétente qu'il est préférable de remplacer.

1.2.4 En quoi consiste son travail ?

Le travail de l'informaticien consiste, partant d'énoncés en langage naturel (français, anglais, etc) à traduire ces énoncés en une série d'opérations clairement définies que l'on appelle algorithme. Ces algorithmes sont ensuite traduits en instructions directement compréhensibles par le microprocesseur de la machine.

Définition : Algorithme

Un algorithme est une succession finie d'actions clairement identifiées exécutées dans un ordre précis.
Le mot algorithme est dérivé du nom du mathématicien persan Al Khwarizmi (vers l'an 820), qui introduisit en Occident la numération décimale (rapportée d'Inde) et enseigna les règles élémentaires des calculs qui en découlaient.

On peut résumer le travail de l'informaticien en disant qu'il doit *être capable de créer un logiciel*. La simplicité de cette expression ne laisse pas présager de l'étendue des compétences qu'elle englobe. On peut, afin de mieux comprendre ce que cela implique, prendre l'analogie avec la construction d'une maison.

Imaginons que vous vouliez faire construire une maison et que vous disposiez, pour cela, d'un terrain. La première étape consiste à rencontrer un architecte qui, en fonction de vos besoins (nombre d'étages, de chambres, disposition des pièces, ...) et des contraintes du terrain (forme, présence d'un dénivelé ou non), dessinera les plans de votre maison. Une fois les plans finalisés, il faut faire appel à une entreprise de BTP (Bâtiments et Travaux Publics) qui contractera différents corps de métiers (terrassier, grutier, maçon, électricien, plombier, carreleur, charpentier, couvreur, peintre, etc) afin de construire votre nouvelle demeure.

La difficulté de la tâche de l'informaticien c'est que, construire un programme informatique, s'apparente à construire une maison, avec une contrainte de taille : l'informaticien doit être à la fois architecte, maçon, électricien, plombier, etc. Il doit être à la fois :

- **concepteur** c'est à dire réfléchir d'un point de vue théorique à l'organisation de la structure de son programme, des classes qu'il va créer et de l'interaction entre ces classes,
- **constructeur**, c'est à dire savoir implanter son code en gardant à l'esprit qu'il doit produire du code efficace, maintenable, lisible et compréhensible par un

relecteur, alors que lisibilité et efficacité sont antinomiques [5].

En ce qui concerne l'évolution du métier d'informaticien, on pourrait caricaturer en disant que dans les années 70, 80, on a eu tendance à séparer conception et codage. La partie conception était considérée comme *noble* et ne demandant pas nécessairement de savoir coder. Elle était réservée à des personnes ayant fait des études au niveau bac + 4 (master) voire à bac + 8 (doctorat). Cette tâche, que certains considèrent comme ingrate ou de bas niveau, qu'est le codage était plutôt réservée aux analystes programmeurs que l'on formait au niveau bac + 2 ou bac + 3 (licence).

Cette vision des choses a évolué à partir des années 90, lorsque l'informatique est devenue de plus en plus complexe avec des programmes contenant des centaines de milliers de lignes de code et donc des centaines de classes ainsi que des paradigmes et des concepts de programmation non triviaux (fonctionnel, logique, généricité, multi-tâches).

Un autre point important est le *passage à l'échelle*, c'est à dire le fait de traiter des volumes de données de plus en plus importants. Lorsque l'on développe un algorithme, on travaille généralement avec un jeu de données en entrée de petite taille de manière à détecter rapidement les erreurs et bogues inhérents à tout programme informatique. Puis une fois le programme finalisé autour de l'algorithme à implanter, on passe à des jeux de données plus importants. Parfois la taille des données va conduire à revoir les structures de données car celles-ci prennent une place trop importante en mémoire, ou alors, on s'aperçoit que notre programme qui mettait quelques secondes à s'exécuter sur un petit jeu de données met finalement plusieurs heures, voire plusieurs jours pour s'exécuter sur un jeu de données plus conséquent car l'accès aux données n'est pas efficace (voir par exemple le Chapitre 11). Il se peut également que la complexité du problème rende impossible le traitement de grandes instances.

Tous ces facteurs concourent à comprendre que l'informaticien, pour accomplir sa tâche de nos jours, doit détenir au moins un niveau master pour disposer des connaissances et de la maturité nécessaires à l'accomplissement de son travail et il faut souvent ajouter à celà plusieurs années d'expérience.

Enfin, un dernier facteur entrant en jeu, et souvent négligé, est le fait que les informaticiens ne conçoivent pas, la plupart du temps, des programmes pour eux-mêmes mais pour les autres. C'est à dire pour des compagnies téléphoniques, des constructeurs automobiles, des avionneurs, des organismes de recherche en médecine, en agronomie, des institutions publiques. Cela représente autant de domaines pour lesquels le domaine d'expertise n'est pas connu de l'informaticien et ajoute une contrainte et une difficulté supplémentaire.

5. c'est à dire contradictoires

1.3 Savoir programmer et savoir réfléchir

Afin d'exemplifier mon propos quant au fait de savoir programmer, je vais prendre trois exemples simples au travers desquels je tenterai de démontrer que l'informaticien se doit de réfléchir, de savoir coder mais également disposer d'un certain recul et d'une expérience qui demande plusieurs années de pratique. S'il en est ainsi sur des exemples aussi simples, que penser s'il s'agit de programmes beaucoup plus complexes ? Le premier exemple touche à la recherche des nombres premiers, le second tient au tri d'un tableau d'entiers et enfin le troisième concerne la recherche et l'élimination de doublons.

1.3.1 Nombres premiers

L'un des problèmes les plus simples que j'aime à demander à mes étudiants de coder est la recherche de nombres premiers. La raison en est que ce problème demande un peu de réflexion. Le problème à résoudre est la recherche des cinquante premiers nombres premiers par exemple. La plupart des étudiants connaît la définition d'un nombre premier mais est incapable de donner le code d'une fonction efficace capable de déterminer si un nombre est premier ou de penser à une méthode plus ingénieuse (cf. ci-après le crible).

Rappelons la définition d'un nombre premier que l'on apprend au collège et lycée :

Definition 1.3.1 (Nombre premier)**.** Un nombre $n \in \mathbb{N}$ est dit premier si il admet uniquement deux diviseurs : un et lui-même. On oublie généralement de préciser que ces deux diviseurs doivent être différents, en conséquence 1 n'est pas premier, le premier nombre premier est donc 2.

Cette définition d'un nombre premier suppose de connaître la notion de divisibilité. Un nombre entier n est divisible par p signifie que $n = p \times q$. Mais la notion la plus intéressante et celle du reste lié à la division entière. Si n n'est pas divisible par p alors il existe un reste r tel que $0 < r < p$ pour lequel $n = p \times q + r$. Pour un informaticien calculer le reste de la division est une opération qui s'appelle l'opération *modulo*. En C elle est représentée par l'opérateur `%` et dans d'autres langages par le mot clé `mod` ou `modulo`.

Une première version de la fonction `est_premier` qui détermine si un nombre n est premier ou non, est celle du Listing 1.3.1. Je l'ai souvent obtenue en réponse à ce problème de la part des étudiants lorsque je leur proposais de le résoudre. Elle traduit simplement la définition que nous avons donnée d'un nombre premier, elle compte le nombre de diviseurs et indique que le nombre passé en paramètre n n'est pas premier si le nombre de diviseurs est différent de 2. Cette version est bien entendu totalement inefficace pour plusieurs raisons :

```
bool est_premier(int n) {
    if (n < 0) return false;
    int nbr_diviseurs = 0;
    for (int i = 1; i <= n; ++i) {
        if ((n % i) == 0) ++nbr_diviseurs;
    }
    return (nbr_diviseurs == 2);
}
```

Listing 1.3.1 – Fonction nombre premier, version inefficace

- si n est divisible par 2 (excepté 2) alors il n'est pas premier et il est inutile de continuer à rechercher d'autres diviseurs
- si n n'est pas divisible par 2, on vérifie quand même qu'il est divisible par des multiple de 2 ce qui n'a aucun intérêt
- si on a obtenu un nombre de diviseurs supérieur à 2, il faudrait simplement s'arrêter plutôt que d'en rechercher d'autres

On peut donc améliorer cette fonction de la sorte (cf. Listing 1.3.2) :

- on teste le cas ou n est égal à 2 ou 3 et dans l'affirmative on indique que le nombre est premier
- on élimine ensuite le cas des nombres pairs en vérifiant si le nombre est divisible par 2
- on ne teste pas les diviseurs au delà de $\sqrt{n}$, car si n est divisible par p, il s'écrit $n = p \times q$ avec $p \leq q$, le cas extrême étant celui où $p = q$. Pour s'en convaincre il suffit de regarder comment se décompose 37 (cf. Table 1.2). A partir de

p	q	r	p	q	r
1	37	0	11	3	4
2	18	1	12	3	1
3	12	1	13	2	11
4	9	1	14	2	9
5	7	2	15	2	7
6	6	1	16	2	5
7	5	2	17	2	3
8	4	5	18	2	1
9	4	1	19	1	18
10	3	7	20	1	17

TABLE 1.2 – Décomposition de 37

$\sqrt{37} \simeq 6$ on ne trouvera pas de diviseur puisqu'on aura déja testé les valeurs de p de 1 à 6 et que q possède des valeurs entre 1 et 5.

- on ne teste que les diviseurs impairs

```
bool est_premier(int n) {
    if (n <= 1) return false;
    if (n <= 3) return true;

    // est-ce un nombre pair ?
    if ((n % 2) == 0) return false;

    // chercher les diviseurs impairs jusqu'à
    // racine carrée de n
    int limit = static_cast<int>(floor(sqrt(n)));
    for (int k = 3; k <= limit; k += 2) {
        if ((n % k) == 0) return false;
    }
    return true;
}
```

Listing 1.3.2 – Fonction nombre premier, version améliorée

On peut améliorer cette version et en donner une version optimisée (Listing 1.3.3) en se basant sur l'élimination des multiples de 2 et 3, puis sur la recherche de diviseurs impairs. Dès lors, on testera beaucoup moins de diviseurs.

```
bool est_premier_v3(int n) {
    if (n <= 3) return n > 1;
    if (0 == (n % 2) || 0 == (n % 3)) return false;

    for (int i = 5; (i * i) <= n; i += 6) {
        if ( ((n % i) == 0) || ((n % (i + 2)) == 0) )
            return false;
    }

    return true;
}
```

Listing 1.3.3 – Fonction nombre premier, version optimisée

Plutôt que de passer par une fonction qui calcule si un nombre est premier, on peut utiliser la méthode du crible d'Ératosthène (voir Listing 1.3.4) qui consiste à remplir un tableau qui indique si un nombre est premier ou non et à éliminer ses multiples. Cette méthode est plus efficace que les précédentes si on doit déterminer dans un intervalle donné quels sont les nombres premiers.

```cpp
// on teste les nombres de 1 a un million
const int N = 1000000;
// tableau qui indique si un nombre est premier ou non
bool *tab = new bool [N+1];

// 0 et 1 ne sont pas premiers
tab[0] = false;
tab[1] = false;
// tous les autres nombres sont initialement premiers
for (int i = 2; i <= N; ++i) tab[i] = true;

// on élimine les multiples de chaque nombre
int n = 2;
while (n <= N) {
    if (tab[n]) {
        for (int j = 2*n; j < N; j+=n) tab[j] = false;
    }
    ++n;
}
```

Listing 1.3.4 – Nombre premier avec crible

Nous présentons Table 1.3, les temps d'exécution en secondes obtenus pour différentes plateformes pour les trois méthodes que nous venons d'évoquer . La première méthode qui consiste à compter le nombre de diviseurs est totalement inefficace. La version améliorée de la fonction `est_premier` est tout à fait acceptable. Le crible représente la méthode la plus efficace. Les temps d'exécution pour cette méthode sont égaux à 0 car de l'ordre de la milliseconde. Elle peut encore être améliorée en ne se focalisant que sur les nombres impairs par exemple.

Au final, on s'aperçoit qu'il ne faut pas simplement répondre en cherchant à *coller* à l'énoncé mais qu'il est nécessaire de réfléchir afin d'améliorer l'efficacité de la fonction `est_premier`. Il faut également parfois chercher une méthode plus

Méthode	AMD Ryzen 7 1700X	AMD Ryzen 5 3600	Intel Core i5 7400	Intel Core i7 8700
`est_premier` (version 1)	1859,59	1726,93	1154,00	895,17
`est_premier` (version 2)	0,20	0,18	0,12	0,07
Crible d'Eratosthène	0,00	0,00	0,00	0,00

TABLE 1.3 – Temps d'exécution en secondes pour la recherche des nombres premiers entre 1 et 1_000_000

adaptée, en l'occurrence le crible. Cette méthode troque en fait la divisibilité par le remplissage d'un tableau ce qui la rend terriblement efficace.

Notons également qu'il existe d'autres améliorations de la fonction `est_premier` : on peut par exemple tester la divisibilité par 3 ou s'appuyer sur le fait que tous les nombres premiers supérieurs à trois sont de la forme $6k \pm 1$.

1.3.2 Tri

Le tri d'un tableau d'entiers représente probablement le sujet le plus étudié par des générations d'étudiants. On apprend qu'il existe différents algorithmes de tri et qu'on peut les classer en fonction de leur complexité. Cependant la complexité est la notion la plus maléable qui soit. Comme on ne sait pas la calculer de manière exacte, on évalue une complexité dans le meilleur des cas, dans le pire des cas ainsi qu'une complexité moyenne qui généralement est la moyenne de la complexité dans le meilleur et dans le pire des cas. Celle-ci varie en effet parfois en fonction des données qu'on manipule. Elle n'est au final qu'un indicateur, mais le programmeur a besoin de plus de précision afin de choisir le meilleur algorithme possible pour traiter ses données. Par exemple deux algorithmes qui possèdent la même complexité n'auront pas forcément le même temps d'exécution et parfois le codage de l'algorithme peut jouer sur son efficacité !

Pour en revenir au tri, on apprend que le tri à bulles (*bubble sort*), le tri par insertion (*insertion sort*) et le tri par sélection (*selection sort*) sont des algorithmes de tri dont la complexité dans le pire des cas est en $O(n^2)$, c'est à dire que si on doit trier un tableau de n entiers, le nombre d'opérations élémentaires à réaliser pour effectuer le tri nécessitera $\alpha \times n^2$ opérations avec α qui est une constante réelle qui peut varier en fonction des opérations de l'algorithme.

Des tris plus efficaces sont les tris en $O(n \times \log(n))$ et on classe dans cette catégorie le tri par tas (*heap sort*), le tri fusion (*merge sort*) et le tri rapide (*quick sort*).

```
1  void bubble_sort(int t[], int n) {
2      for (int i = n-1; i > 0; --i) {
3          for (int j = 0; j < i; ++j) {
4              if (t[j] > t[j+1]) {
5                  swap(t[j], t[j+1]);
6              }
7          }
8      }
9  }
```

Listing 1.3.5 – Tri à bulles en ordre croissant

Le problème est que l'on ne sait pas ce que représente la complexité dans le cas du tri (voir plus loin pour la partie résultats). Nous présentons Listing 1.3.5,

le code du tri à bulles pour un tableau `t` de taille `n`. On peut voir qu'il existe deux opérations qui influent sur la complexité du tri :

- la **comparaison** des valeurs `t[j] > t[j+1]`
- la **permutation** des valeurs `swap(t[j], t[j+1])`

Toute comparaison n'entraine pas forcément une permutation, il est donc difficile de quantifier dans le cas où les données sont aléatoires ce qui peut se passer. De plus, le temps d'exécution d'une comparaison est différent du temps d'exécution de la permutation.

Au final, le seul moyen dont on dispose pour comparer des méthodes de tri ayant la même complexité consiste à obtenir un ordre de grandeur de la complexité réelle (et non théorique) en réalisant de nombreux tests sur des jeux de données en comptabilisant le nombre de comparaisons et le nombre de permutations.

En particulier le tri rapide [6] est le plus efficace dans le cas général sur les tests que j'ai menés.

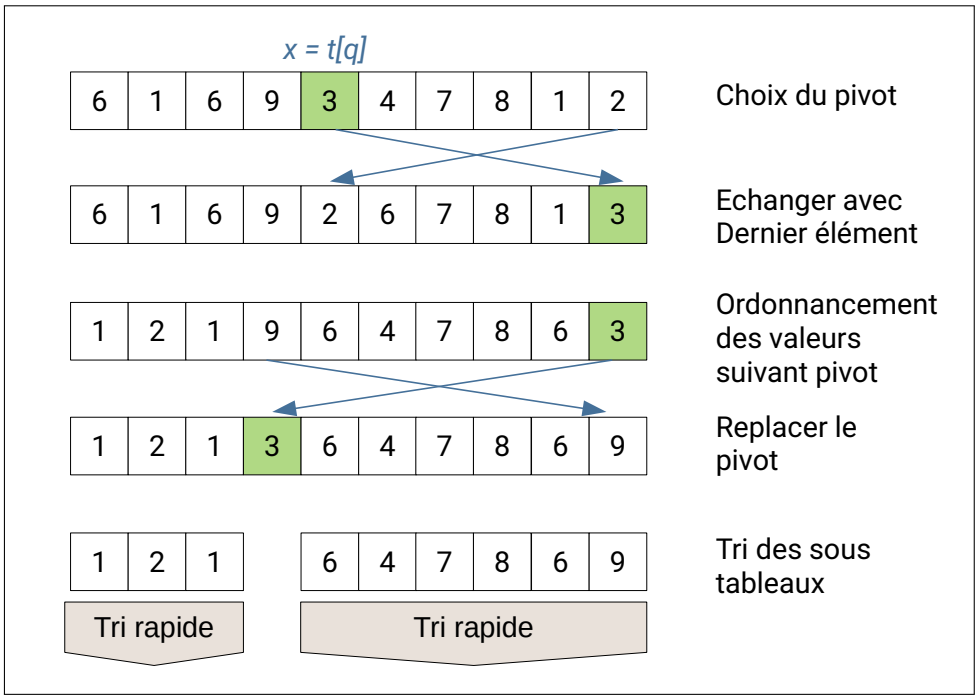


FIGURE 1.1 – Principe du tri rapide

Il se base (cf. Figure 1.1) sur une partition des éléments du tableau initial par rapport à une valeur pivot notée x qui sera placée à un indice q dans le tableau. Toute valeur inférieure à x aura un indice inférieur à q et toute valeur supérieure à x aura un indice supérieur à q. On réitère le partitionnement de manière récursive

6. On pourra consulter le site http://www.rosettacode.org pour découvrir le principes qui sous-tendent les différents tris et découvrir les implantations dans de nombreux langages.

sur les sous-tableaux d'indices $[1..q-1]$ et $[q+1..n]$. Notons que pour simplifier la compréhension le premier indice du tableau est 1 et non 0 comme en C.

Le choix de la valeur pivot est ici essentiel. Elle peut être choisie à un indice compris entre 1 et n. Cependant si on choisit l'indice 1 ou l'indice n cela peut se révéler un très mauvais choix.

Nous présentons Table 1.4 quelques résultats concernant le temps d'exécution de méthodes de tri appliquées à des tableaux d'entiers pour des données qui sont initialement en ordre croissant (c'est à dire déjà triées), puis en ordre décroissant (triées en ordre inverse) et enfin placées aléatoirement.

Méthode	Croissant	Décroissant	Aléatoire
tri par insertion	0.000	169.000	85.300
tri à bulles	0.000	167.040	348.220
tri rapide - version 1	87.420	91.360	0.040
tri rapide - version 2	**0.000**	**0.010**	**0.040**
tri fusion	0.020	0.020	0.060

TABLE 1.4 – Temps d'exécution en secondes pour trier 500_000 entiers sur Intel Core i5 7400 @ 3.00GHz

On remarque que les méthodes de complexité en $O(n^2)$ comme le tri par insertion ou le tri à bulles peuvent se révéler très rapides si les données sont déjà triées. Par contre, si les données sont placées aléatoirement ou en ordre inverse, leur temps d'exécution est prohibitif.

En ce qui concerne le tri rapide, la version 1 qui consiste à choisir la valeur de pivot à l'indice le plus grand du tableau donne de mauvais résultats (comme évoqué précédemment) pour des données triées ou triées en ordre inverse. Par contre, si on choisit le pivot au milieu du tableau (version 2), on obtient des temps de calcul minimes.

Le tri fusion est assez proche du tri rapide mais un peu moins performant. Cela est dû au fait que la fusion qui consiste à créer un seul tableau à partir de deux sous-tableaux triés nécessite de créer un tableau temporaire avec l'implantation que nous avons choisie.

Pour en revenir à la complexité, nous donnons Table 1.5 les complexités observées en nombre de comparaisons et permutations pour trier un tableau de cinq cent mille entiers. Dans le cas du tri fusion il s'agit du nombre de recopies lors de la fusion. On comprend alors mieux pourquoi le tri rapide est le plus efficace, c'est qu'il génère le moins d'opérations de comparaisons et de permutations.

Pour conclure sur cette partie, la connaissance des algorithmes est primordiale mais elle ne donne pas forcément leur efficacité réelle. Les données peuvent influer sur le temps d'exécution, les variantes d'implantation se révèlent plus ou moins

Méthode	Croissant	Décroissant	Aléatoire
tri par insertion	n	n^2	$0.5 \times n^2$
tri à bulles	$0.5 \times n^2$	n^2	$0.75 \times n^2$
tri rapide	$4.8 \times n \times \log(n)$	$4.8 \times n \times \log(n)$	$6.2 \times n \times \log(n)$
tri fusion	$5.1 \times n \times \log(n)$	$6.9 \times n \times \log(n)$	$8.1 \times n \times \log(n)$

TABLE 1.5 – Complexité - Nombre de comparaisons + nombre de permutations ou de recopies pour 500_000 entiers

efficaces. Il est donc nécessaire de ne pas se fier à la théorie et il faut expérimenter par soi-même, implanter les algorithmes et les tester. Nous recommandons au lecteur intéressé la lecture du chapitre 4 de [4] et des chapitres 6 et 7 de [3].

1.3.3 Recherche de doublons

Je fus contacté en 2018 par une étudiante qui rencontrait un problème avec un programme CUDA[7] qui prenait trop de temps à s'exécuter et provoquait un *timeout*, c'est à dire que le programme est arrêté parce que son exécution dure trop longtemps.

Ce timeout est dû au fait que, sur la plupart des machines de bureau ou portables, la carte graphique est utilisée pour l'affichage. On peut également l'utiliser pour faire des calculs parallèle mais dans ce cas l'affichage n'est plus disponible. Si le calcul ne dure que quelques milli secondes cela n'est pas perceptible, mais si le calcul dure plus de quelques secondes, il me semble que le choix a été fait de terminer le programme afin que l'utilisateur récupère la main après 5 secondes.

Le problème à résoudre consiste à supprimer les doublons d'un ensemble d'enregistrements stockés sous forme d'un tableau de N enregistrements de P champs de type entier. Le premier champ contient un identifiant d'enregistrement qui varie de 1 à N. On veut donc connaître les enregistrements dont les champs 2 à P sont similaires. Notons qu'ici nous faisons le choix de stocker ce tableau à deux dimensions sous forme d'un tableau à une dimension de $N \times P$ entiers.

La méthode la plus simple (cf. Listing 1.3.6) et utilisée par l'étudiante qui m'a contacté, consiste à parcourir le tableau et à vérifier que les enregistrements suivants sont identiques ou non à l'enregistrement i. Cependant cette méthode possède une complexité en $O(N^2/2)$ et le nombre d'enregistrements N sur lequel on travaille est de l'ordre de 7 millions. Le calcul de la complexité est assez simple :

- pour l'indice $i = 0$, on aura $N - 1$ comparaisons
- pour $i = 1$ on en aura $N - 2$

7. CUDA pour *Compute Unified Device Architecture* est une technologie mise au point par NVidia pour faire du calcul parallèle sur carte graphique.

```
// nombre d'enregistrements
const int N = 7000000;
// nombre de champs
const int P = 10;

int *enr = new int [N * P];

void recherche(int *enr, bool *elimine) {
    for (int i = 0; i < N-1; ++i) {
        for (int j = i+1; j < N; ++j) {
            if (identique(&enr[i * P], &enr[j * P]) {
                elimine[j] = true;
            }
        }
    }
}
```

Listing 1.3.6 – Recherche de doublons, version simpliste

- ainsi de suite jusqu'à $i = N - 1$, pour lequel on aura une comparaison

Au final on a :

$$\sum_{i=1}^{N-1} i = N \times \frac{N-1}{2}$$

ce qui est proportionnel à N^2.

L'étudiante n'a fait que transposer le code du Listing 1.3.6 sur CUDA. Le premier thread doit donc comparer le premier enregistrement aux $N - 1$ autres enregistrements ce qui est totalement inefficace que ce soit sur une carte graphique ou sur un microprocesseur et c'est ce qui provoque le timeout sur la carte graphique.

Se pose alors l'épineux problème de diminuer la complexité de l'algorithme de recherche de doublons. On diminuera la complexité si on ne doit pas comparer l'enregistrement i aux $N - i$ suivants mais à un plus petit nombre d'enregistrements. Il faut donc trouver un moyen de classer les enregistrements qui sont similaires ou identiques afin de les comparer par la suite. Dans ce but, on peut envisager :

- d'ajouter un champ qui contient une valeur de hachage de l'enregistrement
- de trier les enregistrements suivant la valeur de hachage
- et de finalement comparer uniquement les enregistrements qui ont la même valeur de hachage

Pour rappel, une valeur de *hachage* (*hash value* en anglais) est une valeur entière qui résulte d'un calcul qui prend en compte tout ou partie des champs d'une structure de données et qui a pour but d'identifier de manière unique l'enregistrement.

On peut la voir comme une *signature* des données qu'elle représente. Malheureusement il est difficile de trouver une fonction de hachage qui donne une valeur unique pour chaque enregistrement dès lors que le nombre d'enregistrements est important. Cependant, si cette valeur de hachage permet de distinguer un grand nombre d'enregistrements alors elle peut se révéler utile. Lorsque deux structures de données différentes possèdent la même valeur de hachage on parle de *collision*. Dans certains cas, la collision est problématique si on désire distinguer de manière unique chaque donnée.

Il semble que les fonctions de hachage de type FNV[8] soient très intéressantes car elles permettent une bonne séparation ou distinction des données.

En utilisant la méthode que nous venons de décrire la complexité diminue et se résume à celle du tri des enregistrements qui sera de l'ordre de $N \times \log(N)$, si on choisit un algorithme de tri efficace (cf. section précédente).

Temps de calcul doublons

Au final, sur un microprocesseur récent, pour 7 millions d'enregistrements, la première méthode prendra une quinzaine d'heures pour terminer alors que la deuxième prendra quelques secondes. Soit une amélioration drastique !

Là encore, réfléchir au problème avant de le résoudre et donc choisir le bon algorithme apporte un gain conséquent en terme de temps de calcul. C'est ce genre d'expérience qu'il faut acquérir au cours des ans et parfaire sa connaissance des algorithmes ainsi que des matériels afin de répondre au mieux aux problèmes qui nous sont posés, à nous, informaticiens.

Imaginons que vous soyez un *mauvais* informaticien, votre méthode de résolution prendra donc une quinzaine d'heures pour résoudre le problème précédent. Si votre patron vous indique maintenant qu'il a un client qui aura 1000 problèmes du même type à résoudre dans quelques mois et que ce client désire obtenir les résultats au bout d'une semaine après vous avoir fourni les données, une rapide analyse vous amènera à la conclusion qui suit.

Mille problèmes impliquent 15000 heures de calcul, soit environ 625 jours de calcul sur une seule machine. La solution, pour répondre en une semaine, consiste donc à disposer de plusieurs ordinateurs. Vous proposerez donc à votre patron d'acheter un cluster (ce qui risque de coûter assez cher) et ce cluster devra disposer d'au moins 90 coeurs[9] de calcul. Si dans deux ans, le client dispose non plus de 1000 mais de 10000 problèmes à traiter et qu'il désire toujours obtenir le résultat dans le même délai, cela implique de disposer de 10 fois plus de processeurs, soit plus de 890. Disposer d'un cluster dix fois plus gros engendre un coût non négligeable mais peut également conduire à un surcoût lié à l'achat d'un nouveau local adapté et dimensionné pour accueillir le cluster et le refroidir.

8. Fowler, Noll, Vo, voir http://www.isthe.com/chongo/tech/comp/fnv/index.html.
9. 15000 / (7 jour $\times$ 24 heures) = 89,28

Maintenant, si une nouvelle recrue se voit confier la tâche d'améliorer le temps de résolution et qu'en réfléchissant un peu elle parvient à imaginer que l'utilisation d'une fonction de hachage risque de diminuer le temps de calcul, vous risquez de vous attirer les foudres de votre patron et de passer pour un incapable aux yeux d'un bon nombre de personnes. En effet, si le problème est résoluble en 5 secondes avec fonction de hachage au lieu de 15 heures, alors résoudre 10000 problèmes prend 50000 secondes, soit un peu plus de 14 heures de calcul sur une seule machine. Votre manque de professionnalisme aura donc coûté très cher à votre entreprise.

A titre d'exercice, nous invitons le lecteur à tenter d'implanter la recherche de doublons comme nous l'avons expliquée.

1.4 Le Génie (du) logiciel

De manière générale l'intérêt ou l'engouement pour les sciences, au delà de la découverte, réside dans le fait qu'on est confronté quotidiennement à des problèmes et on se doit d'y apporter une solution, voire la meilleure solution. Parvenir à trouver une solution originale, performante apporte alors une grande satisfaction intellectuelle.

Le travail de l'informaticien consiste à faire exécuter par un ordinateur des traitements qui doivent être pensés pour être les plus efficaces par rapport au matériel dont il dispose. Les méthodes de développement logiciel issues de la mouvance *Agile* [10] préconisent au contraire de commencer par faire ce qui est simple plutôt que de faire ce qui est compliqué, et par conséquent, efficace. C'est le fameux principe KISS (*Keep It Stupidly Simple*). Bien entendu ce genre d'approche est discutable et possède des avantages comme des inconvénients.

Commencer par faire ce qui est simple (par exemple un tri à bulles) permet d'avancer plus vite dans le codage des diverses fonctionnalités d'un logiciel. Cependant, il faudra revenir par la suite sur le code et le modifier pour introduire de l'efficacité. Inversement utiliser des algorithmes efficaces va nous amener à ralentir la cadence de développement. Plus un algorithme est complexe, plus il faudra de temps pour le coder, le tester, et plus on a tendance à introduire de bogues.

Néanmoins, du point de vue utilisateur, c'est souvent l'efficacité qui prime dans le choix d'un logiciel. L'informaticien se trouve donc souvent dans cette position peu confortable, tiraillé entre deux choix contradictoires : utiliser un algorithme simple pour augmenter sa productivité mais ralentir les traitements, ou alors, produire du code efficace (donc complexe) pour diminuer le temps d'exécution des traitements mais ralentir sa productivité.

10. http://agilemanifesto.org/

Définition : Génie Logiciel

En Informatique, le **Génie Logiciel** est une discipline qui a pour but d'apprendre les méthodes qui permettent de mener à terme la réalisation d'un logiciel, en partant de l'expression du besoin d'un client et en passant par la conception, l'implantation, les tests, pour arriver jusqu'au déploiement de l'application et sa maintenance.

Les premières méthodes élaborées dans les années 1970 à 1980 avaient tendance à se fonder sur une approche considérée trop rigide. Pour caricaturer, encore une fois, ces méthodes fonctionnaient sur le modèle suivant : on allait voir le client et on comprenait (plus ou moins bien) son besoin, puis on concevait le logiciel pendant quelques mois et on présentait le résultat final au client. Le problème est que le résultat pouvait ne pas convenir au client :

- soit parce que pendant la phase de développement la vision du client ou son besoin évoluait,
- soit parce que le logiciel final n'était pas ergonomique,
- soit parce que les besoins du client avaient été mal compris par l'équipe de développement dès le début.

En effet, les développeurs ont leur propre vision du logiciel et les utilisateurs en ont une autre. Par exemple, les utilisateurs vont préférer appuyer sur une touche pour ouvrir une fenêtre qui contiendra l'information d'un client, plutôt que de fermer la fenêtre courante qui affiche sa commande et accèder par un menu à la fonctionnalité qui donne l'information du client. Parfois, pour le programmeur, il est plus pratique ou plus simple de faire un choix plutôt qu'un autre en raison de l'implantation qu'il a choisi alors que l'utilisateur se focalise sur l'ergonomie.

Toute modification du logiciel contraint les développeurs à repenser et modifier l'application ce qui peut engendrer plusieurs semaines ou mois de développement supplémentaires. Certaines entreprises, pour éviter ce genre de déconvenue, vont facturer très cher toute modification dans le but de faire comprendre au client que les changements sont pénalisants pour celui qui développe le logiciel et ont, par conséquent, des répercussions sur les délais et le budget alloué au logiciel.

Les méthodes *Agile*, évoquées précédemment, mises au point dans les années 1990 et 2000, tentent de répondre à cette problématique en intégrant le client à l'équipe de développement et en construisant le logiciel par groupes de fonctionnalités, c'est ce que l'on appelle le *développement itératif*. Après avoir développé quelques fonctionnalités pendant trois à quatre semaines, on présente le résultat au client et on prend en compte les modifications qu'il demande dans le prochain cycle de développement qui intègre également de nouvelles fonctionnalités.

Ces nouvelles méthodes *Agile* prônent également la simplication des procédures et l'adaptation (ou adaptabilité) au changement.

Simplifier les procédures signifie obtenir rapidement ce que l'on veut afin d'avancer rapidement et ne se concentrer que sur la tâche principale c'est à dire

développer le logiciel. Un développeur sera plus serein et efficace s'il peut remplacer son écran tombé en panne dans l'heure, plutôt que d'avoir à remplir un formulaire en trois exemplaires, le faire signer par son supérieur hierarchique et attendre quelques jours avant d'obtenir un nouvel écran.

L'adapation au changement, quant à elle, concerne aussi bien les besoins du client que l'arrivée ou le départ d'un nouveau collègue au sein de l'équipe de développement.

Même si les méthodes *Agile* connaissent un engouement certain, elles ne sont pas la panacée. Elles ont pour but, comme nous venons de le voir, l'adaptabilité aux besoins du client ou de l'équipe et ont tendance à considérer que *l'agilité*, qui signifie dans ce cadre, l'acceptation et l'adaptation au changement, lèvera beaucoup de verrous et permettra de surmonter de nombreux problèmes qui tendent à faire capoter certains projets qui, finalement, n'arrivent pas à terme ou qui ne respectent pas les délais ou le budget.

Malheureusement, ces méthodes ne fonctionnent pas toujours car elles oublient l'analogie avec la construction d'une maison que nous avons évoqué précédemment.

En effet, s'adapter aux demandes de changement du client reviendrait, si nous reprenons notre analogie avec la construction d'une maison, à revoir le plan de la maison, à détruire certaines pièces pour en créer de nouvelles, à supprimer des câbles pour en faire passer de nouveaux à un autre endroit. Si une maison devait être construite ainsi, en modifiant les plans en cours de construction, il est presque certain qu'elle n'arriverait pas à terme ou que le résultat serait décevant pour le client. On comprend bien qu'une telle approche risquerait de grèver le budget de construction. Sans compter la démotivation de l'équipe de construction qui serait contrainte de défaire et refaire son travail plusieurs fois et aurait le sentiment de stagner.

Les changements au niveau du logiciel peuvent donc intervenir mais à la marge, pas en profondeur, ce qui impose de bien réfléchir au préalable à l'architecture du logiciel à concevoir.

Un autre facteur qui semble totalement négligé par la totalité des méthodes de génie logiciel est le fait que la vision du logiciel que possède l'équipe de développement évolue au fur à mesure de sa construction. Si vous demandez à un développeur, une fois le logiciel opérationnel, ce qu'il pourrait améliorer, il vous dira sans doute qu'avec le recul et la vision globale qu'il en a, s'il devait refaire le logiciel, il procéderait autrement pour implanter telle partie, qu'il aurait conçu les classes de manière différente, etc. Même si le logiciel fonctionne, le fait qu'il puisse apparaître mal conçu est un facteur psychologique qui peut impacter de manière significative la motivation du développeur et influencer sa volonté à continuer de le modifier ou de l'améliorer.

1.5 Conclusion

Pour résumer, il est primordial pour l'informaticien de disposer d'une connaissance approfondie de son outil de travail. Avant de se lancer dans l'écriture du code source d'une application, il est nécessaire de réfléchir de manière posée et de s'interroger afin de trouver la meilleure architecture possible pour le logiciel que l'on doit concevoir ainsi que la meilleure organisation sous forme de classes, de méthodes et de coopérations entre les classes. Une séance de *brainstorming* à plusieurs est souvent salutaire car on ne pense pas toujours à tout et les autres peuvent nous aider dans notre réflexion globale.

Lors de l'écriture du code, il est du devoir du développeur de **bien commenter son code**, c'est à dire d'expliquer pourquoi une classe ou une méthode existe, quel est son rôle, comment elle réalise le traitement qui lui est demandé, quels sont les paramètres à fournir et quel est le résultat attendu.

Cette nécessité de réflexion et de documentation est d'autant plus vitale que l'on travaille à bas niveau, comme en assembleur, car la relecture du code peut être fastidieuse, d'autant plus qu'il n'y a pas de structures de contrôle. Elle est vitale pour une personne qui serait amenée à relire votre code, mais également pour soi-même. Lorsqu'on laisse de côté un projet qu'on avait commencé et que l'on continue son développement quelques semaines ou mois plus tard, on se demande souvent comment on a réalisé telle fonction. Si on dipose de commentaires de qualité, il sera alors plus simple de progresser.

1.6 Exercices

Exercice 1 - En utilisant les entiers 32 bits du langage C (`int`) écrire un programme qui fait la somme des entiers de 1 à n et trouver à partir de quelle valeur de n, la somme, qui est également de type `int`, n'est plus correcte.

On n'oubliera pas d'inclure l'option de compilation `-fwrapv` de **g++** pour obtenir une comparaison exacte.

Exercice 2 - Reprendre l'exercice précédent mais avec les entiers 32 bits non signés.

Chapitre 2

Représentation de l'information

Karla Mangeait
de Grandes Tortillas
et du Pain de Élote,
Zen, sur son Yacht

Dans ce chapitre nous allons découvrir comment est modélisée l'information afin de pouvoir être traitée par le microprocesseur car le fonctionnement des ordinateurs se fonde sur un modèle logique ou binaire, c'est à dire, un modèle à 2 états distincts qui sont le 1 ou le 0, le vrai ou le faux, l'ouvert ou le fermé. Ce modèle binaire (ou base 2) est utilisé pour représenter l'information de différentes façons en fonction des données à traiter. La compréhension de la représentation de l'information est également essentielle lorsque l'on programme en assembleur car elle permet de réaliser certaines fonctionnalités très rapidement (cf. valeur absolue de la Section 2.4) ou le calcul de certaines valeurs (voir Chapitre 12).

2.1 Introduction

Etre informaticien demande de penser d'une certaine manière qui est différente de la manière de penser des mathématiciens : un informaticien d'un bon niveau ne fera pas forcément un mathématicien d'un bon niveau et inversement.

Par exemple : les mathématiciens travaillent avec la notion d'infini alors que les informaticiens travaillent dans des domaines finis : la taille de la mémoire, la taille du disque dur, le nombre de processeurs utilisés pour réaliser un calcul en parallèle, toutes ces quantités sont finies.

Un mathématicien peut dire que : quand n tend vers l'infini, $1/n$ tend vers 0 mais ne sera jamais égal à 0. Pour un informaticien, à partir d'une certaine valeur de n, il remplacera $1/n$ par 0 car il aura dépassé la capacité de représentation d'un très petit nombre.

Du point de vue de la démarche, un mathématicien va démontrer qu'un problème admet ou non des solutions dans telles conditions mais sans donner ces solutions. La réponse sera généralement de type oui ou non : oui, le problème admet une solution, ou non, il n'en admet pas. le mathématicien peut également nous indiquer comment construire une solution.

L'informaticien va s'attacher à trouver une, ou toutes les solutions, ou à prouver qu'on ne trouvera pas de solution en résolvant le problème : c'est à dire en tentant de trouver une solution et en ne pouvant, au final, n'en trouver aucune en ayant testé tous les cas possibles ; la réponse sera une solution, la ou les meilleures pour un critère donné, ou aucune.

Au niveau de la machine l'information est représentée sous forme binaire avec des suites de 0 et de 1. Il est donc primordial de comprendre comment l'information (entiers, réels, texte) est représentée en informatique si on désire raisonner comme un informaticien puisque c'est de cette représentation :

- que l'on peut déduire la limite des calculs possibles que l'on pourra réaliser
- mais également, trouver les traitements les plus efficaces pour résoudre un problème donné

A titre d'exemple, considérons un traitement qui s'attache à déterminer si un nombre entier est impair ou, en d'autres termes, comment sait-on qu'un nombre entier est impair ?

Facile, me direz-vous, il suffit que ce nombre se termine par l'un des chiffres suivants : 1, 3, 5, 7, 9. Mais comment procéder avec un ordinateur ?

Une première solution consiste à faire ce que font les humains : extraire le chiffre unité du nombre et le comparer à 1, 3, 5, 7 ou 9 :

```
#include <iostream>
using namespace std;

int main( int argc, char *argv[] ) {
    int x = 123789;

    if (argc > 1) x = atoi( argv[ 1 ] );

    // extraire l'unité
    int u = x % 10;

    // la comparer
    if ((u == 1) || (u == 3) || (u == 5) || (u == 7) || (u == 9)) {
        cout << x << " est impair" << endl;
    } else {
        cout << x << " est pair" << endl;
    }

    return EXIT_SUCCESS;
}
```

Voici le code assembleur x86 64 bits qui correspondrait au code C précédent pour la partie comparaison. Ici, on retourne la valeur 1 dans le registre **eax** pour indiquer que le nombre est impair et 0 pour indiquer qu'il est pair :

```
1  global est_impair
2
3  section .text
4
5  ; code 64 bits
6  ; bool est_impair(int n)
7  ; n => edi
8  est_impair:
9      mov     eax, edi    ; eax <- edi
10     xor     edx, edx    ; edx <- 0
11     mov     ecx, 10     ; ecx <- 10
12     div     ecx         ; eax <- eax / ecx, (u) edx <- eax % ecx
13     mov     eax, 1      ; eax <- 1, valeur de retour true
14     cmp     edx, 1      ; si u == 1 alors sortir de la fonction
15     je      .end
16     cmp     edx, 3      ; si u == 3 alors sortir de la fonction
17     je      .end
18     cmp     edx, 5      ; si u == 5 alors sortir de la fonction
19     je      .end
20     cmp     edx, 7      ; si u == 7 alors sortir de la fonction
21     je      .end
22     cmp     edx, 9      ; si u == 9 alors sortir de la fonction
23     je      .end
24     xor     eax, eax    ; sinon, le nombre est pair on sort avec
25                         ; la valeur 0 (false)
26 .end:
27     ret
```

Comme nous n'avons pas encore vu d'instructions assembleur, quelques explications s'imposent. Les lignes 9 à 12 calculent le reste de la division de n par 10, le modulo. Celui-ci est obtenu dans le registre **edx** après utilisation de l'instruction **div** qui réalise la division. On place ensuite en ligne 13 la valeur 1 (true) dans **eax** car c'est, par convention, ce registre qui contient la valeur retournée par la fonction. Les lignes 14 à 23 ne font que comparer le reste de la division à 1, 3, 5, 7 puis 9, et, s'il s'agit de l'une de ces valeurs, on se dirige directement vers la sortie de la fonction. Finalement, si le reste n'est pas un chiffre impair, on met, en ligne 24, **eax** à 0 (false), puis on sort de la fonction.

Un informaticien ne procédera pas ainsi, il sait que la représentation binaire des nombres fait que, si un nombre est impair, il possède son premier bit (bit en position 0) à 1, étant donné que c'est la seule puissance de 2 impaire. Il effectuera donc un ET-binaire avec le nombre et vérifiera que le résultat est égal à 1 ou qu'il est différent de 0, ce qui revient au même :

```
1  #include <iostream>
2  using namespace std;
3
```

```
4  int main( int argc, char *argv[] ) {
5      int x = 123789;
6
7      if (argc > 1) x = atoi( argv[ 1 ] );
8
9      if ((x & 1) != 0) {
10         cout << x << " est impair" << endl;
11     } else {
12         cout << x << " est pair" << endl;
13     }
14
15     return EXIT_SUCCESS;
16 }
```

Au final, le calcul réalisé par un informaticien, ou tout au moins une personne qui possède des connaissances en informatique, est moins coûteux en temps de calcul et moins soumis à certains aléas.

Un test réalisé pour comparer les deux méthodes (cf. Table 2.1), et, qui consiste à répéter 50_000 fois l'application de l'une des deux fonctions précédentes sur les éléments d'un tableau de 100_000 entiers générés dans des conditions spécifiques (voir ci-après), donne les résultats suivants :

Initialisation	Méthode 1	Méthode 2
aléatoire	38.42	5.28
...1	12.04	5.44
...3	12.23	5.41
...5	12.45	5.29
...7	12.86	5.40
...9	14.81	5.50
pairs	12.46	5.49

TABLE 2.1 – Temps d'exécution (en secondes) des méthodes en fonction des nombres à traiter sur Intel Core i7-10850H

La méthode d'initialisation des éléments du tableau peut être :

- aléatoire : on aura autant de nombres pairs que de nombres impairs
- ne générer que des nombres impairs se terminant par 1, 3, 5, 7 ou 9
- ne générer que des nombres pairs

L'analyse des résultats montre que la méthode 1, traduction de la manière dont procède un programmeur non expérimenté, est sensible aux données et se révèle toujours moins efficace que la méthode 2. En effet, trouver si un nombre se termine par 3 prend plus de temps que comparer si un nombre se termine par 1 car on

effectue un test supplémentaire, et ainsi de suite jusqu'à comparer si un nombre se termine par 9, comme le montre le code assembleur ci-dessus.

Dans le cas de données aléatoires (nombres pairs ou impairs sans ordre précis), on note que le temps d'exécution est prohibitif (exorbitant) avec la méthode 1. Cela est dû à la prédiction de branchement (cf. Section 3.7.1) qui ne peut déterminer sur quelle valeur de l'unité sortir de la fonction.

Dans ce cas, la méthode 2 est $7,27$ ($= 38,42/5,28$) fois plus performante que la méthode 1.

2.2 Représentation des entiers

Pour représenter un nombre entier naturel dans une base b, il faut disposer de b chiffres distincts allant de 0 à $b-1$. Tout nombre n s'exprime alors sous la forme :

$$n = \sum_{i=0}^{k} a_i \times b^i \tag{2.1}$$

où chaque a_i est un chiffre. Ainsi en base 10, on peut écrire :

$$\begin{aligned} 1975_{10} &= 1 \times 1000 + 9 \times 100 + 7 \times 10 + 5 \times 1 \\ &= 1 \times 10^3 + 9 \times 10^2 + 7 \times 10^1 + 5 \times 10^0 \end{aligned}$$

En informatique, on utilise la base 2 ou binaire mais il est parfois plus facile d'utiliser d'autres bases comme l'octal (base 8) ou l'hexadécimal (base 16) afin de représenter de grandes quantités ou de faire des calculs.

2.2.1 Le binaire

Dans la notation binaire, également appelée base 2, on ne dispose que de deux chiffres 0 et 1. Par exemple 11001_2 représente la valeur décimale 25 :

$$11001_2 = 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^0 = 16_{10} + 8_{10} + 1_{10} = 25_{10}$$

Notation

J'ai choisi de mettre en indice de chaque nombre la base à laquelle il se rapporte. Quand on ne le précise pas il s'agit par défaut de la base 10.
Dans la suite de l'ouvrage, afin d'améliorer la lisibilité des nombres j'utilise le symbole souligné (_) après chaque quartet pour les nombres binaires ou chaque triplet pour les nombres décimaux :

$$11010101010_2 = 110\_1010\_1010_2 = 1706_{10} = 1\_706_{10} = 1\_706$$

On poura également utiliser la notation suivante :

- *b* pour le binaire : 1011_0001_b ou 1011_0001_2
- *o* pour l'octal : 261_o ou 261_8
- *d* pour le décimal : 177_d ou 177_{10} ou 177
- *h* pour l'héxadécimal : $B1_h$ ou $B1_{16}$

En binaire un chiffre est appelé un **bit** pour *BInary digiT*. On distingue généralement dans un nombre binaire le bit (où chiffre) le plus à gauche qui est appelé bit de *poids fort* (ou *most significant bit* en anglais) et le bit le plus à droite appelé bit de *poids faible* (ou *least significant bit*).

2^n		Valeur	2^n		Valeur
2^0	=	1	2^8	=	256
2^1	=	2	2^9	=	512
2^2	=	4	2^{10}	=	1_024
2^3	=	8	2^{11}	=	2_048
2^4	=	16	2^{12}	=	4_096
2^5	=	32	2^{13}	=	8_192
2^6	=	64	2^{14}	=	16_384
2^7	=	128	2^{15}	=	32_768
2^{16}	=	65_536	2^{31}	=	2_147_483_648
2^{17}	=	131_072	2^{32}	=	4_294_967_296
2^{63}	=	9_223_372_036_854_775_808	2^{64}	=	18_446_744_073_709_551_616

TABLE 2.3 – Liste de puissances de 2 de 2^0 à 2^{64}

Etant donné qu'en informatique on travaille toujours sur une quantité finie, on a introduit des termes pour identifier un nombre de bits consécutifs déterminé :

- un ensemble de 4 bits consécutifs est appelé un *quartet*
- un ensemble de 8 bits consécutifs est appelé un *octet* (*byte* en anlgais)
- deux octets consécutifs (16 bits) forment un *mot* (*word*)

- quatre octets consécutifs (32 bits) forment un *double mot* (*double word*)
- huits octets consécutifs (64 bits) forment un *quadruple mot* (*quad word*)
- seize octets consécutifs (128 bits) forment un *double quadruple mot* (*double quad word*)

Quand on travaille en tant qu'informaticien il est généralement très utile de connaître les puissances de 2 allant de 2^1 jusqu'à 2^{16} (voire jusqu'à 2^{20}) car cela permet de réaliser certains calculs de tête. Je recommande à tout informaticien d'apprendre la Table 2.3.

Il est également nécessaire de connaître les puissances de 2 proches des puissances de 10 (cf. Table 2.4) puisqu'elle définissent la taille des mémoires et espaces de stockage. Elles sont basées sur les préfixes du Système International (SI) d'unités et simplifient la manipulation des grandes quantités.

Puissance	Préfixe	Puissance	Préfixe
2^{10}	kilo	2^{50}	Peta
2^{20}	Mega	2^{60}	Exa
2^{30}	Giga	2^{70}	Zetta
2^{40}	Tera	2^{80}	Yotta

TABLE 2.4 – Liste de puissances de 2 liées aux puissances de 10

La phrase introductive de ce chapitre reprend dans l'ordre, pour chaque mot, la première lettre des préfixes : ainsi le K de Karla correspond à kilo, le M de Mangeait correspond à Mega, etc. Il s'agit d'un moyen mnémotechnique pour se rappeler l'ordre des préfixes du SI.

Certains constructeurs comme les fabricants de disques durs préfèrent utiliser 10^3 au lieu de 1024. Un disque de 80 *Giga* octets, qui possède un espace de stockage de 80×10^9 octets, apparaît pour le système d'exploitation comme un disque de 74 Giga octets [1].

2.2.2 L'octal

La base 8 est utilisée pour représenter des octets comme par exemple des caractères mais elle est en général peu usitée. On la retrouve lors de l'utilisation de commandes Unix comme `chmod` qui change les droits d'un fichier ou `tr` qui permet de transposer ou d'éliminer des caractères dans un fichier ou un flux de données. Voici, par exemple, deux commandes Unix qui utilisent l'octal :

```
1 richer@universe:~$ chmod 644 fichier
2 richer@universe:~$ tr ':' '\012' < fichier
```

1. $80 \times 10^9 / 1024^3 = 74,5$

La première ligne donne au propriétaire les droits de modification et lecture, aux membres du groupe et aux autres uniquement les droits de lecture sur le fichier. La seconde permet de remplacer le caractère `':'` par un saut de ligne car $12_8 = 10$ ce qui correspond au caractère `'\n'`. Il faut noter que le nombre commence par un 0 qui indique qu'il faut lire la valeur en octal.

Binaire vers l'octal

Le passage du binaire à l'octal est simple puisqu'un triplet (3 bits consécutifs) correspond à un chiffre octal.

2.2.3 L'hexadécimal

La base 16 permet de représenter des adresses ou des nombres utilisant plusieurs bits comme les double et quadruple mots. Ainsi un double mot qui occupe 32 bits, soit 32 chiffres en binaire, utilise seulement 8 chiffres hexadécimaux.

Dans la base 16 on utilise les chiffres 0 à 9 ainsi que des lettres pour représenter les chiffres supérieurs ou égaux à 10 en partant de A qui vaut 10 pour aller jusqu'à F qui vaut 15 en décimal :

$$\begin{aligned} \mathrm{A2F8}_{16} &= \mathrm{A} \times 16^3 + 2 \times 16^2 + \mathrm{F} \times 16^1 + 8 \times 16^0 \quad = \quad 41\_720 \\ &= 10 \times 16^3 + 2 \times 16^2 + 15 \times 16^1 + 8 \times 16^0 \end{aligned}$$

On remarquera qu'en C ou en assembleur on peut écrire les nombres hexadécimaux en les préfixant avec `0x`, on écrira donc `0xA2F8`.

Binaire vers l'hexadécimal

Le passage du binaire à l'hexadécimal est simple puisqu'un quartet (4 bits consécutifs) correspond à un chiffre hexadécimal.

2.2.4 Les entiers naturels

Les entiers naturels $\mathbb{N}$ sont des entiers positifs ou nul, ils sont généralement représentés en langage C par le type `unsigned int` ou encore par `uint32_t` du fichier d'entête `stdint.h`, en d'autres termes il s'agit de valeurs dites non signées.

On a souvent besoin de convertir des nombres décimaux en binaire ou en hexadécimal dès lors que l'on programme en assembleur. Pour passer d'un nombre décimal en un nombre dans une autre base il existe plusieurs méthodes :

1. méthode des divisions successives
2. méthode des intervalles de puissances
3. méthode par complémentation

2.2.4.1 Méthode des divisions successives

On réalise des divisions successives par la base b du nombre n à convertir. On s'arrête lorsque le quotient de la division est inférieur à b, puis on prend le dernier quotient et les restes successifs obtenus lors des divisions (cf. Figure 2.1).

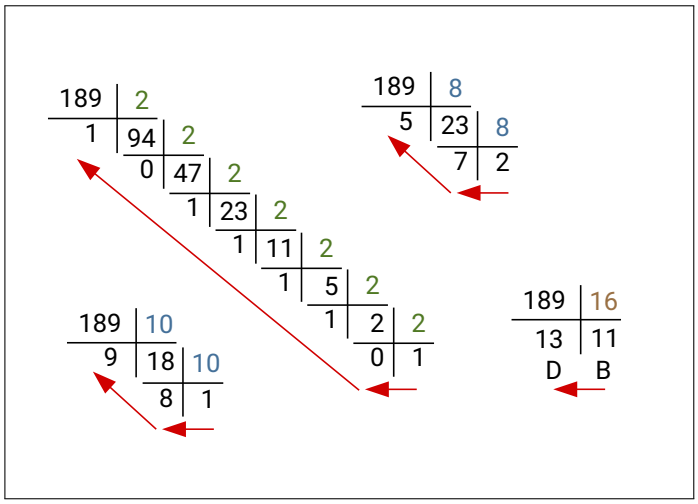


FIGURE 2.1 – Méthode des divisions successives par 2, 8 et 16

Ainsi le nombre 189 en décimal s'écrit également :

- 1011_1101_2
- 275_8
- BD_{16}

2.2.4.2 Méthode des intervalles de puissances

On applique ici la méthode au binaire mais elle peut être transposée à d'autres bases. Si on connait les puissances de 2, il est plus facile de convertir les grands nombres. En effet on n'est jamais à l'abris de commettre une erreur avec la méthode des divisions successives. Avec la méthode des intervalles, on cherche entre quelles puissances de 2 se trouve le nombre à convertir et on retranche la puissance la plus petite comme sur l'exemple suivant :

- 189 est compris entre 128 et 256, il contient donc $128 = 2^7$
- 189 - 128 = 61 est compris entre 32 et 64, il contient donc $32 = 2^5$
- 61 - 32 = 29 est compris entre 16 et 32, il contient donc $16 = 2^4$
- 29 - 16 = 13 est compris entre 8 et 16, il contient donc $8 = 2^3$
- 13 - 8 = 5, et finalement $5_{10} = 101_2$

On retrouve donc comme dans la méthode précédente que $189_{10} = 1011\_1101_2$.

2.2.4.3 Méthode par complémentation

Enfin, si le nombre à convertir se trouve proche d'une puissance 2^k, on peut procéder par complément c'est à dire en calculant $2^k - 1 - n$. Prenons un exemple : 32_745 est proche de $32\_768 = 2^{15}$. Si on calcule $(2^{15} - 1) - 32745$, on obtient 22 qui s'écrit en binaire 1_0110_2. Il suffit alors de retirer les bits à 1 du nombre 22 à $(2^{15} - 1)$ en utilisant par exemple l'opérateur *xor* (ou exclusif) :

$$\begin{array}{rrr} & 111\_1111\_1111\_1111_2 & (32767) \\ xor & 1\_0110_2 & (22) \\ \hline = & 111\_1111\_1110\_1001_2 & (32745) \end{array}$$

2.2.4.4 Intervalles de représentation

En informatique, on travaille à quantité finie et on utilise pour représenter l'information des octets, des mots, des double mots ou des quadruple mots. La Table 2.5 indique pour un nombre de bits fixés (n) le nombre de valeurs différentes que l'on peut représenter ainsi que les valeurs minimum et maximum si on utilise des valeurs non signées.

n	valeur minimum	valeur maximum	nombre de valeurs
8	0	255	256
16	0	65_535	65_536
32	0	4_294_967_295	4_294_967_296

TABLE 2.5 – Entiers naturels représentés avec un nombre fixé de bits

On peut résumer la Table 2.5 en disant qu'avec n bits on peut représenter 2^n entiers naturels différents allant de 0 à $2^n - 1$. Ainsi avec 32 bits on peut représenter un peu plus de 4 milliards de valeurs différentes.

2.2.4.5 Débordement

Que se passe t-il si on si essaye de représenter la valeur 259 sur 8 bits ? Si, par exemple, on ajoute 4 à 255, il se produit alors un dépassement de capacité (*overflow*) ou débordement et la valeur obtenue est alors égale à $259 \bmod 256 = 3$. On rappelle que le modulo est le reste de la division entière, ici, $259 = 256 \times 1 + 3$, donc 3 est le reste de la division entière de 259 par 256. La conséquence est que si on ne prête pas attention au calcul ce dernier risque d'être erroné.

Considérons, par exemple, les entiers non signés sur 32 bits. Si on réalise la somme des entiers naturels de 1 à n, à partir de quelle valeur de n la somme n'est-elle plus exacte ? Ce problème est laissé à titre d'exercice de programmation.

Vous devriez normalement trouver la valeur limite de $n = 92\_681$, ce qui correspond à la somme $4\_294\_930\_221$, soit une valeur proche de 2^{32}, au delà, le calcul est inexact.

2.2.5 Les entiers relatifs

L'ensemble des entiers relatifs $\mathbb{Z}$ représente les nombres entiers positifs, négatifs ou nul. En langage C, il s'agit du type `int` ou `int32_t`, c'est à dire des valeurs signées. Plusieurs représentations existent afin de pouvoir coder nombres positifs et négatifs mais on utilisera la notation *binaire en complément à deux* qui permet de réaliser des opérations arithmétiques dont le résultat sera correct. Dans cette notation, les nombres positifs utilisent le même procédé de réprésentation que la notation binaire de la section précédente.

Pour obtenir le codage en binaire en notation en complément à deux d'un nombre **négatif**, on procède en commençant par fixer la taille de l'espace de codage en nombre de bits, généralement 8, 16, 32 ou 64 bits. Prenons par exemple 8 bits. On réalise ensuite la série d'opérations suivantes :

1. on prend la valeur absolue $|x|$ du nombre que l'on code sur 8 bits
2. on réalise ensuite une opération de complémentation ($\bar{x}$) c'est à dire que l'on remplace les 0 par des 1 et inversement
3. on ajoute 1 au résultat final

Ainsi, pour coder la valeur $x = -1$ sur 8 bits en notation binaire en complément à deux, on obtient :

$$\begin{array}{rcl} |x| & = & 0000\_0001_2 \\ \bar{x} & = & 1111\_1110_2 \\ & + & 0000\_0001_2 \\ \hline & = & 1111\_1111_{\bar{2}} \end{array}$$

Notation

On remarque alors que si le bit de poids fort est à 0, il s'agit d'une valeur positive ou nulle. Par contre si le bit de poids fort est à 1, il s'agit d'une valeur négative.
On notera $x_{\bar{2}}$ un nombre en notation en complément à 2 afin de le différencier avec un nombre en notation binaire.

La Table 2.6 indique pour un nombre de bits fixés (n) quelles sont les valeurs minimum, maximum et le nombre de valeurs différentes que l'on peut représenter.

On peut résumer la table en disant qu'avec n bits on peut représenter 2^n entiers relatifs différents allant de -2^{n-1} à $+2^{n-1} - 1$.

n	valeur minimum	valeur maximum	nombre de valeurs
8	-128	127	256
16	-32_768	32_767	65_536
32	$-2\_147\_483\_648$	$2\_147\_483\_647$	$4\_294\_967\_296$

TABLE 2.6 – Entiers relatifs représentés avec un nombre fixe de bits

2.2.5.1 Débordement

De la même manière qu'avec les entiers naturels on peut avoir un dépassement de capacité. Fixons la représentation sur 8 bits et considérons le nombre 126 auquel on ajoute 3, on obtient alors 129 qui est en dehors de l'intervalle de représentation, puisque pour 8 bits, la valeur maximale que l'on peut représenter est 127. Mais sur 8 bits, $129_{10} = 1000\_0001_{\bar{2}}$. Etant donné que le bit de poids fort est à 1, cela signifie qu'on traite un nombre négatif ! Comment un nombre positif peut il être négatif ?

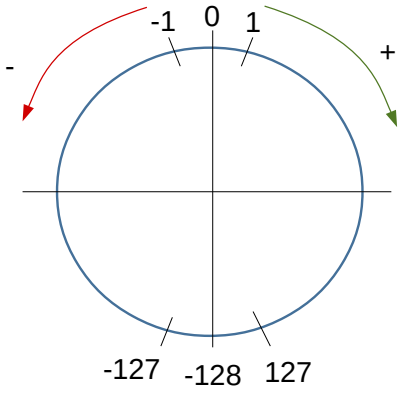


FIGURE 2.2 – Intervalle de représentation des entiers relatifs sur 8 bits

La raison est dûe au débordement. Comment savoir à quel nombre correspond $1000\_0001_{\bar{2}}$? Il suffit de réaliser les opérations inverses qui conduisent au codage des nombres négatifs :

1. on retranche 1
2. on complémente $\bar{x}$ chacun des bits

$$\begin{array}{rcl} y & = & 1000\_0001_{\bar{2}} \\ & - & 0000\_0001_2 \\ \hline & = & 1000\_0000_2 \\ \bar{y} & = & 0111\_1111_2 \end{array}$$

Donc $1000\_0001_{\bar{2}}$ correspond finalement à la représentation binaire en complément à deux de -127. Comme on peut le voir sur la Figure 2.2, à partir de 127, si on se déplace dans le sens des aiguilles d'une montre de deux positions, on se positionne sur -127. La boucle est bouclée !

Une autre méthode qui permet de savoir à quel nombre correspond $1000\_0001_{\bar{2}}$ consiste à ne pas considérer qu'il s'agit d'un nombre signé et de le convertir en décimal et le soutraire à 2^8, si on a choisi une représentation sur 8 bits. Ici on a $128 + 1 = 129$. On retranche alors 129 à 256 ce qui donne $256 - 129 = 127$. Le nombre correspondait alors à -127.

De la même manière, si on a $1111\_1011_{\bar{2}}$, il s'agit de $255 - 4 = 251$. Si on calcule $256 - 251$, on obtient 5. On en déduit que $1111\_1011_{\bar{2}}$ correspond à -5.

2.3 Calculs en binaire avec des entiers

Que ce soit en notation binaire naturelle pour les entiers non signés ou en notation binaire en complément à deux pour les entiers signés, on applique les mêmes schéma d'opérations qu'en arithmétique classique.

2.3.1 Addition

L'addition de deux nombres binaires est assez simple, il suffit d'appliquer les régles suivantes :

- $0_2 + 0_2 = 0_2$
- $0_2 + 1_2 = 1_2$
- $1_2 + 0_2 = 1_2$
- $1_2 + 1_2 = 10_2$, on abaisse le 0 et on génère une retenue de 1
- enfin le dernier cas correspond à une retenue en entrée de 1, dès lors $1_2 + 1_2 + 1_2 = 11_2$, on abaisse le premier 1 et on génère une retenue en sortie de 1

Ces règles s'appliquent aussi bien pour les nombres signés que les non signés.

Considérons une représentation des nombres sur 8 bits, pour le calcul de $1101\_1010_2 + 1110\_1111_2$. Dans ce cas on ne garde que les 8 premiers bits du résultats :

$Retenue(s)$	1	1	1	1	1	1		
	1	1	0	1	1	0	1	0
$+$	1	1	1	0	1	1	1	1
$=$	1	1	0	0	1	0	0	1

Le résultat est-il correct ? Il suffit de traduire les nombres binaires en décimal en décidant si on travaille avec des valeurs signées ou non signées :

- s'il s'agit de valeurs non signées, alors on effectue le calcul $218 + 239 = 457$ qui modulo 256 est égal à 201

- s'il s'agit de valeurs signées, alors on effectue le calcul $-38 + (-17) = -55$ qui est juste car $-55_{10} = 1100\_1001_{\bar{2}}$

2.3.2 Multiplication

La multiplication fonctionne comme en décimal :

$$
\begin{array}{cccccccccc}
 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
\times & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
\hline
 & & & & & & & & \\
 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
+ & & & & & & & & . \\
+ & 1 & 1 & 1 & 1 & 1 & 0 & . & . \\
+ & 1 & 1 & 1 & 1 & 0 & . & . & . \\
+ & 1 & 1 & 1 & 0 & . & . & . & . \\
+ & 1 & 1 & 0 & . & . & . & . & . \\
+ & 1 & 0 & . & . & . & . & . & . \\
+ & 0 & . & . & . & . & . & . & . \\
\hline
 & & & & & & & & \\
= & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0
\end{array}
$$

Le seul problème que l'on rencontre est celui de la somme des valeurs, on peut alors procéder de deux manières différentes :

- soit en faisant la somme comme on le ferait en décimal :

Colonnes	8	7	6	5	4	3	2	1
Retenues				1	←			
		1		←				
	1		←					
	1	←						
	1	1	1	1	1	1	1	0
+	1	1	1	1	1	0	0	0
+	1	1	1	1	0	0	0	0
+	1	1	1	0	0	0	0	0
+	1	1	0	0	0	0	0	0
+	1	0	0	0	0	0	0	0
=	0	0	0	0	0	1	1	0

Pour la cinquième colonne en partant de la droite on a une retenue en entrée et trois 1, ce qui fait 4, soit 100_2, on aura donc une retenue pour la septième colonne. De la même manière, la somme des valeurs de la sixième colonne donne $4 = 100_2$, on aura donc une retenue de 1 pour la huitième colonne. Pour la septième colonne on a 5 plus une retenue en entrée ce qui fait $6 = 110_2$, ce qui provoque une retenue pour la huitième et la neuvième colonne. Comme on a fixé une représentation sur 8 bits, la retenue pour la neuvième colonne ne doit pas être prise en compte. Enfin pour la huitième colonne, on a deux retenues en entrée plus 6, ce qui fait $8 = 1000_2$, on abaisse donc le premier 0 et on ne tient pas compte de la retenue pour la onzième colonne.

- soit en faisant des additions deux à deux :

$$\left.\begin{array}{rcccccccc} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ + & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{array}\right\} = 1\ 1\ 1\ 1\ 0\ 1\ 1\ 0\ (s1)$$

$$\left.\begin{array}{rcccccccc} & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ + & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{array}\right\} = 1\ 1\ 0\ 1\ 0\ 0\ 0\ 0\ (s2)$$

$$\left.\begin{array}{rcccccccc} & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ + & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right\} = 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ (s3)$$

$$\left.\begin{array}{rcccccccc} & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0\ (s1) \\ + & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0\ (s2) \end{array}\right\} = 1\ 1\ 0\ 0\ 0\ 1\ 1\ 0\ (s4)$$

$$\left.\begin{array}{rcccccccc} & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\ (s3) \\ + & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0\ (s4) \end{array}\right\} = 0\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ (s5)$$

Au final on obtient 0000_0110_2. Si on se place dans le cadre de valeurs non signées le calcul effectué est $254 \times 253 = 64\_262$ qui modulo 256 donne 6 car $64\_262 = 251 \times 256 + 6$. Si on considère que l'on traite des valeurs signées, on effectue le calcul $-2 \times -3 = 6$.

2.3.3 Soustraction

Le principe de la soustraction est le même qu'en décimal. Lorsque l'on calcule $25 - 16$, on commence à s'intéresser aux unités. 5 étant inférieur à 6, on ajoute une dizaine à 5 et on calcule $15 - 6$ ce qui donne 9. On passe ensuite aux dizaines. Sachant que l'on a ajouté une dizaine précédemment, on retire cette dizaine et on calcule $2 - 1$ auquel on retranche la dizaine, soit $2 - 1 - 1 = 0$.

Il suffit donc d'appliquer les régles suivantes :

- $0_2 - 0_2 = 0_2$
- $0_2 - 1_2$, étant donné que $0 < 1$ il faut calculer $10_2 - 1_2 = 1_2$ et propager une retenue de 1
- $1_2 - 0_2 = 1_2$
- $1_2 - 1_2 = 0_2$
- $0_2 - (1_2 + 1_2) = 0_2 - 10_2$, on ajoute une dizaine, ce qui donne $10_2 - 10_2 = 0_2$ et on propage une retenue de 1
- $1_2 - (1_2 + 1_2) = 1_2 - 10_2$, on ajoute une dizaine, ce qui donne $11_2 - 10_2 = 1_2$ et on propage une retenue de 1

Voyons cela sur un exemple et calculons $5 - 10$, soit en binaire sur 4 bits $0101_2 - 1010_2$. Dans ce cas on ne garde que les 4 premiers bits du résultats :

$$\begin{array}{rcccc} \text{Retenue(s) en sortie} & 1 & & 1 & \\ & 0 & 1 & 0 & 1 \\ - & 1 & 0 & 1 & 0 \\ \text{Retenue(s) en entrée} & 1 & 1 & & \\ \hline \\ = & 1 & 0 & 1 & 1 \end{array}$$

Soit au final 1011_2 qui dans le cadre de la représentation signée en complément à 2 correspond à -5. On a donc bien le résultat escompté.

2.3.3.1 Soustraire 1

Pour soustraire 1 d'un nombre binaire x, il suffit :

- si le nombre x se termine par un 1, de transformer ce 1 en 0

- par contre, si le nombre se termine par un ou plusieurs 0, il suffit de trouver le premier bit à 1, puis de complémenter sur cette partie

Par exemple :

- $1101_2 - 1_2 = 1100_2$
- $11\_1000_2 - 1_2$ donne 11_0111_2 puisqu'on prend le complément de 1000_2

2.3.4 Division

La division, tout comme en décimal, est difficile à appréhender. Elle consiste à diviser le dividende n par le diviseur d et obtenir un quotient q ainsi qu'un reste r. On a donc $n = q \times d + r$. On va considérer que $n \geq d$ par la suite.

Comment divise t-on en binaire ? Il suffit de rechercher la position (numéro du bit) dans le dividende n où il est possible de soustraire le diviseur d le plus à gauche possible, puis d'effectuer la soustraction. On réitère ensuite l'opération en plaçant un 1 à droite du quotient et en le décalant de $k - 1$ rangs vers la gauche lorsque $k - 1 \geq 0$, avec k qui représente la différence entre deux positions successives comme on peut le voir sur l'exemple de la Figure 2.3.

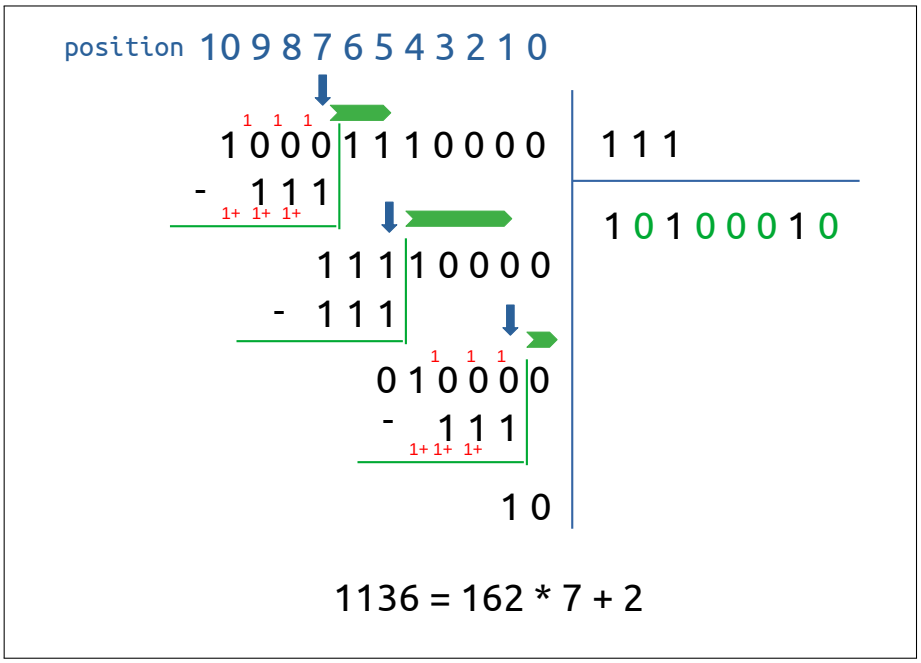


FIGURE 2.3 – Division binaire sur un exemple

Dans cet exemple, on divise 1136 par 7. C'est à partir de la position (ou bit) 7 que l'on obtient au niveau du dividende, un nombre plus grand que le diviseur, en l'occurrence $1000_2 = 8$. On soustrait alors 7 à 8, il nous reste 1 et on abaisse les chiffres restants du dividende. Etant donné que l'on vient de soustraire une fois 7 au dividende, on place un 1 à droite du quotient qui était initialement égal à 0.

On s'intéresse alors au dividende modifié qui est 1111_0000_2 et on trouve que l'on peut lui retrancher le diviseur $7 = 111_2$ à partir de la position 5. On calcule alors $k = 7 - 5 - 1 = 1$, il faut donc décaler le quotient de 1 rang vers la gauche. Le quotient est alors 10_2 et on place un 1 à droite du quotient qui devient 101_2 en raison de la soustraction effectuée.

Le dividende restant est alors 1_0000_2. On peut lui retrancher le diviseur à partir de la position 1. Dans ce cas, $k = 5 - 1 - 1 = 3$. On décale donc le quotient de 3 rangs vers la gauche, celui-ci devient alors 10_1000.

On réalise la soustraction du diviseur au dividende et on place un 1 à droite du quotient qui est à présent égal à 101_0001_2.

Le dividende devient 10_2, il est inférieur au diviseur donc on arrête la division, mais comme la dernière soustraction a été réalisée en position 1, il est nécessaire de décaler le quotient d'un rang vers la droite. Finalement le quotient est 1010_0010_2, soit 162 et le reste est de 2.

On peut en dégager l'algorithme suivant extrait d'une librairie C++ que j'ai écrite :

```
int pos = greater_or_equal_at(dividend, divisor);
while (pos >= 0)
{
    quotient.shl(1);
    quotient.set_bit(BIT_0, 1);
    sub_at(dividend, divisor, pos);
    int next_pos = greater_or_equal_at(dividend, divisor);
    int shift = pos - next_pos - 1;
    if (shift > 0) quotient.shl(shift);
    pos = next_pos;
}
```

Ce code repose sur l'utilisation d'une structure de données appelée `Bits` qui représente une suite de bits par un tableau de caractères, ainsi que sur l'utilisation de deux fonctions :

- `greater_or_equal_at` qui détermine à partir de quel bit dans le dividende on peut soustraire le diviseur de manière à ce que le résultat soit supérieur ou égal à 0

- `sub_at` qui réalise la soustraction à la position trouvée par le sous-programme précédent

2.4 Représentation des réels

La norme IEEE 754 (*Standard for Binary Floating-Point Arithmetic*) date de 1985 [2]. Elle définit initialement quatre représentations de nombres réels qui sont appelés nombres flottants ou nombres à virgule flottante en informatique [3] :

- simple précision (32 bits), correspond au type `float` du langage C
- double précision (64 bits), correspond au type `double` du langage C
- simple précision étendue (43 bits et plus)
- double précision étendue (79 bits et plus), on utilise généralement 80 bits ce qui correspond à la modélisation des nombres au sein du coprocesseur arithmétique (cf. Chapitre 7)

D'autres formats de représentations ont été ajoutés par la suite :

- la demi précision (*half precision*) qui code sur 16 bits et est utilisée dans le cadre des réseaux de neurones
- la quadruple précision qui code sur 128 bits
- l'octuple précision qui code sur 256 bits

Dans la norme IEEE 754, les nombres sont décomposés en trois parties distinctes :

- le Signe (S) qui vaut 0 pour un nombre positif et 1 pour un nombre négatif et qui correspondra au bit le plus significatif
- suivi de l'Exposant (E) que nous appelons `Exposant décalé` (E_d) car on lui ajoute une valeur positive
- et de la Mantisse (M) que nous qualifions de `Mantisse tronquée` M_t car on supprime le premier 1 une fois le nombre normalisé

On peut voir Table 2.7 les caractéristiques des nombres flottants en fonction de la précision de la réprésentation. Nous avons indiqué également les plus petites et plus grandes valeurs que l'on peut représenter.

Ainsi, dans le format IEEE 754 en 32 bits, un nombre n s'exprime par :

$$n = (-1)^S \times M \times 2^{(E)} = (-1)^S \times 1, M_t \times 2^{(E_d - 127)}$$

On voit que l'on retire 127 à l'exposant décalé car celui-ci est augmenté, par convention, de 127 comme nous allons le voir ci-après.

Précision	16	32 bits	64 bits	128 bits
Signe (bits)	1	1	1	1
Exposant (bits)	5	8	11	15
Mantisse (bits)	11	23	52	113
Plus petit nombre	$\pm 6,10310^{-5}$	$\pm 1,17510^{-38}$	$\pm 2,22510^{-308}$	$\pm 3.36210^{-4932}$
Plus grand nombre	± 65504	$\pm 1,70110^{38}$	$\pm 1,79710^{308}$	$\pm 1.18910^{4932}$
Décimales	3	7	16	34

TABLE 2.7 – Caractéristiques des nombres flottants en fonction de la précision.

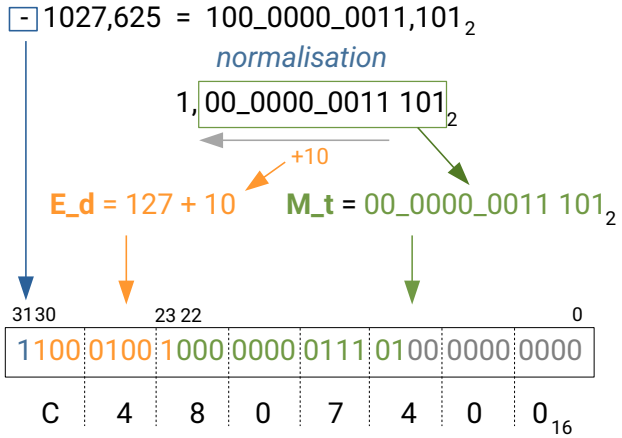


FIGURE 2.4 – Codage d'un nombre flottant en IEEE 754 32 bits

2.4.1 Codage

Comment coder un nombre réel au format IEEE 754 ? Prenons l'exemple de la représentation en simple précision sur 32 bits (cf. Figure 2.4) du codage de $n = -1027,625$. On procède comme suit :

- il s'agit d'un nombre négatif donc $S = 1$
- on code la partie entière en valeur absolue :

$$1027_{10} = 1024_{10} + 2_{10} + 1_{10} = 2^{10} + 2^1 + 2^0 = 100\_0000\_0011_2$$

- on code la partie décimale en utilisant des puissances de 2 négatives :

$$0,625 = 0,5 + 0,125 = 2^{-1} + 2^{-3}$$

2. Il est à noter que la norme IEEE 754 a été révisée en 2008 puis en 2015 et 2019.
3. Les Anglo-saxons utilisent le point pour représenter la virgule, on parle donc de *floating point number*.

- la mantisse qui regroupe partie entière et décimale est alors

$$M = 100\_0000\_0011,101_2$$

Pour obtenir la mantisse tronquée et l'exposant décalé, il suffit de déplacer la virgule vers la gauche derrière le premier 1, on parle alors de *normalisation* du nombre à représenter :

$$1,0000000011101_2$$

Par conséquent, on a déplacé la virgule de 10 rangs vers la gauche (voir Figure 2.4), ce qui correspond à $E = 10$.

- la mantisse tronquée est alors égale à la mantisse à laquelle on a enlevé le premier 1 devant la virgule, on obtient donc $M_t = 0000000011101_2$
- l'exposant décalé est égal, par convention en 32 bits, à $127 + E$, dans notre cas $E = 10$, donc :

$$E_d = 127 + 10 = 137_{10} = 1000\_1001_2$$

On remplit alors chacun des champs du nombre flottant (Figure 2.4) et on complète la mantisse tronquée par des zéros à droite. Au final on obtient une valeur sur 32 bits que l'on exprime généralement en hexadécimal pour plus de lisibilité. On obtient donc $C4\_80\_74\_00_{16}$.

2.4.2 Partie décimale

Pour coder la partie décimale d'un nombre il existe une autre méthode que celle qui consiste à sommer les puissances de deux négatives afin de retrouver la valeur cherchée.

Cette méthode consiste à multiplier la partie décimale par 2 jusqu'à obtenir 0 quand cela est possible.

A chaque étape on garde le chiffre le plus à gauche du résultat de la multiplication qui sera 1 ou 0 puis on réitère la multiplication sur la partie décimale du résultat de la multiplication en supprimant le premier 1 s'il existe.

Prenons un exemple simple, on désire obtenir le codage en binaire de $0,8125$:

n	$n \times 2$	r
$0,8125$	$1,625$	1
$0,625$	$1,25$	1
$0,25$	$0,5$	0
$0,5$	$1,0$	1
$0,0$		

- on multiplie $0,8125$ par deux ce qui donne $1,625$, on garde le premier chiffre 1 et on réitère sur $1,625 - 1$
- on multiplie $0,625$ par deux ce qui donne $1,25$, on garde le premier chiffre 1 et on réitère sur $1,25 - 1$
- on multiplie $0,25$ par deux ce qui donne $0,5$, on garde le premier chiffre 0 et on réitère sur $0,5 - 0$
- on multiplie $0,5$ par deux ce qui donne $1,0$, on garde le premier chiffre 1 et on s'arrête car $1 - 1 = 0$

Au final on obtient $0,8125 = 0,1101_2$.

Un exemple plus problématique est le codage de $0,3$:

n	$n \times 2$	r
$0,3$	$0,6$	0
$0,6$	$1,2$	1
$0,2$	$0,4$	0
$0,4$	$0,8$	0
$0,8$	$1,6$	1
$0,6$	$1,2$	1
$\ldots$		

Dans le cas de $0,3$ le résultat ne tombe pas juste et on obtient une séquence qui se répète à l'infini (et même au-delà) :

```
0, 0 1001 1001 1001 ...
```

Si on code $0,3$ au format IEEE 754 en 32 bits, on a :

- le signe est $S = 0$ car le nombre est positif
- la normalisation du nombre donne $M = 0,0100110011001...$ et donc $E = -2$ car on doit déplacer la virgule de deux rangs vers la droite pour atteindre le premier 1 du nombre
- en conséquence l'exposant décalé est de $E_d = 127 - 2 = 125 = 0111\_1101_2$
- la mantisse tronquée est $00110011...$
- la représentation de $0,3$ est donc `3E_99_99_9A_h`

2.4.3 Remarques

La constante 127 est utilisée pour les nombres flottants en simple précision afin de pouvoir coder les nombres dont la partie entière est égale à 0. Dans ce cas l'exposant E est négatif, par exemple $0,0625 = 2^{-4}$, on ajoute un décalage de 127 pour pouvoir représenter ces nombres.

Notons que l'assembleur `nasm` que nous utiliserons est capable de convertir automatiquement une valeur décimale en sa représentation IEEE 754, nous n'aurons donc pas à réaliser ces calculs fastidieux d'encodage des nombres à virgule flottante.

Symbole	Hexadécimal	Signe	Exposant	Mantisse
0.0	`00000000`	0	00000000_2	$00\ldots0_2$
∞	`7F800000`	0	11111111_2	$00\ldots0_2$
$-\infty$	`FF800000`	1	11111111_2	$00\ldots0_2$
$-NaN$	`FFC00000`	1	11111111_2	$10\ldots0_2$

TABLE 2.8 – Constantes prédéfinies pour les nombres en virgule flottante

Certaines valeurs ont une signification particulière (cf. Table 2.8). Notamment *NaN* qui en anglais signifie *Not a Number* et qui est utilisée pour signaler une erreur lors d'un calcul. Il existe deux types de valeurs NaN :

- qNaN ou *quiet* NaN, ne produit pas d'exception et sera propagée afin que le calcul se termine sans provoquer l'arrêt du programme
- sNaN ou *signaling* NaN est sensée provoquer une exception

Pour les système POSIX/Unix les exceptions générées lors de calculs sur les nombres à virgule flottante mettent fin à l'exécution du programme à moins qu'elles ne soient interceptées par un gestionnaire (*handler*) qui captera le signal **`SIGFPE`**[4].

On pourra utiliser le convertisseur IEEE-754 Floating Point[5] écrit en Javascript qui permet à partir d'un nombre réel d'obtenir son codage en norme IEEE 754 32 bits.

2.4.4 Erreurs de précision

On notera que les puissances de 2 négatives se terminent par des puissances de 5. Par exemple, 2^{-3} se termine par $5^3 = 125$.

Lorsque l'on utilise la représentation IEEE 754, on rencontre deux problèmes :

- le premier est la conséquence de l'utilisation des puissances de 2, car comme on le voit Table 2.9, les puissances de 2 négatives se terminent par 5, on ne peut donc coder la plupart des nombres décimaux qu'en utilisant une combinaison de puissances de 2 négatives et cela engendre une erreur de précision
- le second découle du premier et tient au fait que la taille de la mantisse peut être trop petite pour représenter certains nombres qui comportent beaucoup

4. SIGFPE signifie SIGnal Floating Point Exception (ou Error).
5. https://www.h-schmidt.net/FloatConverter/IEEE754.html

2^{-1}	=	$0,5$	5^1
2^{-2}	=	$0,25$	5^2
2^{-3}	=	$0,125$	5^3
2^{-4}	=	$0,0625$	5^4
2^{-5}	=	$0,03125$	5^5
2^{-6}	=	$0,015625$	5^6
2^{-7}	=	$0,0078125$	5^7
2^{-8}	=	$0,00390625$	5^8
...		...	
2^{-23}	=	$0.00000011920928955078125$	5^{23}

TABLE 2.9 – Puissances de 2 négatives

```cpp
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

float v1 = 1.2;
float v2 = 1.3;
float v3 = 1.3001;
float v4 = 1.3001001;

int main() {
    float diff_v1_v2 = v1 - v2;
    float diff_v2_v3 = v2 - v3;

    cout << setprecision(10);

    cout << "v1-v2 = " << diff_v1_v2 << endl;
    cout << "v2-v3 = " << diff_v2_v3 << endl;

    float diff_v3_v4 = fabs(v3 - v4);
    cout << "|v3-v4| = " << diff_v3_v4 << endl;

    if (diff_v3_v4 < 1E-6)
        cout << "v3 = v4" << endl;
    else
        cout << "v3 != v4" <<  endl;
    return 0;
}
```

Listing 2.4.1 – Précision et nombres flottants

de chiffres, notamment en 32 bits, car on dispose de 7 chiffres significatifs. C'est pour cela que le coprocesseur arithmétique qui réalise les opérations sur les nombres flottants utilise un codage sur 80 bits afin de minimiser les erreurs de précision

On peut voir sur le Listing 2.4.1 un exemple de code très simple qui réalise la différence entre des valeurs proches. Cependant le résultat de l'exécution ne correspond pas à ce que nous devrions obtenir :

```
v1-v2 = -0.09999990463     ! et non -0.1
v2-v3 = -0.0001000165939   ! et non -0.0001
|v3-v4| = 1.192092896e-07  ! et non  0.0000001
v3 = v4
```

- la différence `v1 - v2` devrait être égale à -0.1
- et celle de `v2 - v3` devrait être de -0.0001

Cela est dû au fait qu'il est impossible de coder exactement certaines valeurs comme nous l'avons expérimenté pour représenter $0,3$.

Attention

Le problème lié aux erreurs de précision implique que pour comparer deux valeurs en virgule flottante on ne peut pas utiliser l'opérateur d'égalité (==) du langage C comme on le ferait pour des entiers, il est nécessaire d'utiliser la valeur absolue de la différence des deux valeurs (ligne 20 du Listing 2.4.1) et de vérifier que cette différence est bien inférieure à un ϵ donné (ligne 23).

Si on utilise une précision plus grande de 64 bits, c'est à dire un `double` en langage C, on obtient un résultat qui correspond à un calcul exact :

```
v1-v2 = -0.1
v2-v3 = -0.0001
|v3-v4| = 1.000000001e-07
v3 = v4
```

Néanmoins, on obtiendra les mêmes erreurs de précision dès lors que les nombres à traiter possèdent un nombre de chiffres après la virgule important qui dépasse la capacité de représentation des nombres en double précision.

2.4.5 Intervalle et simple précision

La valeur décimale $0,3$ est codée au format IEEE 754 sur 32 bits par **3E99999A**. De même $0,4 =$ **3ECCCCCD**.

Si nous calculons la différence `0x3ECCCCCD` - `0x3E99999A`, nous obtenons `0x333333` $= 3\_355\_443$, c'est à dire que l'on peut coder un peu plus de $3,3$ millions de valeurs entre $0,3$ et $0,4$.

De la même manière si on code $1,0$ on obtient `0x3F800000`. Le nombre qui suit $1,0$ est donc `3F800001` qui correspond à la valeur décimale $1,0000001$, le suivant est `0x3F800002` qui correspond à la valeur décimale $1,0000002$. On a donc une précision de 7 chiffres après la virugle.

En revanche si on code 1024 on obtient `0x44800000`. Le nombre qui suit 1024 est, en hexadécimal, égal à `0x44800001` et correspond à $1024,0001$, soit une précision de 4 chiffres après la virgule.

Ceci montre qu'en représentation IEEE 754 sur 32 bits on dispose de 8 chiffres significatifs.

2.4.6 Valeur absolue

Pour conclure sur la partie liée au nombres flottants, demandons nous comment coder la fonction `fabs` qui calcule la valeur absolue d'un nombre. Logiquement cette fonction devrait être implantée sous la forme suivante :

```
float fabs(float v) {
    if (v >= 0.0) {
        return v;
    } else {
        return -v;
    }
}
```

C'est à dire qu'il faudrait changer le signe du nombre seulement s'il est négatif et cela implique donc de comparer v à 0.0. Mais il exite une méthode beaucoup plus simple pour obtenir la valeur absolue, puisque le bit de poids fort d'un flottant représente le signe du nombre. Il suffit donc de le mettre à 0 en utilisant un masque. On peut écrire cela en C++ de la manière suivante :

```
float v = -1.5;
unsigned int *p = reinterpret_cast<unsigned int *>(&v);
*p = (*p and 0x7FFFFFFF);
```

On convertit le nombre flottant 32 bits en un entier non signé 32 bits auquel on applique un masque qui préserve tous les bits sauf le bit de signe. En assembleur on peut traduire ce code par une seule instruction :

```
section .data
    v   dd  -1.5

```

```
4 section .text
5     and     dword [v], 0x7FFFFFFF
```

Il faut noter que l'on a utilisé ici une instruction (**and**) qui travaille sur les entiers, mais comme nous le verrons dans le Chapitre 7, il est normalement nécessaire d'utiliser les instructions liées au coprocesseur pour faire des calculs avec les réels.

2.4.7 Division entière non signée par un invariant

Dans le cas d'une division entière (non signée) sur 32 bits par une constante d, il est possible de rendre la division plus efficace en la remplaçant par une multiplication car on exécute alors $x \times (1/d)$. On va alors chercher a et s tels que :

$$\frac{1}{d} = \frac{a}{2^{32+s}}$$

car on ne peut pas représenter une valeur inférieure à 0 comme $0,3$ avec des entiers. Dans ce cas, $x/d = x \times a$ suivi d'un décalage à droite de $32 + s$ bits car la multiplication des deux valeurs 32 bits donne un résultat sur 64 bits.

Prenons un exemple concret avec $d = 10$. Dans ce cas, $a = \text{CCCCCCCD}_{16} = 3\_435\_973\_837$ et $s = 3$. Si on prend $x = 173$, on obtient alors :

$$\begin{aligned} x \times a &= 173 \times 3\_435\_973\_837 \\ &= 594\_423\_473\_801 \\ &= \text{8A\_66\_66\_66\_89}_{16} \\ &= 1000\_1010 \times 2^{32} + \ldots\_1001_2 \end{aligned}$$

Le décalage de 35 bits vers la droite ou dans le cas présent de la partie haute du résultat ($8A_{16}$) de 3 bits vers la droite, donne au final $1\_0001_2 = 17$ et permet d'obtenir un résultat sur 32 bits.

Si on reste en 32 bits, on utilisera le code qui suit pour lequel la partie haute de la multiplication (**edx**) sera décalé de $35 - 32 = 3$ bits à droite :

```
1 mov     eax, 173          ; x
2 mov     edx, 0xCCCCCCCD   ; a
3 mul     edx
4 mov     eax, edx
5 shr     eax, 3
```

Comment trouve t-on a et s ? C'est assez simple, il suffit de calculer $1/d$ et le coder sous forme d'un flottant. Dans l'exemple précédent, $1/d = 0,1$, soit au format IEEE 754 : 3D_CC_CC_CD_{16}. Pour avoir une meilleure précision, on code sur 64 bits sous forme d'un **double**, ce qui donne $\text{3FB9\_9999\_9999\_999A}_{16}$. La taille de l'exposant étant de 11 bits dans le format IEEE 754 64 bits :

- on décale le nombre de 64 bits de 11 bits vers la gauche

$$01001100110011001\ldots_2 = 4\text{CCC}\ldots_{16}$$

- on fixe à 1 le bit de poids fort (bit 63) pour obtenir la mantisse, car on ne disposait que de la mantisse tronquée

$$11001100110011001\ldots_2 = \text{CCC}\ldots\text{C}_{16}$$

- on décale de 32 bits vers la droite pour obtenir une valeur sur 32 bits
- on fixe le bit de poids faible (bit 0) à 1

On retrouve alors CCCCCCCD_{16}.

Une fois qu'on a déterminé la valeur de a, il est assez simple de trouver s, en testant par une boucle `for` le décalage qui donnera le résultat escompté ou en utilisant les instructions assembleur telles que `bsr` ou `bsf`. Le code C correspondant est le suivant et devrait donner dans la majorité des cas les valeurs de a et de s de manière précise :

```
void find_number_and_shift( u32 d ) {

    // on calcule 1/d sous forme d'une double
    double ratio = 1.0 / d;
    u64 *a = (u64 *) &ratio;

    // décalage de 11 bits (exposant)
    // et on fixe le bit de la mantisse tronquée
    // on décale ensuite de 32 bits
    *a = (((*a << 11) | 0x8000000000000000)) >> 32;

    // on fixe le bit de poids faible
    *a = *a | 0x01;

    u32 shift;
    u64 prod = (*a) * d;
    for (shift = 32; shift < 63; ++shift) {
        u64 r = prod >> shift;
        if (r == 1) {
            break;
        }
    }
    cout << "a=" << hex << a << endl;
    cout << "s=" << dec << shift << endl;
}
```

Un test simple qui consiste à réaliser 10 milliards de divisions par différentes valeurs (11, 127, 1027, 11279, 44567187) en emlpoyant soit l'instruction `div` de l'assembleur, soit la multiplication avec décalage pour $d = 10$, donne les temps d'exécution reportés Table 2.10.

On voit clairement que la multiplication suivie d'un décalage est plus performante que la division. On va notamment $3,16$ fois plus vite sur AMD Ryzen 5600g.

Méthode	AMD Ryzen 5 5600g	Intel Core i5 7400	Intel Core i5 12400f	Intel Core i5 10850H
division (div)	13,53	19,67	13,95	13,60
mult + décalage	4,27	7,70	3,37	5,32
gain	$\times 3,16$	$\times 2,55$	$\times 4,13$	$\times 2,55$

TABLE 2.10 – 10 milliards de divisions sur différentes architectures

2.5 Représentation des chaînes de caractères

2.5.1 L'ASCII

Le stockage des caractères ainsi que des chaînes de caractères est généralement réalisé en ASCII 8 bits afin de pouvoir coder sur un octet 256 valeurs différentes. Dans un langage comme le C cela est suffisant si on utilise les langues européennes.

Dans le codage ASCII (voir www.ascii-code.com ainsi qu'en annexe de cet ouvrage), les caractères 0 à 31 sont des caractères de contrôle qui ne représentent pas un symbole mais permettent la mise en page de texte (comme le saut de page FF, le saut de ligne LF, le retour-chariot CR ou la tabulation horizontale HT), ou la transmission d'information pour les liaisons RS232 (port série) comme STX et ETX.

- les plages de caractères de 32 à 47, 58 à 64, 91 à 96, 123 à 126 représentent des symboles tels que l'espace, les opérations arithmétiques, les signes de ponctuations (virgule, point, point-virgule, etc), les parenthèses, les crochets,
- les caractères 48 à 57 sont les chiffres
- les lettres majuscules occupent la plage 65 à 90, alors que les lettres minuscules s'étendent de 97 à 122
- de 128 à 255 les caractères codés sont les lettres avec accents ainsi que des symboles mathématiques ou de ponctuation et des symboles qui permettent la mise en forme de tableaux

On notera que la distance entre les majuscules et minuscules est de 32. Ainsi pour transformer `'A'` en `'a'`, il suffit d'ajouter 32 au code ASCII de `'A'`. Du point de vue du binaire, il suffit de positionner le bit 5 à 1, puisque $2^5 = 32$.

En langage C le codage des chaînes consiste à stocker l'ensemble des caractères de la chaîne de manière contigüe (consécutive) puis à marquer la fin de chaîne par le caractère 0, représenté en en C par `'\0'`. Cette représentation possède l'avantage de pouvoir coder des chaînes très longues puisqu'elle ne pose aucune limitation sur la longueur, si ce n'est celle de la mémoire. Cependant, elle possède un inconvénient dû au fait qu'on ne peut connaître la longueur de la chaîne qu'en la parcourant.

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <cstring>
4  #include <ctype.h>
5  using namespace std;
6
7  int main() {
8      char chaine[] = "abracadabra...";
9
10     int longueur = strlen(chaine);
11     for (int i = 0; i < longueur; ++i) {
12         if (isalpha(chaine[i])) chaine[i] = toupper(chaine[i]);
13     }
14
15     cout << chaine << endl;
16
17     return EXIT_SUCCESS;
18 }
```

Listing 2.5.1 – Convertir une chaine en majuscules

Si une chaîne possède 1000 caractères, elle occupera donc en mémoire 1001 caractères, c'est à dire les 1000 caractères de la chaîne plus le marqueur de fin de chaîne. Si l'on désire changer les caractères minuscules en majuscules il ne faut surtout pas écrire le code du Listing 2.5.1 car cela implique de parcourir deux fois la chaîne : une première fois lors du calcul de sa longueur (ligne 10) et la deuxième fois lors du passage en majuscules (lignes 11 à 13). Il vaut mieux passer par des pointeurs :

```
1  char *s = chaine;
2  while (*s != '\0') {
3      if (isalpha(*s)) *s = toupper(*s);
4      ++s;
5  }
```

2.5.2 l'Unicode

Le problème de l'ASCII est qu'il ne permet de coder que 256 caractères différents ce qui est insuffisant au regard de toutes les langues qui existent ainsi que des symboles (mathématiques, physique, chimie) que l'on peut utiliser dans l'écriture courante.

Le standard Unicode dans sa version 15.0 (Septembre 2022) permet de coder 149_186 caractères ce qui couvre la presque totalité des caractères connus. Le Consortium Unicode a pour but d'identifier de manière précise et distincte l'ensemble des caractères.

Chaque caractère est clairement identifié par son **point de code** qui est en fait un indice entier. Par exemple le symbole € a pour point de code la valeur 8364 soit U+20AC en hexadécimal dans le standard Unicode.

L'UTF (*Universal character set Transformation Format*) permet de transformer le point de code des caractères Unicode en une série d'octets. En fonction des besoins de l'utilisateur on utilisera une représentation 8, 16 ou 32 bits, sachant que l'on peut passer de l'une à l'autre sans perte.

L'encodage par octet, UTF-8, a été conçu pour coder des chaînes à la manière de ce que l'on peut faire avec l'ASCII et est très utilisé par le protocole HTML et les éditeurs de texte :

- les 127 premiers caractères de l'ASCII 7 bits ont les mêmes valeurs en UTF-8 et sont donc codés sur un octet
- pour coder les caractères de valeurs comprises entre 128 et 2047 on utilise deux octets
- puis trois octets pour coder les caractères de valeurs comprises entre 2048 et 65535
- enfin, on utilise quatre octets pour les caractères de valeurs supérieures à 65535

En UTF-32, par contre, chaque caractère est codé par une valeur 32 bits ce qui prend plus de place que l'UTF-8.

Par exemple la chaîne `"abàé€"` sera codée :

```
ASCII
00000000  61 | 62 | e0 | e9 | a4 | 0a
00000006
UTF-8
00000000  61 | 62 | c3 a0 | c3 a9 | e2 82  ac | 0a
0000000a
UTF-16
00000000  ff fe | 61 00 | 62 00 | e0 00 | e9 00 | ac 20 | 0a 00
0000000e
UTF-32
00000000  ff fe 00 00 | 61 00 00 00 | 62 00 00 00 | e0 00 00 00
00000010  e9 00 00 00 | ac 20 00 00 | 0a 00 00 00
0000001c
HTML Entities
00000000 61 62 26 61 67 72 61 76  65 3b 26 65 61 63 75 74 |ab&agrave;&eacut|
00000010  65 3b 26 65 75 72 6f 3b  0a                     |e;&euro;.|
HTML Hexadecimal
00000000 61 62 26 23 78 45 30 3b  26 23 78 45 39 3b 26 23 |ab&#xE0;&#xE9;&#|
00000010  78 32 30 41 43 3b 0a                           |x20AC;.|
```

Pour la transformation en ASCII, j'ai utilisé `konwert` mais comme le symbole de l'Euro n'existe pas en ASCII, il a été traduit par le symbole de code `A4`.

La suite de valeurs `FF FE` en UTF-16 bits et `FF FE 00 00` en UTF-32 indique l'ordre de lecture des caractères. Ici cela signifie qu'il faut commencer par la première valeur trouvée. Dans le cas de l'UTF-32, la séquence d'octets `61 00 00 00` doit donc être considérée comme la valeur hexadécimale `00_00_00_61` $= 97_{10}$ qui correspond au caractère `'a'`.

2.6 Little et big endian

Nous avons vu précédemment que certaines données tels que les nombres entiers ou les nombres flottants peuvent être représentées sur plusieurs octets. L'ordre dans lequel ces octets sont ordonnés en mémoire est appelé endianness [6].

Dans le mode *big endian*, l'octet de poids le plus fort est enregistré à l'adresse mémoire la plus petite alors que dans le mode *little endian* c'est l'inverse.

Pour le monde x86, c'est le mode *little endian* qui est utilisé, ainsi la valeur $\mathrm{FFFE0201}_{16}$ sera stockée en mémoire dans l'ordre croissant des adresses sous la forme : 01_{16} suivi de 02_{16}, FE_{16} et finalement FF_{16}.

2.7 Conclusion

2.7.1 Que retenir ?

- ▷ l'information est codée sur un ordinateur au format binaire et sera modélisée par des types scalaires comme le caractère, l'entier court, l'entier, le flottant simple ou double précision qui occupent un, deux, quatre ou huit octets
- ▷ la représentation binaire en complément à deux permet de modéliser les entiers relatifs et de pouvoir leur appliquer les opérations arithmétiques de base (addition, multiplication, soustraction, division)
- ▷ la norme IEEE 754 définit le format de codage des nombres réels qui sont qualifiés de nombres à virgule flottante, ou encore de manière plus succinte de flottants
- ▷ le codage des nombres à virgule flottante ne permet pas de représenter tous les nombres et cela peut conduire à des erreurs de précision lors de calculs
- ▷ deux nombres à virgule flottante sont égaux si la différence de leur valeur absolue est inférieure à un epsilon donné, ou en d'autres termes, s'ils sont proches.

6. Terme issu du livre les Voyages de Gulliver, conte satirique de Jonathan Swift et qui se traduit en français par *boutisme* ou par *endianisme*.

2.7.2 Compétences à acquérir

Après lecture et travail sur ce chapitre, on doit être capable de :

- ☐ convertir un nombre décimal dans une autre base
- ☐ convertir un nombre en binaire, en octal ou en hexadécimal en décimal
- ☐ convertir un nombre réel en son équivalent flottant
- ☐ convertir un nombre flottant en son équivalent réel
- ☐ réaliser une addition et une multiplication en binaire
- ☐ déterminer si un nombre entier est dans l'intervalle de représentation du codage binaire naturel ou binaire en complément à deux en fonction du nombre d'octets utilisé pour sa représentation

2.8 Exercices

Exercice 3 - Trouvez l'équivalent décimal des nombres suivants :

- 101010_b, 10011_b
- 201_3, 1111_3
- 421_o, 732_o
- A0$_h$, FF$_h$

Exercice 4 - Convertir les nombres décimaux suivants :

- 11 et 10 en base 2
- 26 et 210 en base 8
- 250 et 49 en base 16

Exercice 5 - Utilisez la méthode par complémentation afin de coder en notation binaire naturelle non signée, les nombres suivants :

- 249
- 1011
- 16373
- 131069

Exercice 6 - Réaliser la somme des nombres naturels suivants en base 2. Que remarquez-vous ?

- $0000\_0010\_b + 0000\_0011\_b$
- $0000\_1010\_b + 0000\_1111\_b$

Exercice 7 - Quels sont les plus grands entiers naturels que l'on peut représenter avec 8, 16 ou 32 bits ?

Exercice 8 - Donner la représentation en complément à deux des nombres décimaux suivants : $-1, -2, -127, -128, -129$. Combien de nombres peut-on représenter avec 8 bits en notation en complément à deux ?

Exercice 9 - Calculer la somme des nombres **en complément à deux** suivants. Que remarquez vous ?

- $0000\_0111\_\bar{2}$ + $0000\_0101_{\bar{2}}$
- $0000\_0111\_\bar{2}$ + $1000\_0101_{\bar{2}}$
- $0000\_0011\_\bar{2}$ + $1111\_1011_{\bar{2}}$
- $0100\_0000\_\bar{2}$ + $0100\_0001_{\bar{2}}$

Exercice 10 - Calculer le produit des nombres en complément à deux suivants. Que remarquez vous ?

- 7×5
- 7×-5
- 48×-2
- 48×-3

Exercice 11 -

- comment *multiplier* simplement un nombre binaire par 2, 4, 8 ou 2^n ?
- comment *diviser* simplement un nombre binaire par 2, 4, 8 ou 2^n ?

Exercice 12 - Représentez en norme IEEE 754, les nombres suivants :

- $133,875_{10}$
- $-14,6875_{10}$
- $5,59375_{10}$
- $0,66_{10}$

Exercice 13 - Trouvez à quels nombres réels correspondent les représentations IEEE 754 :

- $42\_C8\_40\_00_{16}$
- $48\_92\_F5\_40_{16}$
- $C2\_92\_F0\_00_{16}$
- $C3\_B0\_30\_00_{16}$

Exercice 14 - Codez la chaîne de caractères *Hola mundo !* en ASCII.

Exercice 15 - Ouvrir un éditeur de texte comme `gedit`, saisir la chaîne "éAèBàCçD€" sans les guillemets et sauvegarder le fichier en le nommant **`a.txt`**. Utilisez ensuite la commande Unix **`hexdump -C a.txt`** afin d'obtenir le contenu du fichier sous forme d'octets. Regardez comment sont codés les caractères accentués et le symbole de l'euro.

Exercice 16 - Programmer la soustraction binaire en C ou tout autre langage que vous maîtrisez. On considère que les nombres sont codés soit sous forme de listes d'entiers ou de booléens. On peut également utiliser des chaînes de caractères ou la classe `bitset` du C++. On réalise l'opération $x - y$ en supposant que $x \geq y$.

Exercice 17 - Programmer la division binaire en C ou tout autre langage que vous maîtrisez. On considère que les nombres sont codés soit sous forme de listes d'entiers ou de booléens. On peut également utiliser des chaînes de caractères. Initialement le quotient est à 0 et la dernière position à laquelle on a réalisé une soustraction (k_1) est égale à la taille du dividende.

Tant que le dividende est supérieur ou égal au diviseur, on effectue les opérations suivantes :

1. on recherche la position k_2 à laquelle on peut soustraire le diviseur dans le dividende
2. on décale le quotient de $(k_1 - k_2 - 1)$ rangs vers la gauche si cette quantité est supérieure à 0
3. on soustrait le diviseur ou dividende à la position k_2 et on place un 1 à droite du quotient
4. $k_1 = k_2$

Enfin, lorsque le dividende est inférieur au diviseur, si k_1 n'est pas égal à 0, on décale le quotient de k_1 rangs vers la gauche

Chapitre 3

Le Fonctionnement du Microprocesseur

Why are Assembly programmers always soaking wet ?
Because they work below C level !

3.1 Introduction

Ce chapitre introduit les bases de l'organisation interne d'un microprocesseur et du sous-système mémoire associé. Il ne s'agit pas d'un cours d'architecture à proprement parler qui nécessiterait à lui seul un ouvrage entier mais d'une revue des notions et principes qui nous serviront plus tard pour coder efficacement. Le lecteur averti voudra bien nous pardonner de prendre parfois certains raccourcis afin de simplifier la machinerie complexe du microprocesseur, notre but étant de focaliser l'attention du novice sur les points cruciaux qui seront exploités par la suite dans la traduction en assembleur des traitements de haut niveau (voir notamment le Chapitre 5)

L'invention du microprocesseur, également qualifié de CPU en anglais pour *Central Processing Unit* où Unité Centrale de Traitement en français, remonte à 1971 avec la commercialisation de l'Intel 4004 en novembre de la même année. Le microprocesseur représente à l'époque une invention majeure car il réunit les fonctions d'un processeur sur un seul circuit intégré. L'utilisation de transistors pour son implantation, la miniaturisation de ces derniers ainsi que le développement de nombreuses techniques issues de la gestion des chaînes de production [1] ont conduit à nos processeurs actuels.

Le microprocesseur est défini comme la partie d'un ordinateur qui exécute les instructions et traite les données des programmes. On le qualifie parfois de cerveau de l'ordinateur mais ce terme est galvaudé car il laisse à penser que l'ordinateur est

1. Que l'on qualifie de Productique.

intelligent. Or, un ordinateur n'est qu'un simple automate, il reproduit une série d'actions prédéterminées et n'a d'intelligence que celle des personnes qui ont mis au point les programmes qu'il exécute.

Les différents travaux qui ont mené à son élaboration datent des inventions de Charles Babbage (1791 - 1871) avec sa machine différentielle dont le but était de calculer des polynômes en utilisant une méthode de calcul dite méthode des différences, puis sa machine analytique qu'il n'achèvera jamais. La machine analytique comprenait beaucoup de concepts repris par la suite durant la seconde guerre mondiale pour la conception des ordinateurs que l'on peut qualifier de préhistoriques. Mais ce furent principalement les travaux et échanges d'idées entre les américains John Von Neumann, John Eckert, John William Mauchly et le britannique Alan Turing dans les années 40 et 50 qui permirent d'aboutir à nos ordinateurs modernes.

Intrinsèquement, le microprocesseur ne sait faire que trois choses : lire des données en provenance de la mémoire, les combiner au travers d'opérations (arithmétiques, logiques) et stocker le résultat de ces opérations en mémoire. Afin de réaliser ces trois opérations de manière efficace il est nécessaire de développer des techniques élaborées, souvent difficiles à concevoir, puis à mettre en oeuvre. C'est ce que nous allons voir au travers de ce chapitre. Nous avons fait le choix de rester synthétique afin de permettre au lecteur de comprendre les principes sous-jacents au traitement des instructions par le microprocesseur. Le lecteur intéressé par plus de détails pourra consulter les ouvrages suivants [5, 27, 26, 2].

3.2 La mémoire centrale

Avant de parler du microprocesseur, il est nécessaire d'évoquer le sous-système mémoire puisque c'est la mémoire qui fournit au processeur sa matière première : les données et les instructions. Il faut savoir qu'il existe deux approches différentes dans la gestion des flux d'instructions et de données : celle de *Von Neumann* que nous utilisons ici, pour laquelle données et instructions sont contenues dans une seule mémoire, et celle dite de *Harvard* où données et instructions sont stockées dans des mémoires séparées.

Du point de vue de l'utilisateur la mémoire centrale apparaît comme un long tableau unidimensionnel d'octets qui permet de stocker les programmes à exécuter. De nos jours il n'est pas rare de trouver sur nos ordinateurs personnels de l'ordre de 8 à 16 Go de mémoire ce qui représente une quantité énorme de stockage généralement sous utilisée pour la plupart des tâches courantes.

La mémoire centrale est de type **DDRx-SDRAM** où x peut prendre des valeurs entre 1 et 4 à l'heure où nous écrivons cet ouvrage. Le sigle RAM (*Random Access Memory*) signifie que l'on peut accéder à n'importe quel octet de la mémoire en lecture ou en écriture.

- Le terme DRAM (*Dynamic RAM*) tient à la composition de la mémoire centrale pour représenter un bit d'information. Celle-ci est formée à l'aide de condensateurs. Si le condensateur est chargé c'est qu'il représente un bit à 1, sinon il représente un 0. Le problème d'un condensateur est qu'il a tendance à se décharger. Pour maintenir l'information valide, il est nécessaire de lire et réécrire les données en mémoire afin de recharger les condensateurs. On appelle cette étape un *rafraîchissement mémoire*. Durant cette période de temps très courte la mémoire est indisponible et il est nécessaire de rafraîchir la mémoire plusieurs fois par seconde.
- Le terme SDRAM (*Synchronous DRAM)* implique que les lectures et écritures se font à intervalles réguliers de manière synchrone.
- Le terme DDR (*Dual Data Rate*) implique que nous doublons le taux de transfert des données en envoyant celles-ci sur le front montant et sur le front descendant du signal d'horloge.
- Enfin le nombre x situé après DDR est un facteur qui définit le nombre d'octets que l'on peut lire ou écrire lors du transfert des données.

Débit mémoire

Le débit où bande passante (*bandwidth* en anglais) de la mémoire est donné par la formule suivante en Mo/s :

$$\text{bandwidth} = [\text{frequency} \times 2 \times 2^{(x-1)}] \times 8 \tag{3.1}$$

- frequency est la fréquence de fonctionnement de la mémoire exprimée en MHz
- le facteur 2 correspond à la DDR qui double le taux de transfert
- x correspond au type de DDRx
- 8 est la largeur du bus de données en octets (soit 64 bits)

Par exemple une DDR4-SDRAM fonctionnant à 100 Mhz possède un débit théorique de $100 \times 2 \times 2^{(4-1)} \times 8 = 12800$ Mo/s. On désigne également ce type de mémoire par les sigles :

- PC4-12800 qui indique le débit en Mo/s
- ou DDR4-1600 qui indique le débit des données en MT/s (Méga Transferts par secondes) et correspond au produit des trois premiers termes de la formule 3.1

Plus le débit est important, plus la mémoire pourra alimenter le processeur en données à traiter. Mais il faut également prendre en compte d'autres facteurs appelés *timings* associés à la fréquence mémoire et liés intrinséquement à la structure matricielle des mémoires vives. Ces timings sont indiqués par les constructeurs ou lisibles par le BIOS de la carte mère grâce à un circuit électronique situé sur les barrettes mémoires appelé SPD pour *Serial Presence Detect*.

La mémoire centrale n'est pas organisée sous forme d'un long tableau unidimensionnel d'octets mais sous forme d'une matrice carrée, c'est à dire qui possède le même nombre de lignes et de colonnes, ceci afin de diminuer l'espace occupé par les circuits d'accès. Comme on a le même nombre de lignes et de colonnes on utilisera un seul canal pour transmettre le numéro de la ligne puis de la colonne qui nous intéresse.

Pour simplifier considérons que nous disposons d'une mémoire de 4 Go organisée sous forme d'une matrice de 65_536 lignes et 65_536 colonnes car $65\_536 = 2^{16}$ et 4 Go $= 2^{32}$. Pour accéder à l'adresse $197\_632 = 3 \times 65\_536 + 1024$, il nous faut dans un premier temps envoyer un signal RAS (*Raw Access Strobe*) au circuit mémoire afin de lui indiquer que nous allons lui envoyer le numéro de ligne. On envoie ensuite le numéro de ligne, suivi du signal CAS (*Column Access Strobe*) pour indiquer qu'on va à présent envoyer le numéro de colonne. Puis on envoie le numéro de colonne. Après quelques cycles d'attente (parfois plusieurs dizaines), on obtient les données sur le bus de données.

Les timings ont pour objectif de définir les délais entre l'envoi d'un signal et le début du signal suivant ou entre l'envoi d'un signal et la réception d'un autre. Le réglage des timings peut donc influer sur l'efficacité de la mémoire mais pour un utilisateur standard il est préférable d'utiliser les valeurs préconisées par le constructeur. Pour plus d'information concernant les timings mémoire je recommande la lecture de l'article du site *hardwaresecrets* [2] qui traite du sujet.

Les technologies liées à la conception des mémoires centrales sont en constante évolution (cf. Table 3.1) et sont régies par un organisme de normalisation des semi-conducteurs appelé JEDEC (*Joint Electron Device Engineering Council*).

Type de mémoire	Année	Transferts (MT/s)	Débit (Go/s)
DDR SDRAM	2000	266-400	2,1-3,2
DDR2 SDRAM	2003	533-800	4,2-6,4
DDR3 SDRAM	2007	1066-3100	8,5-24,8
DDR4 SDRAM	2014	1600-4800	12,8-38,4
DDR5 SDRAM	2020	3200-8400	25,6-67,2

TABLE 3.1 – Types de mémoire et caractéristiques

3.2.1 Alignement des données en mémoire

D'un point de vue conceptuel la mémoire SDRAM est organisée sous forme de bancs mémoire indépendants qui correspondent au nombre d'octets que le circuit mémoire est capable de délivrer en une lecture.

2. https://www.hardwaresecrets.com/understanding-ram-timings

Cette répartition était un facteur important il y a quelques années car le fait d'accéder à des données non alignées ralentissait l'exécution des programmes. Aujourd'hui avec la DDR4-SDRAM et les contrôleurs mémoire dédiés ce ralentissement n'est plus perceptible et ne constitue plus dans la plupart des cas un facteur de ralentissement.

Alignement mémoire

Aligner les données signifie les placer à une adresse multiple d'une puissance de 2 qui dépend du type de mémoire ou des données accédées. En général on prendra un multiple de 16 ou de 32 pour les tableaux. Si on manipule des données qui seront placées dans des registres vectoriels on utilisera un multiple de 16 pour le SSE, de 32 pour l'AVX et de 64 pour l'AVX-512 (cf. Chapitre 8).

Prenons l'exemple de la Figure 3.1 pour laquelle on dispose de 4 bancs mémoire et où l'on a fait figurer les adresses. Lorsque l'on requête un entier de type `int` qui occupe 4 octets à l'adresse `04` la lecture des données peut se faire en une seule fois car les données sont sur la même ligne. Par contre si les données sont situées sur des lignes différentes (adresses `10` et `19`) cela implique l'envoi de deux requêtes au contrôleur mémoire, une pour chaque ligne.

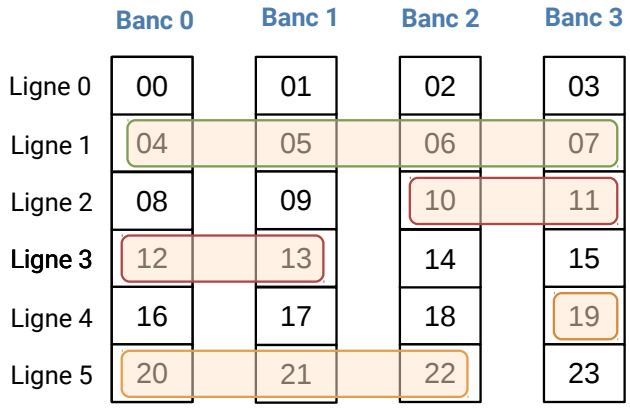


FIGURE 3.1 – Bancs mémoire

L'alignement concerne généralement les variables globales mais peut également être appliqué aux variables locales dans la pile. Le code peut également être aligné de manière à faire commencer le début d'une boucle à une adresse mémoire multiple de 4, 8 ou 16. Par exemple, le compilateur `gcc` comporte des options en ligne de commande comme :

- `-falign-functions` : aligne le début d'une fonction
- `-falign-jumps` et `-falign-labels` : aligne le code des branches du code
- `-falign-loops` : aligne le début des boucles

3.2.2 Double canal

La technologie de **double canal** (cf. Figure 3.2) ou *dual channel*[3] en anglais permet en théorie de doubler le taux de transfert des données en proposant d'utiliser deux canaux au travers desquels on peut échanger des données avec deux barrettes mémoires qui doivent être identiques (même fréquence, même capacité, mêmes caractéristiques). Initialement les deux canaux étaient dépendants c'est à dire que le premier canal lisait 8 octets et le second lisait les 8 suivants, on avait finalement un bus de 128 bits au lieu d'un bus 64 bits d'une mémoire DDR-SDRAM. Ces deux canaux sont ensuite devenus indépendants. Les gains obtenus par le dual channel sont généralement faibles, de l'ordre de quelques pourcents car pour avoir un impact significatif et pouvoir en tirer parti il est nécessaire de travailler sur des tableaux de grande taille ce qui est rarement le cas pour la plupart des applications.

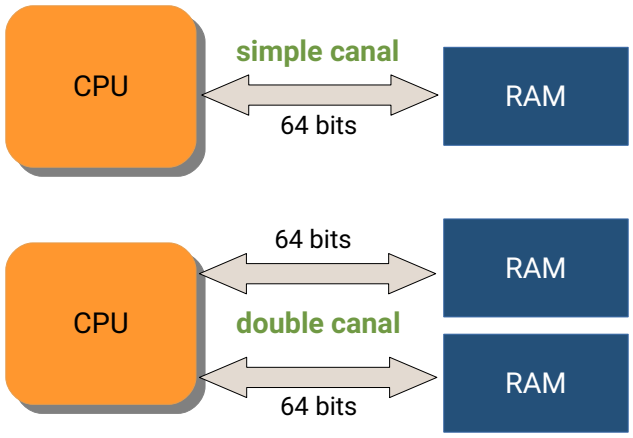


FIGURE 3.2 – Double ou simple canal

Lors de tests que j'ai pu effectuer il y a quelques années avec un Intel Core i5-4570 et de la mémoire de type DDR3 sur le problème de Maximum de Parcimonie en Bioinformatique, je me suis aperçu que l'on pouvait atteindre une diminution du temps d'exécution de l'ordre de 15 à 40% à partir du moment où les séquences ont une longueur de plus de 32_768 octets.

Sur les processeurs haut de gamme on utilise aujourd'hui du *quad channel*, c'est à dire quatre canaux d'accès à la mémoire voire du *hexa channel* sur les derniers Intel Xeon W-3275M.

3.2.3 Mémoire cache

La mémoire centrale est relativement lente par rapport au microprocesseur. Il en a toujours été ainsi et cet écart n'a fait que s'accroître au cours du temps. Le

3. A ne pas confondre avec le Dual Data Rate (DDR) vu précédemment.

microprocesseur est donc pénalisé car il est en attente de données en provenance de la mémoire. Pour palier ce problème différents mécanismes ont été mis en place comme par exemple l'élargissement du bus de données qui consiste à récupérer plusieurs octets consécutifs. Mais le mécanisme le plus intéressant est celui qualifié de mémoire cache. Il consiste à utiliser une mémoire de petite taille mais très rapide d'accès qui contient les données les plus souvent ou les plus récemment utilisées [4].

On dit généralement que la mémoire cache repose sur les principes de *localité* et de *temporalité*. Si on accède une donnée à l'instant t à l'adresse a, il y a une forte probabilité d'accéder à l'instant $t+1$, une donnée à l'adresse $a+\epsilon$. C'est le cas lorsque l'on écrit `x = x + y;` ou que l'on parcourt les éléments d'un tableau.

A tout moment un programme ne travaille que sur une partie de la mémoire , il semble donc intéressant de copier la partie de la mémoire sur laquelle on travaille dans une mémoire locale rapide.

Pour faire une analogie, on pourrait prendre l'exemple du réfrigérateur et du supermarché. Lorsque l'on a besoin de s'approvisionner, on fait des courses au supermarché ce qui prend beaucoup de temps. On stocke les denrées achetées dans un réfrigérateur et elles sont dès lors très rapidement accessibles. Le réfrigérateur représente donc la mémoire cache et le supermarché représente la mémoire centrale.

La rapidité des mémoires cache tient à deux facteurs. Premièrement, un bit de mémoire cache est de type SRAM (*Static RAM*) et est représenté par une bascule (*latch* en anglais) composée de plusieurs transistors et non pas d'un condensateur comme pour les *DRAM*. Le rafraîchissement qui a tendance à ralentir l'accés à la mémoire DRAM n'existe plus dans le cas des SRAM. Deuxièmement, les mémoires caches stockent les données mais également les adresses où se situent ces données. Il est donc très facile de savoir si une adresse est dans le cache ou non.

Au fil des années les mémoires caches se sont développées et sont devenues de plus en plus volumineuses en raison notamment de l'apparition des microprocesseurs multi-coeurs. Initialement absentes, elles ont commencé à être disponibles sur la carte mère, puis ont été progressivement intégrées au microprocesseur.

3.2.4 Niveaux de cache

On distingue aujourd'hui au moins trois niveaux de cache sur les processeurs multi-coeurs. Sur la Figure 3.3, on a fait figurer une architecture de cache pour un microprocesseur quad core, c'est à dire possédant quatre coeurs de calcul.

- le cache le plus proche de la mémoire centrale est le cache de niveau 3, noté L3 pour *Level 3*. Il contient données et instructions et permet de garder la

4. En informatique les algorithmes associés à ces techniques s'appellent LRU pour *Least Recently Used* et LFU pour *Least Frequently Used* et sont utilisés pour remplacer les données les plus anciennes ou les moins souvent utilisées.

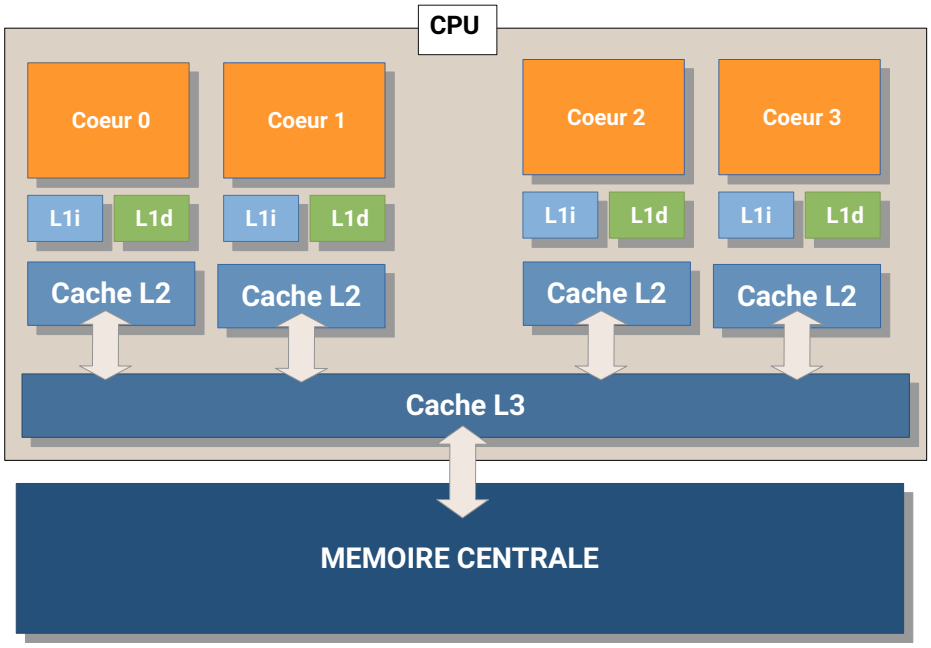


FIGURE 3.3 – Niveaux de mémoire cache

cohérence des données entre les quatre coeurs.

- le cache de niveau 2 (L2) contient également données et instructions mais est plus petit que le cache L3 et est associé à un seul coeur de calcul
- enfin au premier niveau de cache, on scinde le cache en un cache d'instructions L1i et un cache de données L1d car instructions et données ne suivent pas le même cheminement dans le traitement des instructions (cf. Section 3.5)

Sur la Figure 3.3, nous avons fait le choix de montrer une organisation de quatre coeurs disposant chacun de leur propre cache L2. Il est possible, dans des architectures plus anciennes (Intel Core) que deux coeurs partagent (*share*) le même cache L2.

On remarquera que plus on s'éloigne du coeur de calcul plus le cache est de grande taille et plus il sera lent. Pour donner un ordre d'idée on peut consulter la Table 3.2 qui indique la latence des caches pour trois microprocesseurs.

Par exemple, pour le AMD Ryzen 7 1700X, on dispose de 32 ou 64 ko pour les caches L1, 512 ko pour le cache L2 et 16 Mo pour le cache L3. Nous verrons, Section 3.9.1.2, qu'en fait il s'agit de deux fois 8 Mo.

3.2.5 Organisation des mémoires caches entre elles

Il existe deux organisations principales des mémoires cache lorsqu'elles doivent coopérer. On distingue :

Processeur / Cache	L1i (ko)	L1d (ko)	L2 (ko)	L3 (Mo)
AMD Ryzen 7 1700X	64	32	512	16
associativité	4-way	8-way	8-way	16-way
latence (cycles)	4	4 à 5	17	37 à 43
Intel i7-7820X	32	32	1024	11
associativité	8-way	8-way	16-way	11-way
latence (cycles)	4	4 à 5	14	68
Intel i7-1065	32	48	512	2
associativité	8-way	12-way	8-way	16-way
latence (cycles)	5	5	13	30 à 36

TABLE 3.2 – Caractéristiques des caches (taille, latence, associativité) - http ://www.7-cpu.com

- les caches **inclusifs** qui sont conçus de manière à ce que toute donnée présente dans le cache L1 soit aussi présente dans le cache L2. La taille totale du cache L1+L2 est donc celle du cache L2 puisque les données de L1 sont incluses dans L2.
- les caches **exclusifs**, pour lesquels une donnée est soit dans le cache L1, soit dans le cache L2. Lorsqu'une donnée sort du cache L1 pour être placée dans le cache L2 on parle d'éviction. La taille totale du cache L1+L2 est donc la somme des tailles des caches L1 et L2.

Intel a fait le choix des caches inclusifs alors qu'AMD utilise des caches de type exclusifs.

Un autre problème concerne le remplacement des lignes de cache. Lorsque l'on remplace (voir ci-après) une ligne de cache qui est soit la plus ancienne (LRU) ou la moins utilisée (LFU) se pose alors le problème du traitement de la ligne de cache évincée afin de garder la cohérence des données.

Deux stratégies peuvent être envisagées :

- l'écriture immédiate (*Write Through*) consiste à mettre immédiatemment à jour la donnée en mémoire centrale dès lors que sa valeur dans le cache est modifiée
- l'écriture différée (*Write Back*) consiste à mettre à jour la mémoire centrale lors de la modification de l'entrée du cache correspondante

Un compromis entre coût de production, complexité de réalisation et performances doit être trouvé lors de la conception d'une mémoire cache, puis un équilibre doit être trouvé entre les différents niveaux de cache.

Par exemple la stratégie d'écriture immédiate augmente le trafic vers la mémoire centrale. Si la stratégie d'écriture différée pallie ce problème elle en crée un

nouveau : en effet, certains circuits d'entrées sorties de type DMA (*Direct Memory Access*) sont capables de lire ou écrire des données en mémoire sans passer par le processeur et risquent par exemple de lire des données qui ne sont pas à jour. On peut contourner ce problème mais cela augmente la complexité du système.

3.2.6 Cache associatifs par groupe

Les mémoires caches sont organisées sous forme **associative par groupe**, on dit en anglais n-*way set associative*. On peut considérer la mémoire cache comme une table composée de blocs de k lignes et m colonnes qui stockent des données ainsi qu'une partie de l'adresse où se trouvent les données en mémoire.

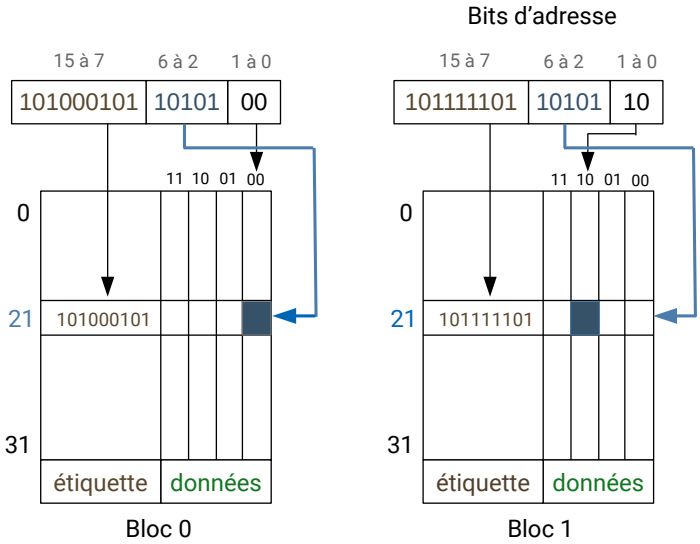


FIGURE 3.4 – Cache associatif à 2 groupes

L'associativité permet d'organiser les adresses sous formes de classes d'équivalence comme le montre la Figure 3.4. Imaginons que le cache contienne deux blocs qui permettent de stocker chacun 32×4 octets. On dit dans ce cas que la taille de la ligne de cache est de 4 octets et que le cache a une taille de $2 \times 32 \times 4 = 256$ octets, soit deux blocs de 32 lignes de 4 octets. On ne compte pas la partie stockant l'adresse.

3.2.6.1 Ajouter une adresse dans le cache

Considérons pour simplifier les choses que notre microprocesseur possède un bus d'adresses de 16 bits c'est à dire que les adresses ont une taille de 16 bits. Pour stocker dans le cache l'octet situé à l'adresse 48_854, on écrit l'adresse en binaire et on la décompose ainsi :

- $48\_854 = 1011\_1110\_1101\_0110_2$

- les deux premiers bits sont mis à 0 pour obtenir l'adresse 48_852 car on charge les données par groupe de 4 octets, c'est à dire la taille d'une ligne de cache, on chargera donc les octets situés aux adresses multiples de 4

- la ligne de cache où on devra placer les quatre octets est obtenue par les 5 bits suivants, car il y a 32 lignes de cache et $32 = 2^5$, soit dans notre cas $10101_2 = 21$

- les 9 bits restants représentent ce que l'on appelle l'étiquette et seront stockés dans le cache car ils permettent de reconstruire l'adresse exacte

On remarque donc pour cet exemple que toute adresse dont les bits d'indices 2 à 6 [5] ont la même valeur sera stockée dans la même ligne du cache associatif. Afin d'améliorer l'efficacité du cache on crée donc n blocs de ce type et on essaye de répartir au mieux les adresses entre les blocs en utilisant un algorithme de type LRU ou LFU comme évoqué précédemment.

3.2.6.2 Vérifier si une adresse est dans le cache

Pour déterminer si l'octet à l'adresse $41687 = 1010\_0010\_1101\_0100_2$ est dans le cache, on calcule la ligne du cache où devrait se trouver l'adresse. Dans le cas présent il s'agit de $1\_0101_2 = 21$ (qui correspond au bits 2 à 6), puis on compare en parallèle dans chaque bloc si l'étiquette 1_0100_0101 est présente. Si c'est le cas, l'adresse et la donnée qui lui correspond sont présentes dans le cache.

Lorsque la donnée recherchée se trouve présente dans le cache, on parle de *cache hit*. Dans le cas où elle est absente il s'agit d'un *cache miss* ou *défaut de cache* en français. Pour donner un ordre de grandeur, déterminer si une donnée est dans le cache L1 prend de l'ordre de 4 à 5 ns, dans le cache L2 environ une dizaine voire une vingtaine de ns, puis dans le cache L3 entre 30 et 70 ns et finalement l'obtenir depuis la mémoire une centaine de ns.

5. Le premier bit à pour indice 0.

Facteur d'amélioration

Un programme s'exécutera donc plus vite si les données sont déjà présentes dans le cache et d'autant plus vite que les données sont présentes dans un cache de niveau faible (L1), c'est à dire, le plus proche du coeur de calcul. C'est le premier facteur d'amélioration de la vitesse d'exécution des programmes : faire en sorte, quand cela est possible, que les données soient présentent dans le cache au moment où on les traite car elles seront accessibles rapidement.
La mémoire cache possède une influence très importante pour certains traitements liés à la lecture et l'écriture en mémoire. Un exemple typique de cette influence est celui du produit de matrices où des techniques d'inversion de boucles, ou de blocage de boucles (*loop blocking*) associée au tuilage (*tiling*) permettent de diminuer le temps de calcul de manière drastique par rapport à une implantation directe de la formule de calcul.
Le produit de matrices est la première étude de cas de ce livre (cf. Chapitre 11).

3.3 Le microprocesseur

Nous nous intéressons dans ce chapitre aux microprocesseurs de la famille x86 c'est à dire compatibles avec le jeu d'instructions de l'Intel 8086. Intel a fait le choix de garder une compatibilité dite descendante (ou ascendante suivant la vision des choses) de son jeu d'instructions assembleur. Ainsi, un microprocesseur en 2018 est capable d'exécuter un programme compilé pour l'Intel 8086 conçu en 1978, c'est à dire il y a 40 ans. Sachez que lorsque vous allumez votre ordinateur il se place en mode 8086, puis ensuite en mode 32 ou 64 bits.

On utilise la dénomination IA-32 (pour *Intel Architecture 32 bits*) pour les processeurs, à partir du Pentium chez Intel, qui utilisent des registres 32 bits.

Les deux grands constructeurs de processeurs x86 sont Intel et AMD, deux sociétés américaines [6] .

Intel (*Integrated Electronics*) fut fondée en 1968 par Gordon Moore, Robert Noyce et Andrew Grove, trois docteurs en chimie et physique issus du monde de l'électronique numérique qui décidèrent de quitter l'entreprise *Fairchild Semiconductor*. Intel est devenu le leader mondial de la famille x86 et ses nombreux produits sont connus du grand public dont notamment le Pentium, Pentium Pro, les processeurs Core, Core 2 et plus récemment les microprocesseurs estampillés i3, i5, i7 et les derniers i9.

AMD (*Advanced Micro Devices*) fut fondée en 1969, soit un an après Intel, par un

6. On pensera également à Cyrix (1988-1997) qui a créé des clones du 80486 et du Pentium d'Intel.

groupe d'ingénieurs et de dirigeants de *Fairchild Semiconductor* dont notamment Jerry Sanders. AMD est entré sur le marché des microprocesseurs x86 en 1975 en produisant par rétro-ingénierie une version de l'Intel 8080. C'est au début des années 80 qu'AMD devint un producteur de microprocesseurs sous licence Intel car la société IBM désirait ne pas dépendre du seul fournisseur Intel pour la production de ses PCs. AMD a également conçu ses propres microprocesseurs faits maison comme les fameux K5, K6, Athlon et dernièrement les microprocesseurs Ryzen et Threadripper.

3.3.1 Fréquence de fonctionnement

Tout comme la mémoire le microprocesseur fonctionne à une fréquence donnée qui donne la cadence d'exécution des différentes unités de traitement. La fréquence de fonctionnement fut durant de nombreuses années le nerf de la guerre. Produire un processeur avec une fréquence de fonctionnement supérieure permettait de surpasser son concurrent. Cependant plus la fréquence est élevée, plus le processeur consomme de l'énergie et dégage de la chaleur. Ces dernières années la fréquence a été régulée afin que l'on puisse gérer de manière plus fine l'énergie. En effet un microprocesseur n'a pas besoin de tourner en permanence à une fréquence élevée, uniquement lorsqu'il est sollicité par un ou plusieurs programmes. Le choix a donc été fait d'utiliser, dans un premier temps, trois fréquences de fonctionnement :

- une fréquence au repos (*idle*), par exemple 1,0 GHz
- une fréquence de fonctionnement rapide (*boost*), lorsqu'un seul coeur est actif (3,5 GHz)
- une fréquence de fonctionnement moyenne lorsque plusieurs coeurs sont actifs (3,0 GHz)

A partir de 2018, Intel et AMD ont introduit une gestion encore plus fine de l'énergie avec une diminution progressive de la fréquence en fonction du nombre de coeurs [7] qui travaillent, comme indiqué Table 3.3.

Nombre de coeurs actifs	1-2	3-4	5-12	13-16	17-18
Intel Core i9-7980XE	4,4	4,0	3,9	3,5	3,4
Intel Core i9-9980XE	4,5	4,2	4,1	3,9	3,8

TABLE 3.3 – Modification de la fréquence en GHz en fonction du nombre de coeurs actifs

L'objectif est de repousser les limites de la fréquence de fonctionnement tout en restant dans l'enveloppe thermique du microprocesseur appelée TDP pour *Thermal Design Power* [8]. Le TDP caractérise l'énergie (chaleur) dégagée par un circuit

7. Cette technologie est appelée *Precision Boost* 2 chez AMD.
8. On parle également parfois de *Thermal Dissipation Power*.

intégré. Si la chaleur est trop forte le circuit risque d'être endommagé. Il risque de fondre ou d'être soumis à des phénomènes d'électro-migration qui consistent en un déplacement d'atomes des parties conductrices du courant électrique vers les parties isolantes.

Il semble que la valeur du TDP soit calculée différemment suivant les fabricants et les gammes de processeurs. Elle sert d'indicateur afin de prévoir un circuit de refroidissement proportionné à la chaleur dégagée[9].

3.3.2 Architectures RISC et CISC

Le microprocesseur n'est en fait capable de réaliser que 3 types d'opérations :

- `LOAD r,[mem]`, c'est à dire, charger dans un registre `r` une donnée située en mémoire à une adresse fournie en paramètre
- `STORE [mem],r` qui permet de stocker une donnée contenue dans un registre `r` dans la mémoire à une adresse fournie en paramètre
- `OP r3, r2, r1` où `OP` est une opération arithmétique ou logique et qui signifie mettre dans le registre `r3` le résultat de `r1 OP r2`

En prévision de ce que nous verrons dans le prochain chapitre, nous indiquons que pour les microprocesseurs de type x86, on utilise seulement deux opérandes dans la plupart des instructions. On note donc `OP r1,r2` ce qui correspond à `r1 = r1 OP r2`. Dans ce cas l'opérande `r1` est appelée *destination* et l'opérande `r2` est qualifiée de *source*.

Architecture	Signe	Description
RISC	MIPS	Microprocessor without Interlocked Pipeline Stage
RISC	ARM	Acorn Risc Machine (1987) ou Advanced Risc Machine
RISC	POWER	Performance Optimization With Enhanced RISC 1-8
CISC	x86	Intel, AMD
CISC	680x0	Motorola

TABLE 3.4 – Exemples d'architectures CISC er RISC

On distingue historiquement deux classes d'architectures (cf. Table 3.4) pour les microprocesseurs :

RISC (*Reduced*[10] *Instruction Set Computer*) : dans ce type d'architecture, on utilise

9. On pourra consulter le site https://www.anandtech.com et notamment l'article *Intel Core i7 10700 vs Intel Core i7 10700k, is 65 W Comet Lake an option ?* afin d'en savoir plus sur le TDP.

10. Notons que le terme *Reduced* c'est à dire *réduit* en français est mal choisi, on devrait plutôt dire simplifié.

le format d'instruction précédent et l'adressage mémoire reste simple (i.e. il n'existe que peu de manières différentes d'accèder à la mémoire)

CISC (*Complex Instruction Set Computer*) pour ce type d'architecture on a tendance à combiner une instruction de chargement ou de stockage avec un calcul et l'adressage mémoire peut être complexe

Prenons un exemple CISC issu du jeux d'instruction x86 :

```
1    add [ebx + ecx * 4 + 8], eax
```

Cette instruction réalise plusieurs opérations, à savoir :

- le calcul de l'adresse mémoire `A = ebx + ecx * 4 + 8`
- le chargement de la donnée `D` stockée sur 4 octets à l'adresse `Mem[A]` dans un registre temporaire `R`, soit `R = Mem[A]`
- l'addition de la donnée `D` stockée dans `R` avec le registre **`eax`** : `R = R + eax`
- l'écriture du résultat à l'adresse `A`, `Mem[A] = R`

Ce qui rend cette instruction *complexe* est le fait qu'elle combine plusieurs choses à réaliser dont un calcul d'adresse, un chargement de donnée depuis la mémoire, un calcul et un stockage du résultat en mémoire.

Les microprocesseurs RISC vont, quant à eux, éviter ce genre d'instruction complexe en ne permettant par exemple que de lire une donnée depuis la mémoire pour la stocker dans un registre. On n'autorise alors que l'addition entre deux registres. L'instruction CISC précédente sera traduite en RISC par quatre instructions.

Acutellement beaucoup de processeurs RISC sont utilisés dans les systèmes dits embarqués (téléphones, tablettes, robots) en raison de leur efficacité énergétique, les processeurs RISC consomment en général moins d'énergie que les CISC.

La conception d'un microprocesseur pose de nombreux problèmes. Plus sa structure est complexe, plus les procédures de test sont longues et plus il est difficile de déterminer d'éventuels défauts de conception. Un processeur RISC, de structure moins complexe qu'un processeur CISC, est donc plus simple à concevoir et tester.

Plusieurs facteurs ont encouragé par le passé la conception de machines à jeu d'instruction complexe (CISC) :

- premièrement, la lenteur de la mémoire par rapport au processeur laissait à penser qu'il était plus intéressant de soumettre au CPU des instructions complexes. Pour réaliser un traitement donné, il était préférable de définir une instruction complexe plutôt que plusieurs instructions élémentaires. De plus une instruction complexe prend alors moins de temps de chargement depuis la mémoire qu'une série d'instructions simples. Dans les années 70 les ordinateurs utilisaient de la mémoire magnétique (conçue à partir de

tores) pour stocker les programmes. Ce type de mémoire était cher et lent. Un premier changement s'opéra avec l'arrivée des DRAM plus rapides mais restait l'épineux problème du prix des DRAM. Par exemple en 1977, 1 Mo de DRAM coûtait environ $5000 alors qu'il ne valait plus que $6 en 1994

- deuxièmement, le développement des langages de haut niveau (Fortran, Pascal, Ada) a posé quelques difficultés quant à la conception de compilateurs capables de traduire efficacement des programmes d'un langage évolué vers l'assembleur. On a donc eu tendance à incorporer au niveau processeur des instructions plus proches de la structure de ces langages. Le processus de compilation des langages de haut niveau comme Pascal et C était lent et le code assembleur obtenu n'était pas toujours optimisé : mieux valait coder à la main. Certains ont proposé de combler le fossé sémantique entre langage de haut niveau et assembleur afin de faciliter la tâche des programmeurs : en d'autres termes ils proposaient de faire en sorte que les instructions assembleur soient adaptées aux instructions des langages de haut niveau.

A partir de la fin des années 70, deux facteurs sont venus ébranler les idées ancrées dans les esprits par les décennies précédentes et qui tendaient à favoriser l'approche CISC. D'une part, les mémoires sont devenues plus rapides, moins chères et de plus grande capacité qu'elles ne l'étaient auparavant et, d'autre part, des études conduites sur des langages de haut niveau montrèrent ([18, 22]) que les programmes sont constitués à 85% d'affectations, d'instructions *if* et d'appels de procédures et que 80% des affectations sont de la forme `variable = valeur`.

Les résultats précédents ont été résumés par la phrase suivante : 80% des traitements des langages de haut niveau font appel à 20% des instructions du CPU. D'où l'idée d'améliorer la vitesse de traitement des instructions les plus souvent utilisées ce qui a conduit à l'architecture RISC.

Aujourd'hui les processeurs modernes de type CISC (comme les processeurs de la famille x86) possèdent des instructions CISC qui, comme nous le verrons plus tard, sont ensuite traduites en une série d'instructions de type RISC (que l'on qualifie de micro-opérations). On peut donc considérer que le coeur de fonctionnement d'un microprocesseur de type CISC est de type RISC.

3.3.3 Architecture x86

Si la fréquence de fonctionnement est un facteur important qui permet de caractériser la puissance d'un microprocesseur un autre facteur primordial est son architecture. Le terme architecture est à différencier de ce que nous venons de voir pour les architectures CISC et RISC. Quand nous parlerons d'architecture d'un processeur nous évoquerons les caractéristiques et l'organisation des éléments qui le constituent. L'architecture détermine la taille des caches, leur organisation mais également toute la partie liée au traitement des instructions. L'accès à la mémoire et le type de mémoire qui pourra être utilisé est généralement déterminé par le

chipset [11] de la carte mère qui définit entre autres choses comment les données sont échangées entre le microprocesseur, la mémoire et les périphériques. Cependant les deux composants (microprocesseur et chipset) étant liés, on peut se demander lequel influe le plus sur l'autre.

La finesse de gravure détermine grosso-modo la taille des transistors. Plus la finesse de gravure est petite, plus les transistors sont petits. Si un transistor est gravé plus finement il est plus rapide, consomme moins d'énergie et possède une plus grande densité d'intégration, c'est à dire qu'on peut en mettre plus sur la même surface, ce qui économiquement est plus intéressant.

Une architecture est aujourd'hui identifiée par un nom (cf. Table 3.5) qui détermine sa finesse de gravure ainsi que l'étape de production et la génération du processeur.

Année	Etape	Architecture	Génération	Finesse
2008	Tock	Nehalem	1	45 nm
2010	Tick	Westmere	1	32 nm
2011	Tock	Sandy Bridge	2	32 nm
2012	Tick	Ivy Bridge	3	22 nm
2013	Tock	Haswell	4	22 nm
2014	Optimization	Haswell Refresh	4	22 nm
2014	Tick	Broadwell	5	14 nm
2015	Tock	Skylake	6	14 nm
2017	Optimization	Kaby Lake	7	14 nm
2017	Optimization	Kaby Lake Refresh	8	14 nm
2017	Optimization	Coffee Lake	8,9	14 nm
2018	Optimization	Whiskey Lake	8	14 nm
2019 ?	Process	Canon Lake	?	10 nm

TABLE 3.5 – Nom des architectures Intel en fonction des années

La société Intel s'est engagée en 2007 sur la voie d'un mode de production de ses microprocesseurs en deux temps appelés tick-tock [12] :

- un tick correspond à une diminution de la finesse de gravure
- un tock correspond à la création d'une nouvelle architecture

Puis, entre 2014 et 2016, ce modèle a été amendé en raison du retard pris par Intel sur la gravure en 10 nm pour inclure une troisième étape qui consiste à améliorer une architecture existante et le modèle a été rebaptisé PAO pour *Process*

11. Un chipset est un ensemble de composants électroniques qui permet la communication entre microprocesseur, mémoire et périphériques.
12. Ce qui en français correspond au tic-tac d'une montre.

- *Architecture* - *Optimization*. Tout cela prête à confusion car il faut également prendre en compte la génération du microprocesseur (cf. Table 3.5).

Il est préférable de consulter la base de données des microprocesseurs Intel [13] afin d'obtenir des informations adéquates.

3.3.3.1 Les lois de Moore

En 1965, Gordon Moore (l'un des fondateurs d'Intel) alors ingénieur chez Fairchild Semiconductor, postule le doublement de la complexité des semi-conducteurs tous les ans à coût constant en se basant sur des données depuis 1959, date de leur invention.

Une dizaine d'années plus tard, Moore révisera sa prédiction pour un doublement tous les deux ans du nombres de transistors dans un microprocesseur. C'est cette seconde prédiction que l'on qualifie de *loi de Moore* mais qui n'est pas une loi au sens strict du terme, c'est à dire toujours vraie, mais définit une tendance que les fabricants de circuits intégrés et de microprocesseurs en particulier, tendent à suivre, plus ou moins bien.

Enfin une troisième version postule le doublement de quoi que ce soit tous les dix-huit mois. Elle est attribuée à David House, travaillant chez Intel qui aurait combiné la multiplication du nombre de transistors et l'augmentation de leur vitesse ou de la fréquence de fonctionnement des microprocesseurs.

3.3.4 Les Registres

Le microprocesseur possède plusieurs variables, appelées registres, qui permettent de stocker de manière temporaire des valeurs qui serviront pour des calculs ultérieurs. En architecture 32 bits, les registres utilisés pour faire des calculs sont au nombre de 8 et stockent des valeurs entières de 32 bits (entier signé, non signé, adresse mémoire).

Ils sont qualifiés de registres généraux (*General Purpose Registers*) où registres à usage général identifiés par les noms `eax`, `ebx`, `ecx`, `edx` ainsi que les registres d'index `edi` (*Destination Index*) et `esi` (*Source Index*). Il existe également des registres qui servent à gérer la pile et à accéder aux paramètres des fonctions et aux variables locales. Ce sont `esp` (*Stack Pointer*) et `ebp` (*Base Pointer*) [14].

On dénombre également plusieurs autres registres comme les registres de segment (`cs`, `ds`, `ss`, `es`), le registre `eflags` qui stocke les effets des opérations (retenue, résultat à 0, débordement, ...) ainsi que le registre `eip` (*Instruction Pointer*) qui stocke l'adresse mémoire de la prochaine instruction à exécuter. Pour plus d'information sur les registres, on consultera le Chapitre 5.

13. https://ark.intel.com/fr
14. ESP et EBP sont généralement décrits comme faisant partie des registres d'index mais je préfère les distinguer des autres registres car ils ont un usage spécifique lié à la pile.

3.3.5 Adressage mémoire

Une adresse mémoire est définie soit par une valeur constante, soit par la valeur d'un registre, soit par une combinaison de registres qui permet une correspondance avec les expressions liées aux tableaux ou aux structures de données (cf. Figure 3.5). Une adresse a donc la forme suivante :

$$\text{adresse} = [\quad \text{base} + \text{index} \times \text{scale} + \text{offset} \quad]$$

- base et index sont des registres avec une restriction concernant index qui ne peut être le registre `esp`
- scale est un facteur d'échelle et peut prendre les valeurs 1, 2, 4 ou 8 qui vont correspondre à la taille d'un octet, d'un mot, d'un double mot ou d'un quadruple mot
- offset est une constante entière positive, négative ou nulle qualifiée de décalage ou de déplacement

Chacun de ces termes est optionnel. Par exemple si on désire accéder au $i$ème élément d'un tableau d'entiers t on pourra choisir de stocker l'adresse de t dans le registre `ebx`, stocker i dans le registre `ecx`, utiliser `eax` pour lire la valeur `t[i]` et ainsi écrire `mov eax,[ebx + ecx * 4]`. Le facteur d'échelle utilisé ici est 4 car un entier occupe 4 octets en mémoire.

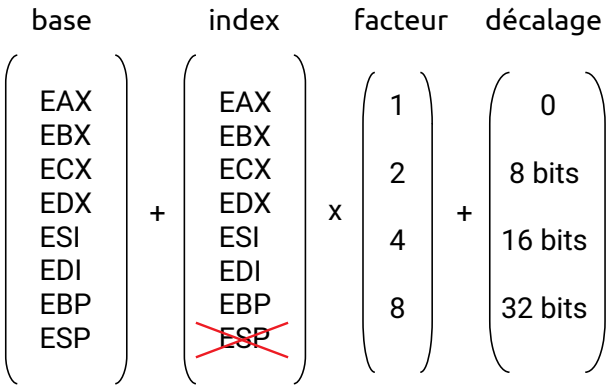


FIGURE 3.5 – Adressage mémoire

Les registres de segment ont été introduits sur l'Intel 8086, un microprocesseur 16 bits, afin de pouvoir gérer 1 Mo de mémoire, alors que 16 bits ne permettent de gérer que 64 ko [15]. Chacun de ces segments est lié à la structure d'un programme : `cs` correspond au segment de code (*Code Segment*), `ds` au segment de données

15. $2^{16} = 65536 = 2^6 \times 2^{10} = 64 \times 1024 = 64$ ko.

(*Data Segment*), `ss` à la pile (*Stack Segment*) et `es` (*Extra Segment*) peut être utilisé pour pointer sur une adresse quelconque de la mémoire.

Grâce à ce mécanisme une adresse mémoire est définie par une combinaison registre de segment et registre général ou registre d'index. Par exemple, `eip` est associé à `cs`. Le registre `edi` est associé à `es` et `esi` est associé à `ds`. Les registres `esp` et `ebp` sont liés au segment de pile `ss`. Lorsque l'on accède à la prochaine instruction au travers de `ip` (ou `eip` en 32 bits), on fait implicitement référence au segment `cs`, l'adresse que l'on accède est donc calculée par :

$$\begin{array}{rcccccc} \text{address} & = & \text{segment} & \times 16 & + & \text{offset} \\ & = & \texttt{cs} & \times 16 & + & \texttt{ip} \end{array}$$

Le fait de multiplier par 16 le registre `cs` consiste à réaliser un décalage à gauche de 4 bits du registre `cs` car $16 = 2^4$, ce qui donne une adresse sur $16 + 4 = 20$ bits.

Notons que les programmes que nous allons écrire par la suite feront référence à ces registres implicitement et nous n'aurons pas à les spécifier car nous fonctionnerons dans une architecture 32 ou 64 bits. Cela permettra de simplifier l'écriture des programmes et les segments seront gérés par l'assembleur `nasm` et le compilateur C ++.

3.4 Amélioration des microprocesseurs

Le temps d'exécution d'un programme est donné par les deux formules suivantes :

$$T_e = \sum_{i=1}^{N} CPI_i \times T_c = N \times CPI_m \times T_c \tag{3.2}$$

La première exprime que le temps d'exécution en secondes (T_e) d'un programme de N instructions est donné par la somme du nombre de cycles que nécessite chaque instruction (CPI_i = *Cycles Per Instruction*) pour s'exécuter multiplié par le temps de cycle (T_c) qui est l'inverse de la fréquence de fonctionnement du microprocesseur.

La seconde est une version synthétique pour laquelle on calcule un nombre moyen de cycles par instruction (CPI_m) pour N instructions :

$$CPI_m = \frac{1}{N} \sum_{i=1}^{n} CPI_i$$

Par exemple, si un sous-programme est composé de :

- 3 instructions qui s'exécutent en 2 cycles
- 1 instruction de 4 cycles

- 1 instruction de 10 cycles

On a alors au total $3 \times 2 + 1 \times 4 + 1 \times 10 = 20$ cycles, soit un CPI moyen de $20/5$ instructions et donc une moyenne de 4 cycles par instruction.

Les différentes évolutions des microprocesseurs ont pour but de diminuer le temps d'exécution des programmes. D'après la formule 3.2, étant donné que le temps d'exécution est un produit de facteurs, il suffit de diminuer l'un des facteurs pour diminuer le temps total d'exécution du programme.

La première amélioration consiste à diminuer le temps de cycle. Pour cela il suffit d'augmenter la fréquence des processeurs. Un processeur doté d'une fréquence de 3 Ghz fonctionne 3 fois plus vite qu'un processeur à 1 Ghz. Cependant augmenter la fréquence pose de nombreux problèmes à résoudre notamment l'élévation de la température dégagée par le circuit électronique.

On peut ensuite diminuer le nombre d'instructions N ou le nombre moyen de cycles par instructions (CPI_m). Or dans ce cas, il semble que le produit $N \times CPI_m$ reste constant, en effet :

- si on diminue le nombre d'instructions on crée des instructions plus complexes (de type CISC) qui nécessitent plus de cycles pour être exécutées, on augmente donc CPI_m
- par contre, si on diminue le nombre moyen de cycles par instructions on crée des instructions simples (de type RISC) et il faut utiliser plusieurs instructions pour réaliser le même traitement qu'une instruction CISC, on augmente donc N

Il a donc fallu élaborer des solutions capables de diminuer le temps nécessaire au traitement des instructions qu'elles soient CISC ou RISC.

Les architectures des processeurs modernes jouent sur plusieurs plans, en tentant de maximiser :

- l'ILP (*Instruction Level Parallelism*) d'un flux d'instructions, c'est à dire, tenter d'exécuter le plus possible d'instructions en parallèle
- le DLP (*Data Level Parallelism*) qui consiste à exécuter la même instructions sur plusieurs données différentes en parallèle
- le TLP (*Thread Level Parallelism*) qui consiste à disposer de plusieurs coeurs d'exécution, c'est ce que l'on appelle les processeurs multi-coeurs

La Table 3.6 indique pour chacun de ces niveaux de parallèlisme quelles techniques peuvent être mises en oeuvre.

Nous allons donc passer en revue ces techniques dans la suite de ce chapitre.

Amélioration	Technique
ILP	pipeline, super-scalaire, exécution dans le désordre, coprocesseur
DLP	registres et calcul vectoriel (unités MMX, SSE, AVX)
TLP	multi-coeurs, SMT, HyperThreading

TABLE 3.6 – Techniques d'amélioration des microprocesseurs

3.5 Traitement des instructions

Afin de simplifier la compréhension du traitement des instructions par le microprocesseur, on peut dire que traiter une instruction consiste à passer par cinq étapes principales (cf. Figure 3.6) :

1. le chargement de l'instruction depuis la mémoire (*Fetch Instruction*)
2. le décodage de l'instruction afin de connaître les opérations à réaliser lors de son exécution (*Decode Instruction*)
3. le chargement des opérandes de l'instruction (*Load Operand*)
4. l'exécution de l'instruction à propement parler (*Execute Instruction*)
5. l'écriture du résultat (*Write Result*)

Le programme à éxécuter réside en mémoire centrale et il se décompose en plusieurs parties :

- le code, c'est à dire les instructions à exécuter
- les données qui peuvent être initialisées, non initialisées ou en lecture seule
- la pile des appels de sous-programmes
- le tas (*heap* en anglais) qui représente le reste de la mémoire et c'est notamment dans cette zone que l'on allouera les données grâce à **malloc** en C ou **new** en C++

L'ensemble des données en mémoire (attention, ici le terme donnée signifie tout octet de la mémoire centrale), s'il est accédé par le processeur au travers d'une adresse, va transiter par les différents niveaux de cache.

Les deux premières étapes de traitement (chargement et décodage) représentent ce que l'on appelle le **frontal** (où *front-end* en anglais), c'est à dire la partie émergée, donc visible de l'iceberg. Les trois dernières sont qualifiées de *back-end* en anglais, que l'on trouve parfois traduit par **dorsal**[16], c'est la partie immergée et la plus complexe.

16. Terme proposé par l'Office québécois de la langue française.

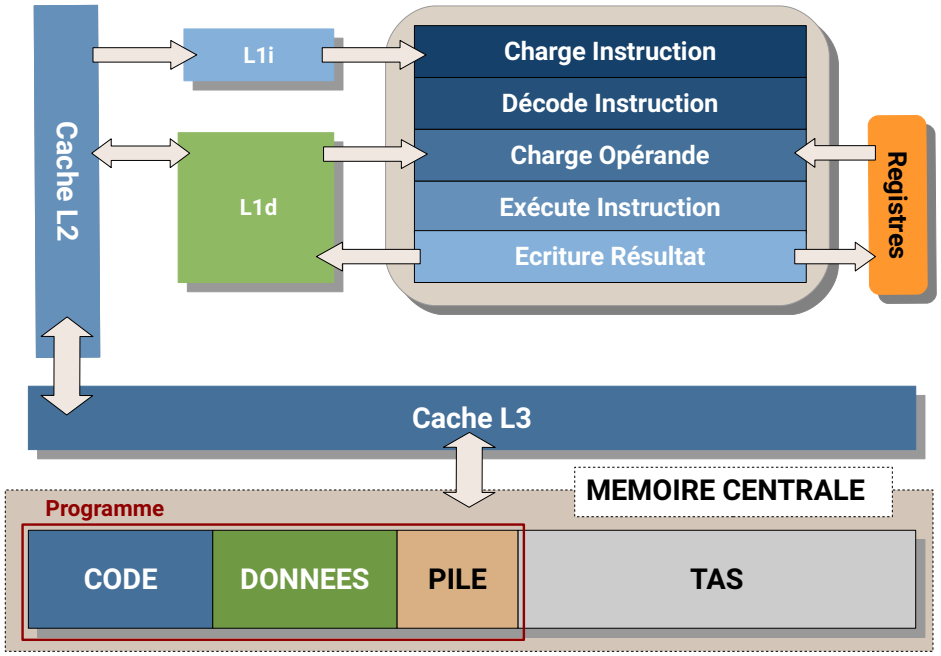


FIGURE 3.6 – Etapes de traitement d'une instruction

Imaginons, de manière grossière que chacune de ces étapes prend une nano seconde (10^{-9} s). Le traitement de chaque instruction demande 5 étapes d'une nano seconde donc 5 ns. En d'autres termes, on traite une instruction toutes les 5 ns.

C'est ce que nous avons représenté sur la partie haute de la Figure 3.7. La première instruction $i1$ passe successivement par les 5 étapes de traitement avant que l'instruction suivante $i2$ puisse être traitée.

3.6 Pipeline d'instructions

Afin d'améliorer la vitesse de traitement des instructions un mécanisme de pipeline a été mis en place. Il consiste à ne pas attendre que l'ensemble des étapes de traitement aient été réalisées avant de passer à l'instruction suivante. Pour cela on rend chaque étape de traitement indépendante. Une première instruction passe dans l'étape de chargement au temps $t = 0$, puis au temps $t + 1$, elle passe dans l'étape de décodage, pendant que l'instruction suivante passe dans l'étape de chargement et ainsi de suite. C'est le même principe qui est utilisé sur les chaînes de montage dans les usines. On qualifie généralement le pipeline de mécanisme d'**amélioration en longueur**.

Ce mécanisme général est utilisé à plusieurs niveaux du traitement des instructions, notamment lorsqu'une instruction est exécutée par une Unité de Traitement (cf. ci-après).

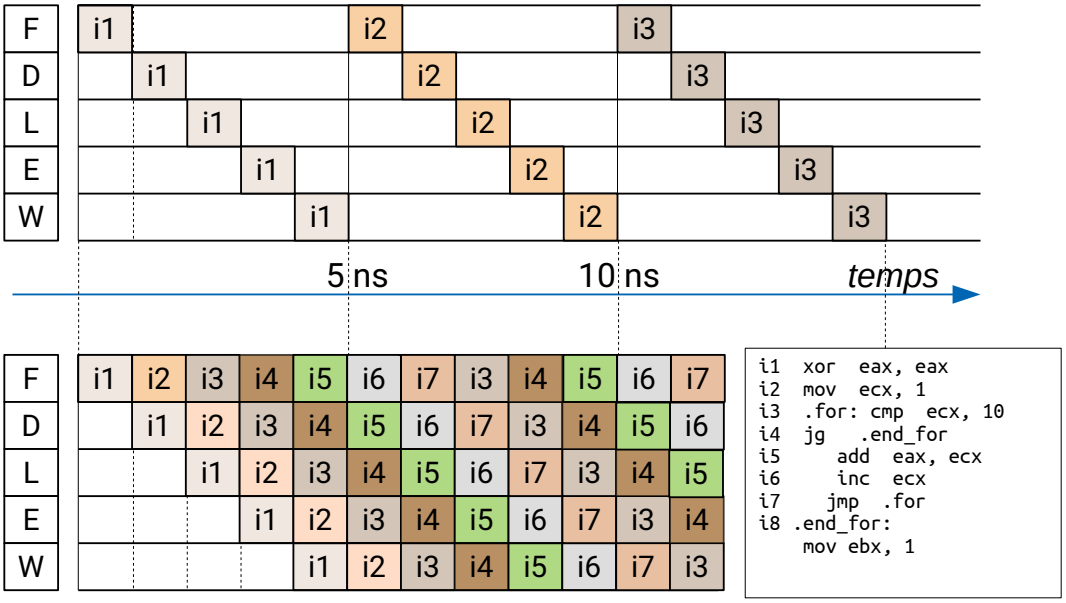


FIGURE 3.7 – Pipeline d'instructions

La question que l'on peut légitimement se poser est : quel gain apporte le pipeline ? Pour répondre à cette question il suffit de comparer les temps d'exécution avec et sans pipeline pour traiter n instructions :

- *sans pipeline* une instruction est exécutée toutes les 5 ns, si on a n instructions à exécuter il faut donc $5 \times n$ ns.
- *avec pipeline*, il faut 5 ns pour que la première instruction soit exécutée, puis $n - 1$ ns pour exécuter les $n - 1$ instructions restantes

Le gain obtenu est donné par le rapport du temps d'exécution sans pipeline par le temps d'exécution avec pipeline :

$$\text{gain} = \lim_{n \to \infty} \left(\frac{5n}{5 + n - 1} \right) \simeq \frac{5n}{n} \simeq 5$$

Un pipeline de k étapes (on parle également d'étages où *stages* en anglais), permet théoriquement de diviser le temps de traitement par k. Cependant, le nombre d'étages de traitement est limité par le nombre d'étapes élémentaires à réaliser mais et est influencé par les accès à la mémoire et le nombre d'unités de traitement (cf. sections suivantes). Plus le pipeline est long, plus il est coûteux de le vider et le réalimenter, c'est ce qui arrive lors de l'exécution des instructions

Micro architecture	Pipeline	Micro architecture	Pipeline
P5 (Pentium)	5	NetBurst (Cedar Mill)	31
P6 (Pentium 3)	10	Core	14
P6 (Pentium Pro)	14	Sandy Bridge	14
NetBurst (Willamette)	20	Haswell	14
NetBurst (Northwood)	20	Skylake	14
NetBurst (Prescott)	31	Kabylake	14

TABLE 3.7 – Nombre d'étages de pipeline pour différentes architectures Intel

conditionnelles ou lors du traitement des boucles. Il se limite à une quinzaine d'étages sur la plupart des microprocesseurs actuels (cf. Table 3.7).

Voyons à présent comment les différentes étapes de traitement des instructions s'enchaînent.

3.7 Frontal : chargement et décodage

Sur le schéma de la Figure 3.8 on a fait apparaître les différentes étapes liées au frontal.

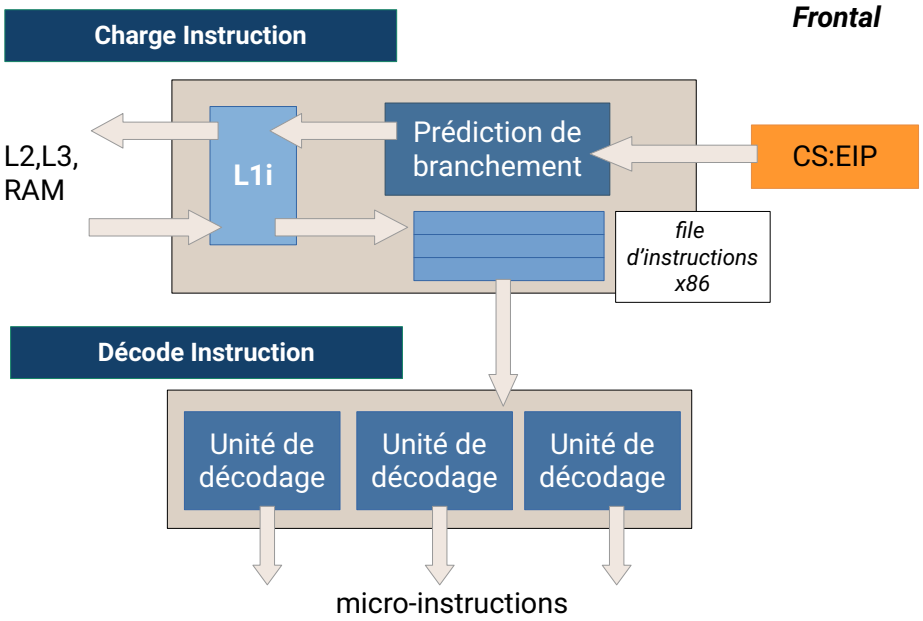


FIGURE 3.8 – Chargement et décodage

A partir de `cs:eip` on obtient l'adresse de la prochaine instruction à exécuter.

Cependant comme nous allons le voir et comme cela a déjà été évoqué, certaines instructions assembleur modifient **eip** et il est nécessaire d'utiliser un mécanisme de prédiction de branchement, représenté sur la figure par BPU pour *Branch Prediction Unit*, afin de savoir si l'on devra lire l'instruction suivante ou si on devra se déplacer à une autre adresse du code.

Une fois que l'on dispose de la bonne adresse, on récupère l'instruction à exécuter dans le cache L1 d'instructions (L1i). Il se peut que l'instruction ne soit pas présente dans le cache L1i, il faudra alors chercher si elle est dans le cache L2, puis dans le cache L3 et finalement, si elle n'est présente dans aucun cache, il faudra lancer une requête d'accès en mémoire pour récupérer les octets situés à l'adresse à lire et les charger dans les différents caches ou dans le cache L1i uniquement.

3.7.1 Chargement et prédiction de branchement

Comme nous venons de le dire, le chargement d'une instruction fait appel à plusieurs mécanismes dits de *prédiction de branchement* qui permettent de prédire à quelle adresse le pointeur d'instruction (**eip**) doit se placer. Généralement il s'agit de l'instruction suivante. Mais dans le cas de branchements, d'une boucle **for** par exemple, il faut revenir au début de la boucle après avoir exécuté son corps ou sortir de la boucle lorsque la condition d'arrêt est atteinte. On dit alors qu'il existe plusieurs chemins d'exécution.

Considérons le code C de la Figure 3.9 pour lequel on calcule la somme des entiers de 1 à 10. On voit sur l'organigramme de gauche qu'il exite deux chemins : le premier est pris lorsque $i \leq 10$ et le second lorsque $i > 10$. En prévision de ce que nous verrons dans le Chapitre 5, nous avons fait figurer le code assembleur sur la même figure. Le registre **eax** contient la somme des valeurs et le registre **ecx** représente la variable de boucle (`i`).

Après l'utilisation de l'instruction `cmp ecx, 10` qui compare le registre **ecx** à la constante 10, on place une instruction de branchement conditionnel `jg .end_for`, qui signifie *jump on greater*.

Ces instructions de branchement conditionnel sont source de ralentissement au sein du pipeline puisqu'il est nécessaire de vider le pipeline si le chemin d'exécution suivi n'est pas le bon. Si **ecx** est supérieur à 10 il faut sortir de la boucle et modifier **eip** pour qu'il pointe sur l'instruction après le label `.end_for`, c'est à dire l'instruction `i8`. Cependant les instructions suivant la comparaison (*i5, i6, i7*) ont déjà été chargées dans le pipeline pendant le traitement de *i3* et *i4*. On doit donc invalider leur traitement en vidant le pipeline et recommencer à partir de l'instruction `i8`.

Afin d'éviter le plus possible de vider le pipeline, la prédiction de branchment, comme son nom l'indique, permet de prédire dès lors qu'une instruction de type branchement est présente, si le branchement sera emprunté ou non. De son efficacité découle une vitesse de traitement accrue.

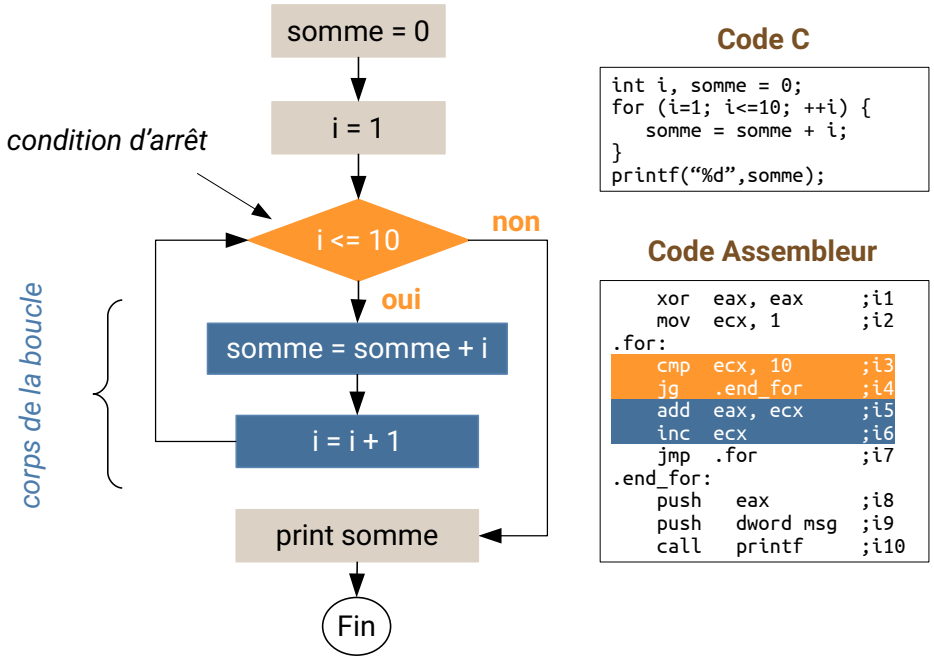


FIGURE 3.9 – Exemple de boucle for

Notons également que les conditionnelles de type `if then` ou `if then else` à l'intérieur d'une boucle (`for` ou `while`) sont les plus pénalisantes et le sont d'autant plus qu'on ne peut prédire la condition du `if` (cf. Section 5.4.11.3).

3.7.2 Décodage d'instructions

Les instructions assembleur peuvent être qualifiées de macro-instructions car elles définissent des traitements parfois très complexes. Au sein du microprocesseur, ces macro-instructions sont décomposées en une série d'instructions plus simples appelées micro-opérations et notées μ-ops.

Nous avons vu Section 3.3.2, l'instruction `add [ebx + ecx * 4 + 8], eax`. Cette instruction sera décomposée en plusieurs micro-opérations beaucoup plus simples afin d'être exécutée :

1. μ-op1 : calul de l'adresse `A = ebx + ecx * 4 + 8`
2. μ-op2 : chargement de la donnée à l'adresse mémoire `A` dans le registre `R`
3. μ-op3 : exécution de l'addition `R + eax` et stockage dans `R`
4. μ-op4 : stockage de `R` à l'adresse mémoire `A`

De retour à la Figure 3.8, nous voyons qu'une fois chargée dans une file d'instructions x86, la prochaine instruction à exécuter doit être décodée en micro-instructions. Généralement, il existe un mécanisme de cache de traduction représenté sur la figure par le μ-Ops Cache. Ce cache a pour objectif de stocker la série de

micro-instructions générées par le décodage d'une instruction x86 précédemment décodée. Si l'instruction x86 est présente dans ce cache, on approvisionnera la file de μ-ops avec les données du cache, sinon on utilisera le décodeur qui est le circuit dédié à la traduction d'une instruction x86 en μ-ops.

De nos jours la partie décodage est capable de décoder plusieurs instructions à la fois, généralement de l'ordre de 3 à 5 sur les microprocesseurs récents.

3.8 Exécution des instructions

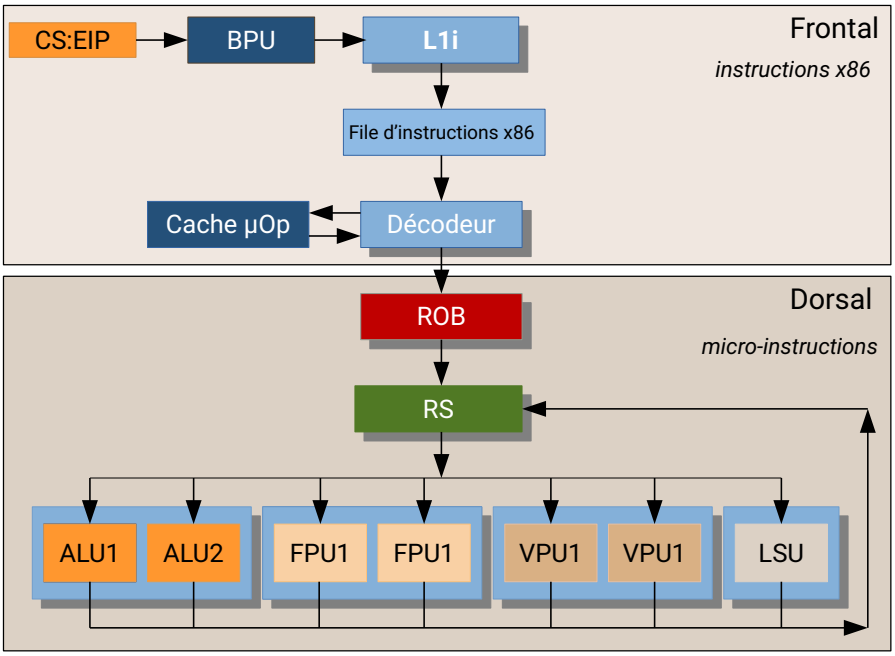


FIGURE 3.10 – Traitement instruction

3.8.1 Exécution dans le désordre

Au niveau du dorsal (voir Figure 3.10), c'est un ensemble de μ-ops associées à des instructions x86 que l'on doit traiter. Afin de diminer les temps d'attentes et ne pas ralentir l'exécution du traitement on utilise un mécanisme d'exécution dans le désordre (*Out Of Order*) qui consiste à traiter les μ-ops dès lors qu'elles disposent de toutes les ressources nécessaires pour être traitées.

Cependant, cette exécution dans le désordre pose un problème crucial à résoudre : faire en sorte qu'au final les instructions x86 soient traitées dans l'ordre dans lequel elles sont entrées dans le pipeline de traitement.

Pour ce faire, on utilise deux tampons (*buffers*) appelés *Reservation Station* et *ReOrder Buffer* notés respectivement RS et ROB. Nous ne détaillerons pas leur

fonctionnement afin de rester le plus concis possible et ne pas désorienter le lecteur, mais ces deux tampons assurent les fonctionnalités suivantes :

- ROB, comme son nom l'indique est chargé de garder la cohérence et maintenir l'ordre d'exécution, il est également chargé de l'*allocation de registres*
- RS est chargé de stocker les instructions et de les garder jusqu'à ce qu'elles soient exécutées

L'*allocation avec renommage de registres* est une technique essentielle pour traiter les instructions dans le désordre. En interne le microprocesseur dispose de plusieurs registres et lorsqu'il traite une instruction x86 il établit une correspondance entre les registres visibles par le programmeur (`eax`, `ebx`, etc...) et ses registres internes de manière à pouvoir traiter chaque instruction de manière indépendante.

3.8.2 Microprocesseur super scalaire

Dès qu'une instruction est prête à être traitée au niveau de la RS elle est envoyée à une unité de traitement dédiée. Les différentes unités sont de type entier, flottant, vectoriel et chargement / stockage de donnée. On les qualifie respectivement d'ALU, FPU, VPU et LSU :

- ALU (*Arithmetic and Logic Unit*) ou unité de traitement arithmétique et logique traite les opérations sur les valeurs entières et travaille avec les registres généraux
- FPU (*Floating Point Unit*) ou unité de traitement des nombres à virgule flottante traite les opérations sur les réels, cette unité s'appelait auparavant coprocesseur
- VPU (*Vector Processing Unit*) ou unité de traitement vectorielle s'intéresse aux vecteurs, ce sont les instructions de type MMX, SSE ou AVX
- LSU (*Load and Store Unit*) ou unité d'accés à la mémoire traite le chargement et le stockage des données ainsi que le calcul des adresses mémoire

Là encore, afin d'améliorer l'efficacité du traitement des instructions on a introduit une technologie qualifiée de super-scalaire (*Superscalar*) qui consiste à disposer de plusieurs unités de traitement de même type afin de mieux répartir la charge de travail. On qualifie cette technique d'**amélioration en largeur** du microprocesseur. Les instructions sont réparties (*dispatch*) sur les différentes unités de traitement dès qu'elles sont prêtes à être exécutées. Le principe est le même que la caisse du supermarché. Si on a une file de dix clients qui attendent pour payer leurs achats et qu'il n'y a qu'une seule caisse d'ouverte, on peut diminuer leur temps d'attente, notamment le temps d'attente des derniers clients en ouvrant une deuxième caisse. Les dix clients vont alors idéalement se répartir en deux files de cinq clients.

3.8.3 Ecriture du résultat

Après son exécution une instruction est finalement traitées par le ROB de manière à propager son résultat dans l'ordre de traitement du flux d'instructions soumis au microprocesseur. Il peut s'agir d'une écriture en mémoire, de la modification de la valeur d'un registre suivie éventuellement par la mise à jour du registre `eflags`.

3.8.4 Amélioration en longueur et en largeur

Nous avons vu que l'utilisation d'un pipeline était qualifiée d'amélioration en longueur et que l'amélioration en largeur consistait à avoir plusieurs unités de traitement. On peut combiner ces deux techniques afin de tirer profit des deux types d'améliorations mais il faut trouver un équilibre entre elles. Un pipeline trop long ou trop court aura un effet de ralentissement. Disposer de nombreuses unités de traitement mais les sous utiliser affecte la rentabilité. Une analogie appropriée pour comprendre l'interaction de ces deux techniques est celle du *fast food* qui est un restaurant dont le but est de servir rapidement ses clients. Lorsque l'on entre dans un fast food on dispose de plusieurs files d'attentes pour lesquelles une personne traite votre commande, va chercher les produits que vous avez commandés et prend en compte votre réglement. Il s'agit d'un système avec de nombreuses unités de traitement mais un pipeline court (une personne pour traiter un client).

L'autre alternative qui s'offre à vous et d'aller au *drive* où se trouve une longue file de clients en voiture qui seront servis par trois personnes différentes : l'une va prendre votre commande, la seconde s'occupe de votre réglement et la troisième vous livre votre commande. Il s'agit d'un système avec peu d'unité de traitement (une seule file de traitement) mais un long pipeline (plusieurs personnes pour traiter un client).

3.8.5 Multi-coeur et SMT

Le *Simultaneous MultiThreading* (SMT) est une technique qui permet le partage d'un coeur de processeur superscalaire entre plusieurs threads dans le but d'optimiser l'utilisation des ressources. Généralement il s'agit d'un seul autre thread, un processeur qui possède le SMT apparaît alors comme ayant le double de coeurs de calculs. Il ne faut cependant pas se laisser leurrer et croire que le microprocesseur possède deux fois plus de coeurs et donc deux fois plus de puissance de calcul puisque comme indiqué, on partage les ressources d'un coeur entre deux threads.

Cette technologie qui date des années 60 a été réintroduite sur le Pentium 4 [17] d'Intel en 2003 et a été qualifiée de technologie *Hyper-Threading* (HT). A cette

17. Il faut noter que le Pentium 4 d'architecture Willamette date de Novembre 2000, c'est seulement à partir de l'architecture Northwood et pour une fréquence de 3.06 GHz que le Pentium 4 possède l'Hyper-Threading.

époque Intel évoque un gain de 30 % par rapport à un même processeur sans HT.

Pour certains traitements parallèles, utiliser 4 coeurs doté du SMT, c'est à dire 8 threads, peut se révéler un atout. Dans d'autres cas, il sera préférable de se cantonner à utiliser les 4 coeurs sans le SMT.

Après ce bref aperçu des technologies mises en oeuvre afin de diminuer le temps d'exécution des instructions, nous allons nous intéresser au matériel et en découvrir les caractéristiques.

3.9 Apprendre à connaître son ordinateur sous Linux

3.9.1 Le microprocesseur

Si vous utilisez Windows comme système d'exploitation vous trouverez de nombreux outils professionnels pour obtenir des informations sur votre machine. L'un des plus connus et les plus intéressants est CPU-Z [18], il est doté d'une interface graphique et donne de nombreux détails sur le matériel. Il existe également GPU-Z [19] pour les cartes graphiques.

Sous Linux l'offre est plus restreinte et les outils comme `hardinfo` et `sysinfo` sont très rudimentaires. On trouve néanmois l'utilitaire graphique I-Nex qui est un clone de CPU-Z mais son installation est assez difficile et il n'est plus maintenu. Un autre outil plus récent, appelé **CPU-X**, est l'équivalent de CPU-Z.

Il existe différentes manières de récupérer l'information sous Linux par l'intermédiaire d'utilitaires en ligne de commande ou de simples commandes shell :

- `lshw` (*list hardware*)
- `dmidecode` décode une table DMI (*Desktop Management Interface*)
- `lscpu` (*list cpu*)
- `cat /proc/cpuinfo` (informations sur le microprocesseur)
- `lstopo` du package hwloc

Par exemple pour obtenir des informations sur le microprocesseur, on peut utiliser la commande `lscpu` ou, de manière équivalente, la commande `lshw` avec les arguments suivants :

```
sudo lshw -C processor
*-cpu
       description: CPU
```

18. https://www.cpuid.com/softwares/cpu-z.html
19. https://www.techpowerup.com/gpuz/

```
produit: AMD Ryzen 5 3600 6-Core Processor
fabricant: Advanced Micro Devices [AMD]
identifiant matériel: 15
information bus: cpu@0
version: AMD Ryzen 5 3600 6-Core Processor
numéro de série: Unknown
emplacement: AM4
taille: 2166MHz
capacité: 4200MHz
bits: 64 bits
horloge: 100MHz
fonctionnalités: lm fpu fpu_exception wp vme de pse tsc msr pae
mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse
sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp x86-64
constant_tsc rep_good nopl nonstop_tsc cpuid extd_apicid
aperfmperf pni pclmulqdq monitor ssse3 fma cx16 sse4_1 sse4_2
movbe popcnt aes xsave avx f16c rdrand lahf_lm cmp_legacy svm
extapic cr8_legacy abm sse4a misalignsse 3dnowprefetch osvw ibs
skinit wdt tce topoext perfctr_core perfctr_nb bpext perfctr_llc
mwaitx cpb cat_l3 cdp_l3 hw_pstate sme ssbd mba sev ibpb stibp
vmmcall fsgsbase bmi1 avx2 smep bmi2 cqm rdt_a rdseed adx smap
clflushopt clwb sha_ni xsaveopt xsavec xgetbv1 xsaves cqm_llc
cqm_occup_llc cqm_mbm_total cqm_mbm_local clzero irperf xsaveerptr
wbnoinvd arat npt lbrv svm_lock nrip_save tsc_scale vmcb_clean
flushbyasid decodeassists pausefilter pfthreshold avic v_vmsave_vmload
vgif umip rdpid overflow_recov succor smca cpufreq
configuration : cores=6 enabledcores=6 threads=12
```

On obtient le nom du microprocesseur, sa fréquence maximale de fonctionnement de 4200 Mhz (capacité) ainsi que les différentes technologies implantées (fonctionnalités) comme `avx2`, `bmi2` et `popcnt` pour celles qui nous intéresseront par la suite. On trouve également le nombre de coeurs (`cores=6`) et le nombre de threads (threads=12).

La ligne `taille` ne correspond en fait à une fréquence et varie si on relance la commande plusieurs fois. Il s'agit probablement de la fréquence de l'un des coeurs.

3.9.1.1 inxi

Un petit utilitaire intéressant sous Linux est `inxi`. Il permet d'afficher dans le terminal les informations principales de la configuration d'une machine. Pour obtenir toutes les informations relatives à une machine, on peut par exemple saisir dans le terminal, la commande :

```
1 inxi -F -c 18
```

L'option `-F` signifie *full* et l'option `-c` permet de choisir un mode de coloration. On obtient alors le compte rendu suivant que nous avons pas affiché en totalité :

```
System:
  Host: zentopia Kernel: 5.4.0-40-generic x86_64 bits: 64
  Desktop: Gnome 3.36.2 Distro: Ubuntu 20.04 LTS (Focal Fossa)
Machine:
  Type: Desktop Mobo: Micro-Star model: MPG X570 GAMING EDGE WIFI (MS-7C37)
  v: 1.0 serial: <superuser/root required> UEFI: American Megatrends v: 1.50
  date: 10/29/2019
CPU:
  Topology: 6-Core model: AMD Ryzen 5 3600 bits: 64 type: MT MCP
  L2 cache: 3072 KiB
  Speed: 2200 MHz min/max: 2200/3600 MHz Core speeds (MHz): 1: 2209 2: 2200
  3: 2200 4: 2199 5: 2200 6: 2199 7: 2200 8: 2198 9: 2199 10: 2200 11: 2199
  12: 2200
Graphics:
  Device-1: NVIDIA GP104 [GeForce GTX 1070] driver: nvidia v: 440.100
...
```

3.9.1.2 lstopo

Pour utiliser `lstopo` il faut installer le paquet `hwloc` sous Ubuntu :

```
sudo apt install hwloc
```

On utilise ensuite la commande `lstopo` ou `lstopo-no-graphics` comme suit :

```
1 lstopo
2 lstopo --no-io file.png
3 lstopo-no-graphics -.ascii --no-io
```

La première commande (ligne 1) ouvre une fenêtre qui donne la topologie du processeur (Figure 3.11) avec les informations des coeurs, l'organisation des caches et la partie concernant les interfaces entrées et sorties. Cette dernière partie peut être supprimées en utilisant l'option **`--no-io`** (ligne 2). Enfin, la ligne 3 affiche les informations au format ASCII dans un terminal.

Sur la Figure 3.11 on obtient une information détaillée sur un AMD Ryzen 7 1700X. On voit clairement comment sont répartis et numérotés les coeurs (cf. ci-après) ainsi que la taille et la répartition des mémoires caches. Ce processeur d'architecture *Summit Ridge* est décrit comme possédant 8 coeurs, 16 threads et est doté de 4 Mo de cache L2 ainsi que 16 Mo de cache L3. Cependant, sur la figure, on voit que le cache L3 est scindé en deux fois 8 Mo chacun associé à 4 coeurs SMT,

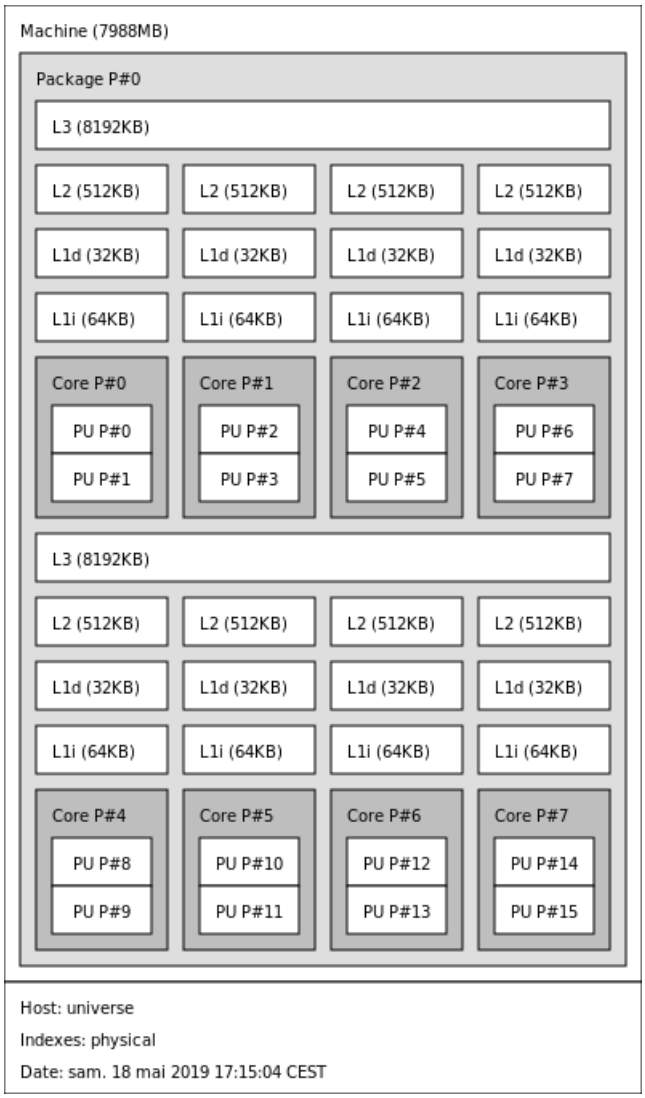


FIGURE 3.11 – Résultat de la commande lstopo sur AMD Ryzen 7 1700X

ce qui fait 8 Mo pour 8 threads. Chaque groupe de 2 threads dispose de 512 ko de cache L2, 64 ko de cache L1i et 32 ko de cache L1d.

Ce schéma indique également la numérotation des threads (`P#0` à `P#15` sur fond gris) :

- les coeurs ont donc un numéro d'identification pair (P#0, P#2 à P#14)
- les coeurs SMT ont des numéros impairs (P#1, P#3, à P#15)

Quant à la mémoire cache, on peut obtenir beaucoup d'informations en listant le contenu du répertoire :

```
1 ls /sys/devices/system/cpu/cpu0/cache/index*
```

Chaque `index` correspond à un cache, l'index 0 est le cache L1 de données, l'index 1 est le cache L1 d'instructions, l'index 2 est le cache L2 et l'index 3 correspond au cache L3. Voici un petit script bash qui permet d'afficher ces informations :

```
#!/bin/sh
input_dir="/sys/devices/system/cpu/cpu0/cache"
levels=`ls -d ${input_dir}/index[0-9]`
levels=`echo $levels | tr ' ' '\n' | wc -l`
level=0
while [ $level -lt $levels ] ; do
        size=`cat ${input_dir}/index${level}/size | awk '{ printf("%6s",$1);}'`
        type=`cat ${input_dir}/index${level}/type | awk '{ printf("%12s",$1);}'`
        levl=`cat ${input_dir}/index${level}/level`
        assc=`cat ${input_dir}/index${level}/ways_of_associativity`
        lnsz=`cat ${input_dir}/index${level}/coherency_line_size`
        echo "L$levl $type $size ${assc}-way-set-associative ${lnsz} bytes"
        level=`expr $level + 1`
done
```

On peut également utiliser la commande `sudo lshw -short -C memory`, (voir ci-après pour la mémoire).

3.9.2 La carte mère

Le framework logiciel DMI pour *Desktop Management Interface* fournit un standard afin de gérer et de suivre les modifications de composants sur un ordinateur. L'utilitaire `dmidecode` permet de lire les informations DMI de l'ordinateur et de les afficher au format binaire ou dans un format texte compréhensible par un humain. Pour obtenir des informations sur la carte mère, on peut utiliser la commande suivante :

```
sudo dmidecode -t baseboard
```

```
# dmidecode 3.2
Getting SMBIOS data from sysfs.
SMBIOS 2.8 present.

Handle 0x0002, DMI type 2, 15 bytes
Base Board Information
        Manufacturer: Micro-Star International Co., Ltd.
        Product Name: MPG X570 GAMING EDGE WIFI (MS-7C37)
        Version: 1.0
        Serial Number: J816453611
        Asset Tag: To be filled by O.E.M.
```

```
        Features:
                Board is a hosting board
                Board is replaceable
        Location In Chassis: To be filled by O.E.M.
        Chassis Handle: 0x0003
        Type: Motherboard
        Contained Object Handles: 0

Handle 0x0039, DMI type 41, 11 bytes
Onboard Device
        Reference Designation: RTL8111EPV
        Type: Ethernet
        Status: Disabled
        Type Instance: 1
        Bus Address: 0000:03:00.0
```

Ici, il s'agit d'une carte MSI X570 GAMING EDGE WIFI.

3.9.3 La mémoire

De la même manière, pour obtenir des informations concernant le sous-système mémoire, il suffit d'utiliser `dmidecode` :

```
sudo dmidecode -t memory
# dmidecode 3.2
Getting SMBIOS data from sysfs.
SMBIOS 2.8 present.

Handle 0x000F, DMI type 16, 23 bytes
Physical Memory Array
        Location: System Board Or Motherboard
        Use: System Memory
        Error Correction Type: None
        Maximum Capacity: 128 GB
        Error Information Handle: 0x000E
        Number Of Devices: 4

Handle 0x0017, DMI type 17, 40 bytes
Memory Device
        Array Handle: 0x000F
        Error Information Handle: 0x0016
        Total Width: 64 bits
        Data Width: 64 bits
        Size: 8192 MB
        Form Factor: DIMM
        Set: None
```

```
24          Locator: DIMM 0
25          Bank Locator: P0 CHANNEL A
26          Type: DDR4
27          Type Detail: Synchronous Unbuffered (Unregistered)
28          Speed: 3200 MT/s
29          Manufacturer: Unknown
30          Serial Number: 00000000
31          Asset Tag: Not Specified
32          Part Number: F4-3200C16-8GVKB
33          Rank: 1
34          Configured Memory Speed: 3200 MT/s
35          Minimum Voltage: 1.2 V
36          Maximum Voltage: 1.2 V
37          Configured Voltage: 1.2 V
38  ...
```

On voit que la carte mère possède quatre slots de connexion (*Number Of Devices*, ligne 13), on peut donc positionner quatre barrettes de mémoires mais il ne faut pas dépasser 128 Go (*Maximum Capacity*, ligne 11) , soit par exemple quatre barrettes de 32 Go, ce qui est déjà énorme. Le premier slot mémoire DIMM0 est occupé par une barrette de 8 Go (ligne 21) de PC-3200 (ligne 34), pour laquelle on n'a pu déterminer le fabriquant (*Manufacturer : Unknown*, ligne 29). Il s'agit en fait de barrettes G-Skill dont la référence est F4-3200C16-8GVKB.

On peut également utiliser la commande `lshw` pour obtenir un sommaire des différentes mémoires :

```
1   sudo lshw -short -C memory
2   Chemin ...   Classe         Description
3   ========================================================================
4   /0/0           memory         64KiB BIOS
5   /0/f           memory         32GiB Mémoire Système
6   /0/f/0         memory         8GiB DIMM DDR4 Synchrone Unbuffered (Unregistered)
7   /0/f/1         memory         8GiB DIMM DDR4 Synchrone Unbuffered (Unregistered)
8   /0/f/2         memory         8GiB DIMM DDR4 Synchrone Unbuffered (Unregistered)
9   /0/f/3         memory         8GiB DIMM DDR4 Synchrone Unbuffered (Unregistered)
10  /0/12          memory         384KiB L1 cache
11  /0/13          memory         3MiB L2 cache
12  /0/14          memory         32MiB L3 cache
13
```

On voit ici que la mémoire centrale possède une taille de 32 Go organisée en 4 fois 8 Go. Les mémoires cache de niveau L1 (Données et Instructions) font 32 ko, le cache L2 possède une taille de 512 ko et le cache L3 fait 32 Mo (2×16 Mo).

Nous avons évoqué la présence d'informations liées au timings mémoires Section 3.2. On peut obtenir ces informations en installant `i2c-tools` :

```
sudo apt install i2c-tools
sudo modprode eeprom
decode-dimms
```

On obtient entre autres informations :

```
EEPROM CRC of bytes 128-253                     OK (0xA01C)

---=== Memory Characteristics ===---
Maximum module speed                            2132 MHz (PC4-17000)
Size                                            8192 MB
Banks x Rows x Columns x Bits                   16 x 16 x 10 x 64
SDRAM Device Width                              8 bits
Ranks                                           1
AA-RCD-RP-RAS (cycles)                          15-15-15-36
Supported CAS Latencies                         16T, 15T, 14T, 13T, 12T, 11T, 10T

---=== Timings at Standard Speeds ===---
AA-RCD-RP-RAS (cycles) as DDR4-1866             13-13-13-31
AA-RCD-RP-RAS (cycles) as DDR4-1600             11-11-11-27

---=== Timing Parameters ===---
Minimum Cycle Time (tCKmin)                     0.938 ns
Maximum Cycle Time (tCKmax)                     1.600 ns
Minimum CAS Latency Time (tAA)                  13.750 ns
Minimum RAS to CAS Delay (tRCD)                 13.750 ns
Minimum Row Precharge Delay (tRP)               13.750 ns
Minimum Active to Precharge Delay (tRAS)        33.000 ns
Minimum Active to Auto-Refresh Delay (tRC)      46.750 ns
Minimum Recovery Delay (tRFC1)                  350.000 ns
Minimum Recovery Delay (tRFC2)                  260.000 ns
Minimum Recovery Delay (tRFC4)                  160.000 ns
Minimum Four Activate Window Delay (tFAW)       21.000 ns
Minimum Row Active to Row Active Delay (tRRD_S) 3.700 ns
Minimum Row Active to Row Active Delay (tRRD_L) 5.300 ns
Minimum CAS to CAS Delay (tCCD_L)               5.625 ns
Minimum Write Recovery Time (tWR)               15.000 ns
Minimum Write to Read Time (tWTR_S)             2.500 ns
Minimum Write to Read Time (tWTR_L)             7.500 ns

---=== Other Information ===---
Package Type                                    Monolithic
Maximum Activate Count                          Unlimited
Post Package Repair                             One row per bank group
Soft PPR                                        Supported
Module Nominal Voltage                          1.2 V
Thermal Sensor                                  No
```

```
42
43 ---=== Physical Characteristics ===---
44 Module Height                                    32 mm
45 Module Thickness                                 2 mm front, 2 mm back
46 Module Reference Card                            A revision 1
47
48 ---=== Manufacturer Data ===---
49 Module Manufacturer                              Undefined
50 Part Number                                      Undefined
51 ...
```

Malheureusement, on n'obtient pas toutes les informations, comme par exemple le fabricant (*Module Manufacturer*).

Il s'agit ici de barrettes de DDR4-SDRAM au format UDIMM, PC4-17000 de 8 Go qui possède plusieurs fréquences de fonctionnement.

3.9.4 CPU-X

Enfin, on dispose à présent d'un outil qui s'appelle CPU-X[20] qui est l'équivalent de CPU-Z. Sur le site du logiciel on trouvera des packages pour Linux pour Debian, OpenSUSE et Ubuntu qui pour la version 4.0 de CPU-X gère les versions 16.04, 18.04, 19.04, 19.10 et 20.04 de Ubuntu (voir Figure 3.12).

Pour installer CPU-X, il suffit de télécharger l'archive correspondant à Ubuntu sur le site du logiciel, puis de lancer les commandes suivantes :

```
1 mkdir install
2 cd install
3 mv ~/Téléchargements/CPU-X_v4.0.1_Ubuntu.tar.gz .
4 tar -xzf CPU-X_v4.0.1_Ubuntu.tar.gz
5 cd xUbuntu_20.04
6 sudo dpkg -i libcpuid15_0.5.0_amd64.deb cpuidtool_0.5.0_amd64.deb cpu-x_4.0.1_amd64.deb
7 cpu-x
```

On peut également obtenir l'information dans le terminal en utilisant l'option en ligne de commande `-ncurses` :

```
1 cpu-x --ncurses
```

20. https://x0rg.github.io/CPU-X/

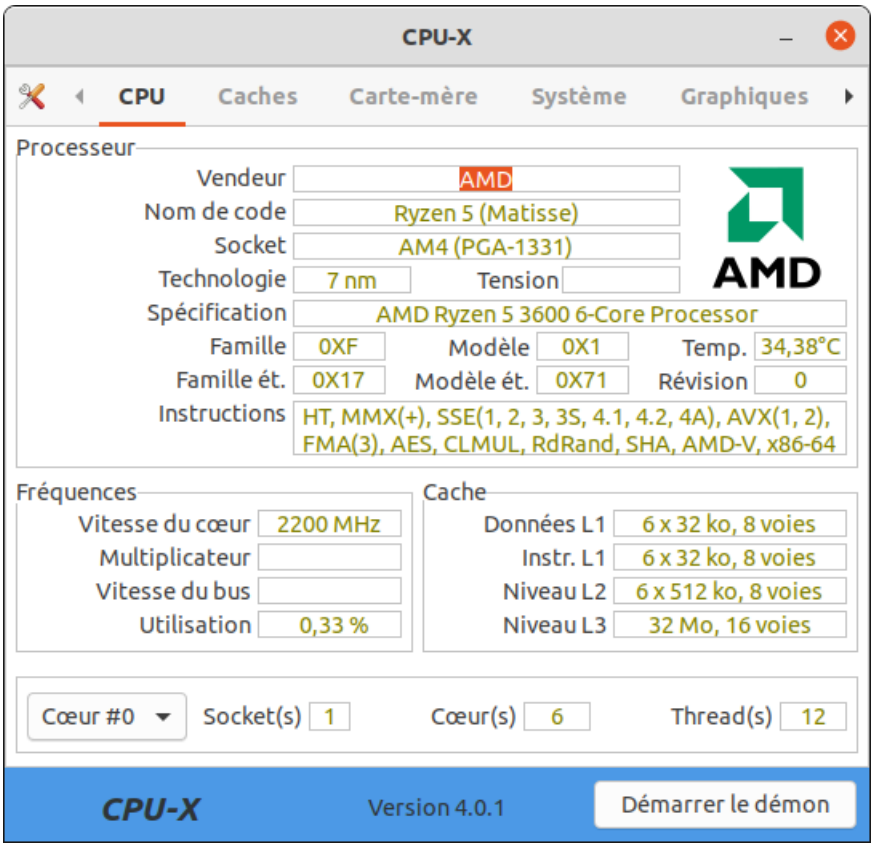


FIGURE 3.12 – Interface de CPU-X

3.10 Outils de tests

3.10.1 Phoronix

Phoronix[21] est un logiciel qui permet d'installer et exécuter une série de *benchmarks* assez fournie.

```
sudo apt-get install phoronix-test-suite
phoronix-test-suite list-available-tests
```

La première commande installe le logiciel Phoronix et la seconde donne la liste des tests disponibles. Il en existe plus de 300 dans la version 5.2.1. On peut ensuite installer les tests qui nous intéressent comme celui des n reines :

```
phoronix-test-suite install n-queens
phoronix-test-suite run n-queens
```

21. On pourra consulter http://www.phoronix-test-suite.com/ pour de plus amples informations.

Le programme demande un identifiant pour le test à réaliser qui pourra être utilisé par la suite pour afficher les résultats :

```
phoronix-test-suite info n-queens-results
phoronix-test-suite result-file-to-csv n-queens-results
...
"N-Queens - Elapsed Time",16.13
```

3.10.2 Sysbench

```
sudo apt-get install sysbench
```

Pour tester le CPU, on exécute le test adéquat qui calcule des décimales de π :

```
sysbench cpu --cpu-max-prime=20000 run
sysbench 1.0.11 (using system LuaJIT 2.1.0-beta3)

Running the test with following options:
Number of threads: 1
Initializing random number generator from current time


Prime numbers limit: 20000

Initializing worker threads...

Threads started!

CPU speed:
    events per second:   182.73

General statistics:
    total time:                          10.0016s
    total number of events:              1828

Latency (ms):
         min:                                    5.37
         avg:                                    5.47
         max:                                   10.31
         95th percentile:                        5.99
         sum:                                 9997.81

Threads fairness:
    events (avg/stddev):           1828.0000/0.00
    execution time (avg/stddev):   9.9978/0.00
```

```
sysbench memory --memory-block-size=256K run
sysbench 1.0.11 (using system LuaJIT 2.1.0-beta3)

Running the test with following options:
Number of threads: 1
Initializing random number generator from current time


Running memory speed test with the following options:
  block size: 256KiB
  total size: 102400MiB
  operation: write
  scope: global

Initializing worker threads...

Threads started!

Total operations: 199824 (19977.75 per second)

49956.00 MiB transferred (4994.44 MiB/sec)


General statistics:
    total time:                          10.0002s
    total number of events:              199824

Latency (ms):
         min:                                    0.05
         avg:                                    0.05
         max:                                    5.53
         95th percentile:                        0.05
         sum:                                 9875.07

Threads fairness:
    events (avg/stddev):           199824.0000/0.00
    execution time (avg/stddev):   9.8751/0.00
```

3.10.3 Geekbench

Geekbench est une suite de test disponible pour Windows, Linux et MacOS qui permet d'évaluer la puissance de calcul du microprocesseur que ce soit en mono core ou en multi-core. La version 4 compare les scores obtenus lors des tests par rapport à un microprocesseur de référence qui est un Intel Core i7-6600U tournant à 2,6 GHz. Pour la version 5, il s'agit d'un Intel Core i3-8100.

Microprocesseur	SC 32 bits	MP 32 bits	SC 64 bits	MC 64 bits
Intel i5-7400	4106	11252		
Intel i7 8700			5153	22744
AMD Ryzen 7 1700X	4029	25046	4507	27207
AMD Ryzen 5 5600g	5627	30348	6677	34098
Intel i7 10850H	5705	26103	6443	28378

TABLE 3.8 – Résultats GeekBench 4.x.x en 32 et 64 bits SC (mono core) et MC (multi core)

Microprocesseur	SC 64 bits	MC 64 bits
Intel i5-7400	990	3200
Intel i3-6100	1014	2269
Intel i7 8700	1230	6448
AMD Ryzen 7 1700X	967	6261
AMD Ryzen 5 3600	1333	7705
AMD Ryzen 5 5600g	1493	8313
Intel i7 10850H	1367	6923

TABLE 3.9 – Résultats GeekBench 5 en 64 bits SC (mono core) et MC (multi core)

Les tests sont liés au calcul sur les entiers, calculs sur les réels, la cryptographie et l'accès mémoire. Pour exécuter les tests il suffit de récupérer une archive sur le site Geekbench [22], de la décompresser et lancer les deux programmes de tests en 32 et 64 bits.

```
tar -xvzf ~/Downloads/Geekbench-4.3.4-Linux.tar.gz
cd Geekbench-4.3.4-Linux
./geekbench4_x86_32
./geekbench4_x86_64
```

Après exécution des tests un lien vers le web est fourni qui permet d'obtenir le détail des résultats. Nous présentons Table 3.8 les résultats obtenus pour plusieurs microprocesseurs en 32 et 64 bits pour une exécution des tests en mono core (*SC = Single Core*) et multi core (*MC = Multi Core*).

La Table 3.9 présente des résultats pour Geekbench dans sa version 5.

22. https://www.geekbench.com/download/linux/

3.11 Comparaison de microprocesseurs

A titre d'exercice, nous allons comparer deux microprocesseurs de la famille Intel. Le premier est un Core i3-6100 et le second un Core i5-7400.

Sur le papier, le Core i5 est plus performant que le Core i3 pour plusieurs raisons :

- c'est un Core i5 qui dispose de plus de mémoire cache et plus de coeurs qu'un Core i3
- le Core i5 est de génération plus récente (7XXX) que le Core i3 (6XXX)
- les trois derniers chiffres de 7400 sont un indicateur de fréquence, donc normalement 400 étant supérieur à 100 (6100), le Core i5 devrait avoir une fréquence de fonctionnement supérieure au Core i3

En pratique, il faut aller sur le site `ark.intel.com` pour obtenir les informations de ces deux microprocesseurs. Nous avons résumé Table 3.10 les caractéristiques comparées de ces microprocesseurs.

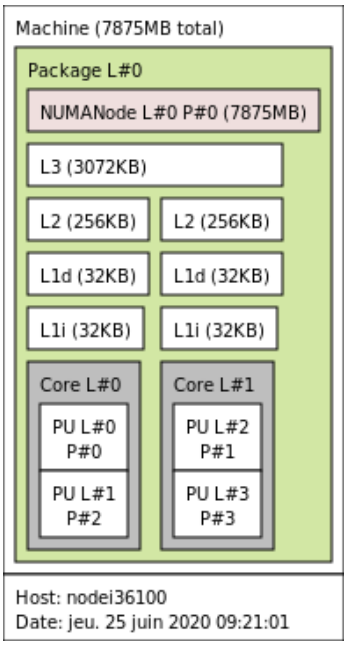
Processeur	Core i5-7400	Core i3-6100
Génération	7 / Kaby Lake	6 / Skylake
Date de lancement	Q1'2017	Q3'2015
Finesse de gravure (nm)	14	14
Prix (dollars)	182	117
Coeurs/Threads	4/4	2/4
Cache L3 (Mo)	6	3
Fréquence de base (GHz)	3,00	3,70
Fréquence turbo (GHz)	3,50	3,70
Technologie vectorielle	AVX2	AVX2

TABLE 3.10 – Caractéristiques des Core i5-7400 et Core i3-6100

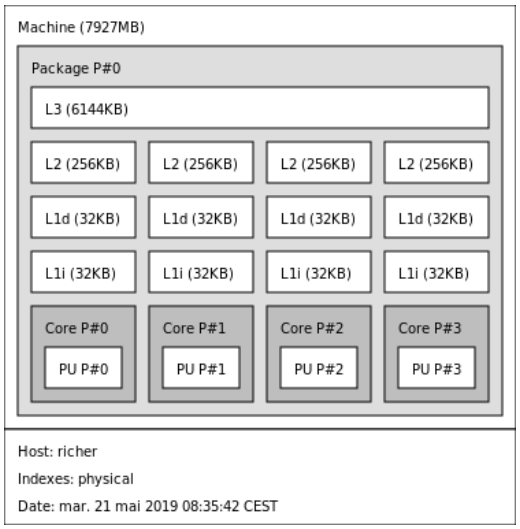
Nous voyons donc que le Core i5 comprend quatre coeurs et 6 Mo de cache L3 alors que le Core i3 comprend deux coeurs dotés de l'HyperThreading et moitié moins de cache L3. Les deux architectures sont présentées Figure 3.13.

Cependant, un détail change beaucoup de chose, c'est la fréquence de fonctionnement qui plafonne à 3,5 GHz pour le Core i5 alors que le Core i3 fonctionne avec 200 MHz de plus.

Il en résulte que pour les programmes monothreads c'est le Core i3 qui sera généralement le plus performant, alors que le Core i5 prendra l'avantage sur les programmes multithreads ou pour lesquels l'accès à la mémoire cache est important, comme indiqué Table 3.11.



(a) Intel Core i3 6100



(b) Intel Core i5 7400

FIGURE 3.13 – Comparaison des architectures Intel Core i3 6100 et i5 7400

Par exemple, pour les tests 3 à 7, la fréquence du CPU est un facteur déterminant, le Core i3 sera donc plus performant que le Core i5. Cependant, pour le test 4 le Core i5 sort grand gagnant car le traitement des instructions AVX a probablement été amélioré sur le Core i5, les deux processeurs ayant 2 ans de différence.

Test No	Description	Core i5-7400	Core i3-6100
1	Produit de matrices 2048x2048	**50,35**	61,84
2	Produit de matrices 2060x2060	36,83	**34,40**
3	SAXPY 524417 FPU	45,20	**43,52**
4	SAXPY 524417 AVX	**10,00**	14,43
5	Popcnt 512333 réf.	8,05	**7,54**
6	Compte Voyelles 524288 (if)	15,50	**14,47**
7	Compte Voyelles 524288 AVX	0,85	**0,79**

TABLE 3.11 – Temps d'exécution en secondes de certaines études de cas pour Core i5-7400 et Core i3-6100

3.12 Conclusion

3.12.1 Que retenir ?

- ▷ la mémoire possède une influence non négligeable lors de certains traitements et notamment le fait que les données à traiter soient présentes ou absentes dans la mémoire cache
- ▷ l'alignement mémoire, c'est à dire le fait que les données soient positionnées en mémoire à des adresses multiples de 8, 16 ou 32 peut influer sur la durée de leur traitement
- ▷ les améliorations liées aux microprocesseurs tentent de maximiser le nombre de traitements que l'on peut réaliser en parallèle que ce soit au niveau du décodage ou du traitement des instructions (pipeline, superscalaire) ou au niveau du traitement des données (vectorisation)
- ▷ les instructions si elles sont traitées dans l'ordre où elles arrivent depuis l'extérieur du microprocesseur sont en fait exécutées dans un mode dit dans le désordre (*Out Of Order*)au sein du microprocesseur, ceci afin d'augmenter l'efficacité de leur traitement

3.12.2 Compétence à acquérir

- □ être en mesure d'obtenir les informations concernant un microprocesseur (architecture, taille des mémoires cache, technologies disponibles)
- □ être en mesure de comparer plusieurs microprocesseurs

3.13 Questions

3.14 Exercices

Exercice 18 - Quelle est la bande passante :

1. d'une mémoire DDR3 fonctionnant à 150 MHz ?
2. d'une mémoire DDR4 fonctionnant à 200 MHz ?
3. d'une mémoire DDR4 fonctionnant à 325 MHz ?

Exercice 19 - Quelle est la fréquence de fonctionnement :

1. d'une mémoire PC3-12800 ?
2. d'une mémoire PC3-17000 ?
3. d'une mémoire PC4-17000 ?

4. d'une mémoire PC4-25600 ?

Exercice 20 - Comparer l'Intel i9 10900K avec l'Intel i9 10850K. En novembre 2020, le 10850K peut être trouvé au prix de 506 €, alors que le 10900K coûte environ 600 €. En général le 10850K coûte entre 50 et 150 € de moins que le 10900K.

Chapitre 4

Outils pour la Programmation Assembleur

Some people relax with a nice drink by the pool,
I relax by playing around with inline assembly code

Linus Torvalds

4.1 Introduction

Dans ce chapitre nous allons installer et découvrir les logiciels qui nous permettront de développer en assembleur. Nous prenons comme plateforme de développement une machine sous Linux Ubuntu 18.10 et suivantes.

Nous aurons besoin de plusieurs types d'outils :

- un **éditeur** qui nous permettra de saisir du code assembleur avec par exemple une coloration syntaxique et une indentation automatique afin de faciliter l'écriture et la relecture du code
- un **assembleur** qui compile le code assembleur et le transforme en fichier objet
- un **compilateur** C/C++ qui permet de faire l'édition de lien avec la bibliothèque C/C++
- un **débogueur** qui permet d'examiner le code généré et de l'exécuter pas à pas afin de détecter des erreurs d'accès à la mémoire par exemple ou vérifier le résultat produit par les instructions

L'édition de lien sera réalisée grâce à `g++` le *GNU C++ Compiler* ou `gcc` la version C du compilateur GNU. Il est bien évidemment possible d'utiliser d'autres compilateurs comme `clang++` de LLVM (*Low Level Virtual Machine*), `icpc` le compilateur Intel ou `pgi++` de PGI (*Portlang Group, Inc*). L'intérêt d'utiliser un compilateur

C pour réaliser l'édition de lien est que l'on a accès aux fonctions de la librairie C, il suffira de déclarer les fonctions C comme externes au niveau des fichiers assembleur.

4.2 Les éditeurs

Il existe de nombreux éditeurs sous Linux mais ils ne sont pas tous forcément adaptés à la structure des programmes assembleurs. En effet, un programme assembleur se compose de trois parties : les étiquettes (ou labels), les instructions assembleur et les commentaires. Il est pratique de pouvoir écrire les étiquettes à gauche, les instructions au centre et les commentaires à droite. Malheureusement les éditeurs sont généralistes et ne permettent pas ce genre d'organisation, il faut donc utiliser les tabulations.

```
1 ; label --- instruction ------- commentaire
2
3 label:      mov     ecx, 1      ; i = 1
```

Choisir un éditeur est toujours une question de goût personnel. Certains préfèrent les environnements de développement cossus avec une interface graphique fournie du type IDE comme **Eclipse** alors que d'autres chérissent les éditeurs épurés comme **nano**, **vim**, **emacs**. Personnellement j'utilise **jedit** qui est intéressant car on peut le configurer simplement et est installable sous Ubuntu sous forme d'un paquet du même nom.

4.2.1 jEdit

jEdit est un éditeur de texte qui a pour particularité d'être codé en Java. Il faudra donc installer une machine virtuelle Java afin de pouvoir l'utiliser. jEdit gère différents langages (Ada, Python, Java, C, ...) ainsi que l'assembleur x86. Son principal intérêt et qu'il peut être configuré simplement et permet d'ajouter de nouveaux modes d'édition pour d'autres langages.

En me basant sur le mode d'édition assembly-x86, j'ai créé un fichier nommé :

`assembly_x86_2020.lang`

qui intégre a priori toutes les instructions x86 décrites sur le site http://nasm.us en 2019 ainsi que les instructions conditionnelles (cmovCC, setCC, cf. Section 5.4.12) et les instructions de saut conditionnel. Cela représente au total 1685 instructions. Pour configurer jEdit, il suffit de consulter la page internet dédiée à l'assembleur sur le site de l'auteur [1].

1. http://leria-info.univ-angers.fr/~jeanmichel.richer/assembleur.php

4.2.2 gedit

`gedit` est un éditeur de texte libre fourni par défaut avec l'environnement graphique GNOME [2]. Il supporte l'encodage UTF-8 et gère la coloration syntaxique de nombreux langages de programmation mais malheureusement pas de l'assembleur. On peut néanmoins ajouter un fichier `.lang` dans le répertoire adéquat afin de donner les régles de coloration syntaxique de l'assembleur x86.

Téléchargez le fichier suivant et copiez le dans le répertoire de `gedit` :

```
wget http://www.info.univ-angers.fr/~richer/assembly/gedit/assembly_x86_2020.lang
sudo cp assembler.lang /usr/share/gtksourceview-*.0/language-specs/
```

4.2.3 kate

Kate est un éditeur de texte spécialement adapté à la programmation issu de l'environnement de bureau KDE. Pour disposer de la coloration syntaxique il faut utiliser le menu *Tools* > *Higlighting* > *Assembler* > *Intel x86 (NASM)*.

4.2.4 emacs

`emacs` (*Editor MACroS*) est à la base un éditeur de texte extensible basé sur le langage LISP. Il a été popularisé avec la version GNU écrite par Richard Stallman [3]. Il faudra par exemple installer les paquets `nasm-mode` et `x86-lookup` afin de pouvoir disposer respectivement de la coloration syntaxique et de l'aide en ligne.

4.2.5 Autres éditeurs

Le concurrent d'emacs, la fameux `vi` ou sa version améliorée `vim` peut également être utilisé pour écrire des programmes en assembleur. Il faudra le configurer pour pouvoir utiliser un mode assembleur.

Eclipse dispose d'un plugin appelé *ASM Plugin* disponible sur `sourceforge.net` [4] et qui permet d'utiliser différents assembleurs comme **`nasm`**, **`tasm`**, **`masm`**. Cependant il n'est plus maintenu depuis 2006.

On pourra utiliser Sublime Text qui possède un mode nasm mais j'ai rencontré beaucoup de problèmes pour installer Package Control qui est censé gérer les packages et celui-ci ne fonctionnait pas pour installer de nouveaux packages.

De ce point de vue, Atom [5] se montre plus convivial et fonctionnel, il suffit

2. Acronyme de *GNU Network Object Model Environment*, il s'agit d'un environnement de bureau libre convivial utilisé sous Linux/UNIX.
3. Fondateur du projet GNU et initiateur du mouvement Logiciel Libre.
4. http://asmplugin.sourceforge.net/
5. https://atom.io/

d'installer le paquet `language-nasmx86`.

4.3 L'assembleur nasm

Le choix de `nasm` (*The Netwide Assembler*) semble assez naturel car il est très simple à utiliser. Nous verrons que la structure des programmes `nasm` liée à l'utilisation du compilateur `gcc` permet une interaction rapide et peu complexe à mettre en oeuvre entre code en C/C++ et code assembleur. `nasm` dispose en outre d'une large documentation sous forme de site web ou de fichier PDF téléchargeable depuis le site `http://nasm.us` et d'une communauté active.

La structure d'un programme `nasm` est également très simple, voici par exemple le fameux programme *hello world !* présenté Listing 4.3.1. Il s'agit de la version en architecture 32 bits. On notera que les commentaires sont introduits par le symbole *point virgule* (`;`) et tout ce qui suit ce symbole jusqu'à la fin de la ligne fait partie du commentaire.

```
1  global main          ; definition de la fonction main
2  extern printf        ; utilisation de printf
3
4  ; ==== DATA ====
5  section .data
6
7      ; declaration d'une chaine
8      msg:     db "hello world!", 10, 0
9
10 ; ==== CODE ====
11 section .text
12
13 ; fonction main
14 main:
15     push    ebp             ; entree dans la fonction
16     mov     esp, esp
17
18     push    dword msg       ; affichage de la chaine
19     call    printf
20     add     esp, 4
21
22     xor     eax, eax
23     mov     esp, ebp        ; sortie de la fonction
24     pop     ebp
25     ret
```

Listing 4.3.1 – Hello world en nasm

Dans la forme la plus épurée d'un programme, il suffit d'utiliser deux sections :

- la section de données (`.data`) qui est une section de données *initialisées*
- la section de code (`.text`) qui contient les instructions

Il existe bien entendu d'autres sections que l'on retrouve en C comme la section `.rodata` pour les données en lecture seule et la section `.bss` pour *Block Started by Symbol* qui contient des données non initialisées.

.bss	.data	Type	Bits	Type C
resb	db	byte	8 bits	char
resw	dw	word	16 bits	short
resd	dd	double word	32 bits	int, float
resq	dq	double précision	64 bits	double

TABLE 4.1 – Définition de variable dans la section `.bss` ou `.data`

On utilise le mot clé `global` (ligne 1) suivi d'un identifiant pour indiquer que cet identifiant sera visible par les autres fichiers objets. En général cet identifiant est le nom d'un sous-programme, en particulier ici il s'agit de la méthode `main` que l'on retrouve dans un programme C.

Le mot clé `extern` (ligne 2) indique, quant à lui les noms de symboles définis dans d'autres fichiers assembleur ou C. On déclarera en externe toute fonction de la bibliothèque C que l'on utilisera.

On remarque que la déclaration d'une chaîne de caractères est réalisée en utilisant le mot clé `db` pour *do byte*. Nous avons fait figurer Table 4.1 les mots clés qui permettent de définir les données en fonction du segment de données (`.data` ou `.bss`). Par exemple pour créer un tableau de 8 octets, il faudra l'initialiser dans la partie `.data` alors que dans la section `.bss`, il suffit d'indiquer le nombre d'octets que l'on utilisera :

```
1 section .data
2     ; tableau initialisé de 8 octets
3     tab1:   db 0, 1, 2, 3, 4, 6, 7
4
5 section .bss
6     ; réserve 8 octets
7     tab2:   resb    8
```

On note que le caractère qui correspond au passage à la ligne en C, le fameux `'\n'` n'existe pas en `nasm` si on définit la chaîne par des guillemets simples ou doubles, il est donc nécessaire de le déclarer en utilisant son code ASCII qui vaut 10. Pour rappel (cf. Section 2.5.1), le dernier caractère qui est 0 marque la fin de la chaîne en langage C. Si on désire utiliser `'\n'` il faut définir la chaîne entre deux symboles *backquote* qui correspond à l'accent grave en français : `` `hello world\n` ``.

Le reste du code implante le sous-programme `main` comme une fonction (voir le Chapitre 6) qui appelle la fonction `printf` de la librairie C pour afficher une chaîne de caractères.

4.3.1 Compilation

La compilation d'un programme en assembleur est réalisée en utilisant sous Linux la commande `nasm` :

```
nasm [arguments] [-o ficher_objet.o] source.asm
```

La partie `-o fichier_objet.o` est optionnelle et permet de modifier le fichier de sortie qui par défaut sera un fichier d'extension `.o` comportant comme identifiant le nom du fichier en entrée. En l'occurrence ce serait ici `source.o`. On peut ajouter à cette commande des arguments qui définissent le format de sortie du fichier objet (cf. Table 4.2).

Arguments	Description
`-f elf`	compilation en 32 bits au format elf
`-f elf64`	compilation en 64 bits au format elf64
`-g -F dwarf`	flags de débogage sous Linux

TABLE 4.2 – Arguments du programme nasm

Pour obtenir un fichier assembleur pour une architecture 64 bits sous Linux avec des options de débogage on utilisera donc :

```
nasm -f elf64 -g -F dwarf source.asm
```

Convention

Lorsque l'on créera un fichier assembleur sous le format **`nasm`** on lui attribuera l'extension : **`_nasm.asm`**.

4.4 Edition de lien avec gcc/g++

Le compilateur C/C++ peut être utilisé pour réaliser l'édition de liens avec la bibliothèque standard du C ce qui permet de ne pas réinventer la roue et disposer de toutes les fonctionnalités du C comme l'affichage (`printf`), la saisie (`scanf`), la conversion (`atoi`, `atof`), l'allocation mémoire (`malloc`), etc.

On pourra bien évidemment utiliser d'autres compilateurs que le compilateur GNU comme par exemple :

- `icpc` d'Intel qui est généralement très performant
- `clang++` de LLVM
- `pgc++` de PGI

4.4.1 Edition de liens avec un seul fichier assembleur

L'édition de liens qui consiste à regrouper plusieurs fichiers objets compilés séparément afin d'obtenir un exécutable est réalisée avec le compilateur C grâce à la commande :

```
g++ -o executable.exe mon_fichier_assembleur.o
```

Dans le cas présent comme nous n'avons qu'un seul fichier objet, celui-ci doit donc contenir une méthode `main`.

4.4.2 Edition de liens avec plusieurs fichiers

Dans les études de cas qui nous intéresserons plus tard nous considérerons une fonction de référence écrite en C et en donnerons plusieurs implantations en assembleur en utilisant les instructions vectorielles par exemple. Toute la machinerie de test sera écrite en C++ dont notamment la récupération et la vérification des paramètres ainsi que l'allocation et l'initialisation des données et leur libération. Les fonctions optimisées seront écrites en assembleur et il faudra regrouper des fichiers objets compilés avec le compilateur C ou l'assembleur.

Pour définir et pouvoir appeler dans un fichier C ou C++ une fonction écrite dans un fichier assembleur il suffit de la déclarer *externe* au fichier C. Cependant la déclaration varie suivant que l'on est dans un fichier C ou un fichier C++ :

```
1 // dans un fichier .c
2 extern int ma_fonction_assembleur(int *t, int size);
3
4 // dans un fichier .cpp
5 extern "C" {
6     int ma_fonction_assembleur(int *t, int size);
7 }
```

On compilera donc les fichiers d'extension `.c` ou `.cpp` séparément et on réalisera l'édition de liens avec l'ensemble des fichiers objets comme suit :

```
g++ -o mon_binaire.exe *.o [options de compilation C/C++]
```

4.4.3 Obtenir le code assembleur d'un fichier C

Il existe deux méthodes pour obtenir le code assembleur d'un fichier C/C++ :

- soit on dispose des sources en C/C++, et dans ce cas on utilise le compilateur C pour traduire le code en assembleur
- soit on dispose de l'exécutable et on peut utiliser l'utilitaire `objdump` pour désassembler le fichier et en obtenir le code

Prenons comme exemple de travail le Listing 4.4.1 qui consiste à afficher la somme des valeurs d'un tableau que l'on aura initialisé avec des valeurs aléatoires comprises entre 0 et 9.

```
#include <stdio.h>
#include <stdlib.h>

#define MAXIMUM 100
int tab[MAXIMUM];
int sum = -1;

int main() {
    int i;

    srand(1970);
    for (i = 0; i < MAXIMUM; ++i) tab[i] = rand() % 10;

    sum = 0;
    for (i = 0; i < MAXIMUM; ++i) sum += tab[i];

    printf("sum = %d\n", sum);
    return 0;
}
```

Listing 4.4.1 – Exemple de traduction

4.4.3.1 utiliser gcc -S

On utilise `gcc` avec l'option `-S` pour obtenir le code assembleur. Il faut également utiliser l'option `-masm=intel` afin de générer une sortie avec une syntaxe Intel que je trouve plus lisible que la syntaxe ATT :

```
gcc -S -masm=intel fichier.cpp
```

En sortie on obtient un fichier d'extension `.s` qui contient le code assembleur.

4.4.3.2 utiliser objdump

`objdump` est un utilitaire qui affiche les informations contenues dans les fichiers objet ou les exécutables :

```
objdump -d -r -l -S -M intel example.exe
```

Il existe de nombreuses options à passer à l'utilitaire `objdump` au format court ou long. Dans l'exemple précédent on a utilisé :

`-d` pour désassembler l'exécutable, c'est à dire en fournir la traduction assembleur
`-r` est utilisée pour la relocation des adresses
`-l` affiche les numéros de lignes
`-S` affiche le code source si disponible
`-M intel` utilisation de la syntaxe Intel

On pourra également consulter le site web `godbolt.org` qui permet de soumettre du code C/C++ et d'obtenir la tradution avec différents compilateurs (GNU, Intel, LLVM, etc.) ainsi que différentes versions de ces compilateurs.

4.5 Le débogueur ddd

`ddd` (*Data Display Debugger*) est une interface graphique qui se base sur le débogueur `gdb` (*GNU debugger*). Elle est plus sympatique que `xxgdb` mais reste néanmoins très basique et pas très ergonomique. On regrettera notamment le fait que la fenêtre de commandes disparaît par moment, que l'affichage du contenu des registres est peu convivial et parfois difficile à lire pour les registres vectoriels SSE et AVX.

On peut bien entendu utiliser `gdb` mais cela implique de connaître les commandes de ce dernier et travailler en mode texte. Il existe également un certains nombre d'assistants qui peuvent être chargés lors de l'initialisation de `gdb` mais ils sont généralement en mode texte et pas très conviviaux :

- PEDA (Python Exploit Development Assistance) : `https://github.com/longld/peda`
- PWNDBG : `https://github.com/pwndbg/pwndb`
- Voltron : `https://github.com/snare/voltron`

On pourra consulter le site BlackArch [6] pour découvrir de nombreux autres outils du même type. On pourra également consulter la section désassembleur (*disassembler*) qui est intéressante.

4.6 Logiciels annexes

Comme évoqué précédemment, lors des études de cas nous comparerons diverses implantations d'une fonction de référence et nous génèrerons des données

6. `https://blackarch.org/debugger.html`

relatives au temps d'exécution de ces variantes de la fonction initiale ainsi que des graphiques.

Pour traiter les résultats, j'utilise le langage PHP (parfois Python) ainsi que des commandes shell telles que `cut`, `sort`, `grep`. En ce qui concerne les graphiques nous utiliserons `gnuplot` qui est un générateur de graphiques en deux ou trois dimensions à partir de données brutes ou de fonctions. `gnuplot` est gratuit et est disponible pour un grand nombre de plateformes dont les principales que sont Linux, Windows et MacOS. L'intérêt de `gnuplot` et qu'il peut être utilisé de manière interactive, mais en ce qui nous concerne c'est la possibilité de générer des graphiques à partir de scripts qui sera notre point d'intérêt.

Chapitre 5

Traitements de base

If you just spend nearly 30 hours
debugging some assembly
soon you will be glad to
Write in C

Write in C
sur la musique de Let it be (The Beatles)

5.1 Introduction

Dans ce chapitre nous allons découvrir et nous familiariser avec les instructions de base de l'assembleur x86. Avant de nous diriger dans les chapitres suivants vers la découverte des instructions du coprocesseur arithmétique et celles des unités vectorielles. Nous définissons de manière plus détaillée les registres des processeurs d'architecture x86 puisque les instructions agissent sur ces registres. Nous présentons les instructions les plus souvent utilisées car la connaissance de l'ensemble des instructions assembleurs et tout ce qui s'y rattache représente 10 volumes de documentation Intel [7, 8, 9, 10, 11, 12, 13, 14, 15, 16], soit plus de `4800 pages`. Nous verrons également comment traduire les structures de contrôle du langage C comme les conditionnelles (*if, switch*) et les boucles (*for, while*).

5.2 Registres

Nous avons déjà évoqué les registres dans le Chapitre 3. Nous allons revoir, en fonction de l'architecture (16, 32 ou 64 bits), quels registres sont disponibles.

5.2.1 Registres 8 et 16 bits

L'Intel 8086 disposait de 8 registres 16 bits dont certains sont manipulables en deux fois 8 bits dits partie haute et partie basse. Ces registres d'usage général (*General Purpose*), comme évoqué antérieurement, sont au nombre de 8 (cf. Section 3.3.4).

Registre 16 bits	Partie Haute bits 15 à 8	Partie Basse bits 7 à 0	Utilisation
`ax`	`ah`	`al`	accumulateur, multiplication, division
`bx`	`bh`	`bl`	accés mémoire
`cx`	`ch`	`cl`	compteur, repétition (`rep`), décalage
`dx`	`dh`	`dl`	`in`, `out`, multiplication, division
`si`	-	-	source index, `lods`, `movs`
`di`	-	-	destination index, `stos`, `movs`
`bp`	-	-	base pointer, pile
`sp`	-	-	stack pointer, sommet de pile

TABLE 5.1 – Registres 16 bits du Intel 8086

Ils sont décrits Table 5.1. Par exemple le registre `ax` est appelé **accumulateur** et permet de réaliser certaines opérations comme la multiplication, la division mais reçoit également le résultat des instructions comme `lodsb`, `lodsw`, `lodsd` (cf. Section 5.3.7). On est en mesure de manipuler la partie haute nommée `ah` ou la partie basse `al`.

Les autres registres (qui ne sont pas d'usage général) qui sont également à prendre en considération sont : le registre `ip` qui est le pointeur d'instruction (*Instruction Pointer*), c'est à dire l'adresse de la prochaine instruction en mémoire, et le registre `flags`. Ce dernier est mis à jour après exécution de la plupart des instructions et donne des informations sur le résultat obtenu. Il n'est pas accessible directement (sauf si on utilise l'instruction `lahf`) mais au travers d'instructions de calcul ou de branchement conditionnel. Chacun de ses bits contient une information comme par exemple :

- le bit 0, CF (*Carry Flag*) le bit de retenue
- le bit 2, PF (*Parity Flag*) indique un nombre pair de bits à 1
- le bit 4, AF (*Adjust Flag*)
- le bit 6, ZF (*Zero Flag*) le bit de zéro, indique une valeur nulle
- le bit 7, SF (*Sign Flag*) le bit de signe indique une valeur négative
- le bit 10, DF (*Direction Flag*) utilisé avec les instructions agissant sur les chaînes comme `lods`, `stos`, etc

- le bit 11, OF (*Overflow Flag*) le bit de débordement pour les calculs signés

Par exemple, le *Carry Flag* est utilisé lors de l'addition, notamment pour signaler qu'une retenue a été produite lors du calcul. Le bit de débordement (*Overflow*) indique qu'une opération arithmétique a produit un résultat signé invalide.

Par exemple si **al** et **bl** contiennent la valeur 128, l'addition de ces deux registres :

- met à 1 le *Carry Flag* puisque le résultat devrait être 256, mais que cette valeur ne peut pas être représentée sur 8 bits
- met à 1 le l'*Overflow Flag* car si on considère que les données sont signées, l'addition de deux valeurs négatives ($-128 + -128$) donne une valeur positive (0), or on devrait obtenir une valeur négative

Si au contraire on utilise les registres **ax** et **bx** alors CF et OF restent à 0 puisque dans les deux cas (signé ou non signé) les valeurs sont positives, on effectue la somme $128 + 128$.

Il existe d'autres cas de positionnement de l'*Overflow Flag* : lors de l'addition de deux valeurs positives (qui donne un résultat négatif) et lors de la soustraction :

- soutraction entre une valeur négative et un nombre positif qui donne un résultat positif alors que le résultat devrait être négatif : $-128 - 1$
- soutraction entre une valeur positive et un nombre négatif qui donne un résultat négatif alors que le résultat devrait être positif : $127 - (-1)$

Nous avons en outre évoqué, Chapitre 3, les registres de segments qui contiennent une adresse mémoire qui indique le début du code (**cs**), des données (**ds**), de la pile (**ss**) et un registre de segment auxiliaire (**es** pour *Extra Segment*).

5.2.2 Architecture et registres 32 bits

En architecture 32 bits, les registres généraux ont une taille de 32 bits et les registres existants du 8086 sont toujours utilisables mais ont été étendus et se nomment **eax**, **ebx**, ... **esp**, **eip**, **eflags**(voir Table 5.2). On peut alors stocker 2^{32} valeurs différentes dans un registre 32 bits ce qui correspond à autant d'adresses mémoires et à un total de 4 Go. On peut toujours utiliser les registres 16 et 8 bits comme **ax**, **ah** ou **al**.

Les registres **eip** et **eflags** sont les extensions de **ip** et **flags** et les registres de segment sont les mêmes. On a cependant ajouté deux autres registres de segment appelés **fs** et **gs**.

5.2.3 Architecture et registres 64 bits

L'architecture 64 bits apportent plusieurs changements importants. Les registres ont bien entendu une taille de 64 bits et sont appelés **rax**, **rbx**, ... **rsp**, **rip**, **rflags**.

Registre 32 bits	Partie Basse 16 bits	Partie Haute 8 bits	Partie Basse 8 bits
eax	ax	ah	al
ebx	bx	bh	bl
ecx	cx	ch	cl
edx	dx	dh	dl
esi	si	-	-
edi	di	-	-
ebp	bp	-	-
esp	sp	-	-

TABLE 5.2 – Registres manipulables en architecture 32 bits

On dispose également de 8 nouveaux registres nommés `r8` à `r15` ce qui laisse une plus grande marge de manoeuvre pour la programmation en permettant de stocker des valeurs temporaires dans ces registres plutôt que de les stocker en mémoire.

Registre 64 bits	Partie Basse 32 bits	Partie Basse 16 bits	Partie Haute 8 bits	Partie Basse 8 bits
rax	eax	ax	ah	al
rbx	ebx	bx	bh	bl
rcx	ecx	cx	ch	cl
rdx	edx	dx	dh	dl
rsi	esi	si	-	sil
rdi	edi	di	-	dil
rbp	ebp	bp	-	bpl
rsp	esp	sp	-	spl
r8	r8d	r8w	-	r8b
...	...	...	...	
r15	r15d	r15w	-	r15b

TABLE 5.3 – Registres manipulables en architecture 64 bits

La Table 5.3 recense les différentes possibilités de manipulation des registres en 64 bits. On notera que l'on peut manipuler les premiers 8 bits de certains registres comme `rsi` alors que cela n'est pas possible en 32 bits, en effet, en architecture 32 bits on ne pourra manipuler que `esi` ou `si`.

5.2.4 Architecture 128 bits

Le passage des microprocesseurs 8 bits à des microprocesseurs 16 bits a permis de gérer une quantité de mémoire plus importante mais également de gérer des nombres plus grands en une seule opération. Le bus de données qui représente les quantités sur lesquelles on réalise des calculs, et le bus mémoire qui représente la taille de l'espace mémoire, sont corrélés puisqu'on stocke dans un registre, soit une adresse mémoire, soit un entier.

Il en fut de même lors du passage de 16 bits au 32 bits, puis du 32 bits au 64 bits. Ainsi, avec 32 bits on gère 4×10^9 valeurs alors qu'avec 64 bits on atteint un ordre de grandeur de 18×10^{18}, soit $18Eo$ (Exa octets).

La question que l'on peut se poser est la suivante, pourquoi ne passe t'on pas à des microprocesseurs 128 bits ce qui permetrait de gérer 34010^{36} valeurs. Simplement parce que seules quelques rares applications ont besoin de gérer des quantitiés si importantes et que globalement nous n'en avons pas l'utilité. Par exemple 1810^{18} représente environ 570 milliards d'années. D'un autre côté, si on rapporte ce temps à des nano secondes cela représente 570 ans.

5.3 Instructions élémentaires

La grande majorité des instructions x86 sont de la forme :

operation *destination*, *source*

- **operation** est un mnémonique, c'est à dire un symbole court de quelques lettres facilement compréhensible et mémorisable qui représente l'opération à exécuter, par exemple `add` pour l'addition
- *source* est une donnée en lecture qui ne sera donc pas modifiée, ce peut être une constante, un registre ou une adresse mémoire
- *destination* est une donnée en écriture qui peut être un registre ou une adresse mémoire
- les deux opérandes *source* et *destination* sont séparées par une virgule

En fait la syntaxe de l'assembleur permet d'écrire plus simplement le traitement qui est effectivement réalisé, à savoir :

$$\text{destination} = \text{destination} \quad \text{operation} \quad \text{source}$$

Ainsi, `add eax, ebx` signifie que l'on doit réaliser le calcul `eax` = `eax` + `ebx`.

Nous verrons qu'il existe d'autres variantes de format d'instruction comme pour les instructions `neg`, `not`, `cmp`, `test`, `div`, `mul`, etc.

Il existe toutefois une contrainte imposée par le format de codage des instructions x86 qui nous force à n'avoir qu'une seule référence mémoire (cf. Section 3.3.5), cela implique que l'on ne peut pas écrire :

```
1    operation   [adresse1], [adresse2]     ; !!! non autorisé !!!
```

Il faudra alors passer par un registre et écrire :

```
1    mov         registre, [adresse2]
2    operation   [adresse1], registre
```

L'instruction `mov` dont nous allons reparler ci-après déplace la donnée située à l'adresse `adresse2` dans un registre.

Référence mémoire

On remarque également que lorsque l'on fait référence à une donnée en mémoire identifiée par son adresse, on place l'adresse entre crochets `[]`. Ainsi :

- `mov eax, [addr]` signifie placer la valeur 32 bits située à l'adresse `addr` dans le registre `eax`
- `mov eax, val` signifie placer la valeur constante codée sur 32 bits `val` dans le registre `eax`

5.3.1 `mov` : chargement et stockage

L'instruction `mov` réalise le chargement (*Load*) et le stockage (*Store*) des données. C'est l'instruction qui est la plus souvent utilisée. Le fait de déplacer une donnée n'influe par sur le registre `flags`. Il faudra donc utiliser une instruction de comparaison pour vérifier si la donnée chargée dans un registre est nulle, postive ou négative.

Voici quelques exemples liés à l'instruction `mov` qui permettent de comprendre comment l'utiliser :

- `mov eax, 0` : affecter la valeur 0 au registre `eax`
- `mov eax, ebx` : affecter le contenu de `ebx` au registre `eax`
- `mov al, bh` : affecter le contenu de `bh` au registre `al`
- `mov eax, [ebx + ecx * 4]` : affecter au registre `eax` la valeur située à l'adresse mémoire indiquée, il s'agit d'une référence mémoire (comme vu Section 3.3.5) comme `eax` est un registre 32 bits on lit le double mot situé à l'adresse indiquée
- `mov [edi + esi], edx` : stocker à l'adresse `edi + esi` la valeur contenue dans le registre `edx`

On trouve deux variantes de l'instruction `mov` :

- `movsx` (*Mov with Sign eXtension*) qui transforme une valeur sur 8 (respectivement 16 bits) en une valeur 16 (respectivement 32 bits) en préservant le fait que la valeur soit négative ou positive
- `movzx` (*Mov with Zero eXtend*) qui transforme une valeur sur 8 ou 16 bits en une valeur 16, 32 ou 64 bits en la complétant avec des 0

L'instruction `movzx` est parfois plus rapide que mov[1]. Il est donc préférable d'écrire, afin de lire l'octect à l'adresse `edi` en mémoire et le stocker dans le registre `al` :

```
1     movzx eax, byte [edi]
```

plutôt que :

```
1     mov   al, [edi]
```

La différence est que `movzx` va modifier `eax` en mettant dans `al` l'octet pointé par `edi` et en mettant à 0 les 24 autres bits. On remarque que dans ce cas il faut préciser la quantité chargée : `byte` pour un octet, `word` pour un mot et dans d'autres instructions `dword` pour un double mot.

5.3.2 Instructions arithmétiques

5.3.2.1 Instructions `add`, `sub`, `inc` et `dec`

Les instructions `add` et `sub` réalisent respectivement l'addition et la soustraction de deux valeurs entières signées ou non signées et prennent deux opérandes.

Les instructions `inc` et `dec` réalisent respectivement l'incrémentation et la décrémentation de leur unique opérande. On peut les utiliser pour réaliser les opérations du C comme `++i` qui correspond à l'incrémentation d'une variable de boucle `for`.

Les deux instructions suivantes sont donc équivalentes :

```
1     add   eax, 1
2     inc   eax
```

Il existe également une instruction `adc` (*ADd with Carry*) qui réalise une addition avec une retenue en entrée. Par exemple, imaginons que l'on travaille avec les registres 16 bits de l'Intel 8086 et que l'on désire réaliser le calcul $a + b$ pour $a = 196607$ et $b = 262145$, soit en hexadécimal $a =$ `2FFFF_h` et $b =$ `40001_h`. Ces deux valeurs sont supérieures à $2^{16} - 1 = 65535$, pour les stocker on va donc devoir utiliser deux registres de 16 bits :

1. voir *Intel 64 and IA-32 Architectures Optimization Reference Manual*, section 3.5.1.8.

- pour a on utilisera par exemple **dx:ax**, c'est à dire la partie haute dans **dx** et la partie basse dans **ax**
- pour b on utilisera **cx:bx**

```
    mov   ax, 0xFFFF
    mov   dx, 0x2          ; dx:ax = 0x2FFFF
    mov   bx, 0x0001
    mov   cx, 0x4          ; cx:bx = 0x40001
```

Si on réalise le calcul suivant ($a = a + b$) en écrivant :

```
    add   ax, bx
    add   dx, cx      ; ne prend pas en compte la retenue !
```

On obtient un résultat faux car alors **dx:ax** $=$ `60000_h` $= 393216_{10}$. Cela est dû au fait que la première addition génère une retenue de 1 qu'il faut utiliser lors de la deuxième addition. On doit donc écrire :

```
    add   ax, bx
    adc   dx, cx      ; prend en compte la retenue
```

Afin d'obtenir le résultat correct qui est `dx:ax` $=$ `70000_h` $= 458752_{10}$. On trouve également l'instruction **sbb** (*SuBstract with Borrow*) pour faire des soustractions si on utilise deux registres 16 ou 32 bits.

Concernant les instructions **inc** et **dec**, elles modifient les flags OF, SF, ZF, AF et PF, mais pas le Carry Flag. Il est de plus conseillé de ne pas les utiliser car elle peuvent produire dans certaines situations des *false dependencies* et des *partial flag register stall*.

Un exemple concret est celui du Chapitre 15 pour lequel on compte des voyelles en mode 64 bits. Le code suivant :

```
.while:
    movzx   eax, byte [rdi + rcx]   ; s[i]
    sub     eax, 'a'                ; s[i] - 'a'
    inc     dword [rbx + rax * 4]   ;++letters[ s[i]-'a' ]
    inc     ecx                     ; ++i

    cmp     ecx, esi                ; if (i < size)

    jne     .while                  ;    goto .while
```

est susceptible dans certains cas et sous certaines architectures de produire de graves ralentissements passant d'un code qui s'exécutait en 6 secondes à un temps d'exécution de 29 secondes, mais environ 6 à 7 fois sur 10 exécutions, ce qui semble totalement aberrant ! On devrait en effet obtenir toujours le même effet, mais ce n'est pas le cas. Si on remplace l'instruction **inc** par un **add** le problème disparaît.

5.3.2.2 L'instruction `mul`

L'instruction `mul` n'accepte qu'une seule opérande source et réalise la multiplication non signée entre un registre 8, 16 ou 32 bits et respectivement `al`, `ax`, `eax` comme indiqué Table 5.4. Les notations `reg8`, `reg16`, `reg32` signifient respectivement un registre général 8, 16 ou 32 bits.

Opération	Source	Résultat
`mul reg8`	`al`	`ax`
`mul reg16`	`ax`	`dx:ax`
`mul reg32`	`eax`	`edx:eax`

TABLE 5.4 – Modes d'utilisation de `mul`

Par exemple, si on écrit `mul bh`, c'est le registre `al` qui est multiplié par `bh` et le résultat est placé dans `ax`.

En architecture 32 bits, on notera qu'avec une opérande source de 16 (resp. 32 bits), le registre `dx` (resp. `edx`) est modifié. Il ne faudra donc pas stocker de donnée dans `edx`, ou alors, sauvergarder cette donnée avant la multiplication en la plaçant dans la pile, puis après la multiplication, la récupérer depuis la pile.

Pour calculer 7×5 ou 5×7, on écrira donc :

```
1     push  edx       ; on sauvegarde edx
2     mov   eax, 5
3     mov   ebx, 7
4     mul   ebx      ; edx:eax= 0:35
5     pop   edx      ; on restaure edx
```

Il existe également une autre instruction appelée `imul` (voir ci-après) qui réalise une multiplication signée et peut prendre trois formes en fonction du fait qu'elle utilise une, deux ou trois opérandes [8].

5.3.2.3 L'instruction `div` et le modulo

L'instruction `div` réalise la division entière **non signée** entre une valeur 64, 32 ou 16 bits par un diviseur sur 32, 16 ou 8 bits respectivement, le reste de la division est également calculé. L'instruction `div` permet donc également de réaliser le modulo (voir Table 5.5).

Par exemple en architecture 32 bits, c'est en fait une valeur sur 64 bits contenue dans deux registres 32 bits `edx:eax` que l'on divise par une opérande 32 bits contenue dans un autre registre. Si on désire travailler avec des valeurs 32 bits, il faut mettre `edx` à 0 avant de faire la division. Pour diviser 1024 par 3, on écrira donc :

Dividende	Diviseur	Quotient	Reste
edx:eax	div reg32	eax	edx
eax	div reg16	ax	dx
ax	div reg8	al	ah

TABLE 5.5 – Comportement de l'instruction `div`

```
1     mov   eax, 1024
2     xor   edx, edx      ; mise à zéro de edx pour rester
3                         ; en 32 bits
4     mov   ebx, 3
5     div   ebx
```

Le registre `eax` contiendra alors la valeur 341 et `edx` sera égal à 1 car $1024 = 3 \times 341 + 1$.

Attention, si le résultat de la division de `edx:eax` par un autre registre 32 bits donne un résultat plus grand que la valeur hexadécimale `0xFF_FF_FF_FF`, le microprocesseur lève une exception. C'est pour cela qu'il est conseillé de mettre `edx` à 0 avant de faire le calcul. Par exemple si `eax` est égal à 6 et que l'on divise par `ebx` qui vaut 2, mais que `edx` contient 8, c'est la valeur `8_00_00_00_06` qui est divisée par 2, ce qui donne `4_00_00_00_03` qui est supérieure à `FF_FF_FF_FF`.

En outre, puisqu'il s'agit d'une division non signée, si on réalise le calcul suivant :

```
1     mov   eax, -6       ; eax = FF_FF_FF_FA_h
2     xor   edx, edx      ; mise à zéro de edx pour rester
3                         ; en 32 bits
4     mov   ebx, 3        ;
5     div   ebx           ; eax = 55_55_55_53_h = 1_431_655_763
6                         ; edx = 0
```

On n'obtiendra pas -2 (`FF_FF_FF_FE`) dans `eax` mais `55_55_55_53`. Pour réaliser une division signée, il faut utiliser l'instruction `idiv` (*Integer Division*) mais pour cela il faut mettre `edx` à -1, sinon le résultat du calcul sera le même que précédemment. En effet, c'est `edx:eax` que l'on divise par `ebx`, il faut donc coder -2 sur 64 bits :

```
1     mov   eax, -6       ; eax = FF_FF_FF_FA_h
2     xor   edx, edx      ; mise à -1 de edx pour réaliser
3     dec   edx           ; -6 (FF_FF_FF_FF_FF_FF_FF_FA) / 3
4     mov   ebx, 3        ;
5     div   ebx           ; eax = FF_FF_FF_F2_h = -2 (quotient)
6                         ; edx = 0 (reste)
```

ou alors, on utilisera l'instruction `cdq` qui réalise cette conversion :

```
1    mov   eax, -6          ; eax = FF_FF_FF_FA_h
2    cdq                  ; edx = FF_FF_FF_FF, eax = FF_FF_FF_FA
3    mov   ebx, 3         ;
4    div   ebx            ; eax = FF_FF_FF_F2_h = -2 (quotient)
5                         ; edx = 0 (reste)
```

5.3.2.4 L'instruction `imul`

Elle réalise la multiplication de valeurs signées et possède trois formats. Voici par exemple avec des opérandes 32 bits les formalismes possibles :

```
1    imul  ecx            ; edx:eax = eax * ecx (comme mul)
2    imul  ebx, ecx       ; ebx = ebx * ecx (edx pas modifié)
3    imul  ebx, 3         ; ebx = ebx * 3 (edx pas modifié)
```

5.3.2.5 L'instruction `idiv`

Elle réalise la division de valeurs signées et possède le même format que `div`. Par exemple, pour diviser 23 par -7 :

```
1    xor   edx, edx
2    mov   eax, 23
3    mov   ecx, -7
4    idiv  ecx
```

On obtient alors -3 dans `eax` et 2 dans `edx`.

5.3.2.6 L'instruction `neg`

L'instruction `neg` réalise **le complément à deux** (*Two's Complement Negation*). Si `eax` contient la valeur -1 alors `neg eax` produira la valeur 1 dans `eax` et inversement. Le *Sign Flag* du registre `flags` sera positionné en conséquence.

Attention cette instruction met le drapeau CF (*carry flag*) à 0 si la valeur initiale est 0, sinon le drapeau CF sera positionné à 1, on calcule ensuite le complément à deux et on ajuste les autres flags en conséquence. On pourra par exemple utiliser l'information sur le drapeau CF pour implanter en 4 instructions la fonction *zero_un_moins_un* (cf. Exercice 5.6).

5.3.2.7 L'instruction `lea`

L'instruction `lea` (*Load Effective Address*) est intéressante car elle réalise une multiplication et une addition. Il ne faut pas se laisser leurrer par son formalisme qui utilise la représentation sous forme de référence mémoire. Ainsi :

```
1     lea eax, [ebx + ecx * 4 + 9]
```

signifie que l'on calcule l'expression `ebx + ecx * 4 + 9` et qu'on affecte le résultat au registre **eax**. **Il n'y a pas d'accès à la mémoire !**

En particulier, si on désire multiplier le registre **eax** par 5, plutôt que d'utiliser une multiplication coûteuse en nombre de cycles on utilisera :

```
1     lea eax, [eax + eax * 4]
```

Comme pour les références à la mémoire le facteur d'échelle qui multiplie le registre **eax** dans l'instruction précédente ne peut être égal qu'à 1, 2, 4 ou 8 (cf. Section 3.3.5).

5.3.3 Instructions logiques

Les instructions logiques (ou binaires, puisqu'elles s'appliquent sur l'ensemble des bits de leurs opérandes) permettent de réaliser bon nombre d'opérations basées sur l'utilisation de masques.

x	y	and(x,y)	or(x,y)	xor(x,y)
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

TABLE 5.6 – Table de vérité pour les fonctions `and`, `or`, `xor`

Nous rappelons Table 5.6, les tables de vérité des opérateurs `and`, `or` et `xor`.

5.3.3.1 Instructions `and` et `or`

L'instruction **and** permet de sélectionner des bits spécifiques d'un registre alors que l'instruction **or** permet de positionner certains bits à 1. Ainsi, le code suivant permet de ne garder la valeur que des 3 bits de poids faible du registre **eax** et de positionner le bit de poids fort à 1.

```
1     and   eax, 0x07         ; sélectionne les 3 bits de poids faible
2     or    eax, 0x80000000   ; fixe le bit de poids fort à 1
```

5.3.3.2 L'instruction `xor`

L'instruction `xor` réalise le *ou exclusif*. Cette instruction est également utilisée sous la forme `xor reg, reg` ou `reg` est l'un des registres généraux. C'est un autre moyen de mettre la valeur 0 dans le registre `reg` car d'après la table de vérité du `xor`, un *ou exclusif* entre deux valeurs identiques donne 0.

```
1     xor   al, al    ; mise à zéro de al
2     xor   ecx, ecx  ; mise à zéro de ecx
```

L'instruction `xor` permet également de modifier un bit en le positionnant à 1 s'il était préalablement à 0 et inversement.

5.3.3.3 L'instruction `not`

Elle réalise le complément, c'est à dire qu'on change les bits à 1 en les positionnant à 0 et inversement. Avec `nasm`, il existe l'opérateur `~` qui réalise le `not`, on peut donc écrire :

```
1     mov    eax, ~7
```

au lieu de :

```
1     mov    eax, 7
2     not    eax
```

5.3.4 Instructions de décalage

5.3.4.1 Instructions `shl`, `shr`

Les instructions `shl` (*SHift Left*) et `shr` (*SHift Right*) réalisent des décalages respectivement à gauche et à droite de l'opérande destination, la source étant une constante ou le registre `cl` qui indique de combien de rangs on réalise le décalage.

Décaler un registre de n rangs vers la gauche consiste à faire une multiplication entière par 2^n. De la même manière, si on décale à droite, on réalise une division entière par 2^n :

```
1     mov   eax, 17
2     shl   eax, 5    ; eax = 17 * 2^5 = 544
3     shr   eax, 7    ; eax = 544 / 2^7 = 4
```

5.3.4.2 L'instruction sar

L'instruction sar (*Shift Arithmetic Right*) permet de réaliser une division par une puissance de 2 tout en préservant le signe de la valeur divisée. Comme on peut le voir sur l'exemple qui suit, shr considère -128 comme une valeur non signée et ne préservera pas son signe, par contre, sar le fera :

```
1     mov   eax, -128 ; eax = 0xFFFFFF80
2     shr   eax, 2    ; eax = 0x3FFFFFE0 = 1073741792
3     mov   eax, -128 ; eax = 0xFFFFFF80
4     sar   eax, 2    ; eax = 0xFFFFFFE0 = -32
```

Il existe également d'autres opérations de décalage comme rcl (*Rotate Carry Left*) et rcr (*Rotate Carry Right*), rol (*ROtate Left*), ror (*ROtate Right*). Pour de plus amples explications sur ces instructions nous conseillons la lecture de [19].

5.3.5 Comparaison

5.3.5.1 L'instruction cmp

L'instruction cmp (*CoMPare*) réalise la comparaison non signée de deux opérandes en calculant leur différence. Les opérandes ne seront pas modifiées seule la différence sera utilisée pour mettre à jour le registre des flags. Voici quelques utilisations de cette instruction :

- cmp eax, 10 : compare eax à la constante 10
- cmp eax, edx : compare eax à edx
- cmp ecx, [ebp-12] : compare ecx à l'entier 32 bits situé en mémoire à l'adresse [ss:ebp-12]

La Table 5.7 montre comment sont modifiés les principaux bits du registre flags lors de la comparaison en fonction des opérandes de la commande cmp.

cmp eax, ebx	CF	SF	ZF
eax == ebx	0	0	1
eax < ebx	1	1	0
eax > ebx	0	0	0

TABLE 5.7 – Influence sur les bits du registre flags de la comparaison de valeurs

5.3.5.2 L'instruction `test`

L'instruction `test` permet également de comparer deux valeurs mais en réalisant un `and` entre les deux opérandes qui ne seront pas modifiées et dont le résultat sera propagé au niveau du registre `flags`. En général cette instruction est utilisée pour vérifier qu'une valeur est nulle ou non ou qu'un bit est positionné à 1. En réalisant par exemple un `test eax, eax`, on vérifie si `eax` est nul plutôt que d'écrire `cmp eax, 0` qui prend plus de place en mémoire puisqu'on code l'opérande 0.

```
1     test   eax, eax     ; si eax == 0 alors aller en .end
2     jz     .end         ;
3
4     test   eax, 1       ; si le bit de poids faible n'est pas
5     jz     .pair        ; à 1, alors il s'agit d'un nombre pair
```

5.3.6 Instructions de branchement

Ce que nous appelons instructions de branchement correspond aux instructions qui modifient le pointeur d'instructions. On distingue les instructions de branchement conditionnel (qui sont liées à une comparaison préalable) des instructions non conditionnelles.

5.3.6.1 Instructions de branchement conditionnel

Ces instructions sont utilisées après une comparaison (`cmp`, `test`) ou un calcul (`add`, `sub`, `mul`, etc) afin de pouvoir modifier le pointeur d'instruction et exécuter une instruction qui n'est pas directement la suivante. Elles sont de la forme `jCC adresse` où `CC` est remplacé par la condition de branchement (cf. Table 5.8).

La documentation Intel indique que les termes *less* et *greater* sont utilisés pour des comparaisons entre des entiers signés alors que les termes *below* et *above* sont utilisés pour des entiers non signés.

Note : il existe également des équivalents à certains instructions comme `jnae` (*Jump on Not Above or Equal*) qui est équivalent à `jb`.

En prévision de ce que nous verrons plus tard au sujet du coprocesseur, nous indiquons que les instructions de saut de type *below* et *above* sont utilisées lors de la comparaison de valeurs flottantes.

5.3.6.2 Loop

Il existe également une instruction spécifique `loop address` qui décrémente le registre `ecx` et, si celui-ci n'est pas égal à 0, se branche à l'adresse indiquée. Elle est donc équivalente aux deux instructions suivantes :

Instruction	Signification	Condition
`jl`	jump on less	SF $\neq$ OF
`jle`	jump on less or equal	ZF = 1 ou SF $\neq$ OF
`jg`	jump on greater	ZF = 0 et SF = OF
`jge`	jump on greater or equal	SF = OF
`je`	jump on equal	ZF = 1
`jne`	jump on not equal	ZF = 0
`jz`	jump on zero	ZF = 1
`jnz`	jump on not zero	ZF = 0
`jb`	jump on below	CF = 1
`jbe`	jump on below or equal	CF = 1 ou ZF = 1
`ja`	jump on above	CF = 0 et ZF = 0
`jae`	jump on above or equal	CF = 0
`jcxz`	jump on `cx` equals zero	`CX` = 0
`jecxz`	jump on `ecx` equals 0	`ECX` = 0
`jrcxz`	jump on `rcx` equals 0	`RCX` = 0
`jo`	jump on overflow	OF = 1
`jno`	jump on notoverflow	OF = 0
`jp`	jump on parity	PF = 1
`jnp`	jump on not parity	PF = 0
`js`	jump on sign	SF = 1
`jns`	jump on not sign	SF = 0

TABLE 5.8 – Instructions de branchement conditionnel et FLAGS affectés

```
1 .begin:
2     ...
3     dec   ecx     ; à remplacer par
4     jnz   .begin  ; loop .begin
```

5.3.6.3 Autres instructions de branchement

Il s'agit des instructions de branchement comme :

- `jmp address` : modifie (*JuMP*) le pointer d'instruction pour qu'il soit égal à l'opérande `address`
- `call address` : réalise un appel de sous-programme (cf. Chapitre 6)
- `ret` : réalise le retour de sous-programme

L'instruction `call` empile l'adresse de l'instruction qui lui succède puis modifie le registre `eip` pour qu'il soit égal à `address`. L'instruction `ret`, utilisée lors du

retour de sous-programme, dépile l'adresse en sommet de pile (placée par `call`) et l'affecte à `eip`.

5.3.7 Instructions complexes

Ces instructions permettent de réaliser des traitements complexes et remplacent la combinaison de plusieurs instructions que nous venons d'évoquer. Elles sont généralement combinées à un **préfixe** comme `rep` pour *REPeat* qui utilise le registre `ecx` pour **indiquer le nombre de répétitions à exécuter**. Il existe également les préfixes `repe` (*REpeat while Equal*) `repne` (*REpeat while Not Equal*), `repz` (*REpeat while Zero*), `repnz` (*REpeat while Not Zero*) qui peuvent être utilisés avec les instructions `cmps` qui permet de comparer deux tableaux et `scas` qui permet de rechercher une valeur dans un tableau.

5.3.7.1 Lecture d'un tableau

`lods(b/w/d)` (*LOaD String of Byte/Word/Double word*) permet de parcourir un tableau en le lisant sous forme d'octets, de mots ou de double mots, les valeurs étant lues depuis `ds:esi` et stockées respectivement dans `al`, `ax`, `eax`.

5.3.7.2 Ecriture d'un tableau

`stos(b/w/d)` (*STOre String of Byte/Word/Double word*) permet d'écrire la même valeur dans un tableau en écrivant sous forme d'octets, de mots ou de double mots, les valeurs étant écrites vers `ds:edi` et lues depuis respectivement `al`, `ax`, `eax`.

5.3.7.3 Déplacement d'un tableau

`movs(b/w/d)` (*MOV String of Byte/Word/Double word*) permet de déplacer un tableau dont l'adresse est stockée dans `ds:esi` vers un tableau dont l'adresse est stockée dans `es:edi`. Il ne faut pas que les tableaux se chevauchent.

Ainsi `rep movsd` correspond à la série d'instructions suivantes :

```
1 .label:
2     mov     eax, [esi]
3     mov     [edi], eax
4     add     esi, 4
5     add     edi, 4
6     dec     ecx
7     jnz     .label
```

Les deux dernières instructions (lignes 6 et 7) peuvent être remplacées par l'instructions `loop .label` comme indiqué précédemment. Attention, après exécution de `rep movsd`, `ecx` est à 0 et `edi` et `esi` sont également modifiés.

5.3.7.4 rep ret

L'utilisation du préfixe `rep` devant une autre instruction n'est pas défini. Cependant, on trouve parfois dans la génération du code assembleur pour les processeurs AMD, la série d'instructions `rep ret`. Il s'agit d'un stratégème qui a été trouvé afin de remédier à un problème de prédiction de branchement lorsque l'instruction `ret` se trouve juste après une instruction de branchement conditionnel. On pourra consulter le site http://repzret.org/p/repzret/ pour de plus amples informations.

5.4 Traitements de base

5.4.1 Langage de GoTo

Nous avons déjà évoqué le fait que l'assembleur est un langage sans structures de contrôle que sont le `if`, le `while`. Le langage assembleur est rudimentaire et se fonde sur le déplacement du pointeur d'instruction en mémoire pour éviter d'exécuter le code d'un `if` ou revenir au début d'un `while`. Il s'agit d'un fonctionnement basé sur l'instruction `goto` que l'on trouve par exemple en langage BASIC.

En BASIC, chaque ligne d'un programme commence par un numéro qui permet de l'identifier. On commence généralement par 10, puis on incrémente de 10 à chaque nouvelle ligne, cela permet, au cas où on aurait oublié certaines instructions, d'en ajouter de nouvelles entre les lignes 11 à 19, 21 à 29, etc. L'instruction `goto` suivie d'un numéro de ligne permet de revenir à la ligne voulue.

On recommande fortement aux programmeurs de ne pas utiliser cette fameuse instruction `goto` dans des langages plus évolués comme le langage C (même si cette instruction est présente) car elle va à l'encontre d'un mode de programmation structuré. Néanmoins, cette instruction est implicitement utilisée, bien que cachée, par les mécanismes d'exception comme en C++, pour lesquels on appelle une fonction nommée `longjmp`.

L'exemple qui suit est un programme BASIC qui calcule la somme des entiers de 1 à 10 puis affiche le résultat.

```
10 i = 1: sum = 0
20 if i > 10 then goto 60
30 sum = sum + i
40 i = i + 1
50 goto 20
60 print "sum=", sum
```

Les programmes en assembleur vont donc suivre ce modèle de programmation. On peut d'ailleurs voir le langage BASIC comme une version plus évoluée de l'assembleur pour laquelle les variables et les entrées / sorties sont gérées de

manière à simplifier la tâche du programmeur.

5.4.2 Association variable registre

Dans la suite de ce chapitre et de l'ouvrage, nous allons traduire du code écrit en C vers l'assembleur. Pour optimiser le code il est nécessaire d'**utiliser le plus souvent possible des registres** car ils sont les plus rapides pour le traitement des données. La première étape préalable à la traduction est donc la réalisation de cette association. On pourra procéder en créant une table de correspondance (voir Table 5.9). Ainsi, pour le code suivant :

```
1 void init(int *tab, int N) {
2     for (int i = 0; i < N; ++i) {
3         tab[i] = 1;
4     }
5 }
```

On peut par exemple décider d'utiliser `ebx` pour stocker l'adresse du tableau `tab` et `ecx` pour réprésenter la variable de boucle `i`. La taille du tableau `N` pourra également être stockée dans un registre comme `edx` ainsi que la valeur 1 qui sera affectée à `tab[i]` et qui sera placée dans `eax`.

Variable/Cste	Type	Paramètre	Registre	Description
tab	int []	[ebp+8]	ebx	tableau de valeurs entières
N	int	[ebp+12]	edx	taille du tableau
i	int		ecx	variable de boucle
1	int		eax	constante 1

TABLE 5.9 – Association entre variables C et registres du microprocesseur en architecture 32 bits

Si on est en architecture 32 bits, les paramètres du sous-programme sont dans la pile (cf. Chapitre 6). Il faudra donc les placer dans des registres. C'est le cas pour les adresses de tableau, mais la longueur `N` peut être référencée depuis la pile.

5.4.3 Notion de label

Un label, également appelé étiquette, permet dans le code assembleur de définir l'adresse d'une instruction ou d'une donnée en utilisant un identifiant alphanumérique. On distingue :

- un label **global** qui commence par une lettre et suivi par des lettres, chiffres et le symbole souligné et permet de définir le nom d'un sous-programme ou le nom d'une variable

- un label **local** qui commence par un point (`.`) et qui indique une adresse de branchement à l'intérieur d'un sous-programme

L'intérêt des labels locaux est que leurs identifiants peuvent être réutilisés alors que les labels globaux sont uniques. Pour pouvoir réutiliser un label local il faut le faire précéder d'un label global. Nous verrons également plus avant dans ce chapitre un autre type de label utilisé par `nasm`.

La définition d'un label, qu'il soit local ou global, est réalisée en le suffixant par un caractère deux points (`:`) alors que lorsqu'on y fait référence ce symbole n'est pas utilisé :

```
1  fonction_1:                  ; définition de fonction_1
2          push    ebp
3          mov     ebp, esp
4  .if:    ...                  ; utilisation du label local if
5          mov     esp, ebp
6          pop     ebp
7          ret
8
9  fonction_2:                  ; définition d'une autre fonction
10         push    ebp
11         mov     esp, ebp
12         call    fonction1
13 .if:    ...                  ; réutilisation du label local if
14         ret
```

Dans l'exemple précédent le label local `.if` (défini en ligne 4) peut être réutilisé en ligne 13 car il est précédé en ligne 9 d'un label global (`fonction_2`).

5.4.4 Si alors

La conditionnelle *si alors* est réalisée en utilisant une comparaison `cmp` suivie d'une instruction de branchement conditionnel que nous notons `jCC`[2], comme présenté sur la Figure 5.1, où nous pouvons voir trois représentations du `if` :

- la première (en haut à droite) est la version en langage C que nous voulons traduire
- la seconde (en haut à gauche) est une vision graphique sous forme d'**organigramme**
- la troisième (en bas à gauche) est un code en BASIC

L'organigramme montre deux chemins d'exécution, celui de droite qui est emprunté lorsque la condition du `if` est *vraie* et qui consiste à exécuter le code du bloc *alors*. Le chemin de gauche est emprunté quand la condition est *fausse* et consiste à se placer après le code du *alors* en *fin_si*.

2. Comme nous ne connaissons pas la condition celle-ci est représentée de manière générique par un double C.

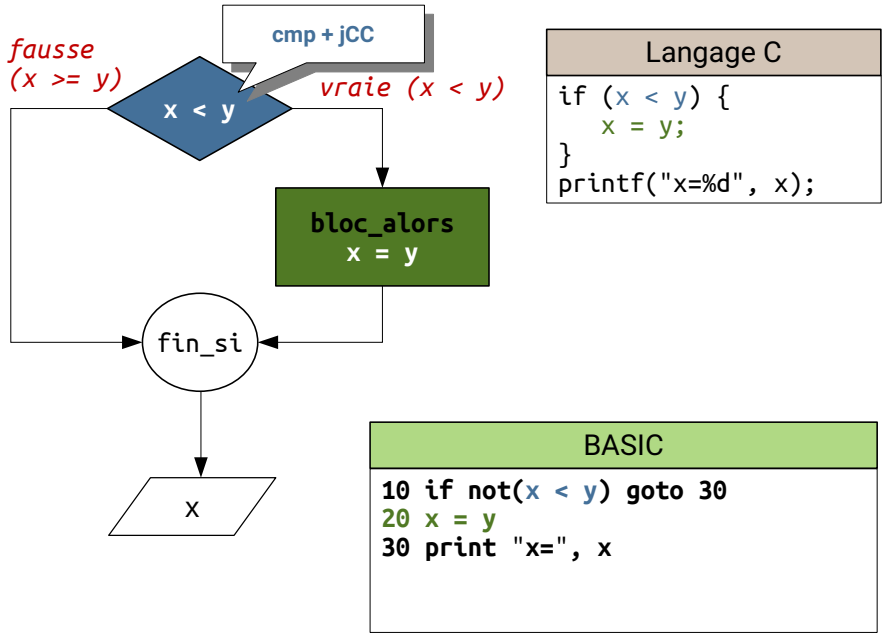


FIGURE 5.1 – si alors

Pour la partie en langage C on considère la condition `x < y`, où `x` et `y` sont deux variables entières que l'on a choisi de modéliser par les registres `eax` et `ebx`. On débute donc la traduction en chargeant la variable `x` dans `eax` et la variable `y` dans `ebx`.

La traduction du `if` est obtenue en réalisant la comparaison entre `x` et `y` au moyen de l'instruction `cmp` qui prend comme opérandes `eax` et `ebx`, suivie par un branchement en fin de conditionnelle si la condition est *fausse*. On prend donc, dans ce cas, la négation de l'expression `x < y`, soit `x >= y`. Ce qui se traduit par l'instruction de branchement conditionnel `jge` (*Jump on Greater or Equal*) :

```
1  .if:
2      mov     eax, [x]
3      mov     ebx, [y]
4      cmp     eax, ebx     ; si  non(x < y) équivalent à x >= y
5      jge     .endif       ; aller en .endif
6  .then:
7      mov     [x], ebx     ; bloc alors
8  .endif:
```

On remarquera que pour clarifier le code on a défini trois labels qui correspondent au *si* (`.if`), *alors* (`.then`) et *fin_si* (`.endif`) mais seul le label `.endif` est utile ici.

5.4.5 Si C1 et C2 et ... et Cn alors

Dans le cas d'une condition complexe qui peut se résumer à des conjonctions (et), il faut que toutes les conditions soient vraies pour exécuter le code du bloc *alors*. On doit donc produire le pseudo code suivant :

- si *C1* est *fausse* alors aller en `.endif`
- si *C2* est *fausse* alors aller en `.endif`
- si *Cn* est *fausse* alors aller en `.endif`

On pourra représenter ce code sous forme d'organigramme pour s'en convaincre.

```
1  if ((x < 3) && (y >= 6) && (x == z)) {
2      x = y;
3  }
```

Listing 5.4.1 – Si Alors avec conjonction de conditions

```
1  ; eax = x, ebx = y, ecx = z
2  .if:
3      cmp  eax, 3     ; C1
4      jge  .endif     ; si non(x < 3) alors aller en finsi
5      cmp  ebx, 6     ; C2
6      jl   .endif     ; si non(y >= 6) alors aller en finsi
7      cmp  eax, ecx   ; C3
8      jne  .endif     ; si non (x == z) alors aller en finsi
9  .then:
10     mov  eax, ebx
11 .endif:
```

Listing 5.4.2 – Si Alors avec conjonction de conditions

Afin de gagner en efficacité lorsque l'on rencontre ce genre de condition complexe il faut **placer la condition qui a le plus de chance d'être fausse en premier** car elle échouera en premier et on n'aura pas à évaluer *C2* à *Cn*.

Prenons l'exemple du Listing 5.4.1 dont la traduction en assembleur est donnée par le Listing 5.4.2. On considère que `x` est représentée par `eax`, `y` par `ebx` et `z` par `ecx`. Il serait alors probablement préférable de placer la condition *C3* en premier si elle a le plus de chance d'échouer, tout dépend bien évidemment des données que l'on traite.

5.4.6 Si C1 ou C2 ou ... ou Cn alors

Dans le cas d'une condition composée de disjonctions (ou), il suffit qu'une seule condition soit vraie pour exécuter le code du bloc *alors*. On doit donc produire le

pseudo code suivant :

- si *C1* est *vraie* alors aller en `.then`
- si *C2* est *vraie* alors aller en `.then`
- si *Cn* est *fausse* alors aller en `.endif`

Afin d'être efficace lorsque l'on rencontre ce genre de condition complexe il faut **placer la condition qui a le plus de chance d'être vraie en premier**.

5.4.7 Si alors sinon

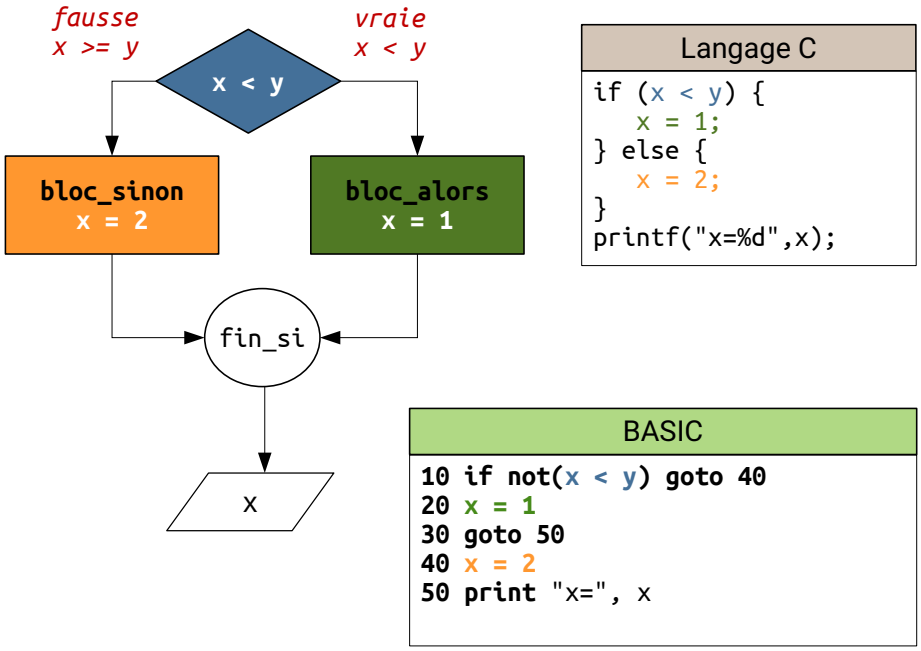


FIGURE 5.2 – si alors sinon

Dans le cas de la conditionnelle *si alors sinon* de la Figure 5.2, il y a également, comme dans le cadre du *si alors*, deux chemins à emprunter en fonction de l'évaluation de la condition du *si*.

On procède comme avec le *si alors* en comparant les valeurs puis en prenant la négation de la condition pour effectuer un branchement conditionnel en `.else`. Après l'exécution des instructions du `.then` il ne faut pas oublier de se brancher en `.endif` sinon on exécuterait également le code du *sinon*.

Notons que `x` et `y` sont deux variables entières et que l'on pourrait n'utiliser qu'un seul registre afin de comparer ces valeurs en chargeant par exemple `x` dans `eax` puis en le comparant à `y` grâce à l'instruction `cmp eax, [y]`. Il n'est pas toujours nécessaire de charger toutes les variables dans les registres.

Le code correspondant est le suivant :

```
.if:
    mov     eax, [x]
    mov     ebx, [y]
    cmp     eax, ebx
    jge     .else
.then:
    mov     [x], 1      ; bloc alors
    jmp     .endif      ; pour ne pas exécuter le code du .else
.else:
    mov     [x], 2      ; bloc sinon
.endif:
```

5.4.8 Tant que

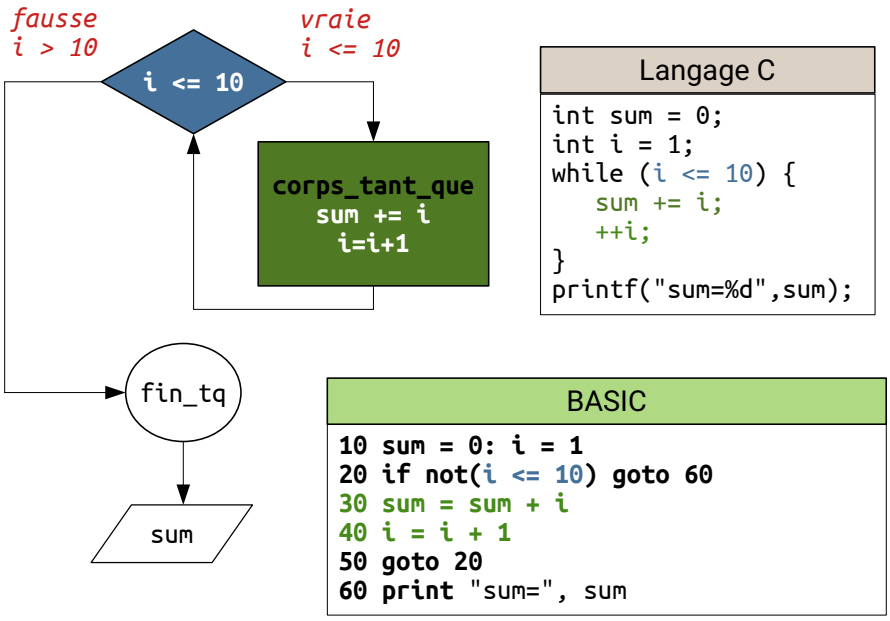


FIGURE 5.3 – Tant que

La structure de contrôle *tant que* est modélisée sur la Figure 5.3. Elle est organisée de la manière suivante et comprend :

- une initialisation `i = 1`
- une condition de poursuite de la boucle `i <= 10`, souvent appelée condition d'arrêt
- le corps de la boucle, c'est à dire les instructions à exécuter à chaque itération de la boucle, soit ici `sum += i;`, suivi de l'incrémentation de `i`

Le *tant que* est traité comme un *si alors* suivi du code du corps de la boucle `.body`, puis le code de l'incrémentation et enfin par une instruction de saut `jmp` (*JuMP*) qui ramène au début de la condition de poursuite (`.while`). Si cette condition est *fausse* on se branchera en `.endwhile`.

Pour traduire la somme des entiers entre 1 et 10 (cf. Listing 5.4.3), on choisit de remplacer la variable `sum` par le registre `eax` et la variable `i` par `ecx` :

```
1     xor     eax, eax        ; sum = 0
2     mov     ecx, 1          ; i = 1
3  .while:
4     cmp     ecx, 10         ; si i > 10 alors sortir du while
5     jg      .endwhile
6     add     eax, ecx        ; sum += i
7     inc     ecx             ; ++i
8     jmp     .while
9  .endwhile:
10    mov     [sum], eax
```

Listing 5.4.3 – Tant que

On remarquera que pour passer à l'itération suivante on a utilisé, ligne 7, l'instruction `inc ecx` et, que sous certaines architectures, un `add ecx,1` qui prend plus de place puisqu'il faut coder la valeur 1 au niveau de l'instruction est plus efficace.

5.4.9 Pour i de 1 à n

L'instruction *pour*, soit `for` en langage C, est en fait un `while` déguisé, il s'agit d'un *sucre syntaxique* [3].

```
1  for (initialisation; condition; incrementation) {
2      corps;
3  }
4
5  initialisation;
6  while (condition) {
7     corps;
8     incrementation;
9  }
```

Listing 5.4.4 – Equivalence des boucles *pour* et *tant que*

3. Expression inventée par Peter J. Landin pour exprimer le fait de donner au programmeur des possibilités d'écriture plus succinctes ou plus proches d'une notation usuelle.

Comme pour la boucle *tant que* on retrouve l'initialisation, la condition de poursuite ainsi que l'incrémentation. Nous avons fait figurer sur le Listing 5.4.4 la correspondance entre la syntaxe du `for` en C et le `while`.

Il est possible de traduire une boucle `for` de deux manières différentes. Considérons le code suivant qui est équivalent au `while` que nous venons de traduire :

```
1  const int N = 10;
2  int sum = 0;
3  for (int i = 1; i <= N; ++i) {
4      sum += i;
5  }
```

On peut traduire cette boucle comme on l'a fait précédemment pour le `while`, le code est donc identique.

L'autre manière de traduire le `for` consiste au préalable à vérifier la condition de poursuite et à ne pas exécuter le corps de la boucle si la condition est *fausse*. Si la condition est *vraie* par contre, la boucle s'exécutera au moins une fois. On traduira donc par le corps de la boucle, l'incrémentation puis la condition de poursuite, qui, si elle est *vraie*, forcera à retourner grâce à un branchement conditionnel au début de la boucle comme présenté Listing 5.4.5.

```
1      xor  eax, eax       ; sum = 0
2      mov  ecx, 1         ; initialisation : i = 1
3      mov  edx, N
4  .pre_for:
5      cmp  ecx, edx       ; test condition
6      jg   .endfor        ; ne pas executer la boucle si i >= n
7
8  .for:
9      add  eax, ecx       ; corps : sum += i
10     inc  ecx            ; incrémentation : ++i
11
12     cmp  ecx, edx       ; test de la condition
13     jle  .for           ; retour au début de la boucle si vraie
14 .endfor:
```

Listing 5.4.5 – Traduction améliorée du for

Dans le premier cas (Listing 5.4.3) on aura à chaque itération de la boucle à exécuter quatre instructions (lignes 4, 5, 7, 8) pour traiter la boucle (sans son corps).

Dans le second cas (Listing 5.4.5), on n'exécutera que trois instructions (lignes 8 à 10) à chaque itération pour traiter la boucle (sans son corps). On peut donc parfois gagner en efficacité en ayant une instruction en moins à exécuter, mais le gain est généralement très faible.

5.4.10 Selon cas

La structure *selon cas*, c'est à dire le `switch` en langage C, est dans le cas général difficile à traduire et dépend des données. Dans certains cas les données sont consécutives et il est facile de remplacer le `switch` par une expression comme c'est le cas du Listing 5.4.6.

```
1  int convert1(char c) {
2      int code = 0;
3      switch(c) {
4          case 'a': case 'A': code = 1; break;
5          case 'b': case 'B': code = 2; break;
6          ...
7          case 'z': case 'Z': code = 26; break;
8      }
9      return code;
10 }
11
12 int convert2(char c) {
13     if (isalpha(c)) return toupper(c) - 64;
14     return 0;
15 }
```

Listing 5.4.6 – Exemple de switch simplifiable par une expression

```
1 switch(c) {
2     case 1: liste_instructions_cas_1; break;
3     case 3: liste_instructions_cas_3; break;
4     case 22: liste_instructions_cas_22; break;
5     default: liste_instructions_defaut;
6 }
```

Listing 5.4.7 – Exemple de switch avec table de conversion

En effet la première fonction `convert1` ne fait que remplacer chaque lettre par un identifiant entier en suivant l'ordre alphabétique. Elle contraint à écrire 26 lignes qui peuvent être remplacées par une seule (ligne 13) de la fonction `convert2`.

Le cas le plus problématique correspond à des valeurs du `case` qui ne suivent pas une suite logique ou calculable. Dans ce cas on passe généralement par un tableau de valeurs (cf. Listings 5.4.7 et 5.4.8).

On crée ainsi deux tables, l'une pour les valeurs du `case`, l'autre qui contient les adresses des labels qui correspondent au code à exécuter pour chaque `case`. On remarquera que l'on utilise des labels spéciaux dotés d'un préfixe `..@` qui sont définis pour ce genre de situation. Les labels qui débutent par ce préfixe n'obéissent

```
section .data
    switch_values_table:    dd 1, 3, 22
    switch_jumps_table: dd ..@case_1, ..@case_3, ..@case_22

section .text
main:
    ...
    xor     ecx, ecx
    mov     edx, 3
.for:
    cmp     ecx, edx
    jge     .default
    cmp     [switch_values_table + ecx * 4], eax
    jne     .endif
    jmp     [switch_jumps_table + ecx * 4]
.endif:
    inc     ecx
    jmp     .for

..@case_1:
    ...
    jmp     .endswitch
..@case_3:
    ...
    jmp     .endswitch
..@case_22:
    ...
    jmp     .endswitch
.default:
    ...
.endswitch:
```

Listing 5.4.8 – Exemple de switch avec table de conversion en assembleur 32 bits

pas aux règles des labels locaux et peuvent être référencés à tout moment dans le code.

On doit donc parcourir la table des valeurs jusqu'à trouver un valeur de cette table, ou alors, si on ne la trouve pas, on éxécutera le `default`.

5.4.11 Techniques d'amélioration liées aux boucles for

Nous allons présenter deux techniques d'amélioration liées aux boucles de type `for` et par extension aux boucles `while`. La première qualifiée de *dépliage* permet de diminuer le nombre d'itérations de la boucle en dupliquant les instructions du corps de la boucle. La seconde appelée *tuilage* augmente l'efficacité des traitements

en réduisant les données placées en mémoire cache lors de l'utilisation de grands tableaux.

5.4.11.1 Dépliage de boucle

Le dépliage de boucle ou *loop unrolling* (voir Figure 5.4) en anglais consiste à augmenter le corps de la boucle en le répétant plusieurs fois. On dépliera une boucle par une puissance de 2 : soit 2, 4, 8 voire 16. Le but de cette technique est double, elle permet :

- de diminuer le nombre de branchements
- d'augmenter l'efficacité en exécutant un plus grand nombre d'instructions avant de passer à la prochaine itération

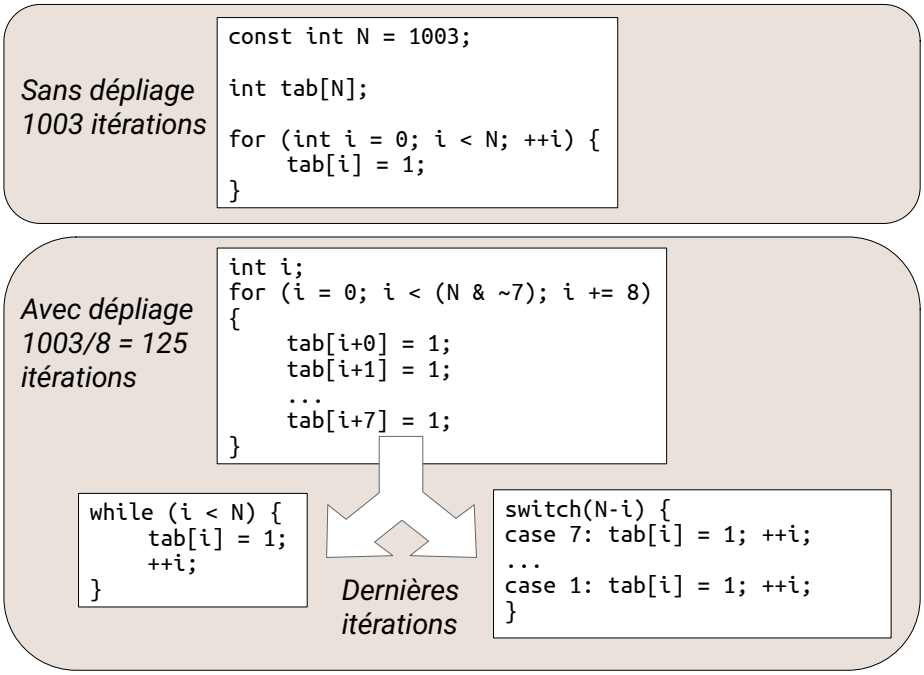


FIGURE 5.4 – Dépliage de boucle

En effet, comme on l'a vu précédemment, le traitement d'une boucle `for` classique demande d'exécuter au moins 4 instructions : une comparaison et une instruction de branchement conditionnel en début de boucle, puis une incrémentation et un branchement pour revenir au début de la boucle.

Si `N = 1003`, comme sur l'exemple de la Figure 5.4, cela fait $1003 \times 4 = 4012$ instructions à exécuter rien que pour traiter la boucle sans son corps.

Par contre si la boucle `for` est dépliée par 8, on traitera dans un premier temps 125 itérations ($1003/8 = 125$) et donc seulement $125 \times 4 = 500$ instructions liées à la boucle.

Reste à traiter les trois dernières itérations potentielles de la boucle, ce qui peut être fait de deux manières différentes :

- soit par une boucle `while` mais on risque de perdre quelque peu en efficacité si le corps de la boucle se traduit par quelques instructions,
- soit par un `switch` dont l'intérêt est d'éviter les branchements répétifs comme on le ferait pour le `while`

On obtient avec le `switch` un code séquentiel puisqu'on ne trouvera pas de `break`. L'utilisation d'un `switch` se révèle donc en général intéressante lorsque la boucle est dépliée par 8 ou 16.

```
1  section .data
2      N        EQU 1003           ; constante
3      tab:     times N dd 0       ; tableau de N entiers dont chaque
4                                  ; élément est initialisé à 0
5
6  section .text
7      mov      ebx, tab           ; ebx = tab
8      mov      eax, 1             ; eax = 1 (constante)
9      xor      ecx, ecx           ; i = 0
10     mov      edx, N             ; edx = N
11     and      edx, ~(8-1)        ; edx = (N / 8) * 8
12 .for_ur8:
13     cmp      ecx, edx           ; fin de boucle si i >= N
14     jge      .endfor_ur8
15     mov      dword [ebx + ecx * 4 +  0], eax
16     mov      dword [ebx + ecx * 4 +  4], eax
17     mov      dword [ebx + ecx * 4 +  8], eax
18     mov      dword [ebx + ecx * 4 + 12], eax
19     mov      dword [ebx + ecx * 4 + 16], eax
20     mov      dword [ebx + ecx * 4 + 20], eax
21     mov      dword [ebx + ecx * 4 + 24], eax
22     mov      dword [ebx + ecx * 4 + 28], eax
23     add      ecx, 8             ; i += 8
24     jmp      .for_ur8
25 .endfor_ur8:
26 .while:
27     cmp      ecx, N  ; dernieres iterations
28     ...
29
```

Listing 5.4.9 – Dépliage de boucle

La traduction assembleur du code de la Figure 5.4 est donné Listing 5.4.9. On commence par définir la constante `N` grâce à l'instruction `EQU` qui signifie *EQUal*, puis le tableau `tab` grâce à l'instructions `times N` qui signifie répéter `N` fois ce qui

suit, c'est à dire `dd 0` qui définit un entier sur 32 bits initialisé à 0. On crée donc 1003 entiers initialisés à 0.

On traduit ensuite le code en utilisant les associations suivantes :

- le tableau `tab` est placé dans `ebx`
- la variable de boucle `i` est remplacée par `ecx`
- le registre `eax` contient la valeur à affecter à chaque élément du tableau soit 1
- le registre `edx` contient le plus proche multiple de 8 de la constante `N` qui représente la taille du tableau

Afin de stocker dans `edx` le multiple de 8 le plus proche de `N`, il existe deux possibilités :

- la plus naturelle consiste à rendre `N` multiple de 8 en le divisant par 8 puis en le multipliant par 8, ce qui peut être fait rapidement en utilisant les instructions de décalage puisque 8 est une puissance de 2. On peut donc traduire par :

```
1    mov     edx, N      ;           edx = 1003
2    shr     edx, 3      ; /8=2^3    edx = 125
3    shl     edx, 3      ; *8        edx = 1000
```

- l'autre solution consiste à utiliser un masque afin d'éliminer les trois premiers bits de `edx` qui permettent de coder des valeurs entre 0 et 7 :

```
1    mov     edx, N      ;  edx = 1003 =   000..0011_1110_1011_b
2    and     edx, ~7     ;           ~7   & 111..1111_1111_1000_b
3                        ;  edx = 1000 =          11_1110_1000_b
```

ici le symbole ~ (tilde) correspond à la complémentation.

On peut en outre utiliser une macro instruction afin de remplacer le corps de la boucle de manière à obtenir un code plus lisible cela est particulièrement vrai si le corps de la boucle contient plusieurs instructions. On définit donc la macro instruction `BODY` qui prend un paramètre représentant le décalage de l'adresse observé listing 5.4.10.

Enfin, on pourra utiliser des macro instructions de `nasm` afin de remplacer les huit macro instructions `BODY` (lignes 14 à 21) par une boucle :

```
1    %assign i 0    ; définition d'une variable affectée à 0
2    %rep 8         ; répéte 8 fois
3    BODY i         ; appel de la macro BODY avec la valeur de i
4    %assign i i +4 ; ajouter 4 à i et l'affecter à i, soit i += 4
5    %endrep        ; fin de la répétition
```

```
%macro BODY 1
    mov     [ebx + ecx * 4 + %1], eax
%endmacro

section .text
    mov     ebx, tab
    mov     eax, 1
    xor     ecx, ecx
    mov     edx, N
    and     edx, ~(8-1)
.for_ur8:
    cmp        ecx, edx
    jge     .endfor_ur8
    BODY     0
    BODY     4
    BODY     8
    BODY    12
    BODY    16
    BODY    20
    BODY    24
    BODY    28
    add     ecx, 8
    jmp     .for_ur8
.endfor_ur8:

; dernières itérations de la boucle
.while:
    cmp     ecx, N
            ...

```

Listing 5.4.10 – Dépliage de boucle avec macro instruction

Attention

Attention cependant, en C l'utilisation des parenthèses est primordial car si on ne les utilise pas, l'expression sera interprétée de manière à ce que la boucle `for` ne soit pas exécutée et c'est le `while` qui traitera toutes les itérations de la boucle.

En effet, si on écrit :

```
for (i = 0; i < N & ~3; ++i) { ... }
```

L'expression de continuité de la boucle sera interprétée comme `i < N`, c'est à dire vrai, ce qui correspond à la valeur 1. On réalise ensuite un **et binaire** entre la

valeur 1 et ~3, c'est à dire une valeur dont les deux premiers bits sont à 0. Au final on obtient la valeur 0, la boucle `for` n'est donc pas exécutée.

Il faut donc bien écrire :

```
for (i = 0; i < (N & ~3); ++i) { ... }
```

5.4.11.2 Tuilage

La technique du tuilage (*tiling* en anglais) aussi appelée blocage de boucle (*loop blocking*) est associée au traitement des tableaux notamment ceux à deux dimensions. Elle permet de décomposer un traitement défini pour un grand tableau en plusieurs traitements plus simples qui interagissent sur des sous parties du tableau que l'on gardera en mémoire cache afin de les réutiliser. Cette technique est particulièrement efficace dans le cas du produit de matrices car elle permet de ne pas saturer la mémoire cache et a pour objectif de cantonner les données aux registres et aux caches L1 et L2.

```
const int N = 1027;
int tab[N];

for (int i = 0; i < N; ++i) {
    A[i] = 0;
}
```

```
const int BLOCK_SIZE = 32;

for (int i = 0; i < N; i += BLOCK_SIZE) {
  for (int ii = 0; ii < std::min(i+BLOCK_SIZE, N); ++ii) {
    A[ii] = 0;
  }
}
```

Cette technique n'a d'intérêt que si on lit et/ou écrit les données plusieurs fois.

5.4.11.3 Perte d'efficacité : if à l'intérieur d'un for

Le cas d'un `if` à l'intérieur d'une boucle `for` (ou d'un `while`) est problématique car c'est le genre de traitement qui peut ralentir l'exécution de la boucle. En effet, si la condition du `if` n'est pas prédictible, la prédiction de branchement (cf. Section 3.7.1) ne sera pas capable de choisir avec certitude la branche de code à exécuter.

Il faut faire en sorte, quand cela est possible, de remplacer le `if` par un calcul plutôt que par un branchement pour gagner en efficacité. La vectorisation du code permet également d'éviter les ralentissements dûs à l'impossibilité de prédire quel chemin emprunter (cf. Chapitre 14).

5.4.12 Instructions pour l'élimination des if

On dispose de deux instructions qui permettent de supprimer les branchements :

- `setCC` (*Set Byte on Condition*) introduite à partir de l'Intel 80386 qui met à 0 ou 1 un registre 8 bits ou un emplacement mémoire 8 bits en fonction des valeurs des bits CF, SF, OF, ZF et PF du registre `eflags` : on remplacera CC par les lettres qui correcpondent aux sauts conditionnels, par exemple `nz` pour *Not Zero*
- `cmovCC` (*Conditional MOVe*) qui déplacera entre registres ou entre un registre et une adresse mémoire les données de la source vers la destination en fonction des valeurs des bits du registre `eflags`

Par exemple, le code suivant :

```
1  if (x % 3 == 0) {
2     return  101;
3  } else {
4     return 7001;
5  }
```

peut être traduit de manière classique par :

```
1      mov     eax, [x]
2      movs    ecx, 3
3      cdq                     ; convertir dans edx:eax
4      div     ecx             ; division par 3
5  if:
6      cmp     edx, 0          ; quotient == 0 ?
7      jnz     .else
8  .then:
9      mov     eax, 101
10     jmp     .endif
11 .else:
12     mov     eax, 7001
13 .endif:
14     ; sortie de la fonction
```

Cette traduction utilise deux sauts : un saut conditionnel `jnz` pour ne pas exécuter le `.then` et un saut `jmp` pour ne pas exécuter la partie `.else`. On peut réécrire le code en utilisant `cmov` afin d'éviter ces sauts qui viennent perturber l'exécution du code :

```
1      mov     eax, [x]
2      mov     ecx, 3
3      cdq
4      div     ecx
5      cmp     edx, 0
6      mov     ecx, 7001
```

```
7       mov     eax, 101
8       cmovnz  eax, ecx
```

Dans le même esprit, lorsqu'on doit par exemple travailler avec des booléens, le code suivant :

```
1   if (expression == 0) {
2       return  0;
3   } else {
4       return 1;
5   }
```

peut être réécrit en :

```
1       ; évaluation de l'expression résultat dans edx
2       xor     eax, eax    ; valeur de sortie à 0
3       cmp     edx, 0      ; ou test edx, edx
4       setnz   al
5       ; sortie de la fonction
```

Ici, le comportement de l'instruction `setnz` est :

```
1   if (NZ) {
2       al = 1;
3   } else {
4       al = 0;
5   }
```

Il est nécessaire de mettre la valeur 0 dans `eax` de manière à ce que le résultat soit 0 ou 1. Ou alors, il faut utiliser l'instruction `movzx eax,al` après `setnz`.

5.4.13 Débit et latence des instructions

On distingue deux grandeurs lorsqu'on évoque le temps de traitement des instructions : la latence (*latency*) et le débit (*throughput*). Malheureusement il est très difficile de distinguer et comprendre la différence entre ces deux valeurs. D'après la documentation Intel :

- *Latency is the number of processor clocks it takes for an instruction to have its data available for use by another instruction*
- *Throughput is the number of processor clocks it takes for an instruction to execute or perform its calculations*

Référence mémoire

En fait, lorsque l'on parle de latence on fait référence à la **chaîne de dépendance des latences** (*dependency chain latency*). Quand une instruction a besoin du résultat d'une instruction qui la précède on parle de dépendance et dans ce cas on mesure son coût de traitement par sa latence.
Le **débit** (*throughput*) est le nombre maximum d'instructions de même type qui peuvent être exécutées par cycle d'horloge quand il n'y a pas de dépendances.

Par exemple, imaginons que nous ayons à traduire le code suivant où les variables sont des entiers :

```
a = a + b + c + d;
```

Convenons que `a` est dans `eax`, `b` dans `ebx`, etc. Une première traduction en assembleur est :

```
    add     eax, ebx ;
    add     eax, ecx ; dépendance avec la ligne 1
    add     eax, edx ; dépendance avec la ligne 2
```

Ce code introduit deux dépendances : de la ligne 2 vers la ligne 1 et de la ligne 3 vers la ligne 2. Etant donné qu'on utilise le même registre (`eax`) pour stocker le résultat du calcul, l'instruction de la ligne 2 doit attendre que l'instruction de la ligne 1 soit terminée pour pouvoir ajouter le résultat de `ecx` à `eax`.

On peut réécrire ce code de manière à pouvoir effectuer deux sommes en parallèle :

```
    add     eax, ebx ; indépendant de la ligne 2: a += b
    add     ecx, edx ; indépendant de la ligne 1  c += d
    add     eax, edx ; dépendance avec les lignes 1 et 2: a += c
```

Le second code peut éventuellement être réalisé de manière plus rapide puisque l'on pourra effectuer les deux premières sommes en parallèle nonobstant le fait que l'on perde la valeur de `ecx`.

Imaginons à présent un exemple plus conséquent qui concerne un tableau de flottants dont on doit faire la somme. On écrit une boucle que l'on déplie par 8 :

```
// ici N, la taille du tableau, est multiple de 8
const int N = 320000;
float array[N];

float sum = 0.0f;
for (int i = 0; i < N; i += 8) {
    sum += array[i+0];
```

```
8       sum += array[i+1];
9       sum += array[i+2];
10      sum += array[i+3];
11      sum += array[i+4];
12      sum += array[i+5];
13      sum += array[i+6];
14      sum += array[i+7];
15  }
```

Si on traduit ce code en utilisant les registres SSE mais sans vectorisation en ne prenant en compte que la partie basse du registre (cf. Chapitre 8), on obtient :

```
1       mov     ebx, array
2       xorps   xmm0, xmm0                  ; xmm0 = sum = 0.0
3       xor     ecx, ecx                    ; i = 0
4   for_i:
5       cmp     ecx, 320000                 ; fin, si i >= 320000
6       jge     .endfor_i
7       addss   xmm0, [ebx + ecx*4 + 0]     ; sum += array[i];
8       addss   xmm0, [ebx + ecx*4 + 4]     ; sum += array[i+1];
9       addss   xmm0, [ebx + ecx*4 + 8]     ; sum += array[i+2];
10      addss   xmm0, [ebx + ecx*4 + 12]    ; sum += array[i+3];
11      addss   xmm0, [ebx + ecx*4 + 16]    ; sum += array[i+4];
12      addss   xmm0, [ebx + ecx*4 + 20]    ; sum += array[i+5];
13      addss   xmm0, [ebx + ecx*4 + 24]    ; sum += array[i+6];
14      addss   xmm0, [ebx + ecx*4 + 28]    ; sum += array[i+7];
15      add     ecx, 8                      ; i += 8
16      jmp     .for_i
17  endfor_i:
```

La partie basse du registre `xmm0` est utilisée pour contenir la somme mais chaque instruction (de la ligne 7 à la ligne 13) dépend de l'instruction précédente. Il existe une chaîne de dépendance de 7 instructions. Si l'instruction `addss, addsd` possède un débit d'une instruction par cycle et une latence de 4 cycles alors le coût du traitement de la ligne 6 à la ligne 13 est d'au moins $7 \times 4 = 28$ cycles. Cependant comme l'exécution se fait dans le désordre, il est possible que le microprocesseur puisse briser en partie ces dépendances.

Maintenant, si on organise le traitement autrement en créant des sommes partielles, on peut briser la chaîne des dépendances :

```
1       mov     ebx, array
2       xorps   xmm0, xmm0                  ; xmm0 = sum1 = 0.0
3       xorps   xmm1, xmm1                  ; xmm1 = sum2 = 0.0
4       xorps   xmm2, xmm2                  ; xmm2 = sum3 = 0.0
5       xorps   xmm3, xmm3                  ; xmm3 = sum4 = 0.0
6       xor     ecx, ecx                    ; i = 0
7   for_i:
8       cmp     ecx, 320000                 ; fin de boucle si i >= 320000
9       jge     .endfor_i
10      addss   xmm0, [ebx + ecx*4 + 0]     ; sum1 += array[i]
```

```
11      addss   xmm1, [ebx + ecx*4 + 4]     ; sum2 += array[i+1]
12      addss   xmm2, [ebx + ecx*4 + 8]     ; sum3 += array[i+2]
13      addss   xmm3, [ebx + ecx*4 + 12]    ; sum4 += array[i+3]
14      addss   xmm0, [ebx + ecx*4 + 16]    ; sum1 += array[i+4]
15      addss   xmm1, [ebx + ecx*4 + 20]    ; sum2 += array[i+5]
16      addss   xmm2, [ebx + ecx*4 + 24]    ; sum3 += array[i+6]
17      addss   xmm3, [ebx + ecx*4 + 28]    ; sum4 += array[i+7]
18      add     ecx, 8                      ; i += 8
19      jmp     .for_i
20  endfor_i:
21      addss  xmm0, xmm1                   ; sum1 += sum2
22      addss  xmm2, xmm3                   ; sum3 += sum4
23      addss  xmm0, xmm2                   ; sum1 += sum3
```

La ligne 13 dépend de la ligne 9, la ligne 14 dépend de la ligne 10, etc. Au final nous avons 4 dépendances, donc trois de moins que dans la première version de la somme. Le code assembleur précédent correspond en fait au code C suivant :

```
1  float sum, sum1, sum2, sum3, sum4;
2  sum1 = sum2 = sum3 = sum4 = 0.0f;
3  for (int i = 0; i < N; i += 8) {
4      sum1 += array[i+0];
5      sum2 += array[i+1];
6      sum3 += array[i+2];
7      sum4 += array[i+3];
8      sum1 += array[i+4];
9      sum2 += array[i+5];
10     sum3 += array[i+6];
11     sum4 += array[i+7];
12 }
13 sum = (sum1 + sum2) + (sum3 + sum4);
```

Nous avons reporté, Table 5.10, certains résultats de l'implantation des solutions précédentes suivant le nombre de cycles nécessaires au traitement de la boucle. Nous avons exécuté 100 fois la somme des éléments d'un tableau de 320_000 flottants :

- la version v1 correspond au code avec 7 dépendances
- la version v2 est un code avec 4 dépendances dues à l'utilisation de 4 registres SSE (`xmm0` à `xmm3`)
- la version v3 ne possède pas de dépendances car on utilise 8 registres SSE (`xmm0` à `xmm7`).

La ligne intitulée gain est le rapport entre la version 1 et la version 3.

Il est alors flagrant que diminuer le nombre de dépendances conduit à obtenir un code qui s'exécute plus rapidement. Sur AMD Ryzen 5 3600, on va $4,47$ fois plus vite sans chaîne de dépendance en utilisant la version 3.

Dépendances	AMD Ryzen 5 3600	Intel Core i5 7400	Intel Core i7 8700	Intel Xeon Silver 4208	AMD Ryzen 5 5600g	Intel Core i7 10850H
v1 (7 dépendances)	103	112	93	105	85	122
v2 (4 dépendances)	35	28	23	29	23	28
v3 (aucune)	23	19	16	25	15	19
gain	$\times 4,47$	$\times 5,89$	$\times 5,81$	$\times 4,20$	$\times 5,66$	$\times 6,42$

TABLE 5.10 – Nombre de millions de cycles pour la somme des éléments d'un tableau avec et sans chaîne de dépendance

Attention

Un dernier point important concerne la version 2 avec 4 dépendances. Celles-ci sont distantes. Si on les place côte à côte, on obtient sur AMD Ryzen 5 5600g, un temps moyen d'exécution de 32 millions de cycles et non plus 23.

5.5 Conclusion

Nous avons vu dans ce chapitre comment traduire les structures de contrôle du langage C et comment on pouvait améliorer l'efficacité des boucles en les dépliant ou en brisant la chaîne de dépendances d'un calcul. Ces améliorations doivent être conjuguées avec l'accélération de l'accès mémoire en tentant de mettre le plus de données dans les caches et en les réutilisant quand cela est possible.

5.5.1 Que retenir ?

- en architecture 32 bits, on dispose de huit registres généraux, cependant seuls 6 sont utilisés pour faire des calculs ou stocker des données (`eax`, `ebx`, `ecx`, `edx`, `edi`, `esi`) ; le registre `esp` contient le sommet de pile et ne doit pas être modifié directement alors que `ebp` est utilisé afin de récupérer les arguments d'un sous-programme
- en architecture 32 bits, si l'on doit réaliser des multiplications ou des divisions les registres `eax` et `edx` seront impactés, ce qui ne laisse plus que 4 registres pour faire les calculs
- en architecture 64 bits, on dispose de 8 registres généraux supplémentaires (`r8` à `15`), ce qui permet de lever le verrou des limitations du 32 bits
- l'assembleur ne dispose pas de structures de contrôle comme la conditionnelle `if`, les boucles `for`, `while`. Ecrire en assembleur est donc une tâche difficile.

- ▷ les techniques de dépliage de boucle ou de tuilage permettent d'améliorer l'efficacité du traitement des boucles
- ▷ positionner un `if` à l'intérieur d'une boucle (`for` ou `while`) conduit à ralentir l'exécution du traitement, il faut alors être en mesure de pouvoir éliminer le `if` soit en le remplaçant par des instructions spécifiques (cf Chapitre 14), soit en le vectorisant (cf. Chapitre 8)

5.5.2 Compétences à acquérir

On doit être capable de traduire :

- ☐ une multiplication, une division, un modulo
- ☐ une conditionnelle avec plusieurs conditions séparées par des et/ou
- ☐ une boucle for
- ☐ une boucle while

5.6 Exercices

Exercice 21 - Traduire le code suivant en assembleur x86 32 bits où x, y et z sont trois variables entières :

```
1 if ((((x % 2) == 0) && (y < 257)) || (z == 9)) {
2     x = x + y - z;
3 }
```

Exercice 22 - Traduire le code suivant en assembleur x86 32 bits :

```
1 const int SIZE = 1000;
2 int tab[SIZE];
3 for (int i = 0; i < SIZE; ++i) {
4     tab[i] = (i + 1) % 7;
5 }
```

Exercice 23 - Ecrire les fonctions qui réalisent les traitements suivants en C puis les traduire en assembleur :

1. vérifier qu'un entier est une puissance de 2
2. trouver le bit le plus significatif d'un entier non signé, c'est à dire la position du bit de poids fort

3. compter le nombre de bits à 1 dans un entier non signé

Exercice 24 - Soit le code suivant :

```
const int N = 1005;
int tab[N];

for (int i = 0; i < N; ++i) {
    tab[i] = i;
}
```

1. réaliser un dépliage par 8 du code C
2. puis le traduire en assembleur 32 bits

Attention, la difficulté lors de la traduction en assembleur provient de l'instruction `tab[i] = i;` pour laquelle il faut augmenter `i` à chaque itération du dépliage.

Exercice 25 - Que se passe t-il si on réalise le traitement suivant (cf. Section 5.3.2.3) ?

```
    mov   eax, -6       ; eax = FF_FF_FF_FA_h
    xor   edx, edx      ; mise à -1 de edx
    dec   edx
    mov   ebx, 3        ;
    div   ebx
```

Exercice 26 - Traduire la fonction suivante en assembleur x86 32 bits de la manière *la plus efficace possible* :

```
int zero_un_moins_un( int x ) {
    if (x < 0) {
        return -1;
    } else if (x > 1) {
        return 1;
    } else {
        return 0;
    }
}
```

Chapitre 6

Appel de sous-programme

Aux cieux, les dieux,
Baptisent des dissidents belliqueux.

6.1 Introduction

Dans ce chapitre nous allons voir comment réaliser l'appel de sous-programme en 32 et 64 bits. Malheureusement les conventions d'appel dans ces deux architectures sont très différentes sous Linux et elles différent également entre Linux et Windows. Il faut donc soit penser méticuleusement au choix des registres si on désire écrire du code en 32 bits pour ensuite passer au 64 bits ou revoir entièrement son code.

6.2 Appel de sous-programme en 32 bits

Regardons dans un premier temps comment est réalisé l'appel de sous-programme en 32 bits.

6.2.1 Rôle de la pile

Lorsque l'on appelle un sous-programme en 32 bits on passe les paramètres dans la pile. La pile est une partie de la mémoire centrale qui sert d'espace de stockage et, en réalité, il existe plusieurs piles. Chaque programme se voit attribuer une pile. Lorsque l'on bascule d'un programme à un autre on réalise un changement de contexte et on doit mettre à jour les différents registres pour qu'ils soient conformes à l'état dans lequel ils étaient avant de basculer vers un autre programme. La pile permet de garder trace des appels de sous-programmes, de passer les paramètres des sous-programmes et de créer des variables locales à un sous-programme.

En architecture 32 bits, le segment `ss` contient l'adresse du début de la pile et Le sommet de pile se trouve dans le registre `esp`. On utilise le registre `ebp` afin de récupérer les paramètres placés dans la pile.

Sous Linux, la pile d'un programme possède une taille de 8 Mo qui peut être éventuellement redéfinie. On peut obtenir la valeur de cette taille initiale de la pile grâce à la commande `ulimit -a` dans le terminal.

Les deux instructions principales utilisées pour manipuler la pile sont `push` et `pop` mais les instructions `call` et `ret` agissent également sur celle-ci.

6.2.1.1 Push pour empiler ou sauvegarder des données

L'instruction `push` consiste à mettre la valeur d'un registre, un emplacement mémoire ou une constante dans la pile. Cependant la pile possède un fonctionnement différent et spécifique par rapport à une pile que l'on pourrait implanter soi-même. La pile est un tableau d'octets que l'on remplit par le haut et non par le bas comme on le ferait classiquement.

Par exemple `push eax`, consiste à **abaisser** le sommet de pile puis y écrire la valeur contenue dans `eax` ce qui se résume en fait aux deux instructions suivantes :

```
1    sub    esp, 4
2    mov    [esp], eax
```

On soustrait ici 4 octets à `esp` car `eax` est un registre 32 bits.

6.2.1.2 Pop pour dépiler ou restaurer des données

L'instruction `pop` fonctionne de manière inverse. Par exemple `pop eax`, lit la valeur en sommet de pile et la stocke dans `eax` puis **remonte** le sommet de pile :

```
1    mov    eax, [esp]
2    add    esp, 4
```

6.2.1.3 pusha, pushad, pushf

Il existe d'autres instructions pour empiler et dépiler des informations dans la pile, dont notamment :

- `pusha` place dans la pile l'ensemble des registres généraux 16 bits (`ax`, `cx`, `dx`, `bx`, `sp`, `bp`, `si`, `di`)
- `pushad` place dans la pile l'ensemble des registres généraux 32 bits (`eax`, `ecx`, `edx`, `ebx`, `esp`, `ebp`, `esi`, `edi`)
- `pushf` place dans la pile le registre 16 bits flags

- `pushfd` place dans la pile le registre 32 bits eflags

Bien entendu, on dispose des instructions `popa`, `popad`, `popf` et `popfd` qui réalisent les opérations inverses.

6.2.2 Réalisation d'un appel de sous-programme

L'appel de sous-programme en 32 bits est l'un des concepts des plus difficiles à appréhender lorsque l'on apprend l'assembleur car il fait appel à diverses notions et conventions. Lors de l'appel d'un sous-programme, on distingue :

- le sous-programme **appelant** (un autre sous-programme) qualifié de *caller* en anglais
- du sous-programme **appelé** (par l'appelant) qualifié de *callee*

Pour réaliser l'appel de sous-programme, on procède de la manière indiquée Table 6.1, qui consiste à suivre la convention d'appel du langage C sous Linux.

	Sous-programme appelant (*Caller*)	**Sous-programme appelé** (*Callee*)
1	placer les paramètres dans la pile dans l'ordre inverse de la définition du sous-programme	
2	appel du sous-programme grâce à l'instruction (`call`)	
3		entrée dans le sous-programme : sauvegarde de `ebp`, mise à jour de `ebp`
4		récupération des paramètres grâce à `ebp`
5		Exécution du sous-programme
6		sortie du sous-programme : mise à jour de `esp`, restauration de `ebp`
7	suppression des paramètres mis dans la pile à l'étape 1	

TABLE 6.1 – Appel de sous-programme en 32 bits convention du langage C

6.2.3 Registres non modifiables

La convention d'appel en 32 bits en langage C impose que le sous-programme appelé ne modifie pas certains registres. Ces registres sont `ebp`, `ebx`, `esi` et `edi`. Il est donc nécessaire, une fois entré dans le sous-programme appelé, de sauvegarder ces registres dans la pile si on a l'intention de les modifier. Il faudra par la suite les dépiler afin de restaurer leur contenu avant de sortir du sous-programme. Les registres qui peuvent être modifiés sont donc `eax`, `ecx` et `edx`.

La conséquence de cela est que si on écrit du code assembleur qui appelle une fonction de la librairie C, on doit garder à l'esprit que `eax`, `ecx` et `edx` pourront être modifiés. Il ne faudra pas stocker de données qui doivent être réutilisées après l'appel du sous-programme ou alors il faudra les sauvegarder dans la pile, puis les restaurer.

Afin de se rappeler des registres qui sont modifiables ou non modifiables, les deux vers qui figurent en préambule de ce chapitre, représentent un moyen mnémotechnique intéressant :

AuX CieuX, les DieuX,
BaPtisent des DIsSIdents BelliqueuX.

Il fait apparaître :

- sur la première ligne les registres modifiables : `ax`, `cx`, `dx` (et par extension `eax`, `ecx`, `edx`)
- et sur la seconde ceux qu'il faut préserver : `bp`, `di`, `si`, `bx` (et par extension `ebp`, `edi`, `esi`, `ebx`)

6.2.4 Valeur de retour de sous-programme en 32 bits

Lorsqu'un sous-programme retourne une valeur il doit le faire en suivant la convention du langage C :

- s'il s'agit d'une valeur entière (entier, pointeur), on la place dans le registre `eax`
- s'il s'agit d'un nombre à virgule flottante, on le place dans `st0` le sommet de pile du coprocesseur (cf. Chapitre 7)

6.2.5 Exemple d'appel en 32 bits

Prenons un exemple simple avec le programme suivant :

```
1 int sum(int a, int b) {
2     int r;
3     r = a + b;
4     return r;
5 }
6
7 int main() {
8     sum(4, 5);
9 }
```

Le *caller* (`main`) appelle le *callee* (`sum`) qui retourne une valeur entière mais que la fonction `main` n'utilisera pas, ceci afin de simplifier le code que nous allons écrire. Nous allons réaliser une traduction très terre à terre de cet exemple afin de montrer tout ce qui doit être réalisé. Pour cela, nous utiliserons, dans la fonction `sum`, le registre `eax` pour représenter `a` et `ebx` pour représenter `b`.

6.2.5.1 Appel du sous-programme

Traduisons dans un premier temps le sous-programme `main` (cf. Listing 6.2.1). Il consiste à passer les paramètres dans la pile. On peut le faire ici de deux manières différentes puisqu'il s'agit de constantes entières : soit on met la constante dans un registre et on empile le registre (lignes 2 et 3 du code qui suit), soit on empile directement la constante (ligne 4). Dans ce dernier cas il faut préciser sur combien d'octets on code la valeur 4. Etant donné qu'il s'agit d'un entier sur 32 bits, on utilise le préfixe `dword` pour *double word*.

On réalise ensuite l'appel du sous-programme `sum` grâce à l'instruction `call`. Cela a pour effet de stocker dans la pile l'adresse de retour du sous-programme, c'est à dire l'adresse de l'instruction située juste après `call`, c'est à dire la ligne 6.

Nous verrons pourquoi, ci-après, nous appelons l'instruction `add esp, 8` en ligne 6 après être sorti du sous-programme.

```
1 main:
2     mov    eax, 5              ; place dans la pile le second parametre
3     push   eax
4     push   dword 4             ; place le premier parametre
5     call   sum                 ; appel du sous-programme
6     add    esp, 8              ; supprime les parametres
7     ret
```

Listing 6.2.1 – Appelant en 32 bits

Convention : ordre des paramètres

Par convention en C (en architecture 32 bits), on place le dernier paramètre du sous-programme appelé en premier dans la pile et le premier paramètre du sous-programme en dernier dans la pile.

Pour notre exemple, on commence donc par le paramètre le plus à droite (5) et on termine par le plus à gauche (4).

6.2.5.2 Le sous-programme appelé

Traduisons dans un second temps le sous-programme `sum` qui figure Listing 6.2.2.

```
1  sum:
2       push  ebp                  ; entrée dans le sous-programme
3       mov   ebp, esp             ;
4       sub   esp, 4               ; création de la variable 'r'
5       push  ebx                  ; sauvegarde de ebx
6
7       mov   eax, [ebp + 8]       ; recupération de 'a'
8       mov   ebx, [ebp + 12]      ; recupération de 'b'
9       add   eax, ebx             ; calcul du résultat
10      mov   [ebp - 4], eax       ; stockage du résultat dans 'r'
11
12      mov   eax, [ebp - 4]       ; mise du résultat dans eax
13
14      pop   ebx                  ; restauration de ebx
15      mov   esp, ebp             ; sortie du sous-programme
16      pop   ebp                  ;
17      ret                        ;
```

Listing 6.2.2 – Appelé en 32 bits

L'entrée dans la fonction `sum` consiste à réaliser les trois étapes suivantes :

1. la première étape est l'entrée dans le sous programme (lignes 2 et 3), elle consiste à sauvegarder `ebp` puisque celui-ci va être utilisé pour accéder aux paramètres `a` et `b` ainsi que la variable locale `r`, puis on affecte à `ebp` la valeur de `esp`
2. on crée ensuite les variables locales si cela est nécessaire en réservant de l'espace dans la pile, ici il s'agit de la variable `r` qui est un entier 32 bits, soit 4 octets, on abaisse donc le sommet de pile de 4 octets (ligne 4)
3. on sauvegarde les registres dont la valeur doit être préservée pour la procédure appelante, ici c'est le cas pour `ebx` en ligne 5 qui sera utilisé pour stocker le paramètre `b`

Le rôle du registre **ebp** est primordial car c'est lui qui permet d'accèder aux paramètres et aux variables locales dès lors que l'on écrit `mov ebp, esp` en ligne 3. On peut voir Figure 6.1 la mise en correspondance entre **ebp** et les paramètres.

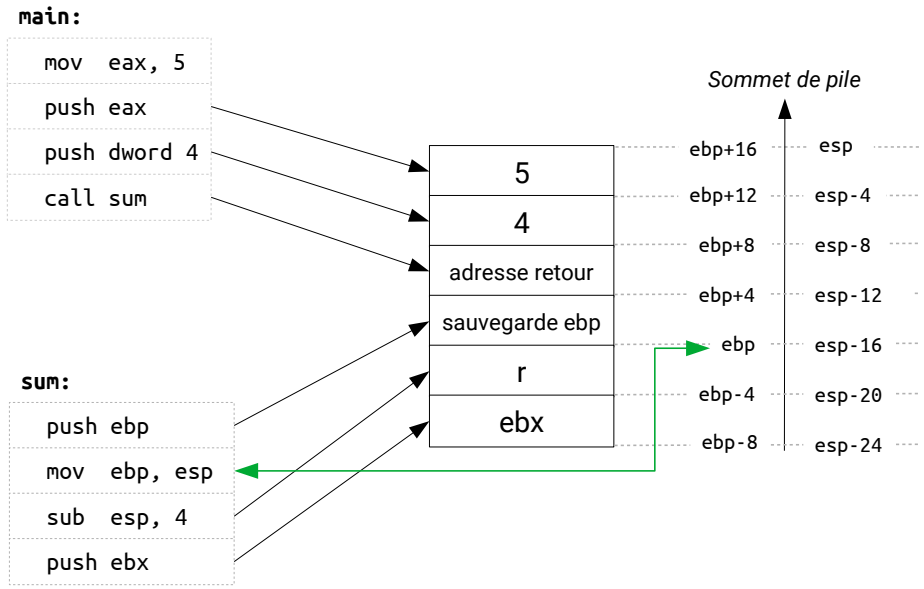


FIGURE 6.1 – Etat de la pile après entrée dans le sous-programme appelé

A l'intérieur de la fonction **sum** :

- le premier paramètre (**a**) est en `[ebp + 8]`
- le second paramètre (**b**) est en `[ebp + 12]`
- par extension le $n$ième paramètre, s'il est défini, est situé à l'adresse `[ebp +` $4 \times (n+1)$ `]`
- les variables locales sont stockées en `[ebp - x]` où `x` dépend du nombre de variables et de leurs types

Taille des paramètres

Que le paramètre soit un octet ou un mot, on le stockera sur 32 bits. S'il s'agit d'une chaîne de caractères ou d'un tableau on passera son adresse sur 32 bits. S'il s'agit d'une valeur 64 bits elle occupera 2 fois 32 bits.

On éxécute ensuite le corps de la fonction : on place le paramètre **a** dans **eax**, puis **b** dans **ebx** (lignes 7 et 8). On additionne ensuite **ebx** à **eax** et on stocke le résultat dans **r** (lignes 9 et 10).

La sortie de la fonction consiste à faire dans l'ordre inverse ce que l'on a fait lors de l'entrée (voir Figure 6.2) :

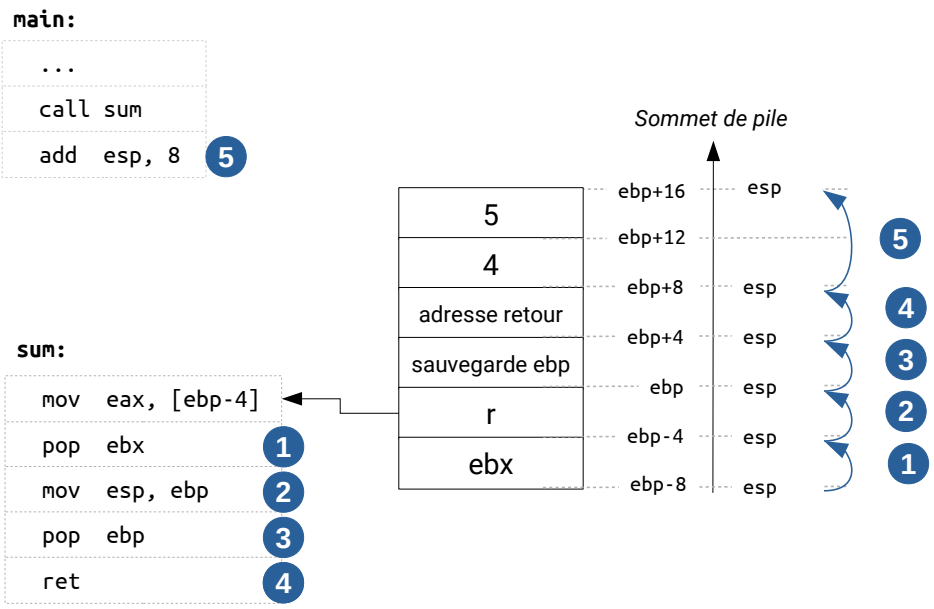


FIGURE 6.2 – Sortie du sous-programme appelé

1. on récupère le résultat, comme il s'agit d'une fonction entière on le place dans `eax` (ligne 12)

2. on restaure le registres `ebx` qui a été sauvegardé (ligne 14)

3. on rétablit `esp` à sa valeur quand on est entré dans la fonction (ligne 15) puis on restaure `ebp` (ligne 16)

4. enfin on exécute l'instruction `ret` qui récupère l'adresse de retour du sous-programme et exécutera l'instruction à cette adresse dans le sous-programme `main`

Définition : Ordinateur

Il faut remarquer que l'on n'a pas supprimé la variable locale `r` qui avait été créée par un `sub esp, 4`. Cela n'est pas nécessaire car l'étape 3 (de la fonction `sum`) qui consiste à rétablir `esp` le fait automatiquement.

On notera que la ligne 12 n'est pas utile puisque le résultat est déjà dans `eax` et, de plus, la variable locale `r` est également inutile puisque les calculs sont réalisés avec les registres `eax` et `ebx`. On notera que pour améliorer le temps d'exécution de la fonction, on pourrait également remplacer `ebx` par `ecx` car `ecx` est un registre modifiable et il n'est alors pas nécessaire de le sauvegarder puis le restaurer comme on l'a fait avec `ebx`.

6.2.5.3 Suppression des paramètres

Les paramètres mis dans la pile dans le sous-programme `main` doivent être supprimés car si ce n'était pas le cas on risquerait de saturer la pile. La manière la plus simple pour réaliser cela consiste à remonter le sommet de pile du nombre d'octets qui correspondent à l'espace occupé par les paramètres. En l'occurrence, on a placé deux entiers 32 bits dans la pile, donc 2×4 octets, on doit donc remonter le sommet de pile de 8 octets.

La convention du langage C impose que ce soit le sous-programme appelant qui supprime les paramètres. C'est pourquoi dans le code du sous-programme `main` nous avons placé une instruction `add esp, 8`, juste après le `call`.

Dans un langage comme le Pascal c'est l'inverse, c'est le sous-programme appelé qui supprime les paramètres. Dans le cas présent nous devrions écrire en Pascal une instruction `ret 8`, pour quitter le sous-programme `sum`. La valeur qui suit `ret` est forcément une constante. Nous faisons la même chose en langage C mais en reportant la suppression des paramètres dans la fonction appelante.

Pourquoi le langage C utilise t-il la convention qui impose au sous-programme appelant de supprimer les paramètres ? La réponse est simple, en C on a la possibilité de créer des fonctions qui peuvent prendre un nombre quelconque de paramètres comme par exemple `printf`. Etant donné que le sous-programme appelé ne connait pas le nombre de paramètres on ne peut pas utiliser l'instruction `ret` avec une valeur constante prédéfinie puisqu'elle varie en fonction du nombre de paramètres. Par contre le sous-programme appelant sait combien de paramètres il a mis dans la pile et il est donc en mesure de les supprimer.

Tout ceci peut paraître complexe mais il s'agit d'une mécanique très simple et il suffit de suivre à la lettre les règles que nous venons de voir.

6.2.6 Enter et leave

L'entrée dans un sous-programme et la sortie peuvent être réalisées également grâce à deux instructions assembleur nommées `enter` et `leave` :

```
1       push  ebp          ; enter 4, 0
2       mov   ebp, esp     ;
3       sub   esp, 4       ;
4
5       mov   esp, ebp     ; leave
6       pop   ebp          ;
7       ret
```

L'instruction `enter` remplace les lignes 1 et 3 du code précédent et `leave` les lignes 5 et 6. On pourra se référer à [19] pour de plus amples informations quant à ces instructions.

6.2.7 Appel rapide (*fast call*)

La manière dont on appelle classiquement un sous-programme est de type *cdecl*, c'est à dire *déclaration du langage C*. Il existe en 32 bits la possibilité d'appeler un sous-programme en utilisant un appel dit rapide, en anglais *fast call*. Un appel rapide signifie que l'on passe les paramètres dans les registres plutôt que de les passer dans la pile.

La raison derrière tout cela est qu'utiliser `ebp` pour accéder aux paramètres est pénalisant pour deux raisons :

- la première est que le sous-programme appelant place une première fois les paramètres dans la pile
- la seconde est que le sous-programme appelé relit les paramètres depuis la pile et utilise plusieurs instructions (`push ebp`; `mov esp, ebp`, etc.) qui servent uniquement à la relecture

Tout cela est finalement pénalisant et source de ralentissement. Lors d'un *fastcall* les données sont passées en 32 bits dans les registres `ecx` et `edx` respectivement. Si un sous-programme dispose de plus de deux paramètres, les premiers paramètres sont placés dans les registres évoqués et le reste des paramètres est placé dans la pile.

Si on désire écrire une fonction avec appel rapide en assembleur et l'appeler depuis un code source en langage C ou C++, il faut préciser dans le fichier C que la fonction est externe (c'est à dire définie dans un autre fichier) et qu'elle est de type *fastcall* grâce à la directive `__attribute__` :

```
1 // en C
2 extern int fonction(int a, int b) __attribute__((fastcall));
3
4 // en C++
5 extern "C" {
6       int fonction(int a, int b) __attribute__((fastcall));
7 }
```

Il existe également une autre possibilité afin d'améliorer l'efficacité des sous-programmes qui consiste à ne pas utiliser `ebp` mais à utiliser directement `esp`. C'est généralement ce que font les compilateurs lorsqu'ils génèrent du code optimisé. Malheureusement cela se révèle très compliqué à gérer lorsque l'on écrit soi-même de l'assembleur car dès qu'on place des données dans la pile, l'accès aux paramètres est décalé et il faut garder trace constamment de l'adresse des paramètres.

6.3 Appel de sous-programme en 64 bits

Afin d'écrire des fonctions en assembleur et de pouvoir les interfacer avec un programme C il est nécessaire de définir une convention d'appel, tout comme en 32 bits. Cette convention indique comment passer les paramètres aux sous-programmes appelés, quels registres devront être préservés dans les sous-programmes appelés et comment retourner les résultats. Il existe différentes conventions d'appel en 64 bits comme celle de Microsoft. En ce qui concerne Linux, on utilise la convention *System V AMD64 ABI* et c'est bien entendu celle que nous allons décrire ici.

Paramètre	64 bits	32 bits	16 bits	8 bits
1	`rdi`	`edi`	`di`	`dil`
2	`rsi`	`esi`	`si`	`sil`
3	`rdx`	`edx`	`dx`	`dl`
4	`rcx`	`ecx`	`cx`	`cl`
5	`r8`	`r8d`	`r8w`	`r8l`
6	`r9`	`r9d`	`r9w`	`r9l`

TABLE 6.2 – Ordre des registres entiers utilisés pour la convention d'appel C sous Linux

- la première différence avec la convention d'appel 32 bits est que les paramètres sont placés dans des registres et non dans la pile, il s'agit donc d'appels de type *fastcall*, cependant, s'il n'y a pas assez de registres on utilise la pile comme en 32 bits
- pour les paramètres entiers on dispose de 6 registres `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9` (cf. Table 6.2) que l'on utilise dans l'ordre indiqué
- les paramètres pour les nombres à virgule flottante sont passés dans la partie basse des registres `xmm0` à `xmm7` (cf. Chapitre 8)
- pour retourner une valeur entière, on la place dans `rax`
- pour retourner un flottant, on le place dans la partie basse de `xmm0`
- les registres qui ne doivent pas être modifiés par le sous-programme appelé sont `rbp`, `rbx`, `r12` à `r15`

On remarque donc qu'il y a un maximum de 14 paramètres (6 entiers, 8 réels) qui peuvent être passés dans des registres avec la convention d'appel 64 bits sous Linux.

6.3.1 Entrée et sortie de la fonction

Pour entrer dans une fonction on peut procéder comme en 32 bits en sauvegardant `rbp` puis en positionnant `rbp` sur `rsp` et lors de la sortie on effectuera les opérations inverses :

```
    push  rbp
    mov   rbp, rsp
    ...
    mov   rsp, rbp
    pop   rbp
    ret
```

Néanmoins si tous les paramètres d'un sous-programme sont placés dans des registres cela n'a aucune utilité et on peut donc faire l'économie de quatre instructions.

6.3.2 Red zone

En 64 bits, sous Linux, il est convenu que la zone mémoire de 128 octets située dans la pile de `[rsp-128]` à `[rsp]`, juste après l'appel d'un sous-programme, et que l'on qualifie de *red zone*, ne sera pas modifiée par les interruptions. On peut donc l'utiliser pour stocker temporairement des valeurs à condition de ne pas faire de `push` ou de `call` car ces instructions vont modifier cette zone. Notons que sous Windows cette zone n'est pas disponible.

6.3.3 Adresses

Les adresses doivent être précisées en utilisant les registres 64 bits uniquement. La manipulation de la pile par l'intermédaire de `push` et `pop` impose également d'utiliser des registres de 64 bits.

6.3.4 Exemple de traduction 64 bits

On considère la fonction C suivante qui calcule la somme des éléments t_i/a d'un tableau t avec a qui est une constante :

```
float sum(float *t, int n, float a) {
    float s = 0;
    for (int i = 0; i < n; ++i) {
        s = s + t[i] / a;
    }
    return s;
}
```

Nous donnons ci-après la traduction selon `g++` du code précédent en Listing 6.3.1 avec option d'optimisation `-O2`.

D'après ce que nous avons vu précédemment, lors de l'entrée dans le sous-programme le premier paramètre `t` est placé dans `rdi`, la taille du tableau `n` est dans `rsi` et la constante `a` est dans `xmm0`. On note que l'on jongle entre les registres

```
1  sum:
2          test    esi, esi                ; si n == 0 alors sortir
3          jle     .L4                     ; du sous-programme
4          lea     eax, [rsi + -1]         ; eax = n-1
5          pxor    xmm2, xmm2              ; s = xmm2 = [0, 0, ... 0]
6          lea     rax, [rdi + rax*4 + 4]  ; rax = tab + (n-1)*4 + 4
7  .L3:                                    ; en fait rax = tab + n*4
8          movss   xmm1, [rdi]             ; xmm1.ps[0] = tab[i]
9          add     rdi, 4                  ; ++tab
10         divss   xmm1, xmm0              ; xmm1.ps[0] /= a
11         addss   xmm2, xmm1              ; s += xmm1.ps[0]
12         cmp     rdi, rax                ; si &t[i] < &tab[N] alors boucler
13         jne     .L3
14         movaps  xmm0, xmm2              ; résultat placé dans xmm0
15         ret
16 .L4:
17         pxor    xmm2, xmm2              ; résultat mis à 0
18         movaps  xmm0, xmm2              ; et placé dans xmm0
19         ret
```

Listing 6.3.1 – Traduction en 64 bits de la fonction sum

32 bits et 64 bits. Par exemple la taille du tableau est un entier de type `int` donc 32 bits, il n'est donc pas nécessaire de considérer cette valeur comme une valeur 64 bits et par conséquent on se cantonne à utiliser `esi`.

Décrivons à présent le comportement de ce sous-programme. Les lignes 2 et 3 vérifient que la taille est bien supérieure à 0, si ce n'est pas le cas on saute en `.L4` pour mettre `xmm0` à 0 et sortir du sous-programme. Etrangement la mise à 0 utilise deux instructions (lignes 17 et 18) alors qu'une seule instruction `xorps xmm0, xmm0` aurait suffit. En ligne 4, on calcule dans `eax` le résultat de `rsi - 1`, c'est à dire `n-1`. De même que précédemment, on aurait pu écrire `lea eax, [esi - 1]`.

En ligne 5, on met `xmm2`, qui représente `s` à 0. On remarque qu'on a utilisé `pxor` qui est normalement destinée aux entiers et qu'il aurait fallu utiliser `xorps` destinée aux valeurs flottantes, mais les deux instructions produisent au final le même résultat à savoir mettre l'ensemble des bits du registre SSE à 0.

En ligne 6, on calcule l'adresse de fin du tableau c'est à dire `rdi + rsi*4`, cependant on réalise le calcul en deux étapes (lignes 4 et 6) et on calcule :

$$rdi + \underbrace{rax}_{rsi-1} * 4 + 4$$

que l'on stocke dans `rax`, ce qui est inutile puisqu'au final :

$$rdi + rax * 4 - 4 = rdi + (rsi - 1) * 4 + 4 = rdi + rsi * 4 - 4 + 4 = rdi + rsi * 4$$

On notera que l'adresse du dernier élément du tableau est `&t[n-1]` et l'adresse

de fin du tableau est `&t[n]`, c'est à dire l'adresse du dernier élément du tableau plus 4 octets.

Les lignes 7 à 13 réalisent la boucle `for` en utilisant `xmm1` pour charger `t[i]`. On augmente `rdi` de 4 octets à chaque itération de boucle pour passer au flottant suivant jusqu'à avoir atteint `rax` qui est l'adresse de fin du tableau.

Enfin, lignes 14 à 15, on met le résultat `s` stocké dans `xmm2` dans `xmm0` qui est le registre qui contient la valeur de retour de la fonction (puisqu'il s'agit d'une valeur en virgule flottante), puis on sort du sous-programme.

Nous avons noté beaucoup de bizzareries dans cette traduction automatique et on peut la réécrire de manière plus concise comme présenté Listing 6.3.2.

```
1  sum:
2          xorps   xmm2, xmm2          ; s = 0
3          test    esi, esi            ; si n <= 0 alors sortir
4          jle     .end                ; du sous-programme
5          lea     rax, [rdi + rsi*4]  ; rax = &tab[N]
6  .while:
7          movss   xmm1, [rdi]         ; xmm1.ps[0] = tab[i]
8          add     rdi, 4              ; ++tab
9          divss   xmm1, xmm0          ; xmm1.ps[0] /= a
10         addss   xmm2, xmm1          ; s += tab[i] / a
11         cmp     rdi, rax            ; si &t[i] < &tab[N] alors boucler
12         jne     .while
13 .end:
14         movaps  xmm0, xmm2          ; résultat placé dans xmm0
15         ret
```

Listing 6.3.2 – Traduction en 64 bits de la fonction sum - version améliorée

Le lecteur aura compris que le calcul peut être simplifié puisque a est une constante :

$$\sum \frac{t[i]}{a} = \frac{1}{a} \sum t[i]$$

On pourra donc sortir l'instruction de division de la boucle `.while`.

6.3.5 Spécificités du mode 64 bits

6.3.5.1 With Respect To (WRT)

L'appel de fonctions externes, comme la fonction `printf`, doit être réalisé en utilisant le mécanisme de PLT (*Procedure Linkage Table*). Ce mécanisme permet l'appel de fonctions dont l'adresse n'est pas connue lors de l'édition de liens et qui sera résolu par l'éditeur de lien dynamique lors de l'exécution.

Il faut alors suffixer les fonctions par `WRT ..plt` où WRT signifie *With Respect to* :

```
call    printf WRT ..plt
```

Cela n'est pas nécessaire pour les fonctions écrites en assembleur par l'utilisateur à moins de faire appel à ces fonctions depuis un autre fichier assembleur.

6.3.5.2 Position Independent Code

En outre, en 64 bits, on utilise généralement le PIC (*Position Independent Code*) qui est simple à mettre en œuvre car on se base sur l'adressage relatif par rapport au registre **`rip`**, le pointeur d'instruction.

De la même manière, le PIE pour *Position Independent Executable* est une fonction de sécurité qui permet aux exécutables d'être chargés à des adresses mémoire aléatoires à chaque fois qu'ils sont exécutés, ce qui peut aider à prévenir certains types d'attaques, telles que la programmation orientée retour (ROP) et certaines formes d'attaques par débordement de mémoire tampon.

Par défaut, certaines distributions modernes de `gcc` génèrent des exécutables indépendants de la position. Cela est utile pour les bibliothèques partagées, qui peuvent être chargées à différentes adresses dans différents programmes. Cependant, cela peut poser un problème pour certains programmes de bas niveau, comme certains programmes d'assemblage, qui peuvent dépendre d'un chargement à une adresse spécifique.

Dans le cadre de la programmation assembleur que nous voyons dans cet ouvrage, il est plus simple de ne pas utiliser cette fonctionnalité car elle est difficile à mettre en oeuvre manuellement. Un compilateur sera plus à même d'automatiser cette tâche.

C'est pourquoi il est nécessaire, pour l'écriture de notre code, d'utiliser lors de l'édition de lien avec `gcc`, l'option `-no-pie` afin d'éviter que le compilateur ne se plaigne du fait que le code n'est pas de type PIC. Il en résulte que le code sera toujours chargé à la même adresse en mémoire.

Par défaut, avec `nasm` l'adressage est absolu, c'est le mode qui nous convient. Si on désire passer à l'adressage relatif il faut, en début du fichier assembleur, ajouter la directive

```
default rel
```

6.3.5.3 Alignement de la pile

Un autre point important qui est à prendre en considération est lié à l'appel des fonctions de la bibliothèque C. Les conventions d'appel en 64 bits imposent qu'avant

l'appel d'un sous-programme, la pile (donc **rsp**) contienne une adresse multiple de 16 octets. Lorsque l'on entre dans un sous-programme, on a préalablement réalisé un **call**, qui en 64 bits place 8 octets dans la pile. Ces 8 octets correspondent à l'adresse de retour de sous-programme qui sera utilisée par l'instruction **ret** afin de passer à l'instruction suivant le **call** dans le sous-programme appelant. Cela implique qu'une fois dans le sous-programme appelé, l'adresse de **rsp** est multiple de 8.

Si on écrit un sous-programme qui fait appel à une fonction de la bibliothèque C, il est nécessaire de rendre **rsp** multiple de 16 avant l'appel à cette fonction. Pourquoi, me direz vous ? Normalement cela ne devrait pas poser de problème. Un appel à `printf` devrait être indifférent au fait que **rsp** soit multiple de 8 ou de 16. Mais pour certaines fonctions ou dans certaines version de la `glibc` se produit une erreur de segmentation ! Ce qui semble bien étrange mais dont l'explication réside dans l'implantation de la fonction.

Regardons ce qui se passe au niveau du code de la fonction `scanf` par exemple, en installant les sources de la bibliothèque C, version 2.31, comme suit :

```
1 sudo apt-get install libc-dbg
2 sudo apt-get install glibc-source
3 sudo tar xvf /usr/src/glibc/glibc-2.31.tar.xz -C ~
```

Avec la dernière commande (`tar`), on installe les sources dans le répertoire *home* de l'utilisateur.

On peut alors lancer le débogueur `gdb` sur un petit programme, ici appelé `prog.exe`, qui fait un simple appel à `scanf`. On obtient bien une erreur de segmentation (ligne 6). Avant de lancer le programme, on indique grâce à la commande `directory` (ligne 2) dans quel répertoire se trouvent les sources de la bibliothèque C :

```
1  > gdb prog.exe
2  (gdb) directory /home/richer/glibc-2.31/
3  (gdb) run
4  Starting program: /home/richer/prog.exe
5
6  Program received signal SIGSEGV, Segmentation fault.
7  0x00007ffff7e0da86 in __vfscanf_internal (s=0x7ffff7f96980 <_IO_2_1_stdin_>,
8      format=0x555555558023 <msg_scanf> "%d", argptr=argptr@entry=0x7fffffffd748,
9      mode_flags=mode_flags@entry=0) at vfscanf-internal.c:339
10 339        vfscanf-internal.c: Aucun fichier ou dossier de ce type.
11 (gdb) bt
12 #0  0x00007ffff7e0da86 in __vfscanf_internal (s=0x7ffff7f96980 <_IO_2_1_stdin_>,
13     format=0x555555558023 <msg_scanf> "%d", argptr=argptr@entry=0x7fffffffd748,
14     mode_flags=mode_flags@entry=0) at vfscanf-internal.c:339
15 #1  0x00007ffff7e0c20f in __scanf (format=<optimized out>) at scanf.c:38
16 #2  0x0000555555555173 in main () at prog.asm:87
```

Grâce à la commande `x/i`, on affiche la dernière instruction exécutée et donc celle qui a provoqué l'erreur de segmentation :

```
(gdb) x/i $pc
=> 0x7ffff7e0da86 <__vfscanf_internal+134>:        movaps %xmm1,-0x600(%rbp)
```

L'instruction est affichée au format AT&T, elle correspond en fait, sous format Intel, à l'instruction :

```
movaps [rbp-0x600], xmm1
```

qui sauvegarde le contenu du registre vectoriel **xmm1** dans la pile. L'instruction **movaps** impose que l'adresse de destination soit multiple de 16 (voir le Chapitre 8).

Si on affiche le contenu de **rbp**, on obtient :

```
(gdb) info reg rbp
rbp            0x7fffffffd738      0x7fffffffd738
```

Soit, si on s'intéresse au trois derniers chiffres hexadécimaux à $738_{16} - 600_{16} = 138_{16}$, donc une adresse multiple de 8, d'où l'erreur de segmentation. Attention, ici il s'agit de **rbp** et non **rsp**, mais **rbp** a été mis à jour en fonction de **rsp** comme on peut le faire en 32 bits.

Pour résoudre le problème, il suffit, dans le sous-programme qui réalise l'appel à `scanf`, de remettre **rsp** à une valeur multiple de 16.

Soit en plaçant un registre dans la pile :

```
push rbp    ; ou tout autre registre 64 bits
```

Soit en abaissant le sommet de pile de 8 octets :

```
sub  rsp, 8
```

Soit en alignant le sommet de pile sur un multiple de 16 inférieur à la valeur actuelle de **rsp** :

```
and  rsp, ~0xF
```

Il faudra bien évidemment supprimer ces octets de la pile avant de sortir du sous-programme. On obtient, par exemple, le code suivant pour lire un entier en 64 bits :

```
1  extern scanf
2
3  section .data
4
5          s: db "%d", 0
6          a: dd 0
7
8  section .text
9
10 ; scanf("%d", &a)
11 my_call_to_scanf:
12         sub   rsp, 8            ; rsp multiple de 16
13
14         lea   rdi, [s]          ; premier paramètre
15         lea   rsi, [a]          ; second paramètre
16         mov   eax, 0            ; pas de flottant traité
17         call scanf WRT ..plt
18
19         add rsp, 8              ; on supprime les octets utilisés
20                                 ; pour l'alignement
21         ret
```

Le désassemblage du code du sous-programme précédent compilé avec un adressage absolu sous forme de fichier objet (`.o`), donne :

```
1  > objdump -d -j .text a.o --show-raw-insn
2  ...
3  my_call_to_scanf():
4     0:        48 83 ec 08                 sub    rsp,0x8
5     4:        48 8d 3c 25 00 00 00        lea    rdi,ds:0x0
6     b:        00
7                           8: R_X86_64_32S        .data
8     c:        48 8d 34 25 00 00 00        lea    rsi,ds:0x0
9    13:        00
10                          10: R_X86_64_32S       .data+0x3
11    14:       b8 00 00 00 00              mov    eax,0x0
12    19:       e8 00 00 00 00              call   1e <my_call_to_scanf+0x1e>
13                          1a: R_X86_64_PLT32      scanf-0x4
14    1e:       48 83 c4 08                 add    rsp,0x8
15    22:       31 c0                       xor    eax,eax
16    24:       c3                          ret
```

Le même code généré avec l'approche PIC ou adressage realtif donne :

```
1  > objdump -d -j .text a.o --show-raw-insn
2  ...
3  0000000000000000 <my_call_to_scanf>:
4  my_call_to_scanf():
5     0:        48 83 ec 08                 sub    rsp,0x8
```

```
 4:        48 8d 3d 00 00 00 00         lea    rdi,[rip+0x0] # b <my_call_to_scanf+0xb>
                     7: R_X86_64_PC32        .data-0x4
 b:        48 8d 35 00 00 00 00         lea    rsi,[rip+0x0] # 12 <my_call_to_scanf+0x12>
                     e: R_X86_64_PC32        .data-0x1
12:        b8 00 00 00 00               mov    eax,0x0
17:        e8 00 00 00 00               call   1c <my_call_to_scanf+0x1c>
                    18: R_X86_64_PLT32        scanf-0x4
1c:        48 83 c4 08                  add    rsp,0x8
20:        31 c0                        xor    eax,eax
22:        c3                           ret
```

On voit apparaître l'adressage relatif par rapport à **rip** qui a été ajouté par nasm. On note également que certains instructions qui n'utilisent pas le PIC sont plus longue d'un octet.

6.3.5.4 Entrée et sortie de sous-programme en 64 bits

Pour l'entrée dans un sous-programme en 64 bits, je recommande d'utiliser un fonctionnement du type 32 bits en passant par **rbp**.

1. on commence par sauvegarder **rbp**
2. on place **rsp** dans **rbp**
3. on sauvegarde les registres que l'on désire ou que l'on doit préserver s'ils sont modifiés
4. on aligne le sommet de pile **rsp** sur un multiple de 16

Dès lors, on peut au travers de **rbp**, accéder aux valeurs des registres sauvegardés dans la pile.

Pour la sortie de sous-programme, on procède de la manière suivante :

1. on récupère les valeurs des registres sauvegardés grâce à **rbp**
2. on place **rbp** dans **rsp**
3. on dépile **rbp**
4. on termine par **ret**

Afin de simplifier l'écriture du code, on peut définir deux macro-instructions en nasm :

- `defsp` pour *define sub program*
- `endsp` pour *end sub program*

La première macro-instruction comporte 1 à plusieurs arguments (`1-*`) dont le premier est le nom du sous-programme. Il est suivi par éventuellement d'autres

arguments qui sont, soit des noms de registres généraux 64 bits, considérés comme des identifiants, soit des entiers positifs. S'il s'agit d'un registre général, il sera empilé, sinon on décrémentera le sommet de pile de la quantité indiquée. Cet espace pourra servir à stocker des variables temporaires.

```
1  %macro defsp 1-*
2  %1:
3      %rotate 1
4      push    rbp
5      mov     rbp, rsp
6      %rep %0 - 1
7          %ifnum %1
8              sub     rsp, %1
9          %else
10             push %1
11         %endif
12         %rotate 1
13     %endrep
14     and     rsp, ~0xF
15 %endmacro
```

L'instruction `%rep`, de la ligne 6, permet de répéter `%0 - 1` fois, c'est à dire $n-1$ fois les instructions qui apparaissent jusqu'à `%endrep`. Ici, n est égal à `%0` qui représente le nombre d'arguments de la macro-instruction. On utilise l'instruction `%rotate 1` qui permet de passer à l'argument suivant en considérant que les arguments sont dans une liste sans fin. Par exemple, `%rotate %0` nous ramène sur le premier argument.

On termine ensuite la macro en rendant le sommet de pile multiple de 16 (ligne 14).

L'appel de cette macro peut être réalisé de la sorte :

```
1  defsp main, rdi, rsi, 100, rbx
```

On note que les paramètres sont séparés par des virgules. Dans ce cas précis, la macro-instruction permettra de générer le code suivant :

```
1  main:
2      push    rbp
3      mov     rbp, rsp
4      push    rdi
5      push    rsi
6      sub     rsp, 100
7      push    rbx
8      and     rsp, ~0xF
```

La seconde macro-instruction qui gére la sortie du sous-programme fonctionne sur le même modèle :

```
%macro endsp 1-*
end_%1:
    %rotate 1
    %assign i 8
    %rep %0 - 1
        %ifnum %1
            %assign i i + %1
        %else
            mov     %1, [rbp - i]
            %assign i i + 8
        %endif
        %rotate 1
    %endrep
    mov     rsp, rbp
    pop     rbp
    ret
%endmacro
```

Elle permet de récupérer les valeurs mises dans la pile. On y fait appel de la même manière que pour `defsp` :

```
endsp main, rdi, rsi, 100, rbx
```

On obtient alors le code suivant :

```
end_main:
    mov     rdi, [rsp - 8]
    mov     rsi, [rsp - 16]
    mov     rbx, [rsp - 124]
    mov     rsp, rbp
    pop     rbp
    ret
```

6.4 Code en 32 ou 64 bits

On peut se demander s'il est préférable de compiler son code en 32 ou 64 bits. A l'heure où nous écrivons cet ouvrage les fournisseurs de systèmes d'exploitation commencent à abandonner le support 32 bits. Il est toujours possible de compiler du code 32 bits sur un système 64 bits mais cela requiert d'installer des librairies spécifiques : avec `gcc` notamment, le package `multilib`.

Un code C compilé en 64 bits s'exécute normalement plus rapidement que du code 32 bits parce que le passage des paramètres des sous-programmes se fait dans les registres et que l'on dispose de plus de registres de calcul pour stocker des résultats temporaires [1]. On dispose de plus de registres en 64 bits ce qui permet

1. Cependant, dans certains cas, un programme 32 bits peut être plus rapide qu'un programme compilé en 64 bits.

de stocker plus de données temporaires dans les registres et faire moins d'appels à la mémoire pour certains traitements, on est donc sensé gagner en efficacité. L'écriture du code est également simplifiée mais plus contraignante qu'en 32 bits.

Il semble donc préférable d'écrire le code assembleur uniquement en 64 bits et compiler ses programmes en 64 bits dorénavant.

6.5 Conclusion

6.5.1 Que retenir ?

- ▷ en architecture 32 bits les paramètres des sous-programmes sont passés dans la pile. Les valeurs de retour des fonctions sont passées dans le registre eax s'il s'agit d'un entier ou d'un pointeur ou dans le sommet de pile de la FPU s'il s'agit d'un nombre à virgule flottante
- ▷ en architecture 64 bits les paramètres des sous-programmes sont passés dans les registres généraux (`rdi`, `rsi`, `rdx`, `rcx`) s'il s'agit d'entiers ou de pointeurs et dans les registres SSE s'il s'agit de nombres à virgule flottante. Les valeurs de retour des fonctions sont passées dans le registre rax s'il s'agit d'un entier ou d'un pointeur ou dans xmm0 s'il s'agit d'un nombre à virgule flottante
- ▷ les conventions d'appel en architecture 32 et 64 bits étant différentes il est très souvent nécessaire de modifier le code assembleur pour passer d'une architecture à l'autre.

6.5.2 Compétence à acquérir

Il faut être en mesure de :

- ☐ réaliser un appel de sous-programme en 32 bits
- ☐ récupérer les arguments d'un sous-programme écrit en 32 bits
- ☐ réaliser un appel de sous-programme en 64 bits
- ☐ savoir dans quels registres se trouvent les arguments d'un sous-programme écrit en 64 bits

6.6 Exercices

Exercice 27 - Réaliser le codage du sous-programme suivant en 32 bits, puis en 64 bits :

```
float procedure(int *tab, int n) {
    float sum = 0;
    for (int i = 0; i < n; ++i) {
        tab[i] = tab[i] / 2;
        sum += tab[i] * 1.25;
    }
    return sqrt(sum);
}
```

Exercice 28 - Réaliser le codage du sous-programme suivant en 32 bits, puis en 64 bits :

```
double procedure(double *tab, int n, double k) {
    double sum = 0;
    for (int i = 0; i < n; ++i) {
        sum += tab[i] / k;
    }
    return sqrt(sum);
}
```

Chapitre 7

Coprocesseur arithmétique

Quelle prétention de dire que l'informatique est récente,
Adam et Eve avaient déjà un Apple !
Anonyme.

7.1 Introduction

Dans ce chapitre nous allons voir comment utiliser le coprocesseur arithmétique afin de réaliser des calculs avec des nombres à virgule flottante. L'ensemble des instructions du coprocesseur commencent par la lettre `'f'` pour *Floating point value*. Le coprocesseur était absent sur les premiers microprocesseurs de la famille x86 mais on pouvait ajouter un circuit externe sur la carte mère chargé de faire les calculs des nombres en virgule flottante. Ce circuit a donc été nommé *coprocesseur* dans la même veine que coéquipier, c'est à dire celui qui vient aider pour réaliser une tâche. Le premier coprocesseur pour l'Intel 8086 fut l'Intel 8087. D'autres coprocesseurs furent produits pour les microprocesseurs suivants dans la gamme x86 : 80287, 80387, 80487, jusqu'à l'intégration du coprocesseur au sein du microprocesseur à partir des Intel 80486DX. On ne parle donc plus à présent de coprocesseur mais plutôt de FPU pour *Floating Point Unit*, comme nous l'avons vu dans le Chapitre 3.

7.2 Organisation de la FPU

La FPU est composée des éléments suivants (cf. Figure 7.1) :

- une pile de 8 registres (*Registers Stack*)
- le registre `opcode` qui contient le code de la dernière instruction exécutée
- le registre de `statut` qui contient le sommet de pile, les exceptions et les flags

- le registre de `contrôle` qui contrôle la précision et l'arrondi des méthodes de calcul
- le registre d'`étiquette` (*Tag Register*) indique le contenu (valid, 0, NaN, infini) de chaque registre du coprocesseur
- le registre dit pointeur de dernière instruction (*Last Instruction Pointer*) qui pointe sur la dernière instruction exécutée
- le registre dit pointeur de dernière donnée (*Last Data Pointer*) qui pointe vers l'opérande de la dernière instruction exécutée

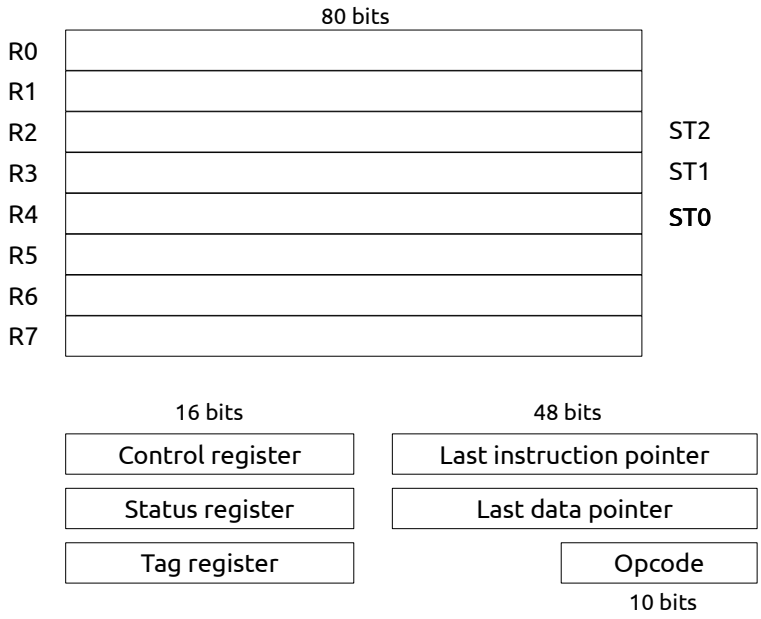


FIGURE 7.1 – Registres du coprocesseur

7.3 Manipulation des données et de la FPU

Le coprocesseur comporte 8 registres notés R0 à R7 mais qui ne sont pas manipulables directement. On y accède au travers d'une pile dotée de 8 registres appelés **`st0`** à **`st7`** que ce soit en architecture 32 ou 64 bits.

Dès que l'on charge une nouvelle donnée dans **`st0`**, les données déjà stockées dans **`st0`** à **`st6`** sont *déplacées* vers **`st1`** à **`st7`**. En réalité, la pile du coprocesseur peut être vue comme une liste, on recule donc le pointeur de sommet de pile et cela revient à ajouter la nouvelle valeur en début de liste.

Les registres **`st0`** à **`st7`** occupent chacun 80 bits ce qui permet de disposer d'une grande précision de représentation afin de réaliser des calculs *justes* comparativement aux nombres en simple ou double précision. L'exposant occupe alors 15 bits et la mantisse 64 bits.

Comme nous le verrons ci-après, si on charge plus de huit valeurs dans la pile du coprocesseur, on génère une exception. Il faut donc prendre l'habitude de ne laisser qu'une valeur dans `st0` qui correspond au réstulat du dernier calcul, puis la supprimer lorsqu'on n'en a plus besoin.

7.3.1 Chargement avec fld

Le chargement des données se fait grâce à l'instruction `fld`, pour *Floating point LoaD*, en précisant la quantité chargée. Par exemple :

- `dword` pour un flottant en simple précision
- `qword` pour un flottant en double précision

Il existe également une instruction `fild` pour *Floating point Integer LoaD* qui permet de charger une valeur entière qui sera convertie en nombre en simple ou double précision.

Attention

On ne peut pas charger une donnée depuis un registre général, uniquement depuis la mémoire ou à partir d'un autre registre du coprocesseur :

```
1  section .data
2
3      a:   dd  1.25              ; float a = 1.25
4      b:   dq  3.75              ; double b = 3.752567871
5      c:   dd  31                ; int c = 31
6      d:   dq  123               ; long int d = 123
7
8  section .text
9
10     fld dword [a]
11     fld qword [b]
12     fld st1
13     fld st2
14     fild dword  [c]
15     fild qword  [d]
16     ret
```

Nous pouvons voir Table 7.1 les effets du chargement des données du programme assembleur précédent. A mesure que l'on charge de nouvelles données les précédentes sont *déplacées* dans la pile du coprocesseur.

Afin de faciliter l'écriture des calculs, un certain nombre de constantes sont prédéfinies et peuvent être chargées en utilisant le mnémonique adéquat (cf. Table 7.2). On pourra donc charger les constantes 0 et $1,0$. Mais si on désire utiliser $2,0$, il faudra stocker cette donnée en mémoire puis la charger dans `st0`.

Instruction	st0	st1	st2	st3	st4	st5	st6	st7
`fld dword [a]`	1.25	?	?	?	?	?	?	?
`fld qword [b]`	3.75	1.25	?	?	?	?	?	?
`fld st1`	1.25	3.75	1.25	?	?	?	?	?
`fld st2`	1.25	1.25	3.75	1.25	?	?	?	?
`fild dword [c]`	31	1.25	1.25	3.75	1.25	?	?	?
`fild qword [d]`	123	31	1.25	1.25	3.75	1.25	?	?

TABLE 7.1 – Effets du chargement de valeurs dans le coprocesseur

Instruction	Constante
`fldz`	0.0
`fld1`	1.0
`fldpi`	π
`fldl2e`	$log(e) = 1.442695$
`fldl2t`	$log_2(10) = 3.312928$
`fldlg2`	$log_{10}(2) = 0.3102999$
`fldln2`	$log_e(2) = 0.693147$

TABLE 7.2 – Constantes prédéfinies du coprocesseur (valeurs approchées)

7.3.2 Stockage avec fst

Le stockage fonctionne comme le chargement, on indique l'emplacement mémoire au format `dword` ou `qword` vers lequel on désire stocker la valeur en sommet du coprocesseur.

Attention

La plupart des instructions que nous allons voir par la suite disposent d'un suffixe formé de la lettre p. C'est le cas pour `fst, fstp`. Ajouter le suffixe `p` signifie qu'on dépile (pop) le résultat.

Voyons cela sur un exemple :

```
1  section .data
2      a:  dd  0
3      b:  dq  0
4
5  section .text
6      fld1
7      fst   dword [a]
8      fstp  qword [b]
```

Pour le code précédent, on commence par charger la valeur 1.0 dans `st0` (ligne 6), puis à la ligne 7, on stocke cette valeur dans la variable `a` sous forme d'un flottant simple précision. A la ligne 8 ,on stocke la valeur contenue dans `st0` qui est toujours 1.0 en `b` sous forme d'une valeur flottante 64 bits, puis on la supprime de `st0`. On se retrouve finalement avec une pile vide.

7.4 Opérations

Nous donnons à présent une liste non exhaustive des opérations que l'on peut utiliser avec la FPU.

7.4.1 Opérations de base

En ce qui concerne les opérations arithmétiques de base que l'on peut utiliser, on dispose de :

- `fadd, faddp` pour l'addition `st0 = st0 + src`
- `fsub` pour la soustraction `st0 = st0 - src`
- `fsubr` pour la soustraction inverse `st0 = src - st0`
- `fmul, fmulp` pour la multiplication
- `fdiv, fdivp, fdivr, fprem` pour la division `st0 = st0 / src`
- `fdivr` pour la division inverse `st0 = src / st0`
- `fprem` reste de la division

Ces opérations peuvent être suffixées par la lettre `p` pour dépiler la valeur au sommet de la pile du coprocesseur. Elles fonctionnent toutes sur le même modèle, par exemple pour `fadd` nous donnons Table 7.3 les différentes variantes de l'instruction sachant que le registre `st0` représente le sommet de pile et `sti` représente l'un des 7 autres registres soit `st1` à `st7`.

Instruction	Interprétation
`fadd [mem]`	`st0 += [mem]`
`fadd sti`	`st0 += sti`
`fadd sti, st0`	`sti += st0`
`faddp sti`	`sti += st0`, puis `st0` est dépilé
`faddp sti, st0`	identique à l'instruction précédente
`fiadd [mem]`	`st0 += (float) [mem]`, où `[mem]` est un entier

TABLE 7.3 – Description de l'instruction `fadd`

Attention

On note que **fadd sti** et **faddp sti** ont un comportement différent alors qu'elle ne devrait normalement différer que d'un *pop* :

- **fadd sti** : `st0 += sti`
- **faddp sti** : `sti += st0`, puis `st0` est dépilé

On dispose également d'opérations comme :

- **fabs** pour le calcul de la valeur absolue
- **fchs** changement de signe
- **fsqrt** calcul de la racine carrée
- **fscale** calcul de $st0 = st0^{st1}$
- **f2xm1** : $st0 = 2^{st0} - 1$
- **fyl2x** : $st0 = st1 \times log2(st0)$
- **fyl2xp1** : $st0 = st1 \times log2(st0 + 1)$
- **fxtract** extrait l'exposant du nombre stocké dans **st0**

7.4.2 Opérations trigonométriques

Pour appliquer les opérations trigonométriques, on utilise les fonctions suivantes qui s'appliquent uniquement sur **st0** qui représente une mesure d'angle exprimée en radians :

- **fcos, fsin, fsincos, fptan, fpatan** pour le cosinus
- **fsin** pour le sinus
- **fptan** calcul de la tangente (partielle)
- **fpatan** calcul de l'arctangente (partielle)
- **fsincos** calcul du sinus et cosinus, **st0** = cos(**st0**), **st1** = sin(**st0**)

Par exemple, le code qui suit commence par convertir un angle de 60° en radians, puis applique la fonction **fsincos** :

```
1  section .data
2      angle:  dd  60      ; degrés
3      cqv:    dd 180      ; degrés
4
5  section .text
6      ; calcul de pi * 60 / 180 pour avoir la mesure en radians
7      fild    dword [angle] ; st0 = 60.0
8      fild    dword [cqv]   ; st0 = 180.0  st1 = 60.0
9      fdivp   st1, st0      ; st1 = 60.0 / 180.0, puis pop st0 = 0.3333...
10     fldpi                 ; st0 = pi, st1 = 0.3333...
11     fmulp   st1, st0      ; st0 = pi * 0.3333...
12     fsincos
```

On obtient donc dans `st0` la valeur 0.5 et dans `st1` la valeur 0.86 qui correspondent respectivement au cosinus et sinus d'un angle de 60 degrés.

7.4.3 Manipulation de la pile de la FPU

La FPU n'a pas directement accès au flux d'instructions du microprocesseur. Celui-ci lui transmet les intructions qui la concernent. Pendant que la FPU exécute les instructions qui lui sont envoyées, le microprocesseur peut continuer à exécuter d'autres instructions qui n'agissent pas sur les flottants car certaines instructions de la FPU peuvent être relativement lentes en comparaison des instructions qui agissent sur les entiers.

Le microprocesseur garde néanmoins la main et est en mesure de lire ou d'écrire les registres de statut et de contrôle de la FPU. Dans certains cas il est préférable d'accèder à ces registres sans attendre, par contre d'en d'autres situations il est nécessaire d'attendre que la FPU ait terminé le calcul de l'instruction en cours d'exécution.

Les instructions suivantes permettent de manipuler la pile du coprocesseur soit de manière locale en modifiant un registre (`ffree`) soit de manière globale (`fsave`, `frstor`). Certaines de ces instructions commencent par les lettres `fn`, la lettre `n` signifiant *no-wait*.

- `fwait` synchronisation des exceptions avant de passer à la prochaine instruction
- `finit` initialise ou réinitialise le coprocesseur
- `fclex`, `fnclex` supprime les exceptions qui auraient été levées
- `fsave`, `fnsave` sauve l'état et les registre du coprocesseur, soit 108 octets
- `frstor` restaure l'état et les registres du coprocesseur
- `ffree` libère un registre

7.5 Erreurs liées à la FPU

Deux types d'erreurs peuvent se produire lors de calculs avec les nombres flottants :

- les erreurs de calcul : calcul d'une racine carrée négative, débordement, division par zéro
- les erreurs liées à la gestion de la pile de la FPU : pile pleine, pile vide

Pour gérer l'ensemble des erreurs, la FPU utilise deux registres appelés **registre de statut** dont la description est donnée Table 7.4, ainsi qu'un **registre de contrôle**

Bit	Identifiant	Description	Exception
0	`FPU_IE`	Opération Invalide	oui
1	`FPU_DE`	Opérande dénormalisée	oui
2	`FPU_ZE`	Division par Zéro	oui
3	`FPU_OE`	Débordement (Overflow)	oui
4	`FPU_UE`	Débordement (Underflow)	oui
5	`FPU_PE`	Précision	oui
6	`FPU_SF`	Erreur Pile (Stack Fault)	
7	`FPU_ES`	Résumé Erreur	
8	`FPU_C0`	$C0$ (Retenue)	
9	`FPU_C1`	$C1$ (Débordement)	
10	`FPU_C2`	$C2$ (Parité)	
11-13	-	Sommet de pile	
14	`FPU_C3`	$C3$ (Zéro)	
15	`FPU_B`	`Busy Bit`	

TABLE 7.4 – Description du registre de statut de la FPU

que nous ne détaillerons pas car le registre de statut est suffisant pour traiter les erreurs qui nous intéressent.

Pour le registre de statut les bits 0 à 5 correspondent à des exceptions qui peuvent être interceptées au niveau d'un programme C en utilisant un gestionnaire de signal (*Signal Handler*). Le signal levé est `SIGFPE` soit *Signal Floating Point Exception*.

Les bits 11 à 13 codent sur 3 bits le sommet de pile, celui-ci est initialement à 7 puis descend jusqu'à 0. Si on place plus de 8 valeurs dans la pile de la FPU on génère une erreur `FPU_SF` et les prochaines valeurs chargées dans la FPU seront remplacées par $-NaN$ (moins *Not a Number*).

On trouvera sous Linux dans le fichier `/usr/include/fenv.h` les signaux liés à la FPU et on pourra consulter le fichier `signal_handler.cpp` dans les sources des études de cas car il gère les différents signaux qui peuvent être levés grâce à la classe `SignalHandler`. On consultera également la méthode `main` qui appelle la classe `SignalHandler` dont le constructeur a pour but d'intercepter les signaux principaux.

Enfin, les bits qui correspondent aux identifiants `C0` à `C3` sont des bits dits de condition (*Condition Flags*) qui sont calqués sur les bits du registre des `flags`. Ces bits qui sont utilisés lors des comparaisons peuvent être copiés dans le registre des `flags` mais il faut utiliser la série d'instructions suivantes pour réaliser cette opération ce copie nécessaire pour certaines instructions de comparaison :

```
    fstsw ax
    sahf
```

7.6 Comparaison

Il existe différentes instructions pour la comparaison de valeurs flottantes. On pourra consulter la documentation de `ftst`, `fcom`, `fcomp`, `fcompp`, `fucom`, `fucomp`, `fucompp` dans la documentation Intel.

Les instructions un peu plus intéressantes pour le développeur sont `fcomi`, `fcomip`, `fcomu`, `fcomup`, `fcomip`, `fcucomi` et `fcuomip` car elles réalisent la comparaison de deux valeurs et mettent directement à jour le registre `flags` à partir des bits de condition de la FPU.

7.6.1 Comparaison en architecture 32 bits

Par exemple pour comparer deux valeurs flottantes :

```
1 int compare_32bits(float x, float y) {
2     if (x > y) {
3         return 1;
4     } else {
5         return 3;
6     }
7 }
```

En 32 bits, on commence par charger `y`, puis `x`. On compare ensuite `x` à `y` grâce à l'instruction `fcomip` qui réalise la comparaison et dépile `x` de `st0`. On dépile ensuite `y` (ligne 8). On utilise l'instruction de branchement conditionnel `jbe` (*Jump Below or Equal*) pour exécuter le `.else` dans le cas où $x \leq y$.

```
1  compare_32bits:
2      push    ebp
3      mov     ebp, esp
4      fld     dword [ebp+12]  ; on charge y
5      fld     dword [ebp+8]   ; puis x
6      fcomip  st0, st1        ; x est en st0, y en st1, on compare
7                              ; x à y et on dépile x
8      fstp    st0             ; supprime y
9      jbe     .else
10     mov     eax, 1
11     jmp     .endif
12 .else:
13     mov     eax, 3
14 .endif:
15     mov     esp, ebp
```

```
16      pop     ebp
17      ret
```

Instructions de saut pour les flottants

Pour des raisons historiques les instructions de comparaison sur les flottants ne mettent pas à jour les mêmes bits du registre `EFLAGS` que l'instruction `cmp`. On a donc défini d'autres instructions de branchement comme `jb` (au lieu de `jl`) et `ja` (au lieu de `jg`). Il existe également `jbe` et `jae`.

Pour résumer, pour comparer deux valeurs (x à y), on commence par charger y dans le coprocesseur, puis x et on exécute `fcomip` ou `fcomip st0, st1`. On prend alors le même raisonnement que pour un `if` entre valeurs entières. On prend la négation de la condition pour se brancher sur le `else`. Il ne faut pas oublier de supprimer les valeurs chargées au niveau du coprocesseur.

7.6.2 Comparaison en architecture 64 bits

Lorsque l'on travaille en 64 bits, ce sont les registres SSE qui sont utilisés (cf. Chapitre 8) pour réaliser les calculs avec les nombres flottants. En prévision de ce que nous verrons dans le prochain chapitre nous montrons comment le compilateur C traduit le code suivant :

```
1  void equal_64bits(float x, float y) {
2      if (fabs(x - y) <= 1e-6) {
3          // code du then
4      } else {
5          // code du else
6      }
7  }
```

On a vu Chapitre 6 que les variables `x` et `y` sont respectivement placées dans les registres `xmm0` et `xmm1` :

```
1  equal_64bits:
2      subss     xmm0, xmm1
3      andps     xmm0, XMMWORD PTR .LC0[rip]
4      cvtss2sd  xmm0, xmm0
5      movsd     xmm1, QWORD PTR .LC1[rip]
6      comisd    xmm1, xmm0
7      jb        .else
8  .then:
9      ....
10     jmp       .endif
11 .else:
12     ....
13 .endif:
```

```
14      ret
15
16
17  .LC0:
18      .long   2147483647 ; 0x7FFFFFFF
19      .long   0
20      .long   0
21      .long   0
22  .LC1:
23      .long   2696277389 ; 0xA0B5ED8D
24      .long   1051772663 ; 0x3EB0C6F7
```

On commence donc par calculer la différence `x-y` (ligne 2), puis on calcule la valeur absolue (ligne 3) en appliquant un masque qui ne garde que les 31 premiers bits du registre **xmm0** qui lui, occupe 128 bits (variable `.LC0`). En ligne 4, on convertit le résultat simple précision en double précision et on le compare à 10^{-6} codé en 64 bits au format IEEE 754 en `.LC1`. Le reste du code exécute le `.then` dans le cas ou la condition du **if** est vérifiée, sinon il exécute le `.else`.

Il faut noter que la constante $1e^{-6}$ placée en `.LC1` est donnée au format IEEE 754 double précision dans le code et représente la valeur hexadécimale `0x3EB0C6F7A0B5ED8D` qui correspond en fait à la valeur :

$$9.99999999999999954748111825886E-7$$

Il existe également une instruction **\gls {cmpss} xmm1, xmm2, imm8** qui permet de comparer deux valeurs 32 bits en partie basse des registres qui sont passés en paramètre. La constante `imm8` indique quel type de comparaison doit être réalisée (cf. Table 7.5).

imm8	Type de comparaison
0	`xmm1.ps[0] == xmm2.ps[0]`
1	`xmm1.ps[0] < xmm2.ps[0]`
2	`xmm1.ps[0] <= xmm2.ps[0]`
3	unordered
4	`xmm1.ps[0] != xmm2.ps[0]`
5	`xmm1.ps[0] >= xmm2.ps[0]`
6	`xmm1.ps[0] > xmm2.ps[0]`
7	ordered

TABLE 7.5 – Comparaison avec cmpss

L'implantation de **cmpss** est donc la suivante :

```
1  bool cmp(float x, float y, u8 imm8) {
2      switch( imm8 ) {
3      case 0: return x == y;
4      case 1: return x < y;
```

```
    case 2: return x <= y;
    case 3: return (x == Nan) || (y == NaN);
    case 4: return x != y;
    case 5: return x >= y;
    case 6: return x > y;
    case 7: return (x != Nan) && (y != NaN);
    }
}

bool res = cmp(xmm1.ps[0], xmm1.ps[1], imm8)
if (res == true) {
    xmm1.ps[0] = 0xFFFFFFFF;
} else {
    xmm1.ps[0] = 0x00000000;
}
```

Notamment la relation *unordered* est vraie si au moins un des opérandes est égale à *NaN*, alors que la relation *ordered* est vraie si aucune des opérandes n'est égale à *NaN*

Il existe l'instruction **cmpsd** (*Compare Scalar Double-Precision Floating-Point Value*) pour comparer deux *double*, instruction qui possède le même mnémonique que **cmpsd** (*Compare String Operands*). Il ne faut donc pas les confondre. La première utilise des registres vectoriels comme opérandes alors que la seconde ne possède pas d'opérande.

7.7 Traduction des expressions réelles

Pour traduire une expression en utilisant les instructions assembleur du coprocesseur il suffit de procéder en trois étapes :

1. représenter l'expression sous forme d'un arbre binaire
2. la traduire en notation polonaise inverse (RPN[1]) en réalisant un parcours postfixe de l'arbre
3. traduire la version en notation polonaise inverse par des instructions du coprocesseur en suivant les règles de traduction décrites ci-après

Prenons l'exemple suivant :

$$\frac{(x+1)\times(x-1)}{3-\sqrt{x}}$$

La représentation sous forme d'arbre binaire de cette expression est donnée Figure 7.2. On notera que l'arbre n'est pas un arbre binaire au sens strict puisque

1. ou *Reverse Polish Notation* est une notation post-fixée qui permet d'écrire de façon non ambiguë les formules arithmétiques sans utiliser de parenthèses.

pour la racine carrée ($\sqrt{x}$), on a qu'une seule branche. Dans le cas des opérateurs unaires on ne disposera que d'une seule branche au niveau de l'arbre et on peut convenir qu'il s'agit de la branche gauche ; la branche droite étant nulle (pointer non représenté).

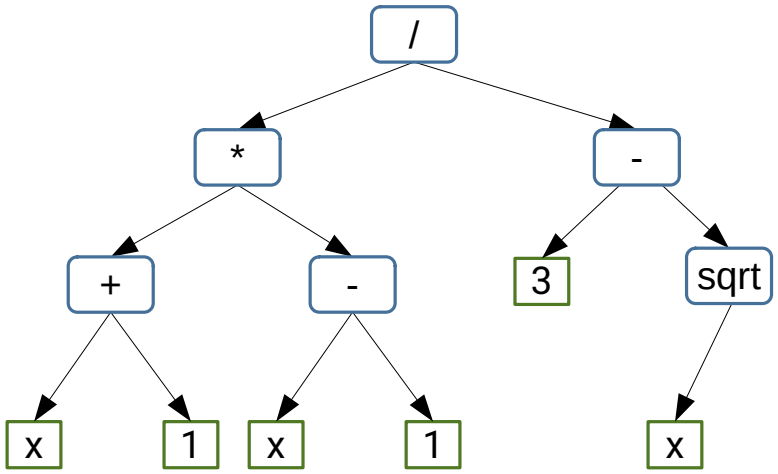


FIGURE 7.2 – Représentation arborescente de $\frac{(x+1)\times(x-1)}{3-\sqrt{x}}$

Le parcours **postfixe** de l'arbre consiste à visiter récursivement le fils gauche, puis le fils droit s'il existe et enfin le noeud. Si l'un des fils est également un noeud, on réitère le processus jusqu'à parvenir à une feuille de l'arbre, représentée ici par un rectangle au bords carrés vert sur la figure. Les noeuds internes sont représentés par des rectangles aux bords arrondis de couleur bleu. Au final l'expression postfixe est :

```
x 1.0 + x 1.0 - * 3.0 x sqrt - /
```

Les régles de traduction en assembleur à partir de l'expression RPN sont très simples :

- s'il s'agit d'une constante ou d'une variable, on la charge au niveau du coprocesseur grâce à l'instruction `fld`
- s'il s'agit d'un opérateur unaire, on l'applique sur `st0`
- s'il s'agit d'un opérateur binaire, on applique la formule `f<Oper>p st1,st0`, où `Oper` = add, sub, mul, div

A partir de l'expression précédente, on obtient donc :

```
section .data
    trois:  dd 3.0              ; constante

section .text
                                ; x + 1
    fld     dword [x]
    fld1
    faddp   st1, st0
                                ; x - 1
    fld     dword [x]
    fld1
    fsubp   st1, st0
                                ; (x+1)*(x-1)
    fmulp   st1, st0
                                ; 3 - sqrt(x)
    fld     dword [trois]
    fld     dword [x]
    fsqrt
    fsubp   st1, st0
                                ; (x+1)*(x-1) / (3 - sqrt(x))
    fdivp   st1, st0
```

On notera que dans le cas des constantes, si la constante n'est pas une des constantes prédéfinies du coprocesseur il est nécessaire de la stocker en mémoire.

7.8 Affichage d'une valeur flottante

7.8.1 Architecture 32 bits

En 32 bits l'affichage est assez simple, il suffit de déclarer en externe la fonction `printf` et de suivre la convention d'appel du langage C avec cependant une spécificité. *Une valeur de type simple précision doit être convertie en une valeur double précision avant l'affichage*, comme sur le code suivant :

```
float x = 3.14;
printf("%f\n", x);
```

est donc traduit en :

```
extern printf

section .data
    x:   dd 3.14
    msg: db `%f\n`, 0

section .text
    fld    dword [x]        ; chargement simple précision
    sub    esp, 8           ; réservation de 64 bits pour
```

```
                                ; double précision
    fstp    qword [esp]         ; conversion en 64 bits dans la pile
    push    dword msg
    call    printf
    add     esp, 12
```

Afin de convertir la valeur en une valeur double précision, on réserve 8 octets dans la pile et on utilisera ces 8 octets comme paramètre de la fonction `printf`.

7.8.2 Architecture 64 bits

Pour une architecture 64 bits, d'après ce qui a été vu en Section 6.3.5, on doit procéder ainsi :

```
extern printf
default rel                     ; utilisation de l'adressage relatif

section .data
    x:   dd 3.14
    msg: db `%f\n`, 0

section .text

affiche:
    push     rbp                ; | ces deux instructions sont
    mov      rbp, rsp           ; | normalement inutiles
    movss    xmm0, [x]          ; utilisation de xmm0 pour stocker x
    cvtss2sd xmm0, xmm0         ; conversion au format double précision
    lea      rdi, [msg]
    mov      rax,  1            ; indique qu'il y a une valeur flottante
    call     printf WRT ..plt
    mov      rsp, rbp           ; | ces deux instructions sont
    pop      rbp                ; | normalement inutiles
    ret
```

Le registre `rdi` contient l'adresse de la chaine du format d'affichage. La valeur flottante est placée dans le registre `xmm0` et est convertie en double précision comme en 32 bits. Enfin, le registre `rax` doit contenir le nombre de valeurs flottantes à traiter avant l'appel à `printf`.

7.9 Conclusion

7.9.1 Que retenir ?

- le coprocesseur arithmétique permet de réaliser les calculs des nombres à virgule flottante

- ▷ initialement absent, puis par la suite positionné sur la carte mère, il est aujourd'hui intégré au microprocesseur et est qualifié de FPU (Floating Point Unit). On trouve généralement plusieurs unités FPU au sein du microprocesseur
- ▷ la FPU fonctionne comme une pile dotée de huit registres **st0** à **st7**
- ▷ les instructions assembleur liées à la FPU commencent par la lettre `f`
- ▷ toute donnée empilée dans la FPU doit être dépilée

7.9.2 Compétences à acquérir

Après lecture et travail sur ce chapitre, on doit être capable de :

- ☐ traduire un calcul avec des nombres à virgule flottante sous forme d'une série d'instructions assembleur, pour cela on modélise l'expression sous forme d'un arbre binaire que l'on traduit en utilisant les règles données dans la Section 7.7.
- ☐ comparer deux nombres flottants

7.10 Exercices

Exercice 29 - Montrer comment, en utilisant les registres généraux et les instructions associées, on peut réaliser les opérations de la FPU comme **fabs** qui calcule la valeur absolue ou **fchs** qui change le signe d'une valeur flottante sur 32 bits. On chargera la valeur flottante dans **eax** par exemple avant de réaliser l'opération.

Exercice 30 - Implantez la fonction `iota` en architecture 32 bits en utilisant la FPU :

```
1 void iota(float *t, int n) {
2     for (int i=0; i<n; ++i) {
3         t[i] = (float) i;
4     }
5 }
```

Exercice 31 - Implantez l'expression suivante en architecture 32 bits en utilisant la FPU :

$$\frac{(x-5)\times(x+6)}{\cos(x-5)^2}\times\sin(x+6)$$

1. on commencera par dessiner l'expression sous forme d'arbre binaire
2. puis on identifiera les sous-expressions qui sont répétées comme $x-5$ et $x+6$

3. on traduira l'expression non optimisée
4. puis on donnera une version en optimisant les calculs en ne recalculant pas à chaque fois les sous-expressions répétées

Exercice 32 - Implantez la fonction `puissance` en architecture 32 bits en utilisant la FPU :

```
float puissance(float x, int n) {
    float result = 1;
    for (int i=0; i<n; ++i) {
        result *= x;
    }
    return result;
}
```

Faire de même en 64 bits.

Exercice 33 - Ecrire un programme assembleur qui calcule l'expression suivante sous forme d'un développement limité :

$$\frac{1}{(1-x)} = 1 + x + x^2 + x^3 + \ldots + x^n$$

Cette formule fonctionne lorsque x est proche de 0, on pourra essayer avec $x = 0.2$ par exemple et déterminer à partir de quelle valeur de n on peut s'arrêter car n ne modifie plus la précision du calcul.

Exercice 34 - Ecrire un programme assembleur qui permet de trouver les solutions à valeurs dans $\mathbb{R}$ d'une équation du second degré $ax^2 + bx + c = 0$. On utilisera la FPU pour réaliser les calculs. On rappelle qu'il faut calculer le discriminant $\Delta = b^2 - 4ac$, puis si $\Delta \geq 0$, on trouvera des solutions réelles :

$$x_1 = \frac{-b + \sqrt{\Delta}}{2a} \qquad x_2 = \frac{-b - \sqrt{\Delta}}{2a}$$

On utilisera l'instruction `fcomip st0, st1` afin de réaliser une comparaison entre Δ et 0.

Chapitre 8

Unités vectorielles

8.1 Introduction

Les unités vectorielles permettent de **vectoriser** le code, en d'autres termes, de le paralléliser au sein du microprocesseur. On exécutera la même instruction sur plusieurs données différentes stockées dans un registre de type MMX (64 bits), SSE (128 bits) ou AVX (256 bits).

Par exemple avec la technologie SSE, au lieu d'écrire :

```
float v1[4], v2[4], v3[4];

void vector_sum(float *x, float *y, float *z, int size) {
    for (int i = 0; i < 4; ++i) {
        z[i] = x[i] + y[i]
    }
}

vector_sum(v1, v2, v3, 4);
```

On réalise une seule opération en parallèle sur un registre capable de contenir 4 floats, ce que l'on note :

```
z[0:3] = x[0:3] + y[0:3]
```

La notation `x[0:3]` symbolise `x[0]` à `x[3]`, elle n'est pas utilisable en langage C, elle nous permet seulement d'exprimer de manière concise le traitement réalisé.

On parle alors de traitement SIMD pour *Single Instruction Multiple Data*, cela signifie que la même instruction est appliquée sur des données différentes et pour que cela ait un intérêt en terme de performance, on réalise les calculs en parallèle et non pas de manière séquentielle.

Par la suite nous allons nous intéresser aux technologies SSE et AVX et nous ne

traiterons pas du MMX désuet à présent.

La technologie MMX pour *MultiMedia eXtensions* est apparue sur les processeurs Intel Pentium MMX en 1997. Il s'agit d'un jeu d'instructions composé de 57 instructions qui traitent uniquement des entiers d'un maximum de 64 bits. Le MMX souffre d'un défaut majeur qui fait qu'il rend indisponible la FPU puisqu'il en utilise une partie des registres. On ne peut donc travailler simultanément avec la FPU et le MMX. Ce défaut a été corrigé avec l'introduction du SSE.

Notons également que nous allons suivre la convention de représentation Intel pour les instructions SSE qui consiste à écrire les valeurs d'un vecteur en mémoire ou d'un registre en plaçant la partie haute à gauche et la partie basse à droite. Cette convention reprend en fait le principe du *little endian* vu Section 2.6.

8.2 SSE

Sous le sigle SSE (*Streaming SIMD Extensions*) nous réunissons tous les jeux d'instructions successifs SSE, SSE2, SSE3, SSSE3, SSE4A, SSE4.1 et SSE4.2. Nous n'allons faire qu'effleurer l'ensemble des instructions SSE qui sont bien trop nombreuses et diverses pour être toutes passées en revue et nécessiteraient à elles seules un ouvrage. Le lecteur intéressé pourra consulter à ce sujet l'excellent livre de [19]. Nous nous intéresserons et décrirons dans la suite de ce chapitre les instructions qui entrent en jeu dans les études de cas que nous ménerons par la suite.

La première version du SSE est un jeu de 70 instructions apparu en 1999 sur le Pentium III en réponse à la technologie 3DNow ! d'AMD, née un an plus tôt. Ces instructions traitent des entiers ou des réels. Les versions suivantes ont apporté de nouvelles instructions de manipulation des données ou de calcul comme par exemple `dpps` (*Dot Product of Packed Single Precision Floating-Point Values*) du jeu d'instructions SSE4.1 qui réalise le produit scalaire de deux vecteurs.

Sur les Pentium III notamment, l'efficacité du SSE était nettement moindre que sur son successeur, le Pentium 4, car bien que le Pentium III disposât de registres de stockage de 128 bits, il ne possédait que de registres 64 bits pour réaliser les calculs. De ce fait, une instruction SSE était traitée en deux fois 64 bits, on commençait par traiter la partie basse, puis la partie haute, ce qui est moins efficace que de traiter 128 bits en une seule fois.

En architecture 32 bits, il existe 8 registres SSE de 128 bits nommés `xmm0` à `xmm7` [1]. Ce nombre de registres est doublé en architecture 64 bits avec l'ajout des registres `xmm8` à `xmm15`.

Les registres SSE possèdent des instructions qui traitent les valeurs qu'ils contiennent (cf. Figure 8.1) :

- soit sous forme d'entiers au format :

1. A ne pas confondre avec les registres MMX qui sont appelés `mm0` à `mm7`.

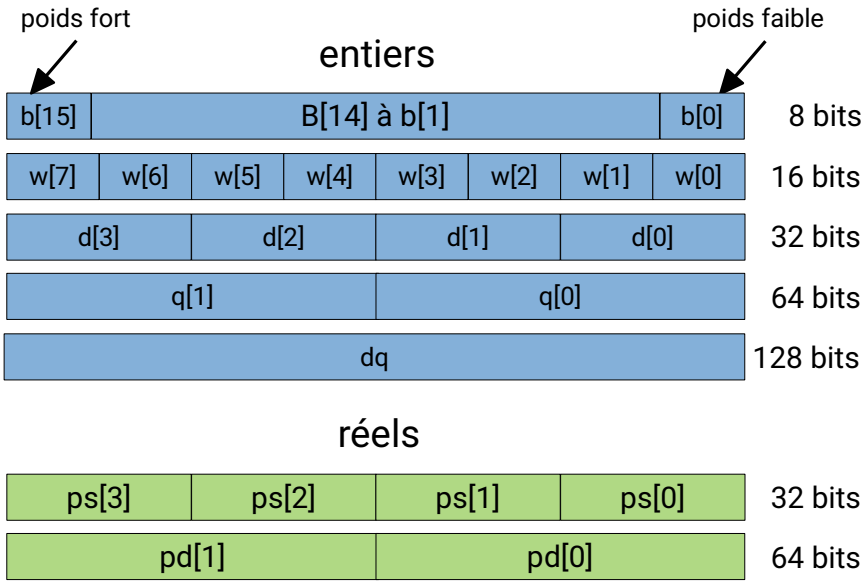


FIGURE 8.1 – Types de données contenues dans un registre SSE

- ▷ 16 octets
- ▷ 8 mots
- ▷ 4 double mots (4 entiers 32 bits signés ou non)
- ▷ 2 quadruples mots (2 entiers 64 bits signés ou non)
- ▷ double quadruple mot (*double quad word*) soit un total de 128 bits

- soit sous forme de nombres à virgule flottante (32 et 64 bits)
  - ▷ 4 flottants simple précision (`float`)
  - ▷ 2 flottants double précision (`double`)

On disposera donc de plusieurs instructions similaires mais avec des mnémoniques différents en fonction que l'on traite des entiers ou des flottants. La grande majorité de ces instructions seront suffixées par une à deux lettres (cf. Table 8.1) qui correspondent au type de donnée manipulée.

Ainsi, l'instruction **`paddb`** réalise une addition entière en parallèle entre les 16 octets de ses deux opérandes, alors que **`paddb, paddw, paddd`** réalise une addition entière en parallèle sur 4 entiers 32 bits. De la même manière **`addps, addpd`** réalise une addition en parallèle sur 4 flottants en simple précision et **`addpd`** traite 2 flottants en double précision.

On note également pour les flottants les suffixes `ss` et `sd` qui ne traitent que la partie basse du registre SSE (respectivement 32 et 64 bits). Ces instructions liées à des flottants simple ou double précision permettent de remplacer la FPU car en 64 bits les paramètres de type float ou double sont passés dans les registres SSE et les calculs sont réalisés avec ces mêmes registres.

Type	Taille en octets	Nom	Quantité dans 128 bits	Suffixe
entier	1	byte	16	`b`
entier	2	word	8	`w`
entier	4	double word	4	`d`
entier	8	quad word	2	`q`
flottant	4	float	4	`ps`
flottant	8	double	2	`pd`
flottant	4	float	1	`ss`
flottant	8	double	1	`sd`

TABLE 8.1 – Suffixes des instrutions SSE

Facteur d'amélioration

Il n'est pas possible de manipuler un registre vectoriel en faisant directement référence à son $i$ème élément (sauf pour des instructions utilisant un masque de sélection) mais afin de simplifier la compréhension de certaines instructions et traitements nous introduisons la notation suivante qui nous permettra de décrire le comportement des instructions SSE et AVX sous forme de petits programmes C :

```
xmm0.T[i]
```

où `T` représente le type (`b`, `w`, `d`, `q`, `ps`, `pd`, présenté Table 8.1) et `i` le $i$ème élément. Ainsi **`xmm0.b[15]`** représente le dernier octet du registre **`xmm0`**, donc l'octet de poids fort, l'octet de poids faible étant **`xmm0.b[0]`**.
Notons que dans la documentation Intel on fait référence aux bits du registre. Ainsi, pour représenter **`xmm0.b[15]`** on indiquera **`xmm0[127:120]`** qui est l'intervalle de bits qui correpond au seizième octet du registre.

8.2.1 Chargement et stockage des données

Le chargement des données vers les registres SSE ou le stockage des valeurs contenues dans les registres vers la mémoire se font à l'aide des instructions de déplacement de type *mov*.

Pour les entiers, on utilisera **`movdqu`** (*MOV Double Quad word Unaligned*) ou **`movdqa`**, **`movdqu`** (*MOV Double Quad word Aligned*). Dans le cas du SSE les données sont alignées si l'adresse depuis laquelle on lit ou on écrit est un multiple de 16. Nous renvoyons le lecteur au Chapitre 3, Section 3.2.1 concernant la notion d'alignement des données en mémoire.

Le format des instructions de chargement de données est de la forme :

```
1    movdqa  xmm1, [ebx]            ; opérande SSE et référence
2                                   ; mémoire (Load)
3    movdqa  [edi + ecx * 4], xmm7  ; idem (Store)
4    movdqa  xmm3, xmm1             ; deux opérandes SSE
```

Pour les flottants, on utilisera les instructions `movups` ou `movaps` qui fonctionnent sur le même modèle.

Cependant, on notera que l'on peut utiliser `movdqa` (ou `movdqu`) avec des flottants et `movaps, movups` (ou `movups`)avec des entiers puisqu'il n'y a a priori aucune conversion ou modification des données, on se contente de lire les données et les stocker dans un registre ou en mémoire. J'avais tenté, il y a quelques années, de contacter Intel afin de savoir pourquoi ils existaient deux types d'instructions différentes mais je n'ai jamais eu de retour.

Enfin il existe des instructions qui ne traitent que la partie basse du registre SSE comme `movd` pour les entiers et `movss, movsd`, `movsd` pour les flottants simple et double précision :

```
1  mov     eax, 0x01010101
2  movd    xmm1, eax        ; xmm1.d[0] = 0x01010101, xmm1.d[1:3] = ?
3  movss   xmm2, [edi]      ; xmm2.ps[0] = [edi], xmm2.ps[1:3] = ?
4  movsd   xmm2, [edi]      ; xmm2.pd[0] = [edi], xmm2.pd[1] = ?
```

On charge ici la valeur hexadécimale sur 32 bits `0x01010101` dans la partie basse du registre `xmm1`, les 3 autres valeurs 32 bits ne sont pas modifiées.

Attention

Le fonctionnement est identique pour `movss` avec la particularité qu'on ne peut charger une valeur depuis un registre mais seulement depuis la mémoire comme pour la FPU.

8.2.2 Instructions arithmétiques

Pour les entiers, on utilisera les instructions `padd` pour l'addition, `psub` pour la soustraction et `pmull` pour la multiplication, suffixées par la quantité traitée. Notons qu'**il n'existe pas** d'instruction `pdiv` qui réaliserait une division entière.

Pour les flottants, on trouve les instructions `addps`, `subps`, `mulps`, `divps` ainsi que `addpd` et consorts.

Il existe également des instructions comme `addsubps xmm1, xmm2` dont le comportement est le suivant :

```
xmm1.ps[0] -= xmm2.ps[0]
xmm1.ps[1] += xmm2.ps[1]
xmm1.ps[2] -= xmm2.ps[2]
xmm1.ps[3] += xmm2.ps[3]
```

et **haddps xmm1, xmm2** qui réalise une addition dite *horizontale*.

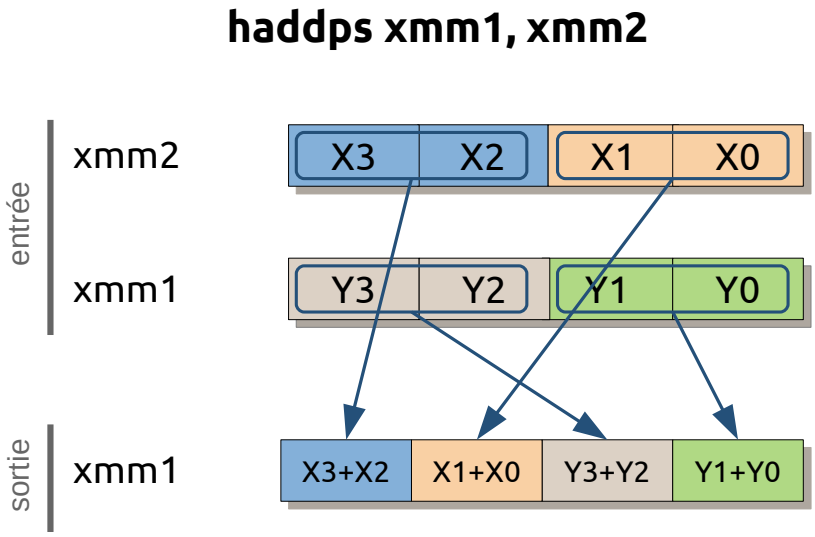


FIGURE 8.2 – Instruction haddps

```
; haddps xmm1, xmm2
; on utilise un registre temporaire xmmt
xmmt.ps[0] = xmm1.ps[0] + xmm1.ps[1]
xmmt.ps[1] = xmm1.ps[2] + xmm1.ps[3]
xmmt.ps[2] = xmm2.ps[0] + xmm2.ps[1]
xmmt.ps[3] = xmm2.ps[2] + xmm2.ps[3]
xmm1 = xmmt
```

L'intérêt de l'instruction **haddps** (cf. Figure 8.2) est qu'elle permet de faire la somme des quatre valeurs flottantes simple précision contenues dans un registre SSE [2] en procédant ainsi :

```
haddps xmm1, xmm1
haddps xmm1, xmm1
```

On réalise deux fois l'addition horizontale d'un registre avec lui même. Au final on obtient :

2. On appelle cette opération une réduction.

```
xmm1.ps[0:3] = xmm1.ps[0] + xmm1.ps[1] + xmm1.ps[2] + xmm1.ps[3]
```

On trouve également **phaddw** et **phaddd** pour les entiers 16 et 32 bits respectivement qui réalisent l'addition horizontale de mots et double mots.

8.2.3 Fonctions trigonométriques, logarithme, exponentielle

Il n'existe pas d'instructions qui réalisent les calculs des fonctions trigonométriques, logarithmiques ou exponentielles. Il faut utiliser des librairies spécialisées comme l'Intel MKL[3] (*Math Kernel Library*) ou l'AMD libM[4] (*Math Library*). Vous trouverez également d'autres librairies non propriétaires sur internet.

8.2.4 Instructions binaires

Les instructions que nous qualifions de binaires agissent sur la totalité des 128 bits d'un registre SSE (ou les 256 bits d'un registre AVX). Il s'agit de **pand, por, pxor**, **por**, **pxor**. Ces instructions réalisent respectivement un *et binaire*, le *ou binaire*, le *ou exclusif binaire*. Il existe également l'instruction **pandn** (*Parallel AND Not*) qui réalise un et binaire entre le complémentaire de l'opérande de destination et l'opérande source.

```
pand    xmm1, xmm2      ; xmm1 = xmm1 and xmm2
por     xmm2, [esi]
pandn   xmm1, xmm4      ; xmm1 = not(xmm1) and xmm4
```

On pourra voir comment utiliser ces instruction dans un cas concret en consultant le Chapitre 14.

8.2.5 Instructions de conversion

Il existe de nombreuses instructions de conversion dont le mnémonique commence par `cvtss2sd, cvtps2pd, cvtss2si, cvtsi2sd` pour *convert*. Elles permettent de convertir des flottants en simple ou double précision ou de convertir des flottants en entiers :

- **cvtss2sd** convertit un flottant simple précision en double précision
- **cvtsd2ss** convertit un flottant double précision en simple précision
- **cvtps2pd** convertit des flottants simple précision en double précision
- **cvtpd2ps** convertit des flottants double précision en simple précision

3. https://software.intel.com/en-us/mkl
4. https://developer.amd.com/amd-cpu-libraries/amd-math-library-libm/

- **cvtss2si** convertit un flottant simple précision en entier dans un registre 32 ou 64 bits
- **cvtsi2ss** convertit un entier situé dans un emplacement mémoire ou un registre 32 ou 64 bits en un flottant simple précision
- **cvtsd2si** convertit un flottant double précision en un entier dans un registre 32 ou 64 bits
- **cvtsi2sd** convertit un entier situé dans un emplacement mémoire ou un registre 32 ou 64 bits en un flottant double précision

Par exemple, le code suivant charge les 4 valeurs flottantes de `v` dans `xmm0` puis convertit la partie basse du registre vectoriel en une valeur entière dans `eax`. Au final `eax` contient la valeur 12.

```
1  section .data
2      v:  dd 12.0, 14.0, 16.0, 20.0
3
4  section .text
5
6  movups      xmm0, [v]
7  cvtss2si    eax, xmm0
```

8.2.6 Instructions de réarrangement

Les instructions **pshufd** pour les entiers et **shufps** pour les flottants permettent de sélectionner ou réorganiser les données au sein d'un registre SSE mais ont un comportement différent. La plupart de ces instructions utilisent une troisième opérande qualifiée de masque et notée `imm8` ce qui signifie qu'il s'agit d'une constante sur 8 bits et elle est utilisée pour indiquer quels champs sélectionner.

Par exemple **pshufd xmm1, xmm2, imm8**, qui est présentée Figure 8.3, réalise une sélection et réorganisation des valeurs de `xmm2` vers `xmm1` :

```
1  ; pshufd xmm1, xmm2, imm8
2  xmm1.ps[0] = xmm2.ps[ imm8 & 0x03 ];
3  xmm1.ps[1] = xmm2.ps[ (imm8 >> 2) & 0x03 ];
4  xmm1.ps[2] = xmm2.ps[ (imm8 >> 4) & 0x03 ];
5  xmm1.ps[3] = xmm2.ps[ (imm8 >> 6) & 0x03 ];
```

L'utilisation de cette instruction sur la même opérande avec un masque de 0 (**pshufd xmm1, xmm1, 0**) a pour effet de recopier la valeur `xmm1.d[0]` dans `xmm1.d[1:3]`. Au final on obtient donc quatre fois la même valeur dans `xmm1`.

On peut bien entendu l'utiliser pour des flottants simple précision car l'instruction **shufps**, qui possède la même syntaxe, prend en considération `xmm1` et `xmm2` pour la sélection des valeurs mais possède un comportement quelque peu différent :

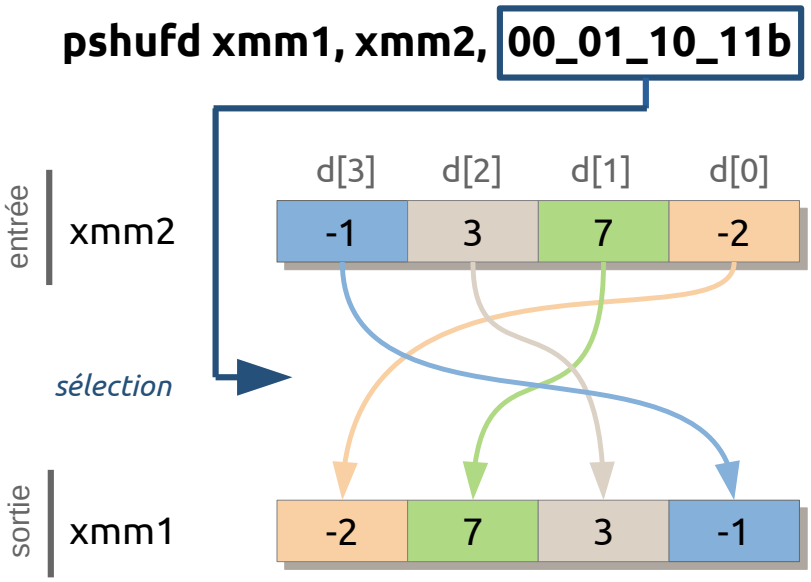


FIGURE 8.3 – Instruction PSHUFD

```
; shufps xmm1, xmm2, imm8
xmmt[0] = xmm1.ps[ imm8 & 0x03 ];
xmmt[1] = xmm1.ps[ (imm8 >> 2) & 0x03 ];
xmmt[2] = xmm2.ps[ (imm8 >> 4) & 0x03 ];
xmmt[3] = xmm2.ps[ (imm8 >> 6) & 0x03 ];
xmm1 = xmmt
```

Une autre instruction intéressante est `blendps`, mais elle n'utilise que les 4 premiers bits de la constante `imm8`. Elle permet de remplacer les valeurs du registre de destination par des valeurs du registre source :

```
// blendps xmm1, xmm2, imm8
for (int index = 0; index <= 3; ++index) {
    xmm1.ps[ index ] = (imm8 & (1 << index)) == 0
        ? xmm1.ps[ index ] : xmm2.ps[ index ];
}
```

Ainsi, le code suivant remplacera `xmm1.ps[1]` par `xmm2.ps[1]` :

```
blendps xmm1, xmm2, 00000010b
```

Une instruction très utile est `pblendvb` (*Variable Blend Packed Bytes*). Elle travaille sur les octets d'un registre SSE et utilise par défaut un masque de sélection basé sur le registre `xmm0` :

```
// pblendvb xmm1, xmm2
int i, byte;
```

```
for (byte = 0, i = 7; i <= 127; i += 8, ++byte) {
  xmm1.b[ byte ] = (xmm0.bits(i) == 1) ? xmm2.b[ byte ] :
    xmm1.b[ byte ];
}
```

Elle permet de sélectionner les octets de **xmm1** ou de **xmm2** en fonction des octets de poids fort de **xmm0** positionnés à 0 ou 1.

Dans la même veine, mais pour les valeurs flottantes, on trouve **blenvps** (*Variable Blend Packed Single Precision*) :

```
blendvps xmm1, xmm2    <xmm0>
```

Il existe une série d'instructions **vpbroadcast(b,w,d,q)** qui permettent de recopier une valeur dans plusieurs emplacements d'un registre SSE ou AVX. Par exemple **vpbroadcastb xmm1, xmm1** recopie le premier octet du registre **xmm1** dans les 15 autres emplacements du registre :

```
// vpbroadcastb xmm1, xmm1
for (int index = 1; index <= 15; ++index) {
  xmm1.b[ index ] = xmm1.b[ 0 ];
}
```

Enfin, l'instruction **insertps xmm1, xmm2, imm8** réalise plusieurs opérations

1. en premier lieu, elle sélectionne l'une des 4 valeurs de la source **xmm2** grâce aux bits 6 et 7 de la constante imm8
2. elle recopie ensuite cette valeur dans **xmm1** à la position indiquée par les bits 4 et 5 de imm8
3. elle met enfin, en fonction des bits 0 à 3 positionné à 1 de imm8, les valeurs correspondantes dans **xmm1** à 0

Le code qui suit donne comme résultat un registre **xmm1** contenant les valeurs [7.0, 3.0, 0.0, 0.0].

```
section .data
    a   dd 1.0, 2.0, 3.0, 4.0
    b   dd 5.0, 6.0, 7.0, 8.0

section .text
    movups  xmm1, [a]
    movups  xmm2, [b]
    insertps xmm1, xmm2, 10_11_0011b
```

On commence par charger dans **xmm1** le vecteur [4.0, 3.0, 2.0, 1.0], puis dans **xmm2** le vecteur [8.0, 7.0, 6.0, 5.0]. On choisit alors la valeur d'indice 10_b de **xmm2**, c'est à dire 7.0 et on la recopie en position 11_b de **xmm1**. La partie basse de la constante **imm8**, soit 0011_b indique que les valeurs d'indices 0 et 1 de **xmm1** doivent être mises à zéro.

8.3 AVX, AVX2

8.3.1 Spécificités

Sous le sigle AVX nous plaçons les jeux d'instructions AVX (*Advanced Vector eXtensions*) et AVX2 256 bits. Nous ne nous intéresserons qu'en fin de chapitre à l'AVX 512 bits. Tout comme en architecture 32 bits, il existe 8 registres AVX de 256 bits nommés `ymm0` à `ymm7`. Ce nombre de registres est doublé en architecture 64 bits avec l'ajout de `ymm8` à `ymm15`. Les principaux changements par rapport au SSE sont les suivants :

- les instructions AVX commencent par la lettre `v` pour les distinguer des instructions SSE
- les instructions AVX peuvent agir sur les registres `ymm` ou `xmm` et vont utiliser la même syntaxe
- cependant, une instruction AVX peut prendre une opérande supplémentaire qui sera le registre de destination

Par exemple, en SSE, si on écrit `paddd xmm1, xmm2`, les quatre entiers de `xmm2` sont ajoutés à `xmm1`, en d'autres termes on a `xmm1.d[0:3] += xmm2.d[0:3]`. Les valeurs présentes dans `xmm1` sont donc perdues. On aura le même comportement si on utilise `vpaddd xmm1, xmm2`.

```
; avec deux opérandes
 paddd  xmm1, xmm2  ; xmm1.d[0:3] = xmm1.d[0:3] + xmm2.d[0:3]
vpaddd  xmm1, xmm2  ; xmm1.d[0:3] = xmm1.d[0:3] + xmm2.d[0:3]
```

Par contre, si on écrit `vpaddd xmm3, xmm1, xmm2`, le registre `xmm3` recevra le résultat de la somme de `xmm1` et `xmm2`. Les registres `xmm1` et `xmm2` ne seront donc pas modifiés.

```
; avec trois opérandes
vpaddd  xmm3, xmm1, xmm2  ; xmm3.d[0:3] = xmm1.d[0:3] + xmm2.d[0:3]
```

8.3.2 Partie haute

Certaines instructions, comme `insertps`, dont nous avons parlé précédemment, travaillent uniquement avec la partie basse des registres AVX. Cela est dû à la constante `imm8` qui interagit avec l'un des quatre flottants simple précision d'un registre SSE. L'extension AVX de cette instruction `vinsertps` ne permet pas d'identifier les flottants dans la partie haute d'un registre AVX.

Il est donc nécessaire pour transposer l'utilisation du SSE vers l'AVX de travailler sur la partie basse du registre AVX puis de déplacer la partie basse vers la

partie haute. On dispose par exemple des instructions `vinsertf128`, `vextractf` ou `vpbroadcast` qui réalisent ces manipulations.

En particulier, l'instruction `vinsertf128 ymm3, ymm2, xmm1, 0/1` copie `ymm2` dans `ymm3` puis remplace la partie haute (1) ou la partie basse (0) de `ymm3` par les valeurs de `xmm1`.

`vextractf128 xmm1, ymm2, 0/1`, copie la partie basse (0) ou la partie haute (1) de `ymm2` dans `xmm1`.

La série d'instructions `vpbroadcast(b/w/d/q) x/ymm, reg` recopie les 8/16/32 ou 64 bits d'un registre général respectivement vers tous les octets, mot, double mots ou quadruples mots d'un registre SSE ou AVX.

Ainsi pour recopier 32 fois l'octet 0x85 dans le registre `ymm1`, on écrira :

```
1 mov           eax, 0x85    ; ou mov al, 0x85
2 vpbroadcastb  ymm1, eax
```

8.3.3 Instructions singulières

Certaines instructions n'ont pas le même comportement en AVX et en SSE. C'est le cas de `haddps` dont nous avons parlé Section 8.2.2. Nous avons vu que l'utilisation de deux fois cette instruction sur le même registre permet de calculer la somme des quatre valeurs qu'il contient. Malheureusement cela ne fonctionne pas avec les 8 valeurs 32 bits que contient un registre `ymm` lorsque l'on utilise `vhaddps`.

En effet, le code suivant :

```
1 section .data
2     v   dd 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0
3
4 section .text
5
6     vmovups ymm0, [v]
7     vhaddps ymm0, ymm0
8     vhaddps ymm0, ymm0
9     vhaddps ymm0, ymm0
```

produira successivement les résultats :

Instruction	ymm0							
`vmovups`	8	7	6	5	4	3	2	1
`vhaddps`	15	11	15	11	7	3	7	3
`vhaddps`	26	26	26	26	10	10	10	10
`vhaddps`	52	52	52	52	20	20	20	20

Or on aimerait obtenir la somme des valeurs c'est à dire 36. Il faut alors procéder comme suit :

```
    vhaddps ymm0, ymm0           ; ymm0 = [15, 11, 15, 11, 7, 3, 7, 3]
    vhaddps ymm0, ymm0           ; ymm0 = [26, ... , 26, 10, ..., 10]
    vextractf128  xmm1, ymm0, 1  ; xmm1 = [26, 26, 26, 26]
    addps   xmm0, xmm1           ; xmm0 = [36, 36, 36, 36]
```

On réalise la somme horizontale deux fois comme en SSE, puis on transfert la partie haute de `ymm0` vers `xmm1`. Il suffit alors d'additionner les deux registres `xmm0` et `xmm1` pour avoir dans `xmm0` le résultat escompté.

8.4 Affichage d'un registre

8.4.1 Architecture 32 bits

Nous présentons, ci-après, deux macro-instructions qui permettent d'afficher un registre SSE et qui peuvent être adaptées pour les registres AVX. Il serait intéressant d'en faire une librairie que l'on peut inclure lors du débogage de certains programmes. Nous laissons cette tâche au lecteur à titre d'exercice.

```
extern printf

section .data

str_sse_int: db `[%d %d %d %d]\n`, 0
str_sse_flt: db `[%f %f %f %f]\n`, 0

%macro print_sse_int 1
    sub     esp, 16
    ; affichage Intel
    pshufd  %1, %1, 00011011b
    movdqu  [esp], %1
    push    dword str_sse_int
    call    printf
    add     esp, 20
    ; rétablir les valeurs initiales
    pshufd  %1, %1, 00011011b
%endmacro

%macro print_sse_flt 1
    sub     esp, 48             ; 16 + 4*8
    movups  [esp], %1           ; stocke le registre
    fld     dword [esp + 32]
    fstp    qword [esp + 24]
    fld     dword [esp + 36]
    fstp    qword [esp + 16]
    fld     dword [esp + 40]
    fstp    qword [esp + 8]
    fld     dword [esp + 44]
    fstp    qword [esp]
```

```
31      push    dword str_sse_flt
32      call    printf
33      add     esp, 48+4
34  %endmacro
```

La première macro appelée `print_sse_int` affiche un registre SSE passé en paramètre sous la forme de 4 entiers signés. Les entiers sont affichés dans l'ordre décroissant des adresses mémoires, l'entier à l'adresse mémoire la plus haute est donc affiché en premier.

La seconde macro `print_sse_flt` affiche le contenu d'un registre SSE en considérant qu'il contient quatre flottants en simple précision, mais comme on affiche des flottants, il faut les convertir en double précision avant l'affichage.

8.4.2 Architecture 64 bits

En mode 64 bits, le code est plus long et un peu plus complexe. Dans le cas de l'affichage du registre en considérant qu'il contient quatre entiers, ces derniers doivent être passés en paramètres. On doit donc préserver les registres `rdi`, `rsi`, `rdx`, `rcx` et `r8` car `rdi` contiendra la chaîne du format d'affichage et les autres registres les quatre entiers à afficher.

```
1   extern printf
2   default rel
3
4   section .data
5
6   str_sse_int: db `[%d %d %d %d]\n`, 0
7   str_sse_flt: db `[%f %f %f %f]\n`, 0
8
9   section .text
10
11  %macro print_sse_int 1
12      sub     rsp, 128+16+8*6
13      mov     [rsp], rdi
14      mov     [rsp+8], rsi
15      mov     [rsp+16], rdx
16      mov     [rsp+24], rcx
17      mov     [rsp+32], r8
18      mov     [rsp+40], rax
19      lea     rdi, [str_sse_int]
20      movdqu  [rsp+56], %1
21      mov     esi, [rsp+56]
22      mov     edx, [rsp+60]
23      mov     ecx, [rsp+64]
24      mov     r8d, [rsp+68]
25      xor     rax, rax
26      call    printf WRT ..plt
27      mov     rax, [rsp+40]
28      mov     r8, [rsp+32]
```

```
29      mov     rcx, [rsp+24]
30      mov     rdx, [rsp+16]
31      mov     rsi, [rsp+8]
32      mov     rdi, [rsp]
33      add     rsp, 128+16+8*6
34  %endmacro
35
36  %macro print_sse_flt 1
37      sub     rsp, 128+5*16
38      movdqu  [rsp+16], xmm0      ; sauvegarde des registres
39      movdqu  [rsp+32], xmm1      ; utilisé pour le passage
40      movdqu  [rsp+48], xmm2      ; des flottants
41      movdqu  [rsp+64], xmm3
42      movdqu  [rsp], %1
43      cvtss2sd xmm0, [rsp]        ; convertion des flottants
44      cvtss2sd xmm1, [rsp+4]      ; simple précision en flottants
45      cvtss2sd xmm2, [rsp+8]      ; double précision
46      cvtss2sd xmm3, [rsp+12]
47      lea     rdi, [str_sse_flt]
48      mov     eax, 4
49      call    printf WRT ..plt
50      movdqu  xmm3, [rsp+64]      ; restauration des registres
51      movdqu  xmm2, [rsp+48]
52      movdqu  xmm1, [rsp+32]
53      movdqu  xmm0, [rsp+16]
54      add     esp, 128+5*16
55  %endmacro
```

Pour ce qui est de l'affichage du registre sous forme de quatre flottants, on doit préserver `xmm0` à `xmm3` car on va les utiliser pour passer les flottants que l'on doit convertir en quatre valeurs double précision grâce à l'instruction `cvtss2sd`. On indique que quatre valeurs sont à afficher en plaçant la valeur 4 dans le registre `eax` (ligne 48).

Chacune des macros commence par abaisser le sommet de pile de 128 octets (afin de préserver la *red zone* du sous-programme dans lequel on se trouve) plus le nombre d'octets nécessaires pour préserver les valeurs des registres qui seront modifiés temporairement pour réaliser l'affichage.

8.5 Intrinsics

Les *intrinsics* sont un apport important pour toute personne qui ne désire pas forcément écrire des programmes en assembleur. Les intrinsics sont en fait une interface entre le langage C et les instructions assembleur vectorielles. Elles autorisent l'écriture de sous-programmes C en utilisant les instructions vectorielles au travers de l'appel de fonctions, un peu à la manière des fonctions *built-in* évoquées Section 1.1.

Définition : Fonction intrinsèque

Une fonction **intrinsèque** est, dans la théorie des compilateurs, une fonction disponible dans un langage de programmation donné et dont l'implémentation est assurée par le compilateur.
L'avantage des fonctions intrinsèques est double :

- on écrit directement en C, le code est donc portable et on dispose des structures de contrôle de haut niveau comme la boucle `for`
- le compilateur se charge de l'optimisation du code

Néanmoins, on rencontre quelques difficultés lorsque l'on apprend à utiliser les intrinsics pour trois raisons :

- il existe une réelle difficulté à connaître le nom des fonctions par rapport aux instructions assembleurs
- les paramètres sont parfois mal décrits ou mal ordonnés (cf. `_mm_set_ps`)
- il est nécessaire de typer les données soumises aux instructions (`__m128`, `__m128i`)

Heureusement, il existe un site web très bien fait, l'*Intel Intrinsics Guide* [5] qui nous permet de retrouver les intrinsics en fonction de leur nom ou de l'instruction assembleur qu'elles remplacent.

Selon le jeu d'instructions utilisé, il faudra inclure le fichier d'entête de la librairie C adéquat (cf. Table 8.2) :

Fichier	Jeu
`<mmintrin.h>`	MMX
`<xmmintrin.h>`	SSE
`<emmintrin.h>`	SSE2
`<pmmintrin.h>`	SSE3
`<tmmintrin.h>`	SSSE3
`<smmintrin.h>`	SSE4.1
`<nmmintrin.h>`	SSE4.2
`<ammintrin.h>`	SSE4A
`<wmmintrin.h>`	AES
`<immintrin.h>`	AVX

TABLE 8.2 – Inclusion des fichiers entête selon le jeu d'instructions SSE ou AVX utilisé

5. https://software.intel.com/sites/landingpage/IntrinsicsGuide/

8.5.1 Types et format des instructions

Comme les intrinsics sont des fonctions, il est nécessaire de redéfinir des types afin d'indiquer au compilateur C la taille des variables qu'il manipule. Nous avons fait figurer Table 8.3, les trois types qui sont utilisés.

Type	Description	Exemple d'instruction
`__m128`	1 ou 4 floats	_mm_add_ps, _mm_add_ss
`__m128d`	2 doubles	_mm_add_pd, _mm_add_sd
`__m128i`	entiers	_mm_add_epi32

TABLE 8.3 – Types intrinsics

Les instructions se basent généralement sur le format `_mm_<oper>_<suffix>` où `oper` est le type d'opération (add, sub, mul, ...) et `suffix` est donné Table 8.4. Par exemple, `epi32` représente 4 valeurs 32-bits et signifie *Extended Packed Integers*. Le terme `epu` est utilisé pour les valeurs non signées (*Unsigned*).

nom	type
ss	1 float
ps	4 floats
d	double
i128	registre 128 bits
i64, u64	2 x 64 bits
i32, u32, epi32, epu32	4 x 32 bits entiers
i16, u16, epi16, epu16	8 x 16 bits
i8, u8, epi8, epu8	16 x 8 bits

TABLE 8.4 – suffixes des intrinsics

A titre d'exemple, voici quelques instructions et leur format :

- `__m128 _mm_add_ss(__m128 a, __m128 b)` additionne les 2 flottants simple précision en partie basse des registres vectoriels a et b
- `__m128 _mm_add_ps(__m128 a, __m128 b)` additionne 4 flottants simple précision en parallèle
- `__m128d _mm_add_pd(__m128d a, __m128d b)` additionne 2 flottants double précision en parallèle
- `__m128i _mm_add_epi32(__m128i a, __m128i b)` additionne 4 entiers en parallèle
- `__m128i _mm_and_si128(__m128i a, __m128i b)` réalise un et binaire entre deux registres SSE

On note que certaines instructions sont équivalentes, et bien que les formats de données sur lesquelles elles agissent soient différents, elles ont le même effet.

Pour en revenir à la fonction initiale de ce chapitre, qui consiste à additionner deux vecteurs de quatre `float`, celle-ci serait traduite en intrinsics sous la forme :

```
void vector_sum(float *x, float *y, float *z) {
    __m128 vx, vy, vz;

    vx = _mm_load_ps(&x[0]); // vx.ps[0:3] = x[0:3]
    vy = _mm_load_ps(&y[0]); // vy.ps[0:3] = y[0:3]
    vz = _mm_add_ps(vx, vy); // vz.ps[0:3] = vx.ps[0:3] + vy.ps[0:3]
    _mm_store_ps(&z[0], vz); // z[0:3] = vz.ps[0:3]
}
```

Sachant que les vecteurs `x`, `y`, `z` ont une taille de 4 éléments. En utilisant le compilateur `g++` avec l'optimisation `-O1` en 32 bits, le code précédent est traduit en :

```
    mov     eax,  [esp + 4]
    movaps  xmm0, [eax]       ; charge x[0:3] dans xmm0
    mov     eax,  [esp + 8]
    addps   xmm0, [eax]       ; xmm0 += y[0:3]
    mov     eax,  [esp + 12]
    movaps  [eax], xmm0       ; z[0:3] = xmm0
```

On se rend alors compte que les vecteurs que l'on avait défini dans la partie intrinsics `vx`, `vy`,`vz` sont finalement soit ignorés (cas de `vy`), soit remplacés par des registres SSE (cas de `vx` et `vz`).

On aurait pu éviter de déclarer les variables `vx`, `vy`, `vz` en écrivant simplement :

```
_mm_store_ps(&z[0], _mm_add_ps(_mm_load_ps(&x[0]), _mm_load_ps(&y[0])));
```

8.5.2 Travailler avec les flottants

8.5.2.1 Chargement et initialisation

Nous présentons, Table 8.5, les différentes possibilités offertes pour le chargement des données depuis la mémoire ou l'initialisation d'un registre avec des flottants simple précision.

A titre d'exemple, la fonction `_mm_set_ps` utilisée ainsi :

```
#include <xmmintrin.h>
#include <pmmintrin.h>

float function(float a, float b, float c, float d) {
```

nom	type	[3]	[2]	[1]	[0]
`__m128 _mm_load_ss(float *p)`	charge un réel en partie basse	0	0	0	p[0]
`__m128 _mm_load1_ps(float *p)`	charge un réel et copie 4 fois	p[0]	p[0]	p[0]	p[0]
`__m128 _mm_loadu_ps(float *p)`	charge 4 réels non alignés	p[3]	p[2]	p[1]	p[0]
`__m128 _mm_load_ps(float *p)`	charge 4 réels alignés	p[3]	p[2]	p[1]	p[0]
`__m128 _mm_set_ss(float w)`	affecte un réel en partie basse	0	0	0	w
`__m128 _mm_set1_ps(float w)`	affecte 4 fois le même réel	w	w	w	w
`__m128 _mm_setzero_ps (void)`	met à 0 les 4 valeurs	0	0	0	0
`__m128 _mm_set_ps(float z, float y, float x, float w)`	affecte les 4 floats	z	y	x	w

TABLE 8.5 – Chargement des flottants depuis la mémoire

```
    __m128 x;
    x = _mm_set_ps(a, b, c, d);
    x = _mm_hadd_ps(x,x);
    x = _mm_hadd_ps(x,x);
    float y;
    _mm_store_ss(&y, x);
    return y;
}
```

sera traduite par gcc/g++ en architecture 32 bits avec option `-O3` par :

```
function:
    sub         esp, 28
    movss       xmm1, [esp + 36]        ; xmm1 = [ -, -, -, b]
    movss       xmm0, [esp + 44]        ; xmm0 = [ -, -, -, d]
    insertps    xmm1, [esp + 32], 0x10  ; xmm1 = [ -, -, a, b]
    insertps    xmm0, [esp + 40], 0x10  ; xmm0 = [ -, -, c, d]
    movlhps     xmm0, xmm1              ; xmm0 = [ a, b, c, d]
    haddps      xmm0, xmm0
    haddps      xmm0, xmm0
    movss       [esp + 12], xmm0
    fld         dword [esp + 12]
    add         esp, 28
    ret
```

alors que les premières instructions (lignes 3 à 7) peuvent être remplacées par :

```
1     movups  xmm0, [esp + 4]
2     pshufd  xmm0, xmm0, 00011011b
```

Il est alors, dans certains, préférable d'écrire du code assembleur qui sera plus rapide.

8.5.2.2 Stocker des flottants en mémoire

On retrouve les opérations similaires à celles de chargement comme celles de la Table 8.6.

nom	type	[3]	[2]	[1]	[0]
`__m128 _mm_store_ss(float *p)`	stocke le réel en partie basse				p[0]
`__m128 _mm_store1_ps(float *p)`	stocke un réel et copie 4 fois	p[0]	p[0]	p[0]	p[0]
`__m128 _mm_storeu_ps(float *p)`	stocke 4 réels non alignés	p[3]	p[2]	p[1]	p[0]
`__m128 _mm_store_ps(float *p)`	stocke 4 réels alignés	p[3]	p[2]	p[1]	p[0]
`__m128 _mm_set_ss(float w)`	affecte un réel en partie basse	0	0	0	w
`__m128 _mm_set1_ps(float w)`	affecte 4 fois le même réel	w	w	w	w
`__m128 _mm_setzero_ps (void)`	met à 0 les 4 valeurs	0	0	0	0
`__m128 _mm_set_ps(float z, float y, float x, float w)`	affecte les 4 floats	z	y	x	w

TABLE 8.6 – Stockage des flottants en mémoire

8.5.3 Travailler avec les entiers

On trouve le même genre d'instructions que pour les réels avec bien entendu quelques différences ainsi que les instructions évoquées dans les sections précédentes. Concernant le chargement des données, on dispose, entre autre, des intrinsics suivantes :

- `__m128i _mm_load_si128(__m128i const* mem_addr)` permet de charger 16 octets situés à l'adresse `mem_addr` multiple de 16 dans un registre SSE, il s'agit de l'instruction `movdqa`
- `__m128i _mm_loadu_si128(__m128i const* mem_addr)` permet de charger 16 octets situés à l'adresse `mem_addr` dans un registre SSE, il s'agit de l'instruction `movdqu`
- `__m128i _mm_loadu_si32(void const* mem_addr)` charge en partie basse d'un registre SSE la valeur 32 bits située à l'adresse `mem_addr`, il s'agit de l'instruction `movd`
- `__m128i _mm_loadu_si64(void const* mem_addr)` charge en partie basse d'un registre SSE la valeur 64 bits située à l'adresse `mem_addr`, il s'agit de l'instruction `movq`
- `__m128i _mm_set_epi32(int e3, int e2, int e1, int e0)` remplit un registre SSE avec quatre valeurs entière, `e0` étant positionné en partie basse du registre et `e3` en partie haute
- `__m128i _mm_set1_epi32(int a)` stocke quatre fois la valeur entière `a` dans un registre SSE
- `void _mm_store_epi32 (void* mem_addr, __m128i a)` stocke le registre SSE à l'adresse indiquée qui doit être multiple de 16
- `void _mm_store_si128 (__m128i* mem_addr, __m128i a)`, équivalente à la précédente

8.5.4 Exemple de programme

On considère deux vecteurs d'entiers `u` et `v` de `size` éléments et on réalise l'addition `u[i] += v[i]`.

La première version qui n'utilise pas les intrinsics est évidente :

```
void add_no_SSE(int *u, int *v, int size) {
    for (int i = 0; i < size; ++i) {
        u[i] += v[i];
    }
}
```

La seconde qui utilise les intrinsics, et, dans notre cas, les registres SSE de 128 bits, nécessite de déplier la boucle `for` par 4.

```
void add_SSE(int *u, int *v, int size) {
    int i = 0;
    for (; i < size & ~3 ; i += 4) {
        // charger quatre entiers de chaque tableau
        __m128i x1 = _mm_loadu_si128((__m128i*) &u[i]);
        __m128i x2 = _mm_loadu_si128((__m128i*) &v[i]);

```

```
        // additionner en parallèle de quatre entiers
        x1 = _mm_add_epi32(x1, x2);

        // stocker le résultat
        _mm_storeu_si128((__m128i*) &u[i], x1);
    }

    // dernières itérations
    while (i < size) {
        u[i] += v[i];
        ++i;
    }
}
```

A chaque itération de la boucle `for` on charge dans le vecteur `x1` les éléments `u[i:i+3]` et on fait de même avec `x2` qui stocke `v[i:i+3]`. On réalise ensuite l'addition de `x1` avec `x2` et on met le résultat dans `x1`. Puis, en fin de boucle, on stocke le résultat contenu de `x1` dans `u[i:i+3]` en mémoire.

La version AVX demande d'utiliser des intrinsics qui débutent par `_mm256` et de déplier la boucle par 8 :

```
void add_AVX(int *u, int *v, int size) {
    int i = 0;
    for (; i < size & ~7 ; i += 8) {
        // charger huit entiers de chaque tableau
        __m256i x1 = _mm256_loadu_si256((__m256i*) &u[i]);
        __m256i x2 = _mm256_loadu_si256((__m256i*) &v[i]);

        // additionner en parallèle
        x1 = _mm256_add_epi32(x1, x2);

        // stocker le résultat
         _mm256_storeu_si256((__m256i*) &u[i], x1);
    }

    // dernières itérations
    while (i < size) {
        u[i] += v[i];
        ++i;
    }
}
```

La Table 8.7 rapporte les temps d'exécution d'un test de performance avec une version non vectorisée (`no_sse`) et les version vectorisée en SSE et AVX. Les lettres a et u indiquent si les données sont alignées ou non alignées.

On note donc que la vectorisation permet de diminuer le temps de calcul, mais également que le fait d'aligner ou non les données peut n'avoir aucune influence (Ryzen 5), ou diminuer le temps de calcul (Xeon), ou alors l'augmenter (i3).

Méthode	Intel Core i3 6100	AMD Ryzen 5 3600	Intel Xeon Silver 4208
add_no_sse	5.61	4.43	8.40
add_sse_u	2.33	2.20	2.94
add_avx_u	2.32	2.18	2.72
add_sse_a	2.62	2.19	2.67
add_avx_a	2.41	2.17	2.59

TABLE 8.7 – Temps d'exécution en secondes pour le calcul de la somme de vecteurs de 131_079 entiers répété 100_000 fois

8.6 AVX 512

8.6.1 Spécificités

L'AVX 512 a été proposé par Intel en 2013, puis a ensuite été implantée dans les Xeon Phi et différents processeurs haut de gamme de type Skylake X comme le Core i9 7980XE. Plusieurs catégories du jeu d'instruction existent, on en dénombre près d'une vingtaine comme l'AVX512-F (*Foundations*), l'AVX512-BW (*Byte and Word Instructions*) ou encore l'AVX512-VNNI (*Vector Neural Network Instructions*) dédié à l'apprentissage artificiel par réseau de neurones.

Par exemple, sur un Intel Xeon Silver 4208 on trouve les jeux suivants : AVX512-F, AVX512-DQ, AVX512-CD, AVX512-BW, AVXx512-VL et AVX512-VNNI.

On dispose avec l'AVX 512 de 32 registres de 512 bits nommés **`zmm0`** à **`zmm31`**. On note également l'apparition de 8 registres de masque nommés **`k0`** à **`k7`** (**`k0`** ayant un rôle particulier) qui permettent de sélectionner les octets, mots, double ou quadruple mots que l'on utilise dans une opération. Ces registres ont chacun une taille de 64 bits si on dispose du jeu d'instructions AVX512-BW. L'ensemble des instructions qui manipulent les registres de masque commence par la lettre `k`.

Certains testeurs ont remarqué que l'AVX512 peut causer des problèmes de ralentissement dans certains cas. Ces ralentissements seraient dûs au fait que les unités de traitement AVX512 fonctionnent à une fréquence inférieure à celle des unités de traitement de l'AVX2 pour certaines instructions qui demandent un calcul intensif ou pour des portions de code qui n'utilisent que des instructions AVX512. L'abaissement de la fréquence consiste probablement à diminuer (ou tout au moins à ne pas augmenter) la dissipation thermique.

Une autre explication, trouvée sur le site d'Intel, donne pour cause probable le fait que les processeurs modernes disposent de trois unités de traitement AVX2 (ports p0, p1, p5) alors qu'il se limite à deux unités de traitement AVX512 car

le port p1 serait rendu non utilisable quand des instructions AVX512 sont dans l'ordonnanceur.

Il est également recommandé d'utiliser l'instruction `vzeroupper` après utilisation des instructions AVX512 car le processeur vérifie si les bits les plus significatifs des registres vectoriels sont *propres* (initialisés à zéro) ou *sales* (contenant potentiellement des données). Lorsque les registres sont propres, le processeur peut traiter les registres de 128 bits comme de véritables registres de 128 bits. Néanmoins, si le registre est sale, le processeur doit en fait traiter le registre comme un registre de 512 bits. On conseille également d'utiliser `vzeroupper` avant de passer d'un code AVX à un code SSE. La documentation Intel indique que cela permet d'éviter les penalités (soit à peu près 70 cycles) liées à la performance engendrée par les fausses dépendances (*it will eliminate performance penalties caused by false dependencies*), ce qui n'est pas très explicite, on aimerait savoir à quoi correspondent ces fausses dépendances.

De plus amples explications sont données sur divers blogs.

8.6.2 Manipulation des masques

On utilise l'instruction `kmov`, suffixée par une quantité (B,W,D,Q), pour manipuler les registres de masque entre eux ou pour échanger une valeur de masque avec un registre général.

Les autres opérations de manipulation des registres de masque sont :

- l'opération d'addition (`kadd`),
- les opérations logiques (`kand`, `kandn`, `kor`, `kxor`, `kxnor`),
- des opérations de test (`ktest`, `kortest`),
- de décalage (`kshiftl`, `kshiftr`),
- et enfin des opérations de décompactage (`kunpck`)

8.6.3 Données vectorielles

En ce qui concerne les données vectorielles, on utilise les mêmes opérations qu'en AVX ou AVX2 avec la possibilité de combiner ces instructions avec un masque, ce qui peut rendre certains traitements plus simples à coder. On pourra se référer aux Chapitres 14 et 15 pour de plus amples explications.

A titre d'exemple, voici dans le cadre de la parcimonie, comment utiliser les instructions AVX512. On considère que le registre `zmm0` a été mis à 0 et que `zmm3` contient le résultat d'un **et binaire** en parallèle entre les valeurs contenues dans les registres `zmm1` et `zmm2`. De la même manière, `zmm4` contient le résultat d'un **ou binaire** en parallèle entre `zmm1` et `zmm2`. Pour obtenir le résultat final qui consiste

à compter le nombre d'octets à 0 dans zmm3 et à les additionner à eax, puis à remplacer les octets de zmm3 qui sont à 0 par ceux de zmm4, on utilise :

```
    vpcmpeqb    k1, zmm0, zmm3      ; compare zmm3 a zmm0 = [0,...,0]
                                    ; et affecte k1 en consequence

    vmovdqu8    zmm3 {k1}, zmm4     ; selectionne les octets de zmm4 en
                                    ; utilisant k1 et remplace dans zmm3
    vmovdqu8    [rdx + r9], zmm3    ; affecte le résultat final

    kmovq       r8, k1              ; met le masque k1 dans r8
    popcnt      r8, r8              ; compte le nombre de bits à 1
    add         eax, r8d            ; additionne à eax
```

On fait usage en premier lieu de l'instruction de comparaison octet par octet vpcmpeqb en indiquant le masque k1. Les registres zmm0 et zmm3 ne seront donc pas modifiés et chaque bit i de k1 sera positionné à 1 si le $i$ème bit de zmm3 est égal à 0. L'instruction suivante remplace chaque octet i de zmm3 par l'octet i de zmm4 si le $i$ème bit de k1 est à 1. On stocke ensuite le résultat en mémoire. Enfin, on place k1 dans le registre 64 bits r8 afin de compter le nombre de bits à 1 de k1 en utilisant l'instruction popcnt. Ce résultat est ensuite ajouté à eax qui comptabilise le nombre de mutations, c'est à dire dans le cas présent, le nombre d'octets à 0 après calcul du **et binaire** entre zmm1 et zmm2.

Voici un autre exemple de code qui utilise le mnémonique vcompressps avec un masque de sélection. On désire réaliser le traitement suivant en C :

```
int compress(float *x, float *y) {
    int count = 0;
    for (int i = 0; i < 8; ++i) {
        if (x[i] > 0.0) {
            y[count++] = x[i];
        }
    }
    return count;
}
```

Etant donnés deux vecteurs de 8 float, on désire sélectionner les éléments de x qui sont supérieurs à 0 et les recopier dans y avec une contrainte qui impose qu'ils doivent être positionnés les uns à la suite des autres. On désire également retourner le nombre d'éléments recopiés. La traduction en assembleur avec des instructions AVX512 utilise l'instruction de comparaison cmpps qui stocke dans le masque k1 les éléments qui correspondent au critère de comparaison, en l'occurrence les éléments qui vont être supérieurs à 0.

```
    vxorps      ymm0, ymm0          ; ymm0 = [0.0, ..., 0.0]
    vmovdqu     ymm1, [rdi]         ; ymm1 <- x[0:7]
    vcmpps      k1, ymm0, ymm1, 5   ; compare ymm1[i] à ymm0[i]
    knotq       k1, k1              ; choix des éléments à garder
```

```
5       vcompressps ymm0 {k1}, ymm1     ; recopie des éléments de ymm1
6                                       ; vers ymm0
7       vmovdqu     [rsi], ymm0         ; stockage dans y
8       kmovq       rax, k1             ; compte le nombre d'éléments
9       popcnt      rax, rax            ; sélectionnés
10      vzeroupper
11      ret
```

Comme indiqué dans la documentation Intel[6], on doit inverser le masque de `k1` puis recopier les bits qui seront sélectionnés par `vpcompressps`. Pour compter le nombre d'éléments recopiés, il suffit de comptabiliser grâce à l'instruction `popcnt` le nombre de bits dans `k1` après l'avoir inversé (ligne 4). Pour cela, on transfère `k1` dans `rax` (ligne 8) et on applique `popcnt` sur `rax`.

Sur un Intel Xeon Silver 4208, un test de performance donne les résultats suivants (le code C est compilé avec `gcc` 8.3.0) :

- fonction C compilée avec `gcc -O3 -march=native` (avec AVX512) : 4,31 s
- fonction assembleur AVX512 écrite à la main : 1,77 s

8.7 AVX 10

Au mois de Juillet 2023, Intel a introduit l'AVX10 (*Advanced Instruction Extensions 10*) qui apparaît comme un sur-ensemble de l'AVX-512 que pourrait également supporter des processeurs avec des registres de 256 bits. En fait, avec l'introduction d'une architecture hybride avec des P-Core et E-Core avec l'arrivée d'Alder Lake la 12ème génération de processeurs Core d'Intel en 2021, l'utilisation de l'AVX devient problématique car les P-Cores sont capables de gérer de l'AVX-512 alors que les E-Cores ne gèrent que l'AVX2. On a donc deux codes incompatibles actuellement. Afin de simplifier l'écriture du code et pouvoir le transposer sur les E-Cores il semble intéressant de pouvoir reprendre les principes de l'AVX-512 et le transposer à l'AVX2, notamment l'utilisation de masques qui rend le code plus compréhensible.

La première version nommée AVX10.1 doit permettre la transition vers le 256 bits et ne supportera donc que le 512 bits sur les processeurs Xeon Granite Rapids. La version AVX10.2 devrait parachever le support pour tous les processeurs qui supporteront l'AVX2.

6. The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

8.8 Conclusion

Vectoriser son code est une opération peu coûteuse et qui permet d'obtenir une amélioration importante en terme de diminution du temps de calcul. On pourra consulter les chapitres qui concernent les différentes études de cas que nous présentons pour constater que la vectorisation est un outil incontournable de l'efficacité.

8.8.1 Que retenir ?

- ▷ les unités vectorielles sont qualifées de SIMD (*Single Instruction Multiple Data*) et permettent de paralléliser les calculs en effectuant la même opération sur des données différentes
- ▷ les unités SSE permettent de traiter 128 bits alors que les unités AVX sont capables de traiter 256 bits de différents formats allant de l'octet au nombre à virgule flottante double précision
- ▷ la technologie AVX apporte généralement, dans la plupart des traitements, un gain négligeable mais peut, dans certains cas, se révéler deux fois plus efficace que le SSE
- ▷ les *intrinsics* sont un moyen détourné pour permettre au programmeur d'utiliser les instructions assembleurs liées à la vectorisation tout en programmant en C. Elles assurent la portabilité du code tout en autorisant l'amélioration de l'efficacité des traitements de manière importante.
- ▷ les registres de l'AVX 512 sont au nombre de 32 et se nomment `zmm0` à `zmm1`.
- ▷ avec l'AVX 512 on utilise des bits de masque (`k0` à `k7`) qui définissent les emplacements dans un registre de 512 bits qui devront être sélectionnés pour une opération ultérieure

8.8.2 Compétences à acquérir

Au cours du temps et au fil des différents projets de programmation que vous pourrez rencontrer il faudra tenter de vectoriser les traitements les plus lents. Pour cela il faut connaître les instructions vectorielles.

8.9 Exercices

Exercice 35 - Implantez, en utilisant les registres SSE, la fonction `iota` définie par :

```
void iota(float *t, int n) {
    for (int i=0; i<n; ++i) {
        t[i] = (float) i;
    }
}
```

On commencera par écrire une version dépliée par 4 de la fonction dans le cas général (c'est à dire quand `n` n'est pas multiple de 4), puis on traduira la partie vectorielle en utilisant `addps` et pour les itérations restantes on utilisera `addss`.

On peut faire en sorte que qu'initialement le registre `xmm1 = [4, 3, 2, 1]` et que le registre `xmm0 = [3, 2, 1, 0]`.

A chaque itération on stocke `xmm0` à l'adresse de `t[i]` puis on augmente `i` de 4 et on ajoute `xmm1` à `xmm0`.

Exercice 36 - Donnez une version intrinsics de la fonction `iota`.

Exercice 37 - Donnez une version vectorielle de la fonction suivante qui convertit une chaîne de caractères en majuscules :

```
void string_to_upper(char *s, size_t size) {
    for (size_t i = 0; i < size; ++i) {
        if (isalpha(s[i])) s[i] = toupper(s[i]);
    }
}
```

Exercice 38 - Donnez une version vectorielle de la fonction suivante qui convertit une chaîne de caractères en minuscules et retourne le nombre de changements effectués :

```
size_t string_to_lower(char *s, size_t size) {
    size_t changes = 0;
    for (size_t i = 0; i < size; ++i) {
        if (isalpha(s[i])) {
            if (islower(s[i])) {
                s[i] = tolower(s[i]);
                ++changes;
            }
        }
    }
    return changes;
}
```

Chapitre 9

Algèbre de Boole

2B or not(2B)

là est la question !

9.1 Introduction

Nous rappelons que ce chapitre est quelque peu digressif par rapport à l'apprentissage de la programmation en assembleur mais il se base sur l'algèbre de Boole qui utilise le *et*, le *ou* et le *non*, opérations disponibles en assembleur en tant que mnémoniques `and`, `or`, `not`.

L'algèbre de Boole, du nom du mathématicien, logicien et philosophe britannique *George Boole* (1815-1864), est une partie des mathématiques qui s'intéresse à une approche algébrique de la logique, alors que la logique se fonde sur des systèmes de réécriture qui consistent à manipuler des symboles. La logique possède bien évidemment un volet sémantique et l'algèbre de Boole vient renforcer la sémantique logique en remplaçant le vrai et le faux par le 1 et le 0, le *et* et le *ou* par les opérateurs $+$ et $\cdot$ (addition et multiplication).

Cette vision arithmétique de la logique a permis de mettre au point un système de calcul qui possède des applications dans la mise au point de circuits électroniques et autorise à aborder les problèmes de la logique sous un angle différent, ce qui peut, dans certains cas, donner la possibilité de résoudre un problème beaucoup plus simplement ou rapidement. Nous en verrons un exemple en fin de chapitre avec le problème des pigeons.

Concernant les circuits électroniques ce sont eux qui exécutent les mnémoniques de l'assembleur comme `add`, `mul`, `div`, ... Nous verrons, Section 9.6, comment on implante un demi-additionneur et un additionneur.

On suppose que le lecteur possède des notions de logique propositionnelle. Si ce n'est pas le cas, nous recommandons la lecture des premiers chapitres de [23].

9.2 Définition

Soit un ensemble $\mathcal{A} = \{0, 1\}$ pour lequel on a $0 \leq 1$. On définit alors les opérations suivantes sur $\mathcal{A}$:

- l'addition : $a + b$ dont la sémantique est $max(a, b)$
- la multiplication : $a \cdot b = min(a, b)$
- la complémentation : $\overline{0} = 1$ et $\overline{1} = 0$

Une variable complémentée $\overline{x}$ est également dite *signée* ou négative.

Le résultat des opérations $+$ et $\cdot$ apparaît Table 9.1 d'après la sémantique que nous avons donnée.

a	b	$a + b$	$a \cdot b$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

TABLE 9.1 – Interprétation semantique des opérations $+$ et .

Le quadruplet $(\mathcal{A}, +, \cdot, \overline{x})$ est appelé *algèbre de Boole* s'il respecte les axiomes suivants :

1. l'addition et la multiplication sont **commutatives** et **associatives** :

$$a + b = b + a$$
$$a \cdot b = b \cdot a$$
$$(a + b) + c = a + (b + c)$$
$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

2. 0 est **élément neutre** pour l'addition et 1 est **élément neutre** pour la multiplication :

$$0 + a = a$$
$$1 \cdot a = a$$

3. l'addition et la multiplication sont **distributives** l'une par rapport à l'autre :

$$(a + b) \cdot c = a \cdot c + b \cdot c$$
$$(a \cdot b) + c = (a + c) \cdot (b + c)$$

4. la complémentation est telle que $\overline{\overline{a}} = a$ et vérifie les propriétés suivantes :

$$a + \overline{a} = 1$$
$$a \cdot \overline{a} = 0$$

Si l'on rapporte ces opérations à la logique, alors :

- 1 indique le caractère *vrai* d'une propriété ou d'un énoncé
- 0 indique le caractère *faux*
- l'addition $(+)$ correspond au *ou*
- la multiplication $(.)$ correspond au *et*
- la complémentation $\overline{a}$ correspond à *non* (noté $\neg$ en logique), i.e. le *contraire* de a

Ainsi, l'expression $a + \overline{a} = 1$ peut s'interpréter : dire qu'une chose est vraie ou n'est pas vraie est toujours vrai. Je peux par exemple remplacer a par l'énoncé *il pleut*, et donc, dans ce cas, il est vrai que : il pleut ou il ne pleut pas.

De la même manière $a \cdot \overline{a} = 0$ signifie qu'on ne peut pas dire une chose et son contraire. Je ne peux pas à la fois être grand et ne pas être grand.

C'est grâce à cette modélisation de la logique sous forme d'opérations arithmétiques que l'on peut simplifier certains traitements modélisés sous forme de fonctions booléennes.

9.3 Fonction booléenne, table de vérité

On appelle fonction *booléenne*, une application de $\mathcal{A}^n$ dans $\mathcal{A}$:

$$(x_1, x_2, \cdots, x_n) \longrightarrow f(x_1, x_2, \cdots, x_n)$$

La manière la plus simple de définir une fonction booléenne f est de donner sa table de vérité, c'est à dire, l'ensemble des n-uplets :

$$(x_1, x_2, \cdots, x_n, f(x_1, x_2, \cdots, x_n))$$

Les variables x_i prenant leurs valeurs dans $\mathcal{A} = \{0, 1\}$, une fonction de n variables possèdent donc $card(\mathcal{A})^n = 2^n$ lignes, avec $Card()$ qui donne la cardinalité d'un ensemble.

Prenons par exemple une fonction $f_1(x, y, z)$, définie par sa table de vérité, comme suit :

Ligne	x	y	z	$f_1(x,y,z)$
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

A partir de la table de vérité d'une fonction, on est en mesure de donner une expression de celle-ci sous forme algébrique en tant que **somme de monômes** :

- il suffit d'exprimer les monômes pour les lignes pour lesquelles la fonction $f_1(x,y,z) = 1$
- si une variable est à 0 sur cette ligne, on utilise son complément

Avec l'exemple précédent, on obtient :

$$f_1(x,y,z) = \underbrace{\overline{x}.\overline{y}.\overline{z}}_{Ligne\,0} + \underbrace{\overline{x}.y.z}_{Ligne\,3} + \underbrace{x.y.\overline{z}}_{Ligne\,6} + \underbrace{x.y.z}_{Ligne\,7}$$

On remarque dans la table de vérité que les variables x, y et z suivent la notation binaire et que pour la ligne 6, on a bien $x = 1, y = 1, z = 0$ qui correspond à $110_2 = 6$.

Notation

Il existe un moyen plus simple et plus rapide de décrire la table de vérité d'une fonction booléenne en indiquant les lignes de la table de vérité qui comportent des 1 et qui définissent la fonction. Ainsi, une fonction f peut être est décrite par : $f(x,y,z,t) = (3,4,5,6,7,9,13,14,15)$ ou encore par $f(x,y,z,t) = (3-7,9,13-15)$, où l'expression $3-7$ signifie de 3 à 7.

9.3.1 Fonctions de deux variables

Dans le cas particulier des fonctions à 2 variables $f(x,y)$, on peut définir 16 fonctions différentes dont certaines sont identifiée par un nom. On retrouve notamment :

- $or(x,y)$, c'est à dire le *ou* : $x + y$
- $and(x,y)$, le *et* : $x \cdot y$

- $nor(x, y) = \overline{or(x, y)}$, le *non ou* : $\overline{x + y}$
- $nand(x, y) = \overline{and(x, y)}$, le *non et* : $\overline{x \cdot y}$
- $xor(x, y)$, le *ou exclusif* qui est vrai uniquement si l'une de ses opérandes est vraie : $\overline{x} \cdot y + x \cdot \overline{y} = x \oplus y$

x	y	$or(x, y)$	$and(x, y)$	$xor(x, y)$	$nor(x, y)$	$nand(x, y)$
0	0	0	0	0	1	1
0	1	1	0	1	0	1
1	0	1	0	1	0	1
1	1	1	1	0	0	0

9.3.1.1 La fonction $and(x, y)$ (ET logique)

Le ET logique (voir Table 9.2) vaut 1 uniquement si ses deux opérandes x et y valent 1.

x	y	$and(x, y)$
0	0	0
0	1	0
1	0	0
1	1	1

TABLE 9.2 – La fonction $and(x, y)$

En d'autres termes, dans un programme, pour que la condition `(x and y)` soit vraie, il faut que les deux sous-conditions `x` et `y` soient vraies.

```
if ((0 < a) and (a < 11)) {
    // a est compris entre 1 et 10
} else {
    // a est en dehors de l'intervalle [1..10]
}
```

9.3.1.2 La fonction $or(x, y)$ (OU Logique)

Le OU logique (voir Table 9.3) vaut 0 uniquement si ses deux opérandes x et y valent 0.

En d'autres termes, pour que la condition `(x or y)` soit vraie, il faut que l'une des deux sous-conditions `x` ou `y` (ou les deux) soient vraies.

x	y	$or(x, y)$
0	0	0
0	1	1
1	0	1
1	1	1

TABLE 9.3 – La fonction $or(x, y)$

```
if ((a > 10) or (a < 1)) {
    // a est en dehors de l'intervalle [1..10]
} else {
    // a est compris entre 1 et 10
}
```

9.3.1.3 La fonction $xor(x, y)$ (OU Exclusif Logique)

Le OU Exclusif logique (voir Table 9.4) vaut 0 uniquement si ses deux opérandes x et y ont la même valeur.

x	y	$xor(x, y)$
0	0	0
0	1	1
1	0	1
1	1	0

TABLE 9.4 – La fonction $xor(x, y)$

En d'autres termes, pour que la condition `(x xor y)` soit vraie, il faut que l'une des deux sous-conditions `x` et `y` soit vraie, mais pas les deux en même temps.

```
if ((est_un_poisson(a) xor vole(a))) {
    // élimine les poissons volants
    // Traite les poissons ou les animaux qui volent
    // mais pas les deux en même temps
}
```

9.3.1.4 Lois de De Morgan

Les lois de **De Morgan** (mathématicien et logicien britannique, 1806-1871) permettent d'exprimer la transformation de la négation d'un ET ou d'un OU logique.

- NON(x ET y) est équivalent à NON(x) OU NON(y), que l'on peut énoncer sous la forme : le complémentaire du produit est la somme des complémentaires :
$$\overline{x.y} = \overline{x} + \overline{y}$$
- NON(x OU y) est équivalent à NON(x) ET NON(y), que l'on peut énoncer sous la forme : le complémentaire de la somme est le produit des complémentaires :
$$\overline{x + y} = \overline{x}.\overline{y}$$

Par exemple, soit la condition `(0 < a) and (a < 11)` vue précédemment dans l'exemple sur le ET. Il existe différentes manières d'en prendre la négation :

- `not( (0 < a) and (a < 11) )`
- `not(0 < a) or not(a < 11)`, transformation par **De Morgan**
- `(0 >= a) or (a >= 11)`, application du **not** sur chaque sous-condition
- `(a <= 0) or (a >= 11)`
- `(a < 1) or (a > 10)`

9.4 Simplification des fonctions booléennes

L'intérêt de l'algèbre de Boole est qu'elle permet de simplifier les fonctions booléennes comme on le ferait d'une expression algébrique sur les entiers ou les réels.

9.4.1 Règles de simplification algèbriques

Deux fonctions booléennes sont dites identiques si elles possèdent la même table de vérité. Cette propriété nous permet d'établir un certain nombre d'identités et de règles de simplification :

Loi		Forme $+$		Forme $\cdot$
élément neutre	$(R1)$	$x + 0 = x$	$(R2)$	$x \cdot 1 = x$
d'idempotence	$(R3)$	$x + x = x$	$(R4)$	$x \cdot x = x$
d'inversion	$(R5)$	$x + \overline{x} = 1$	$(R6)$	$x \cdot \overline{x} = 0$
d'absorption	$(R7)$	$x + x \cdot y = x$	$(R8)$	$x \cdot (x + y) = x$
de De Morgan	$(R9)$	$\overline{x + y} = \overline{x} \cdot \overline{y}$	$(R10)$	$\overline{x \cdot y} = \overline{x} + \overline{y}$
de commutativité	$(R11)$	$x + y = y + x$	$(R12)$	$x \cdot y = y \cdot x$
d'associativité	$(R13)$	$x + (y + z) = (x + y) + z$	$(R14)$	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$
de distributivité	$(R15)$	$x \cdot (y + z) = x \cdot y + x \cdot z$	$(R16)$	$x + y \cdot z = (x + y) \cdot (x + z)$

Essayons d'appliquer ces règles pour simplifier la fonction $f_2(x, y, z)$:

$$f_2(x, y, z) = \overline{x} \cdot y \cdot z + \overline{x} \cdot y \cdot \overline{z} + x \cdot z$$

Cette fonction peut être réécrite sous une forme simplifiée en :

$$\begin{array}{lll} f_2(x, y, z) & = \overline{x} \cdot y \cdot z + \overline{x} \cdot y \cdot \overline{z} + x \cdot z & \\ & = \overline{x} \cdot y \cdot (z + \overline{z}) + x \cdot z & (factorisation) \\ & = \overline{x} \cdot y \cdot 1 + x \cdot z & (R5) \\ & = \overline{x} \cdot y + x \cdot z & \end{array}$$

La fonction ainsi réduite possède deux termes et est plus facile à concevoir sous forme de schéma électronique car elle utilise moins de symboles donc moins de portes logiques. Ces portes logiques sont implantées sous forme de transistors. Si on utilise moins de transistors on peut réduire la taille des circuits électroniques.

On utilise également d'autres formules de simplification parmi lesquelles :

$$\begin{array}{lrcl} (R17) & x + \overline{x} \cdot y & = & x + y \\ (R18) & x \cdot y + \overline{x} \cdot z + y \cdot z & = & x \cdot y + \overline{x} \cdot z \end{array}$$

9.4.2 Méthode des tableaux de Karnaugh

La méthode des tableaux de Karnaugh a été développée par *Maurice Karnaugh*, ingénieur américain en télécommunications aux Bell Labs en 1953. Il s'agit d'un procédé de simplification visuel, pratique, qui ne s'applique qu'à des fonctions booléennes composées au maximum de 6 variables, car au delà, cela devient extrêmement complexe de visualiser les simplifications. Elle consiste à représenter sous une forme particulière la table de vérité de la fonction afin de procéder à des regroupements qui correspondent à l'élimination d'une variable qui apparaît sous une forme positive dans un terme et négative dans un autre terme :

$$x \cdot y \cdot z + x \cdot \overline{y} \cdot z = x \cdot z \cdot \underbrace{(y + \overline{y})}_{(R5)=1} = x \cdot z$$

La simplification est réalisée en deux étapes :

1. on commence par créer un tableau de Karnaugh de la fonction à simplifier
2. puis on simplifie l'expression par réunion de 2^n cases adjacentes (c'est à dire par groupe de $2, 4, 8, 16, 32$) en évitant les regroupements redondants

9.4.3 Création et remplissage du tableau de Karnaugh

Nous présentons Figures 9.1, 9.2, deux agencements d'un tableau de Karnaugh pour une fonction de trois variables. La Figure 9.3 représente, quant à elle, une représentation pour une fonction de quatre variables.

On remarque que les lignes ou les colonnes ont une organisation particulière : quand on passe d'une ligne (ou d'une colone) à la suivante (ou la précédente) on ne change le signe que d'une seule variable.

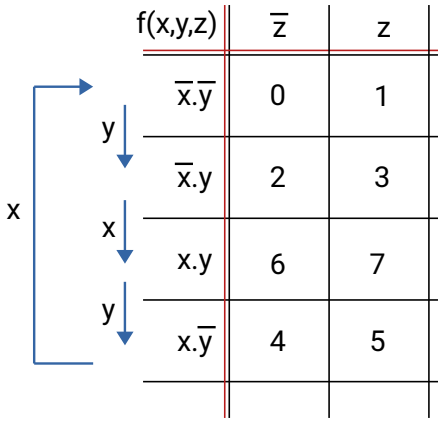


FIGURE 9.1 – Tableau de Karnaugh pour une fonction de 3 variables (version 1)

f(x,y,z)	$\overline{y}.\overline{z}$	$\overline{y}.z$	y.z	y.$\overline{z}$	
$\overline{x}$	0	1	3	2	
x	4	5	7	6	

FIGURE 9.2 – Tableau de Karnaugh pour une fonction de 3 variables (version 2)

Dans chaque case des tableaux nous trouvons le numéro de la ligne de la table de vérité qui lui est associé. Ainsi pour la Figure 9.3, l'expression $x \cdot \bar{y} \cdot \bar{z} \cdot v$ correspond en binaire à 1001_2 soit 9 en décimal. La valeur 9 se trouve donc à l'intersection de la ligne $x \cdot \bar{y}$ et de la colonne $\bar{z} \cdot v$.

Pour remplir le tableau de Karnaugh on ne garde que les numéros des cases qui correspondent à des valeurs pour lesquelles la fonction booléenne vaut 1.

f(x,y,z,v)	$\overline{z}.\overline{v}$	$\overline{z}.v$	z.v	z.$\overline{v}$	
$\overline{x}.\overline{y}$	0	1	3	2	
$\overline{x}.y$	4	5	7	6	
x.y	12	13	15	14	
x.$\overline{y}$	8	9	11	10	

FIGURE 9.3 – Tableau de Karnaugh pour une fonction de 4 variables

9.4.4 Simplification du tableau de Karnaugh

Comme indiqué précédemment, on regroupe les 2^n cases adjacentes. Dans certains cas il n'est pas possible de regrouper des cases du tableau car elles ne sont pas adjacentes, on n'aura donc aucune simplification pour la case en question.

Tableau de Karnaugh : adjacence de deux cases

On peut définir l'adjacence de deux cases du tableau par le fait que leurs expressions algébriques ne diffèrent que par le signe d'une seule variable. Par exemple :

- $x \cdot y \cdot z$ et $x \cdot \overline{y} \cdot z$ sont adjacentes car elles ne diffèrent que par le changement de signe de y en $\overline{y}$
- alors que $x \cdot y \cdot z$ et $x \cdot \overline{y} \cdot \overline{z}$ diffèrent par le changement du signe de deux variables y et z et ne sont donc pas adjacentes

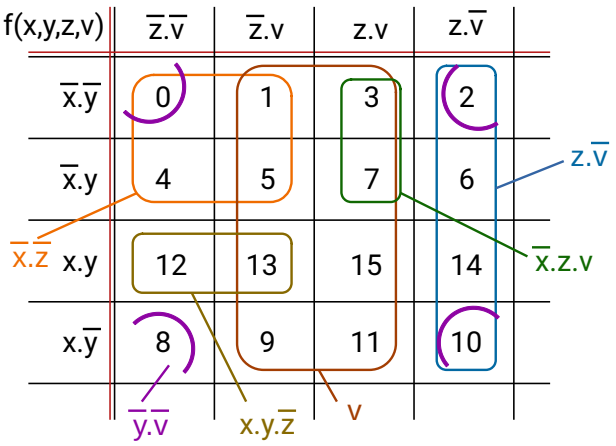


FIGURE 9.4 – Exemples d'adjacence dans un tableau de Karnaugh

La Figure 9.4 représente plusieurs situations d'adjacence. Pour une tableau de quatre variables, on peut regrouper les variables adajacentes par deux, quatre, huit ou seize. On a donc adjacence si, partant d'une case, on atteint une autre case :

- en passant d'une ligne à la suivante ou la précédente
- en passant d'une colonne à la suivante ou la précédente

Il nous suffit alors de regrouper des cases adjacentes afin de simplifier des expressions.

Tableau de Karnaugh : adjacence de plusieurs cases

Plusieurs cases sont adjacentes si elles sont au nombre de 2^n et qu'elles forment un carré ou un rectangle.

Attention : il est possible d'utiliser plusieurs fois la même case pour faire des regroupements différents. Cependant, si on regroupe des cases contenues dans un regroupement déjà effectué alors on produit un terme inutile ou redondant.

9.4.4.1 Exemple simple de simplification par tableau de Karnaugh

Voyons cela sur un exemple concret. Considérons la fonction booléenne donnée par l'expression algébrique :

$$\begin{array}{rcl} f_4(x,y,z) & = & \overline{x}\cdot\overline{y}\cdot\overline{z} \;+\; \overline{x}\cdot y\cdot z \\ & + & x\cdot\overline{y}\cdot z \;+\; x\cdot y\cdot\overline{z} \\ & + & x\cdot y\cdot z \end{array}$$

Le tableau de Karnaugh de cette fonction est représenté Figure 9.5.

La case 0 n'étant adjacente à aucune autre, elle restera seule et ne subira aucune simplification. Les cases $(3,7)$, $(6,7)$ et $(5,7)$ sont adjacentes on va donc les regrouper :

- pour 3 et 7, on a $\overline{x}\cdot y\cdot z + x\cdot y\cdot z = (\overline{x}+x)\cdot y\cdot z = y\cdot z$
- pour 5 et 7, on a $x\cdot\overline{y}\cdot z + x\cdot y\cdot z = (\overline{y}+y)\cdot x\cdot z = x\cdot z$
- pour 6 et 7, on a $x\cdot y\cdot\overline{z} + x\cdot y\cdot z = (\overline{z}+z)\cdot x\cdot y = x\cdot y$

Comme indiqué précédemment, bien que les cases $3,5,6,7$ soient adjacentes, on ne peut pas les regrouper car elles ne forment pas un carré ou un rectangle. Notamment 3 n'est pas adjacente à 5 ou 6. Au final, on obtient la simplification :

$$f_4(x,y,z) = \underbrace{\overline{x}\cdot\overline{y}\cdot\overline{z}}_{0} + \underbrace{y\cdot z}_{3+7} + \underbrace{x\cdot z}_{5+7} + \underbrace{x\cdot y}_{6+7}$$

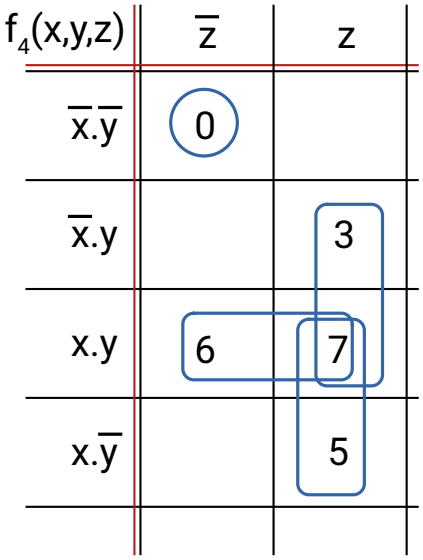


FIGURE 9.5 – Tableau de Karnaugh de $f_4(x, y, z)$

9.4.4.2 Exemple plus problématique

On considère la fonction de 3 variables $f_5(x, y, z) = (1, 3, 6, 7)$

Le tableau de Karnaugh de cette fonction est représenté Figure 9.6.

Si l'on y prête pas attention, on peut envisager de réaliser les regroupements suivants :

- $\alpha = (1, 3)$
- $\beta = (3, 7)$
- $\delta = (6, 7)$

On obtiendra alors la simplification :

$$f_5(x, y, z) = \underbrace{\overline{x} \cdot z}_{\alpha} + \underbrace{y \cdot z}_{\beta} + \underbrace{x \cdot y}_{\delta}$$

Cependant, comme le montre les regroupements sur la Figure 9.6, le terme associé à β est redondant, puisque le recouvrement avec α et δ suffit à regrouper tous les termes. On peut effectivement montrer que le terme β est redondant et donc inutile par simplification algébrique :

$$\begin{array}{rcll} \overline{x}z + yz + xy & = & \overline{x}z + yz \cdot (\overline{x} + x) + xy & 1 = (x + \bar{x}) \\ & = & \overline{x}z + \overline{x}yz + xyz + xy & (developpement) \\ & = & \overline{x}z \cdot (1 + y) + xy \cdot (1 + z) & (factorisation) \\ & = & \overline{x}z + xy & \end{array}$$

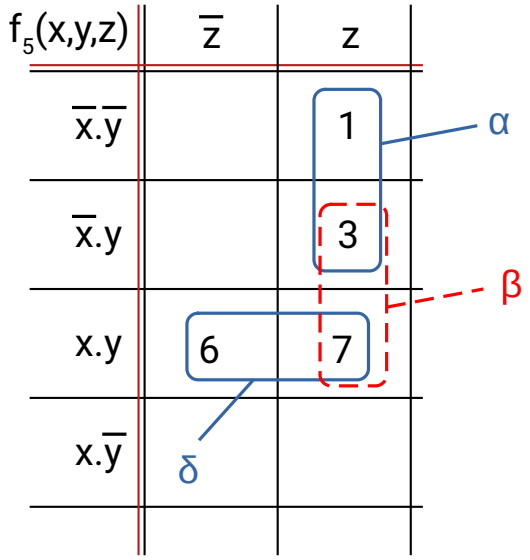


FIGURE 9.6 – Tableau de Karnaugh de $f_5(x, y, z)$

Il s'agit en fait d'une simplification qui s'apparente à la formule $(R18)$ vue précédemment.

9.5 Représentation des portes logiques

Les expressions booléennes sont représentables de manière graphique en utilisant des portes logiques. Les portes logiques sont généralement représentées par des symboles composés d'une ou plusieurs entrées et d'une sortie. Pour relier deux portes il suffit de relier la sortie de l'une à l'une des entrées de la seconde.

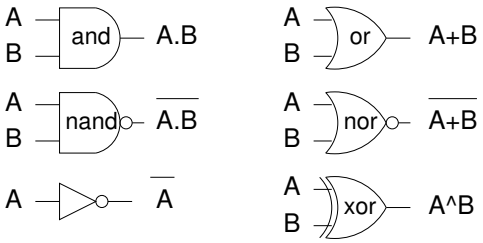


FIGURE 9.7 – Représentation des portes logiques

Par exemple, la fonction $f_6(x, y, z) = \overline{x} \cdot y \cdot z + \overline{x} \cdot y \cdot \overline{z} + x \cdot z$ est représentée Figure 9.8.

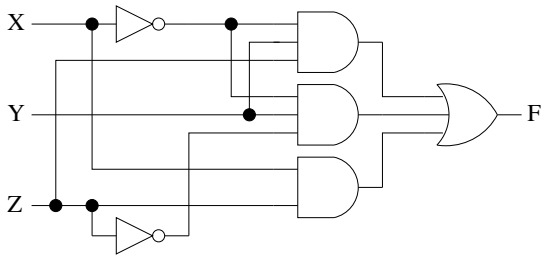


FIGURE 9.8 – Conception de la fonction F

9.5.1 Universalité des portes NAND et NOR

Les expressions booléennes se résument à des combinaisons entre trois types de portes couramment utilisées : NOT, AND, et OR. On remarquera que ces trois portes sont modélisables en fonction d'un seul type de porte : la porte NAND.

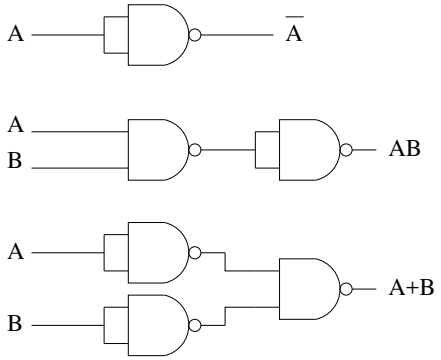


FIGURE 9.9 – Portes NOT, AND et OR en fonction de portes NAND

$$\overline{x} = \overline{x \cdot x}$$

$$x \cdot y = \overline{\overline{x \cdot y} \cdot \overline{x \cdot y}}$$

$$x + y = \overline{\overline{x \cdot x} \cdot \overline{y \cdot y}}$$

On peut également écrire les portes NOT, AND et OR en fonction de portes NOR (cf. figure 9.10).

$$\overline{x} = \overline{x + x}$$

$$x \cdot y = \overline{\overline{x + x} + \overline{y + y}}$$

$$x + y = \overline{\overline{x + y} + \overline{x + y}}$$

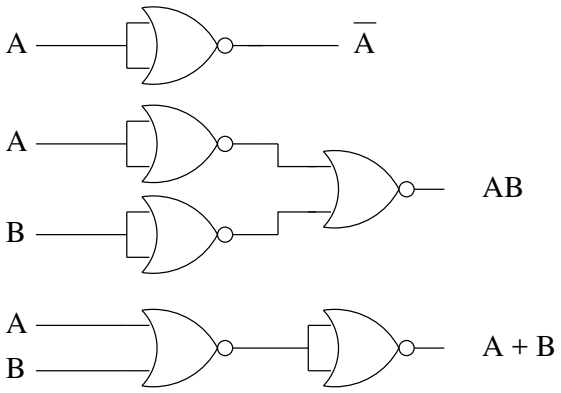


FIGURE 9.10 – Portes NOT, AND et OR en fonction de portes NOR

9.6 Algèbre de Boole et circuits

L'algèbre de Boole permet de modéliser le fonctionnement des circuits logiques et d'en simplifier l'implantation. Nous prendrons ici l'exemple de l'additionneur et du demi-additionneur qui sont deux circuits logiques très simples.

9.6.1 Le demi-additionneur

Le demi-additionneur est un circuit logique qui comporte deux entrées (x et y) et deux sorties et qui a pour but de calculer la somme $x + y$. Les deux entrées correspondent aux deux bits à additionner et les sorties à la somme S ainsi que la retenue en sortie R_s qui peut être générée. S et R_s sont donc des fonctions booléennes dont la table de vérité est donnée Table 9.5.

x	y	S	R_s
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

TABLE 9.5 – Table de vérité du demi-additionneur

Les fonctions S et R_s s'expriment donc sous forme algébrique par :

$$\begin{array}{rcccl} S(x,y) & = & \overline{x} \cdot y + x \cdot \overline{y} & = & x \oplus y \\ R_s(x,y) & = & x \cdot y & & \end{array}$$

9.6.2 L'additionneur

Un additionneur est un circuit qui comporte trois entrées et deux sorties. Il calcule la somme de ses trois entrées. Les trois entrées sont x, y et la retenue en entrée R_e qui correspond à un calcul précédent. Les deux sorties sont comme précédemment S et R_s. La table de vérité de l'additionneur est donc la suivante :

x	y	R_e	S	R_s
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

TABLE 9.6 – Table de vérité de l'additionneur

Le calcul en décimal est équivalent à $x + y + R_e = R_s \times 2 + S$.

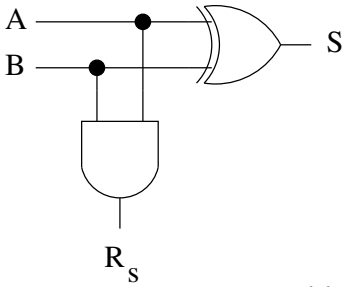


FIGURE 9.11 – Demi additionneur

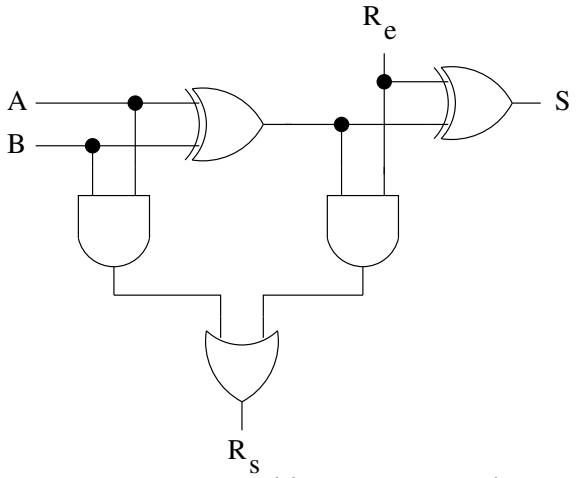


FIGURE 9.12 – Additionneur complet

Les fonctions booléennes S et R_s pour l'additionneur (complet) s'expriment sous forme algébrique par :

$$\begin{aligned} S(x, y, R_e) &= x \oplus y \oplus R_e \\ R_s(x, y, R_e) &= x \cdot y + (x \oplus y) \cdot R_e \end{aligned}$$

9.6.3 Le soustracteur

Conformément aux règles énoncées Section 2.3.3, on obtient la table de vérité suivante :

Re	x	y	S	R_s
0	0	0	0	0
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	1

TABLE 9.7 – Table de vérité du soustracteur

Le calcul en décimal est équivalent à $R_s \times 2 + x - y = R_e + S$.

Les fonctions booléennes S et R_s pour le soustracteur s'expriment sous forme algébrique par :

$$\begin{array}{rcl} S(x, y, R_e) & = & x \oplus y \oplus R_e \\ R_s(x, y, R_e) & = & \overline{x} \cdot y + \overline{x \oplus y} \cdot R_e \end{array}$$

On obtient donc pour S la même formule que pour l'additionneur.

9.7 Algèbre de Boole et arithmétique

Il est possible d'exprimer les portes logiques sous forme de formules arithmétiques. Considérons deux variables entières x et y qui prennent leur valeurs dans l'intervalle $[0, 1]$. Alors, les portes logiques de base peuvent s'exprimer ainsi :

- $not(x) = 1 - x$
- $or(x, y) = x + y - (x \times y)$
- $and(x, y) = x \times y$
- $xor(x, y) = x + y - (2 \times x \times y)$

Pour évaluer des expressions booléennes en C, on peut utiliser les opérateurs classiques comme `&&` (and), `||` (or), associés aux booléens ou les opérateurs binaires `&`, `|`. Il est également possible de passer par les expressions arithmétiques :

```
int not1(int x) { return 1 - x; }

int or2(int x, int y) { return x + y - x*y; };

int and2(int x, int y) { return x*y; };

int xor2(int x, int y) { return x + y - 2*x*y; }

int or3(int x, int y, int z) { return or2(x, or2(y,z)) ;}

int and3(int x, int y, int z) { return x*y*z; }
```

Ainsi pour obtenir la table de vérité de la fonction booléenne $f(x,y,z) = \overline{x}.y.\overline{z} + x.\overline{y}.\overline{z} + x.y.\overline{z} + \overline{x}.\overline{y}.\overline{z}$, on pourra écrire le code suivant :

```
for (int i = 0; i < 8; ++i) {
    int x = (i & 4) >> 2 ;
    int y = (i & 2) >> 1;
    int z = i & 1;

    int v1 = or2( and3(not1(x),y,not1(z)), and3(x,not1(y),not1(z)));
    int v2 = or2( and3(x,y,not1(z)), and3(not1(x),not1(y),not1(z)));
    int f = or2(v1, v2);
    cout << x << " " << y << " " << z << " : " << f << endl;
}
```

9.8 Algèbre de Boole et logique

Dans cette section nous tentons de montrer le lien qui existe entre logique propositionnelle et algèbre de Boole en nous basant sur le problème des pigeons.

9.8.1 Définition du problème

On s'intéresse au problème des pigeons, également appelé *tiroirs et chaussettes*, qui est finalement un problème d'affectation ou de mise en correspondance ou d'injection, c'est à dire que tout élément de l'espace d'arrivée possède au plus un antécédent.

Problème des pigeons

Etant donné n pigeons et q pigeonniers, chaque pigeon devant trouver un pigeonnier et un pigeonnier ne pouvant accueillir qu'au maximum un seul pigeon, restera t-il des pigeons sans pigeonnier ?

Ce problème est très simple à résoudre du point de vue des mathématiques puisqu'il suffit que $n \leq q$ pour qu'il ait une solution et que toute permutation d'une

solution est également une solution.

9.8.2 Modélisation du problème en logique

Pour modéliser le problème en logique propositionnelle, il faut utiliser un ensemble de variables propositionnelles et exprimer des clauses entre les variables. On rappelle qu'une clause est une disjonction de variables propositionnelles, ces dernières sont donc séparées par le symbole $\vee$ qui correspond au *ou*. Un problème est alors un ensemble de clauses liées par le symbole $\wedge$ qui correspond au *et*. Si le problème possède une solution, on dit également qu'il est *satisfiable*, alors toutes les clauses sont interprétées à vrai.

On utilisera une matrice de $n \times q$ variables propositionnelles pour représenter le problème des pigeons :

$$X(n,q) = \begin{bmatrix} x_1^1 & x_1^2 & \dots & x_1^q \\ x_1^2 & x_2^2 & \dots & x_2^q \\ \vdots & & & \vdots \\ x_n^1 & x_n^2 & \dots & x_n^q \end{bmatrix}$$

Si la variable x_i^j est à *vrai* cela signifie que le pigeon i est dans le pigeonnier j. Cela implique donc que les variables $x_i^1, ..., x_i^{j-1}, x_i^{j+1}, ..., x_i^q$ sont à *faux*. En d'autres termes, le pigeon étant dans le pigeonnier j, il ne peut pas être dans les autres pigeonniers.

On doit donc exprimer deux types de contraintes :

- contrainte de type Ctr_L (sur les lignes) : un pigeon est dans **un et un** seul pigeonnier
- contrainte de type Ctr_C (sur les colonnes) : un pigeonnier contient **au plus** un pigeon, il peut donc ne pas en contenir si $n < q$

Dans le cas ou $n = 3$ et $q = 3$, le problème est modélisé comme suit, on renomme les variables afin de faciliter l'écriture des clauses :

$$X(3,3) = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{bmatrix} x_1^1 & x_1^2 & x_1^3 \\ x_2^1 & x_2^2 & x_2^3 \\ x_3^1 & x_3^2 & x_3^3 \end{bmatrix}$$

- le pigeon 1 est dans un seul pigeonnier (4 clauses) :

$$a \vee b \vee c$$
$$\neg a \vee \neg b$$
$$\neg a \vee \neg c$$
$$\neg b \vee \neg c$$

- le pigeon 2 est dans un seul pigeonnier (4 clauses) :

$$d \vee e \vee f$$
$$\neg d \vee \neg e$$
$$\neg d \vee \neg f$$
$$\neg e \vee \neg f$$

- le pigeon 3 est dans un seul pigeonnier (4 clauses) :

$$g \vee h \vee i$$
$$\neg g \vee \neg h$$
$$\neg g \vee \neg i$$
$$\neg h \vee \neg i$$

- le pigeonnier 1 contient au plus un pigeon (3 clauses) :

$$\neg a \vee \neg d$$
$$\neg a \vee \neg g$$
$$\neg d \vee \neg g$$

- le pigeonnier 2 contient au plus un pigeon (3 clauses) :

$$\neg b \vee \neg e$$
$$\neg b \vee \neg h$$
$$\neg e \vee \neg h$$

- le pigeonnier 3 contient au plus un pigeon (3 clauses) :

$$\neg c \vee \neg f$$
$$\neg c \vee \neg i$$
$$\neg f \vee \neg i$$

soit un total de 21 clauses pour $n = 3$ et $q = 3$.

Le nombre de clauses générées est défini par la formule :

$$n \times \left(1 + \sum_{i=1}^{q-1} i\right) + q \times \left(\sum_{i=1}^{n-1} i\right)$$

Si $n = q = 10$, on génère 910 clauses.

9.8.3 Résolution du problème en logique

Pour résoudre ce problème en logique il faut utiliser un solveur ou un démonstrateur automatique de théorème comme *Otter* (*Organized Techniques for Theorem-proving and Effective Research*, [20]).

La méthode qui permet de résoudre le problème sous forme de clauses consiste à appliquer la règle dite du *Principe de Résolution* définie par *John Alan Robinson* [24].

A partir d'une clause qui possède une variable propositionnelle p et une autre clause qui possède la variable sous forme négative ($\neg p$), on génère une nouvelle clause :

$$\frac{p \vee L \qquad \neg p \vee M}{L \vee M}$$

L et M étant des variables propositionnelles séparées par le symbole $\vee$.

On applique cette règle entre toutes les clauses quand cela est possible. On va donc générer un ensemble de plus en plus important de clauses à mesure que le nombre de pigeons et de pigeonniers augmentent. Les nouvelles clauses ajoutées à l'ensemble initial seront utilisées pour générer encore plus de clauses. Il se peut que l'on génère plusieurs fois la même clause, dans ce cas, si elle existe déjà, on ne l'ajoutera pas. On n'ajoutera pas également les tautologies qui sont de la forme $\neg p \vee p \vee L$, car dans ce cas $\neg p \vee p = 1$, il en résulte alors que $1 \vee L = 1$.

On terminera dans les deux cas suivants :

- soit le problème possède des solutions, on dit qu'il est **satisfiable**, et arrivera un moment ou on ne générera pas de nouvelle clause
- soit le problème n'a pas de solution, il est **insatisfiable**, et on génèrera la clause vide :

$$\frac{p \qquad \neg p}{\bot}$$

Si le problème est insatisfiable c'est qu'on a pu générer à partir d'un sous-ensemble de clauses la clause qui se résume à un seul littéral p, et qu'à partir d'un autre sous-ensemble on a généré son contraire. Le problème est donc insatisfiable car on ne peut affirmer une chose et son contraire.

Par exemple dans le cas ou $p = 3$ et $q = 2$, le problème n'a pas de solution et la résolution se fait en 18 étapes. Voici la preuve trouvée par Otter :

```
---------------- PROOF ----------------
4 [] -p1_1| -p2_1.
5 [] -p1_1| -p3_1.
6 [] -p2_1| -p3_1.
7 [] -p1_2| -p2_2.
8 [] -p1_2| -p3_2.
9 [] -p2_2| -p3_2.
10 [] p1_1|p1_2.
11 [] p2_1|p2_2.
12 [] p3_1|p3_2.
13 [hyper,11,7,10] p1_1|p2_1.
```

```
14 [hyper,12,9,11] p2_1|p3_1.
15 [hyper,12,8,10] p1_1|p3_1.
16 [hyper,15,6,13] p1_1.
17 [hyper,16,5,14] p2_1.
18 [hyper,17,4,16] $F.
```

Les variables x_i^j sont renommées en pi_j car Otter considère que le symbole `x` est une variable du calcul des prédicats et non une variable du calcul propositionnel.

Otter utilise ici une règle appelée *hyper-résolution* (HR) qui est dérivée du Principe de Résolution et on génère la clause vide, matérialisée par `$F`. Par exemple, la clause 13 est obtenue par :

$$\overbrace{p_1^1 \vee p_1^2}^{\text{clause 10}} \qquad \overbrace{-p_1^2 \vee -p_2^2}^{\text{clause 7}} \qquad \overbrace{p_2^2 \vee p_2^1}^{\text{clause 11}}$$

$$(HR) \qquad \frac{p_1^1 \vee \underbrace{p_1^2 \qquad -p_1^2}_{\emptyset} \vee \underbrace{-p_2^2 \qquad p_2^2}_{\emptyset} \vee p_2^1}{p_1^1 \vee p_2^1}$$

A mesure que p et q augmentent, le nombre de clauses augmente et la résolution prend plus de temps. Si on prend $p = 5$ et $q = 4$, Otter prouvera qu'il n'y a pas de solution en 142 étapes.

9.8.4 Modélisation sous forme de contraintes de cardinalité

Pour résoudre plus simplement ce problème, on peut le modéliser sous un autre formalisme qui utilise des contraintes de cardinalité [6] [1]. Nous utilisons ici une spécialisation de l'opérateur pour la logique :

$$\#(\alpha, \beta, \{L\})$$

où α et β sont des entiers positifs ou nuls tels que $0 \leq \alpha \leq \beta$ et L est une liste de variables booléennes. La contrainte signifie qu'au minimum α et au plus β variables de L sont vraies.

On modélise alors le problème par une matrice de variables booléennes $X(n, q)$, telles que $x_i^j = 1$ signifie que le pigeon i est dans le pigeonnier j. Le problème des pigeons s'exprime alors par deux types de contraintes :

1. Opérateur introduit dans le cadre de la Programmation Logique avec Contraintes par Pascal Van Hentenryck et Yves Deville en 1991.

- un pigeon est dans un et un seul pigeonnier (contrainte de type Ctr_L) :

$$\#(1,1,\{x_1^1,...,x_1^q\})$$
$$...$$
$$\#(1,1,\{x_n^1,...,x_n^q\})$$

- un pigeonnier accueille au plus un pigeon (contrainte de type Ctr_C) :

$$\#(0,1,\{x_1^1,...,x_n^1\})$$
$$...$$
$$\#(0,1,\{x_1^q,...,x_n^q\})$$

9.8.5 Contraintes $\#(1,1)$ et $\#(0,1)$

A quoi correspondent les contraintes $\#(0,1,\{L\})$ et $\#(1,1,\{L\})$ et peut-on les traduire en calcul propositionnel ?

Par exemple pour la contrainte $\#(1,1,\{x,y,z\})$, on peut utiliser une table de vérité et modéliser la contrainte de cardinalité sous forme d'une fonction booléenne $cc(x,y,z)$:

Ligne	x	y	z	$cc(x,y,z)$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	0
7	1	1	1	0

On a donc $cc(x,y,z)=1$ pour les lignes 1, 2 et 4 de la table de vérité car cela correspond aux cas où seule l'une des variables est à 1 parmi x,y et z. On obtient alors l'expression de la fonction :

$$cc(x,y,z)=\overline{x}.\overline{y}.z+\overline{x}.y.\overline{z}+x.\overline{y}.\overline{z}$$

Il s'agit la d'une somme de produits, en logique propositionnelle on parle de disjonctions ($+$) de conjonctions ($.$). Or les clauses sont des conjonctions de disjonctions et donc, si on veut obtenir des clauses, on doit prendre la négation du complémentaire de $cc(x,y,z)$ soit $\overline{\overline{cc(x,y,z)}}$. On va donc dans un premier temps simplifier $\overline{cc(x,y,z)}$ puis en prendre le complémentaire.

On peut réaliser la simplification de manière algébrique mais elle prend plus de temps que la méthode du tableau de Karnaugh. Il s'agit du tableau de la Figure 9.5 vue précédemment.

$$\begin{aligned}\overline{\overline{cc(x,y,z)}} &= \overline{\overline{x}\cdot\overline{y}\cdot\overline{z}+y\cdot z+x\cdot z+x\cdot y}\\ &= (x+y+z)\cdot(\overline{y}+\overline{z})\cdot(\overline{x}+\overline{z})\cdot(\overline{x}+\overline{y})\end{aligned}$$

Un contrainte de cardinalité $\#(1,1,\{L\})$ ou $Card(L)=k$, remplace donc $1+\sum_{i=1}^{k-1} i$ clauses, ce qui est très avantageux, et cette même contrainte de trois variables est équivalente en logique à :

$$(x\vee y\vee z)\wedge(\neg y\vee\neg z)\wedge(\neg x\vee\neg z)\wedge(\neg x\vee\neg y)$$

9.8.6 Résolution avec des contraintes de cardinalité

On peut vérifier la consistance (ou l'inconsistance) du problème en utilisant un système de déduction comme celui de [23] (cf. pages 80 et 81). On utilise la règle d'extension (Ext) définie ainsi :

$$\frac{\#(\alpha_1,\beta_1,\{L_1\})\quad\#(\alpha_2,\beta_2,\{L_2\})}{\#(\alpha_1+\alpha_2,\beta_1+\beta_2,\{L_1\cup L_2\})}$$

On remplace deux contraintes de cardinalité par une seule en faisant en quelque sorte la somme des deux contraintes initiales. Pour cela, il est généralement préférable de faire en sorte que $L_1\cap L_2=\emptyset$.

Si on applique la règle (Ext) sur l'ensemble des contraintes liées à Ctr_L, puis sur l'ensemble des contraintes liées à Ctr_C, on obtient respectivement :

- un pigeon est dans un et un seul pigeonnier Ctr_L :
$$\#(n,n,\{X(n,q)\})$$
- un pigeonnier accueille au plus un pigeon Ctr_C :
$$\#(0,q,\{X(n,q)\})$$

Le problème se résume donc à ces deux contraintes. La règle d'inconsistance (Inc_1) permet de déduire rapidement si le problème possède ou pas une solution. Cette règle stipule que si on dispose de deux contraintes de cardinalité sur le même ensemble de variables alors on a une inconsistance (donc pas de solution) si l'intersection des intervalles $[\alpha_1,\beta_1]$ et $[\alpha_2,\beta_2]$ est vide :

$$[\alpha_1,\beta_1]\cap[\alpha_2,\beta_2]=\emptyset\quad\frac{\#(\alpha_1,\beta_1,\{L\})\quad\#(\alpha_2,\beta_2,\{L\})}{\bot}$$

Et c'est bien le cas pour ce problème :

- si $n > q$ le problème n'a pas de solution et l'intersection des intervalles est vide, la règle d'inconsistance peut être appliquée et son résultat ($\perp$) indique que le problème n'a pas de solution
- par contre si $n \leq q$, l'intersection des intervalles est non vide, la régle ne sera pas appliquée et le problème possède des solutions

9.8.7 Solveur

Il est tout à fait envisageable de créer un solveur en logique qui se base sur les contraintes de cardinalité. Toute clause de la forme $x_1 \vee \ldots \vee x_n$ peut être traduite par une contrainte de la forme :

$$\#(1, n, \{x_1, \ldots, x_n\})$$

L'intérêt des contraintes de cardinalité est qu'elles nous permettent de déduire la valeur de certaines variables booléennes. Par exemple dans le cas des pigeons, dès qu'une variable d'une contrainte $\#(1, 1, \{L\})$ ou $\#(0, 1, \{L\})$ est vraie (donc égale à un), on peut en déduire que toutes les autres variables de L doivent être affectées à la valeur 0.

A titre de comparaison, voici, Table 9.8, quelques résultats sur un AMD Ryzen 5 3600 concernant la recherche de toutes les solutions du problème des pigeons avec deux solveurs très simples : le premier gère des clauses et le second des contraintes de cardinalité. Dans le cas des contraintes de cardinalité, on utilise la déduction des valeurs des variables comme évoqué au paragraphe précédent.

Nombre de Pigeons n	Nombre de Pigeonniers q	Nombre de Solutions	Solveur Clauses Temps (s)	Solveur Contraintes Temps (s)	Facteur d'Amélioration
10	10	3628800	428	7	$\times 61$
10	9	0	44	0,7	$\times 62$
11	11	39916800	6728	87	$\times 77$
11	10	0	644	7	$\times 92$

TABLE 9.8 – Temps de résolution en secondes de deux solveurs pour le problème des pigeons

La première, deuxième et troisième colonnes de la Table 9.8 indiquent respectivement le nombre de pigeons, de pigeonniers et le nombre de solutions trouvées. Dans le cas de problèmes insatisfiables, on n'a aucune solution. Les colonnes quatre et cinq donnent le temps de résolution en secondes avec un solveur simple basé sur des clauses et un solveur avec contraintes de cardinalité. Enfin, la dernière colonne indique le facteur d'amélioration qui est défini comme le rapport entre le temps de résolution avec le solveur à base de clauses et le temps de résolution avec le serveur à base de contraintes.

Il est indéniable que l'utilisation des contraintes de cardinalité apporte un gain substantiel lors de la résolution du problème. Dans le cas où le problème ne possède pas de solution, le temps de résolution avec le solveur à base de contraintes de cardinalité pour 10 pigeons et 9 pigeonniers est de $0,7$ seconde, soit un facteur d'amélioration de $44/0,7 = 62$. On prend donc 62 fois moins de temps pour résoudre le problème. A mesure que le nombre de pigeons et pigeonniers augmentent, le facteur d'amélioration augmente également.

9.9 Conclusion

Au travers de ces deux exemples que sont les circuits électroniques comme l'additionneur et la résolution du problème des pigeons en logique, nous avons vu comment l'algèbre de Boole pouvait nous aider à simplifier certaines expressions algébriques. Pour les circuits électroniques, nous pouvons diminuer le nombre de portes logiques nécessaires à leur implantation sur le silicium. Concernant la logique, on peut créer un solveur assez simple basé sur les contraintes de cardinalités ou sur des clauses mais les contraintes de cardinalités, dans le cas de problèmes structurés comme les pigeons ou le problème de Ramsey [2] vont permettre de simplifier la recherche en déduisant les valeurs à affecter à de nombreuses variables.

9.10 Exercices

Exercice 39 - Démontrez algébriquement les égalités suivantes :

(a) $\overline{Y}Z + Y\overline{Z} + YZ + \overline{Y} \cdot \overline{Z} = 1$

(b) $AB + A\overline{B} + \overline{A} \cdot \overline{B} = A + \overline{B}$

(c) $\overline{A} + AB + A\overline{C} + A \cdot \overline{B} \cdot \overline{C} = \overline{A} + B + \overline{C}$

(d) $A\overline{B} + \overline{A} \cdot \overline{C} \cdot \overline{D} + \overline{A} \cdot \overline{B}D + \overline{A} \cdot \overline{B} \cdot C \cdot \overline{D} = \overline{A} \cdot \overline{C} \cdot \overline{D} + \overline{B}$

(e) $XY + \overline{X}Z + YZ = XY + \overline{X}Z$

(f) $X + \overline{X}Y = X + Y$

Exercice 40 - Simplifiez les expressions suivantes :

(a) $ABC + AB\overline{C} + \overline{A}B$

(b) $(\overline{A + B})(\overline{A} + \overline{B})$

(c) $(A + \overline{B} + A\overline{B})(AB + \overline{A}C + BC)$

2. Le problème de Ramsey, d'après Frank Ramsey mathématicien, économiste et logicien anglais (1903 - 1930) consiste à colorier les arcs d'un graphe complet à l'aide de trois couleurs sans qu'il n'existe de triangle monochromatique.

(d) $X + Y(Z + \overline{X + Z})$
(e) $\overline{W}X(\overline{Z} + \overline{Y}Z) + X(W + \overline{W}YZ)$

Exercice 41 - Simplifiez les fonctions suivantes à l'aide d'un tableau de Karnaugh
(a) $F(X, Y, Z) = (1, 3, 6, 7)$
(b) $G(X, Y, Z) = (0, 3, 4, 5, 7)$
(c) $H(A, B, C, D) = (1, 5, 9, 12, 13, 15)$

Nous décrivons dans le reste de cette section quelques exercices qui demandent une certaine maîtrise de la programmation en C/C++. Ils sont réservés au programmeur expérimenté.

Exercice 42 - Ecrire un programme appelé `pigeon_hole_generator.exe` qui prend en paramètres le nombre de pigeons, le nombre de pigeonnier ainsi que la méthode de génération. On génèrera le problème sous forme de clauses ou de contraintes de cardinalité. Lors de la génération du problème on donnera le nombre de variables propositionnelle ainsi que le nombre de clauses ou le nombre de contraintes. Ainsi pour trois pigeons et trois pigeonniers, on obtient pour les clauses :

```
clauses
9 21
3 1 2 3
2 -1 -2
2 -1 -3
2 -2 -3
3 4 5 6
2 -4 -5
2 -4 -6
2 -5 -6
3 7 8 9
2 -7 -8
2 -7 -9
2 -8 -9
2 -1 -4
2 -1 -7
2 -4 -7
2 -2 -5
2 -2 -8
2 -5 -8
2 -3 -6
2 -3 -9
2 -6 -9
```

Ici, on a 9 variables propositionnelles numérotées de 1 à 9 et 21 clauses. Chaque clause est décrite par le nombre de variables qui la composent, puis le numéro des variables, précédées du signe - si elle apparaît dans clause sous forme négative ($\neg$).

Sous forme de contraintes de cardinalité, on aura :

```
constraints
9 6
1 1 3 1 2 3
1 1 3 4 5 6
1 1 3 7 8 9
0 1 3 1 4 7
0 1 3 2 5 8
0 1 3 3 6 9
```

On obtient 9 variables, 6 contraintes. Chaque contrainte est décrite par le nombre minimum et maximum de variables à vrai, puis le nombre de variables et la liste des variables.

Exercice 43 - Ecrire un programme appelé `clauses_solver.exe` qui prend en paramètre un fichier qui contient un ensemble de clauses et résoud le problème de manière récursive en tentant d'instancier la prochaine variable à vrai, puis à faux. Initialement les variables sont non instanciées.

Exercice 44 - Ecrire un programme appelé `constraints_solver.exe` qui prend en paramètre un fichier qui contient un ensemble de contraines de cardinalité et résoud le problème de manière récursive en tentant d'instancier la prochaine variable à vrai, puis à faux. On pourra mettre en place la déduction en vérifiant que si le nombre de variables à vrai est égal au nombre maximum (β) alors toutes les autres variables non instanciées sont positionnées à faux.

Chapitre 10

Etudes de cas

10.1 Introduction

Avant d'aborder les différentes études de cas dont le but est de traiter les points cruciaux que nous avons évoqués, nous allons détailler dans ce chapitre les caractéristiques communes à chacune de ces études de cas que nous qualifions également de projets car il s'agit en fait de projets de programmation.

J'ai tenté d'établir une sorte de squelette de projet en utilisant la même organisation des répertoires ainsi que les mêmes scripts shell et PHP. Les diverses actions (compilation, exécution des tests de performance, ...) sont automatisées au moyen de l'utilitaire `make` ainsi que du `makefile` associé.

Chaque projet est conçu autour d'un ensemble d'implantations, appelées méthodes, d'un traitement informatique. L'une de ces méthodes est un sous-programme dit de référence qui nous permet de vérifier que les autres méthodes sont correctes et produisent le même résultat. Chaque nouvelle méthode tente d'apporter une amélioration par rapport à la méthode de référence en

- utilisant une réécriture du code C (dépliage, élimination des `if`, vectorisation par *intrinsics*)
- ou en codant la fonction C en assembleur (dépliage, élimination des `if`, vectorisation)

10.2 Organisation des sources et binaires

Les sous-répertoires de chaque projet sont les suivants :

- le sous-répertoire `cfg` contient des fichiers de configuration utiles à la compilation
- le sous-répertoire `src` contient le code source, c'est à dire les fichiers C/C++ et les fichiers assembleur

- le sous-répertoire `build` contient les fichiers générés lors de la compilation
  - `build/obj` contient les fichiers objets (`.o`)
  - `build/bin` contient les exécutables (`.exe`)
- le sous-répertoire `results` contient les fichiers de données et graphiques générés lors de l'exécution des tests de validité et de performance, on trouve un répertoire par processeur
- le sous-répertoire `output` contient les fichiers générés par le script `table.php` qui permet de générer des tableaux de donnée à partir des résultats obtenus

L'arborescence est donc la suivante :

```
.
├── build
│   ├── bin
│   └── obj
├── cfg
├── output
├── results
│   └── Intel-Celeron-CPU-N3050-1_60GHz
└── src
```

Dans le répertoire principal on trouvera un fichier `README` ainsi qu'un fichier `INSTALL`. Le but du fichier `README` est de renseigner l'utilisateur sur l'objectif du projet ainsi que de lui donner un accès rapide aux fonctionnalités de base du projet. Le fichier `INSTALL`, quant à lui, décrit comment installer le projet, en d'autres termes, comment le compiler et quels programmes annexes doivent être installés.

Le choix a été fait de rédiger l'ensemble des sources en anglais ainsi que les fichiers `README` et `INSTALL` de manière à ce qu'ils puissent être utilisés, voire modifiés par un large public. L'ensemble des fichiers est sous Licence GNU.

10.2.1 Cibles make

On considère que le lecteur possède des connaissances de base liées à la création d'un makefile. Les différentes cibles (*targets*) de make sont les suivantes :

- `configure` appelle le script `cpu_technos.sh` (cf. ci-après)
- `build` permet de générer les exécutables au format *release* c'est à dire avec les options de compilations sensées générer du code performant ; on obtient le même résultat en lançant *make* sans arguments
- `debug` génére les exécutables en incluant les informations nécessaires au débogage
- `clean` supprime les fichiers objets et les exécutables

- `validity` exécute les tests de validité afin de vérifier que les fonctions donnent le même résultat pour un test de base
- `performance` exécute les tests de performance ce qui permet de déterminer quelle fonction est la meilleure
- `archive` génére une archive de l'ensemble des fichiers du projet

10.2.2 Scripts shell et PHP

On dispose dans le répertoire du projet d'un ensemble de scripts :

- `cpu_name.sh` récupère le nom du microprocesseur en éliminant certains caractères inutiles, en remplaçant les espaces par le signe moins (`'-'`), cet identifiant est ensuite utilisé pour créer un sous-répertoire dans le répertoire `results` afin de stocker les résultats spécifiques au microprocesseur
- `cpu_technos.sh` détermine quelles technologies (SSE2, SSE4.1, SSE4.2, AVX2, POPCNT, BMI, FMA) sont implantées au niveau du microprocesseur afin de savoir si on peut utiliser leurs jeux d'instructions. Ce script génère en conséquence les fichiers `src/asm_config.inc`, `src/cpp_config.inc` ainsi que `cpu_technos.mak`
- `validity_test.php` réalise un test de validité en vérifiant pour différents paramètres que les méthodes donnent le même résultat que la fonction de référence
- `performance_test.php` réalise un test de performance afin de déterminer pour différents paramètres quelle est la méthode la plus efficace puis appelle le script `performance_graph.php` afin de générer un graphique des données obtenues en utilisant *gnuplot*
- `table.php`, comme indiqué précédemment, ce script permet de générer dans le sous-répertoire `output` des tableaux de données au format CSV (*Comma Separated Values*), HTML ou LaTeX

Cette liste est non exhaustive, on trouvera également en fonction de l'angle d'attaque de l'étude de cas des scripts spécifiques. Par exemple, dans le cas du produit de matrices nous avons ajouté un script nommé `samples_test.php` qui génère les temps d'exécution du produit de la méthode de référence pour différentes dimensions de la matrice afin de démontrer que la méthode de référence est sensible à la dimension.

10.2.3 Fichiers sources

Dans le répertoire des fichiers sources, on trouve un fichier `common.h` qui contient les définitions de types, constantes et fonctions communes à l'ensemble des sources. On trouvera également les fichiers `cpp_config.h` et `asm_config.inc`, générés par le

script `cpu_technos.sh`, qui définissent des macro instructions liées aux différentes technologies implantées au sein du microprocesseur. Ainsi, si le microprocesseur possède la technologie SSE 4.2 on définit la macro `CPU_SSE42_COMPLIANT`. Ces macro instructions peuvent être utilisées par la suite pour inclure le code d'une méthode utilisant le jeu d'instructions SSE 4.2.

J'utilise également `cpu_timer.h` et `cpu_timer.cpp` qui définissent la classe `CPUTimer` qui est un chronomètre basé sur l'instruction `rdtsc` (*ReaD TimeStamp Counter*) afin de lire le nombre de cycles d'horloge du processeur et calculer par différence le nombre de cycles utilisés lors de l'exécution de chaque méthode.

La classe `SignalHandler` a été implantée et elle est chargée d'intercepter la plupart des exceptions levées par la commande `signal` comme par exemple SIGFPE (cf. fichiers `signal_handler.h` et `signal_handler.cpp`). Pour utiliser cette classe et intercepter différents signaux il suffit de déclarer une instance de la classe au tout début de la méthode `main` dans le fichier `src/main.cpp` :

```
1 int main( int argc, char *argv[]) {
2     SignalHandler sh;
3     ...
4 }
```

Le fichier `src/main.cpp` qui contient le programme principal se base sur `getopt` afin de gérer les options en ligne de commande du programme. Parmi les options communes à tous les projets on trouve :

- `-h` ou `-help` pour obtenir l'aide du programme exécutable
- `-l` ou `-list` donne la liste des méthodes
- `-t` ou `-test` réalise un test de l'ensemble des méthodes et indique si une méthode produit un résultat différent de celui de la méthode de référence
- `-m` ou `-method` permet de choisir la méthode à tester en donnant son identifiant entier, la méthode de référence ayant pour valeur 1
- `-n` ou `-name` permet de sélectionner une méthode en précisant son nom
- `-v` ou `-verbose` introduit le mode verbeux, c'est à dire que l'on affichera des informations supplémentaires lors de l'exécution

Le reste des fichiers contient le code source des différentes implantations de la fonction de référence.

10.3 Redéfinition des types et constantes

Afin de simplifier l'écriture du code et sa compréhension, j'ai pris pour habitude de redéfinir les types utilisés (fichier `src/common.h`). Ainsi plutôt que d'utiliser `unsigned int`, `size_t` ou `uint32_t` pour définir un type entier non signé, j'utilise

`u32` pour *Unsigned* 32 bits. De la même manière, le type `float` du langage C est renommé en `f32` :

```
typedef uint8_t u8;
typedef uint32_t u32;
typedef int32_t  i32;
typedef float f32;
typedef double f64;
```

10.4 Tests et matériels

10.4.1 Matériels

L'ensemble des méthodes que nous allons implanter seront testées sur différents matériels afin d'étudier l'influence de telle réécriture, de telle technologie par rapport à une autre, ou l'influence de la taille des caches. La liste des microprocesseurs utilisés figure Tables 10.1, 10.2, 10.3. Nous avons séparé les matériels en trois catégories :

- les processeurs produits avant 2015, qualifiés d'architectures anciennes,
- ceux produits entre 2015 et 2019, qualifiés d'architectures modernes
- et ceux produits en 2020 et après, qualifiés d'architectures récentes

Il s'agit ici d'ordinateurs personnels ainsi que d'ordinateurs disponibles à l'Université d'Angers dotés de microprocesseurs comme l'Intel i7 8700 ou l'Intel Xeon Silver 4208 qui dispose du jeu d'instructions AVX512.

Marque	Intel	Intel	Intel	AMD	Intel	Intel
Modèle	Pentium D	Core 2	Core i7	Phenom II	Core i5	Core i7
Sous-modèle	925	Q9300	860	1090 T	i5 3570k	4790
Année	2006	2008	2009	2010	2012	2014
Architecture	Presler	Yorkfield	Lynnfield	Thuban	Ivy Bridge	Haswell
Lithographie (nm)	65	45	45	45	22	22
Fréquence de base (GHz)	3000	2500	2800	3200	3400	3600
Fréquence de boost (GHz)	3000	2500	3460	3600	3800	4000
Cores	2	4	4	6	4	4
Threads	2	4	8	6	4	8
Cache L1i (ko)	12	32	32	64	32	32
Cache L1d (ko)	16	32	32	64	32	32
Cache L2 (ko)	1024	3072	256	512	256	256
Cache L3 (Mo)	–	–	8	6	6	8

TABLE 10.1 – Caractéristiques des matériels : architectures anciennes (avant 2015)

Nous donnons pour chaque microprocesseur les informations suivantes :

- la marque du microprocesseur (Intel ou AMD)
- le modèle, par exemple Core 2
- le sous-modèle, par exemple Q9300
- l'année de production
- le nom de l'architecture du microprocesseur
- la finesse de gravure (Lithographie) en nanomètres
- la fréquence de base en GHz ainsi que la fréquence maximale (boost)
- le nombre de coeurs et de threads
- les tailles des différents caches

Marque Modèle Sous-modèle	Intel Core i3 6100	AMD Ryzen 7 1700X	Intel Core i5 7400	Intel Core i7 8700	AMD Ryzen 5 3600	Intel Xeon 4208
Année	2015	2017	2017	2017	2019	2019
Architecture	Skylake	Zen	Kaby Lake	Coffee Lake	Zen 2	Cascade Lake
Lithographie (nm)	14	14	14	14	7	14
Fréquence de base (GHz)	3700	3400	3000	3200	3600	2100
Fréquence de boost (GHz)	3700	3800	3500	4600	4200	3200
Cores	2	8	4	6	6	8
Threads	4	16	4	12	12	16
Cache L1i (ko)	32	64	32	32	32	32
Cache L1d (ko)	32	32	32	32	32	32
Cache L2 (ko)	256	512	256	256	512	1024
Cache L3 (Mo)	3	8+8	6	12	16+16	11

TABLE 10.2 – Caractéristiques des matériels : architectures modernes (2015 à 2019)

10.4.2 Tests

Nous avons défini deux types de tests principaux :

- le test de **validité** qui comme indiqué précédemment vérifie que l'ensemble des méthodes implantées produisent le même résultat et assure ainsi qu'une méthode très rapide ne l'est pas parce qu'elle est boguée
- le test de **performance** qui en faisant varier certains paramètres (dimension des matrices, taille des vecteurs) évalue le temps d'exécution de chacune des méthodes afin de générer par la suite un graphique qui permet de déterminer visuellement quelles méthodes sont les plus performantes

Pour évaluer le temps d'exécution d'une méthode deux facteurs sont à prendre en compte :

Marque Modèle Sous-modèle	Intel Core i7 10850H	AMD Ryzen 5 5600g	Intel Core i5 12400F
Année	2020	2021	2022
Architecture	Comet Lake	Zen3	Alder Lake
Lithographie (nm)	14	7	Intel 7
Fréquence de base (GHz)	2700	3900	2500
Fréquence de boost (GHz)	5100	4400	4400
Cores	6	6	6
Threads	12	12	12
Cache L1i (ko)	32	32	32
Cache L1d (ko)	32	32	48
Cache L2 (ko)	256	512	1280
Cache L3 (Mo)	12	16	18

TABLE 10.3 – Caractéristiques des matériels : architectures récentes (2020 et suivantes)

- le premier concerne ce que l'on mesure
- le second tient à la manière dont on mesure

10.4.2.1 Quantités mesurées

Nous reportons deux mesures lors des tests : le temps d'exécution du programme ainsi que le nombre de cycles de l'exécution d'une méthode. Mais c'est le temps d'exécution du programme qui est utilisé pour générer les graphiques des tests de performance et comparer l'efficacité de chaque fonction.

Pour obtenir le temps d'exécution du programme nous utilisons la commande `/usr/bin/time` qui est différente de la commande `time` de la plupart des shells. Notamment elle permet de spécifier un format d'affichage grâce à l'option `-f`. Nous reportons le *user time*. Le nombre de cycles lié à l'exécution d'une méthode est obtenu grâce à la classe `CPUTimer` et ne prend donc pas en compte l'allocation des données et leur initialisation.

10.4.2.2 Comment mesurer

Pour les méthodes qui s'exécutent en quelques millisecondes se pose un réel problème d'évaluation car les temps d'exécution peuvent parfois varier de manière importante entre deux exécutions successives. Cela tient à différents facteurs comme la variation des fréquences qui dépend du nombre de processus qui tournent en parallèle par exemple. Nous avons vu, Section 3.3.1 ; que la fréquence de fonctionnement est maximale si le microprocesseur est sollicité par un seul processus mais qu'elle diminue si plusieurs processus sont actifs. Il est donc préférable lorsque

l'on réalise les tests de performance de ne pas utiliser souris et clavier et éviter tout traitement en tâche de fond qui viendrait perturber les résultats.

Afin d'obtenir une valeur proche de la réalité il est nécessaire de réaliser plusieurs exécutions (une dizaine est généralement suffisante) et de calculer la moyenne des temps d'exécution. Nous avons conçu une petite librairie PHP appelée `ezlib.php` (lire easy lib) qui comprend notamment une méthode `average_time()` qui calcule la moyenne des temps d'exécution pour dix exécutions d'une même commande.

Nous attirons l'attention du lecteur sur le fait notable suivant : sur les dernières versions d'Ubuntu, nous avons pu observé que la fréquence de fonctionnement du processeur est très en dessous de sa valeur maximale. Ce qui donne, lors de l'exécution des tests sur une machine qui vient d'être mise en route, des temps de calculs bien supérieurs à ce qu'ils devraient être. Il peut donc être nécessaire de forcer le système d'exploitation à passer en mode *performance* grâce à l'utilisation des commandes suivantes :

```
1 sudo apt-get install cpufrequtils
2 echo 'GOVERNOR="performance"' | sudo tee /etc/default/cpufrequtils
3 sudo systemctl disable ondemand
```

10.4.3 Tests du Chapitre 2

Le chapitre 2 contient deux tests, le premier concerne l'utilisation de l'instruction assembleur `bsr` ou son remplacement sous forme de fonction (cf. Section 1.1). Le second concerne le calcul des nombres premiers (cf. Section 1.3.1). Les résultats de ces tests figurents Tables 10.4, 10.5, 10.6.

Marque Modèle Sous-modèle	Intel Pentium D 925	Intel Core 2 Q9300	Intel Core i7 860	AMD Phenom II 1090 T	Intel Core i5 i5 3570k	Intel Core i7 4790
bsr (fonction C)	29,23	25,70	15,47	17,90	15,79	13,48
bsr (`__builtin_clz`)	3,39	1,80	1,52	1,81	1,20	0,55
bsr (asm + `bsr`)	1,78	1,33	0,67	1,70	0,60	0,42
Premier (v1)	7154,93	3003,00	3513,00	6214,00	1327,00	1163,00
Premier (v2)	0,64	0,18	0,28	0,44	0,13	0,08
Crible (v3)	0,02	0,01	0,01	0,02	0,01	0,00

TABLE 10.4 – Tests des matériels anciens

Bien évidemment, on note une diminution du temps de calcul à mesure que l'année de production du microprocesseur devient plus récente.

Marque Modèle Sous-modèle	Intel Core i3 6100	AMD Ryzen 7 1700X	Intel Core i5 7400	Intel Core i7 8700	AMD Ryzen 5 3600	Intel Xeon 4208
bsr (fonction C)	13.56	10.52	14.42	10.40	9.44	16.43
bsr (`__builtin_clz`)	0.63	0.49	0.61	0.46	0.45	0.21
bsr (asm + `bsr`)	0.47	1.04	0.44	0.33	0.96	0.51
Premier (v1)	1082.25	1859.59	1154.00	895.17	1726.93	1263.00
Premier (v2)	0.11	0.20	0.12	0.07	0.18	0.15
Crible (v3)	0.00	0.00	0.00	0.00	0.00	0.00

TABLE 10.5 – Test des matériels modernes en architecture 32 bits

On remarquera que pour le test de l'implantation de `bsr`, sur un Xeon Silver 4208, le temps de calcul pour la fonction qui fait appel à `builtin_clz` est très faible, de l'ordre de $0,09$ secondes. Cela est dû au fait que le compilateur a optimisé le code notamment en faisant appel à l'instruction `vplzcntd`, une instruction vectorielle qu'il l'applique sur un vecteur AVX. Cette instruction fait partie du jeu d'instruction AVX512-VL et AVX512-CD.

Par contre, pour les processeurs AMD l'instruction `bsr` ne semble pas être très efficace comparativement aux microprocesseurs Intel.

Marque Modèle Sous-modèle	Intel Core i7 11850H	AMD Ryzen 5 5600g	Intel Core i5 12400F
bsr (fonction C)	10.01	7.65	11.33
bsr (`__builtin_clz`)	0.44	0.46	0.33
bsr (asm + `bsr`)	0.31	0.92	0.25
Premier (v1)	815.96	676.94	691.73
Premier (v2)	0.08	0.07	0.08
Crible (v3)	0.00	0.00	0.02

TABLE 10.6 – Test des matériels récents 2020 et après en architecture 32 bits

Chapitre 11

Etude de cas produit de matrices

11.1 Introduction

Nous abordons dans ce chapitre le problème du produit de matrices qui est un problème classique en informatique. L'intérêt de ce problème est que la formule mathématique qui donne la manière de calculer le produit est totalement inefficace si elle est implantée directement car elle génère beaucoup de défauts de cache pour certaines dimensions de la matrice. En conséquence, sur des architectures anciennes ne possèdant qu'un cache L1 et L2, le temps d'exécution est anormalement plus important. Les architectures multicoeurs disposant d'un cache L3 se révèlent en général moins sensibles à ces variations.

On rappelle qu'une matrice est un tableau à deux dimensions de n lignes et p colonnes de réels. On notera $A(n,p)$ la matrice A composée de n lignes et p colonnes. Le produit d'une matrice A avec une matrice B n'est possible que si le nombre de lignes de B est égal au nombre de colonnes de A. Le résultat est une matrice C (cf. Figure 11.1) dont le nombre de lignes est celles de A, et le nombre de colonnes celles de B. En d'autres termes, on a :

$$C(n,q) = A(n,p) \times B(p,q)$$

On note c_i^j le coefficient de la matrice C en ligne i colonne j, dont la formule de calcul est donnée par la somme des produits de la ligne i de A par la colonne j de B :

$$c_i^j = \sum_{k=1}^{p} a_i^k \times b_k^j$$

Afin de simplifier la compréhension des calculs nous allons nous cantonner

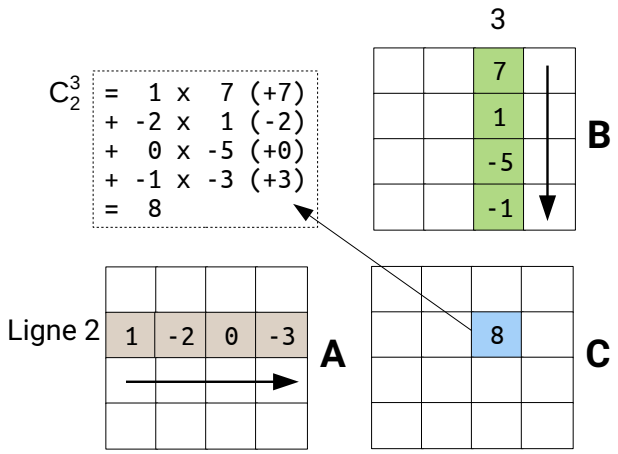


FIGURE 11.1 – Produit de matrices

à des matrices carrées pour lesquelles le nombre de lignes est égal au nombre de colonnes, soit $n = p = q$ et nous appelerons donc dimension (ou taille) de la matrice carrée cette quantité qui sera également identifiée par la variable `dim` dans les sources C. On peut voir, Figure 11.1, le calcul du coefficient c_2^3 pour une matrice carrée de dimension 4. On multiplie la ligne 2 de la matrice A par la colonne 3 de la matrice B et on somme l'ensemble des produits afin d'obtenir le résultat final.

11.2 Stockage des matrices

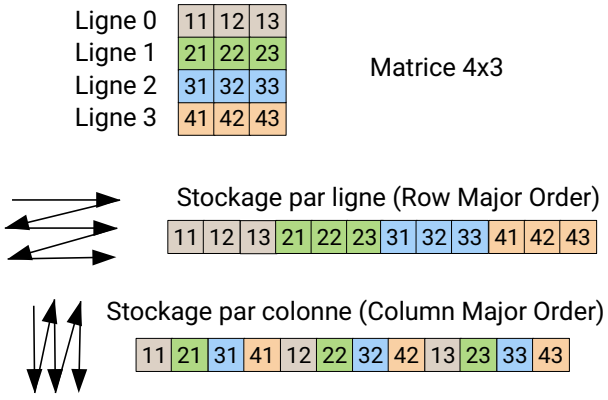


FIGURE 11.2 – Stockage d'une matrice

On dispose de deux stratégies de stockage des matrices (voir Figure 11.2) :

- par lignes (*Row Major Order*), c'est le cas de langage C et c'est la façon la plus naturelle de le faire

- par colonnes (*Column Major Order*), c'est le cas du langage Fortran

Nous allons bien évidemment nous focaliser sur le langage C. Voyons sur un exemple comment définir une matrice. Soient les variables entières non signées `rows = 10`, qui correspond au nombre de lignes d'une matrice, et `cols = 99` qui est son nombre de colonnes. Dès lors, nous avons en langage C quatre alternatives pour créer une matrice :

- la première, dite *statique* [1], consiste à déclarer un tableau à deux dimensions :

```
typedef float f32;
f32 M1[rows][cols];
```

- la seconde, statique également, utilise un tableau à une dimension dont la taille est le produit `rows * cols` :

```
f32 M2[rows * cols];
```

- la troisième, *dynamique*, permet de créer un tableau à deux dimensions en utilisant un premier tableau de pointeurs sur les lignes qui sont ensuite créées en allouant le nombre de colonnes nécessaires :

```
f32 **M3;
M3 = new f32 * [rows];
for (u32 i = 0; i < rows; ++i) {
    M3[i] = new f32 [cols];
}
```

- enfin la quatrième, dynamique également, permet de créer un tableau à une dimension :

```
f32 *M4;
M4 = new f32 [rows * cols];
```

L'accès aux tableaux à deux dimensions pour la ligne i et la colonne j s'écrit naturellement `M1[i][j]` alors que pour les tableaux à une dimension il faut utiliser `M2[i * cols + j]`.

Généralement, lorsqu'on travaille avec des matrices dont on ne connaît pas les dimensions *a priori* on utilise une création dynamique. De plus, afin d'améliorer l'efficacité des traitements, on utilise une représentation avec une seule dimension car dans ce cas les données sont contigües, c'est à dire que les coefficients sont à des adresses consécutives en mémoire (voir Figure 11.3). On utilisera donc l'alternative qui correspond à la création de la matrice `M4`.

Cela est d'autant plus intéressant que dans certains traitement, comme l'initialisation, les données étant contigües, on chargera en mémoire cache les données

1. Une variable statique est une variable immuable définie une fois pour toute.

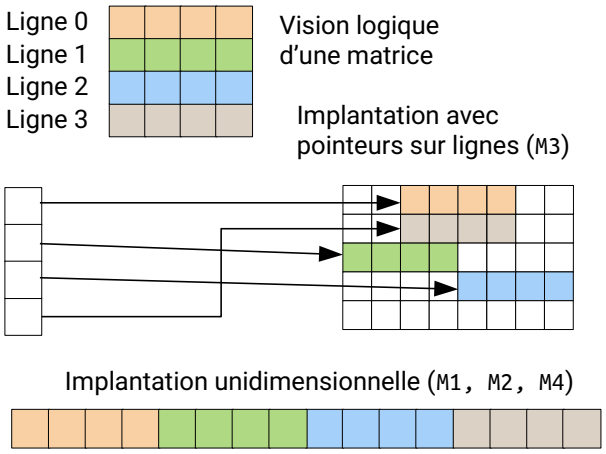


FIGURE 11.3 – Stockage d'une matrice

en `M[i]` et `M[i+1]`, etc. Dans le cas de la matrice `M3`, les données d'une ligne sont consécutives en mémoire, mais quand on passe à la ligne suivante, ce n'est pas forcément le cas.

Afin de créer les matrices de manière dynamique, plutôt que d'utiliser `malloc()` en C, ou `new` en C++, on utilisera `_mm_malloc()` disponible en incluant le fichier `xmmintrin.h` et qui permet d'aligner les données (cf. Section 3.2.1). On libère l'espace alloué en utilisant la fonction `_mm_free()` :

```
size_t size = rows * cols * sizeof(f32);
f32 *M = (f32 *) _mm_malloc(size, CPU_MEMORY_ALIGNMENT);
...
_mm_free(A);
```

La fonction `_mm_malloc()` possède deux paramètres : le premier est le nombre d'octets à allouer et le second, figuré ici par la variable `CPU_MEMORY_ALIGNMENT`, définit l'alignement. Il doit être égal à 16 pour le SSE ou 32 pour l'AVX/AVX2 et 64 pour l'AVX-512. Si les données sont alignées, leur chargement est normalement plus rapide, on peut alors utiliser les instructions comme `movdqa` plutôt que `movdqu` lors de la vectorisation.

Pour information, la fonction `_mm_malloc` est en fait une macro instruction [2] qui se base sur la fonction suivante :

```
int posix_memalign(void **memptr, size_t alignment, size_t size);
```

2. Avec gcc 10 elle est définie sous Ubuntu 20.04 dans le fichier `/usr/lib/gcc/x86_64-linux-gnu/10/include/mm_malloc.h`.

11.3 Fonction de référence

La fonction à implanter est donnée Listing 11.3.1. Elle comporte quatre paramètres qui sont les adresses des matrices `A`, `B` et `C` ainsi que la dimension des matrices `dim`. Il s'agit de l'implantation directe de la formule mathématique. On a une seule dimension car, pour rappel, on considère le produit de matrices carrées qui ont le même nombre de lignes et de colonnes.

```
1  #define a(y,x) A[(y)*dim+(x)]
2  #define b(y,x) B[(y)*dim+(x)]
3  #define c(y,x) C[(y)*dim+(x)]
4
5  void mp_reference(f32 *A, f32 *B, f32 *C, u32 dim) {
6      for (u32 i = 0; i < dim; ++i) {
7          for (u32 j = 0; j < dim; ++j) {
8              f32 sum = 0.0;
9              for (u32 k = 0; k < dim; ++k) {
10                 sum += a(i,k) * b(k,j);
11             }
12             c(i,j) = sum;
13         }
14     }
15 }
```

Listing 11.3.1 – Produit de matrice, fonction de référence

Afin de simplifier l'écriture, on a créé trois macro instructions `a()`, `b()` et `c()` qui font référence aux coefficients a_i^j, b_i^j et c_i^j. Plutôt que d'écrire `A[i*dim+j]`, on préfère utiliser `a(i,j)` plus lisible et compréhensible lors de l'écriture mais également lors de la relecture du code.

11.4 Analyse des premiers résultats

Cette implantation de référence possède une compléxité en $O(n^3)$ étant donné que l'on a trois boucles `for` imbriquées de taille `dim`. On remplace bien évidemment `dim` par n. Si on réalise des multiplications de matrices en faisant varier la taille, on obtient les résultats de la Figure 11.4 qui sont différents de ce à quoi on est en droit de s'attendre. On pourrait en effet prétendre trouver une courbe lisse mais on observe pour un processeur Intel Celeron N3050 que pour certaines tailles de matrices, le temps de calcul est bien plus important que la normale.

On peut voir d'ailleurs, Table 11.1, plusieurs résultats pour des processeurs différents. Pour $n = 1024$ et $n = 2048$, on observe l'accroissement du temps de calcul. Cet acroissement n'existe pas sur des processeurs récents pour $n = 1280$ ou

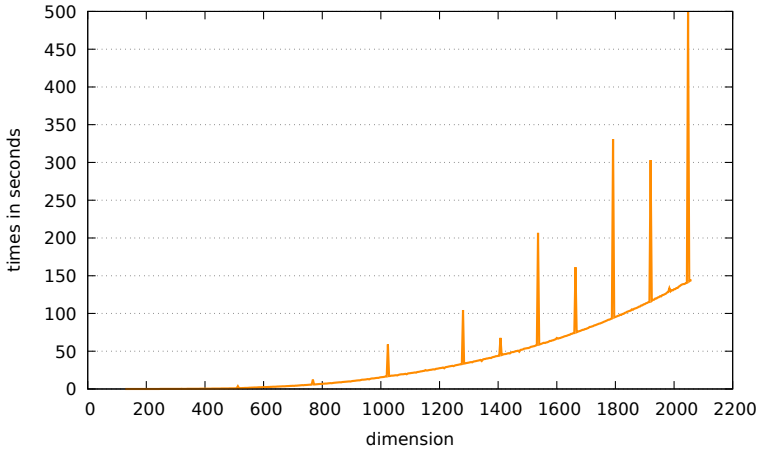


FIGURE 11.4 – Echantillons Produit de matrices sur Intel Celeron N3050

tout dumoins n'est pas perceptible, mais pour un Pentium E5300 qui ne dispose pas de cache L3, le temps de calcul est deux fois plus important (28,46 s) par rapport aux dimensions $n = 1279$ et $n = 1281$.

	AMD Ryzen 7 1700X	AMD Ryzen 5 3600	Intel Core i5 7400	Intel Core i7 8700	Intel Pentium E5300
Cache L2 (ko)	512	512	256	256	32
Cache L3 (ko)	8192	16384	6144	12288	2048
1023	1,49	1,34	1,33	0.95	7,26
1024	**6,35**	**5,98**	**2,28**	**1.53**	**25,39**
1025	1,52	1,37	1,26	0.95	7,24
1279	2,95	2,48	3,00	2.49	14,56
1280	2,98	2,52	3,28	2.43	**28,46**
1281	2,98	2,47	3,05	2.26	14,80
2047	20,14	15,34	37,48	28.48	62,71
2048	**53,88**	**42,21**	**51,84**	**44.24**	**227,36**
2049	20,26	15,17	37,79	28.59	63,07

TABLE 11.1 – Temps d'exécution en secondes de la méthode de référence du produit de matrice pour différentes architectures

Ce phénomène peut être expliqué par de nombreux défauts de cache. Les matrices étant de grande taille elles ne tiennent pas dans le cache L1 ou L2. Mais cela est commun à toutes les matrices. Par exemple, une matrice carrée de dimension 1024 occupe $1024 \times 1024 \times 4$ où 4 est la taille en octets d'un flottant en simple précision, soit 4 Mo. De plus, on travaille avec trois matrices pour réaliser le produit.

Un autre facteur important de ralentissement est l'accès aux coefficients de la matrice B. En effet, on accède la matrice A par ligne, ce qui est efficace, car quand on charge a_i^k, les données suivantes comme a_i^{k+1}, ..., a_i^{k+7} sont déjà dans le cache. Par contre l'accès aux coefficients de la matrice B est pénalisant car lorsque l'on charge b_k^j, le coefficient suivant b_{k+1}^j se trouve `dim * 4` octets plus loin en mémoire et ne sera donc probablement pas dans le cache. On peut démontrer cela en utilisant des outils comme **perf**.

11.5 Analyse du cache avec perf

Il existe sous Linux un outil d'analyse du cache comme *valgrind* (option cachegrind) mais celui-ci ne fait que simuler les caches L1 et LLC (*Last Level Cache*, c'est à dire le cache L3 sur la plupart des microprocesseurs modernes). Il est préférable d'utiliser un outil comme **perf** qui lit et collecte les données des *Performance Monitoring Units* (PMUs) qui sont des registres liés au matériel des processeurs modernes. Pour utiliser **perf**, il faut installer deux packages sous Ubuntu :

```
1 sudo apt install linux-tools-common linux-tools-generic
```

On lance ensuite une analyse en demandant à **perf** de collecter les différents types d'informations qui nous intéressent. Par exemple pour le cas où **dim = 1024**, sur un Intel Celeron N3050, on obtient :

```
1  sudo perf stat -e task-clock,cycles,instructions,cache-references,cache-misses
2      build/bin/asm_matprod_32.exe -s 1024 -m 1
3
4        69975,730658  task-clock (msec)  #  0,982 CPUs utilized
5     150 326 657 774  cycles             #  2,148 GHz                  (49,97%)
6       5 858 594 620  instructions       #  0,04  insn per cycle       (74,96%)
7       2 535 065 211  cache-references   # 36,228 M/sec                (75,01%)
8       2 319 680 000  cache-misses       # 91,504 % of all cache refs (75,02%)
9
10       71,243961985 seconds time elapsed
11
12       69,865177000 seconds user
13        0,111656000 seconds sys
```

Le nombre de défauts de cache (`cache-misses`, ligne 8) est de l'ordre de 91,5% pour un temps d'exécution de 71,24 s (ligne 10) alors que pour **dim = 1023** et **dim = 1025**, on obtient de l'ordre de 3% de défauts de cache pour un temps d'exécution de 17 à 18 s.

Il est possible d'obtenir une information plus fine quant aux défauts de cache notamment pour savoir dans quelle partie du code ils apparaissent. Pour cela il faut

compiler le code C avec l'option `-g` ou `-ggdb` de `gcc`. La commande `perf` que nous venons d'utiliser avec l'option `stat` donne un résumé global des informations pour l'exécution du programme alors que `perf record` réalise un échantillonnage des parties du code où apparaissent les différents événements, comme les défauts de cache, et les stocke dans un fichier appelé `perf.data` :

```
sudo perf record -e cache-misses build/bin/asm_matprod_32.exe -s 1024 -m 1
```

On utilise ensuite la commande `perf report` avec l'option `–stdio` afin d'obtenir une liste des sous-programmes touchés par le défaut de cache par ordre décroissant :

```
# To display the perf.data header info, please use --header options.
#
#
# Total Lost Samples: 0
#
# Samples: 68K of event 'cache-misses'
# Event count (approx.): 89605829
#
# Overhead  Command          Shared Object      Symbol
# ........  ...............  .................  ........................
#
    97.01%  asm_matpr...exe  matprod_main.exe   [.] _Z12mp_referencePfS_S_j
     0.22%  asm_matpr...exe  [kernel.kallsyms]  [k] clear_page_erms
     0.17%  asm_matpr...exe  [kernel.kallsyms]  [k] rcu_check_callbacks
     0.16%  asm_matpr...exe  [kernel.kallsyms]  [k] task_tick_fair
     ...
```

On voit que les défauts de cache apparaissent dans la fonction `mp_reference`, qui est la fonction de référence du produit de matrice, pour 97% (ligne 12 ci-dessus). On peut également obtenir pour chaque ligne de code assembleur le pourcentage de défauts de cache en utilisant `perf annotate` :

```
 Samples: 66K of event 'cache-misses', 4000 Hz, Event count (approx.): 86922764
_Z12mp_referencePfS_S_j  ...build/bin/asm_matprod_32.exe
Percent|       cmp    0x8(%esp),%ecx
       |     → je     2792 <mp_reference(float*, float*, float*, unsigned int)>
  0,00 |       mov    (%esp),%edi
       |       mov    0x18(%esp),%esi
       |                                    sum += a(i,k) * b(k,j);
  0,04 |       movss  (%edx),%xmm2
  3,67 |       movss  (%eax),%xmm3
 13,14 |       add    \$0x20,%ecx
  0,10 |       insertps \$0x10,(%edx,%esi,1),%xmm2
 13,10 |       insertps \$0x10,(%eax,%esi,1),%xmm3
 12,13 |       add    %edi,%edx
```

```
  0,00 |       add    %edi,%eax
  0,01 |       movups -0x20(%ecx),%xmm5
  0,11 |       movss  (%edx),%xmm6
 11,05 |       movlhps %xmm2,%xmm3
  0,05 |       insertps \$0x10,(%edx,%esi,1),%xmm6
 11,27 |       movss  (%eax),%xmm7
 11,49 |       add    %edi,%edx
  0,10 |       mulps  %xmm5,%xmm3
  0,11 |       insertps \$0x10,(%eax,%esi,1),%xmm7
 11,51 |       movups -0x10(%ecx),%xmm1
Press 'h' for help on key bindings
```

Ici c'est l'instruction `sum += a(i,k) * b(k,j);` qui subit de nombreux défauts de cache. Le code est donné avec une syntaxe de type AT&T et l'instruction `movss (%eax),%xmm3` (ligne 9) est équivalente dans la syntaxe Intel (pour laquelle il faut inverser les arguments) à :

```
movss xmm3, [eax]
```

Il semble que les défauts de cache soient reportés sur la ligne suivante. Ainsi pour l'instruction précédente, on observe un défaut de cache de $13,14\%$.

Temps de référence

Pour l'implantation que nous venons de donner, l'exécution dure environ 71 secondes sur un Intel Core i7 4900MQ pour le produit de matrice de référence avec une taille de matrice de 2048 éléments.

11.6 Amélioration avec inversion des boucles j et k

Une première amélioration très efficace consiste à inverser les boucles j et k. Dans ce cas la formule de calcul doit être modifiée, on doit utiliser `c(i,j) +=` lors de chaque calcul. La matrice C devra dans ce cas être initialisée avec des 0 avant de réaliser le produit. On observe, Listing 11.6.1, que j étant la boucle la plus interne, `a(i,k)` reste constant et que les coefficients des matrices `B` et `C` sont consécutifs car j apparaît en second paramètre de `c()` et `b()`, ce qui rend le calcul très efficace (cf. Section 11.9).

Amélioration inversion de boucles j et k

En inversant les boucles `j` et `k` on ne met plus que $2,46$ secondes, on va donc environ 29 fois plus vite.

```
void mp_inv_jk(f32 *A, f32 *B, f32 *C, u32 dim) {
    for (u32 i = 0; i < dim; ++i) {
        for (u32 k = 0; k < dim; ++k) {
            for (u32 j = 0; j < dim; ++j) {
                c(i,j) += a(i,k) * b(k,j);
            }
        }
    }
}
```

Listing 11.6.1 – Produit de matrice, Inversion de boucles j et k

11.7 Version SSE de l'inversion de la boucle j, k

Nous allons créer une version vectorisée en SSE de la variante de l'inversion de boucle `j`, `k` en considérant, pour simplifier les choses, que `dim` est multiple de 4. La boucle la plus interne sera donc dépliée en

```
for (u32 j = 0; j < dim ; j += 4) {
    c(i, j+0) += a(i, k) * b(k, j+0);
    c(i, j+1) += a(i, k) * b(k, j+1);
    c(i, j+2) += a(i, k) * b(k, j+2);
    c(i, j+3) += a(i, k) * b(k, j+3);
}
```

Le coefficient `a(i,k)`, comme indiqué précédemment, reste constant, on le chargera dans `xmm0`. On utilisera `xmm1` pour stocker `b(k, j:j+3)` et `xmm2` pour stocker `c(i, j:j+3)`

L'analyse du code du Listing 11.3.1 montre qu'il existe 7 variables entières et qu'il faut prendre en compte également le calcul des adresses des coefficients `a(i,k)`, `b(k,j)` et `c(i,j)`. Les calculs entre les coefficients sont réalisés par les unités vectorielles. On va donc tenter d'optimiser la boucle la plus interne (boucle `j`) et on stockera `i` et `k` dans la pile en tant que variables locales :

Le sous-programme débute par la création des variables `i` et `k` dans la pile, puis on initialise `i` à 0 et on commence à écrire le code de la boucle `for i` :

```
        push    ebp
        mov     ebp, esp
        sub     esp, 8              ; réserve l'espace pour i et k
                                    ; i est en [ebp-4]
                                    ; k est en [ebp-8]
        push    ebx                 ; sauvegarde des registres
        push    edi
        push    esi

        xor     eax, eax            ; i = 0
```

Cste/Param/Var	Type	Paramètre	Registre	Description
A	f32 *	[ebp+8]		matrice A
B	f32 *	[ebp+12]		matrice B
C	f32 *	[ebp+16]		matrice C
dim	u32	[ebp+20]	edx	dimension
&a(i,k)	f32 *		ebx	adresse de a(i,k)
&b(k,0)	f32 *		esi	adresse de b(k,0)
&c(i,0)	f32 *		edi	adresse de c(i,0)
i	u32		[ebp-4]	variable de boucle
k	u32		[ebp-8]	variable de boucle
j	int		ecx	variable de boucle
a(i,k) x 4	f32 [4]		xmm0	4 fois a(i,k)
b(k,j:j+3)	f32 [4]		xmm1	
c(i,j:j+3)	f32 [4]		xmm2	

TABLE 11.2 – Association variable C avec registres pour l'inversion de boucle

```
11          mov     [ebp-4], eax        ;
12  .for_i:
13          mov     eax, [ebp-4]        ; fin de boucle si i >= dim
14          cmp     eax, [ebp + 20]     ;
15          jge     .endfor_i           ;
16
17          <<<1>>>                     ; reste du code
18
19          inc     dword [ebp-4]       ; ++i
20          jmp     .for_i
21  .endfor_i:
22
23          pop     esi                 ; restauration des registres
24          pop     edi
25          pop     ebx
26          mov     esp, ebp
27          pop     ebp
28          ret
```

La partie notée <<<1>>> est développée ci-après. On écrit le code de la boucle k :

```
1   <<<1>>>
2           xor     ecx, ecx            ; boucle k
3           mov     [ebp-8], ecx        ; k = 0
4   .for_k:
5           mov     ecx, [ebp-8]        ; fin de boucle si k >= dim
6           cmp     ecx, [ebp+20]       ;
7           jge     .endfor_k
```

```
8
9           <<<2>>>
10
11          inc     dword [ebp-8]       ; ++k
12          jmp     .for_k
13  .endfor_k:
```

On poursuit avec le calcul de l'adresse de la valeur `a(i,k)`, valeur que l'on charge dans la partie basse de `xmm0`, puis que l'on recopie 3 fois dans `xmm0` grâce à l'instruction `pshufd`. On calcule également dans `edi` l'adresse de `c(i,0)` et l'adresse de `b(k,0)` dans `esi`

```
1   <<<2>>>
2           mov     eax, [ebp-4]        ; i
3           mul     dword [ebp+20]      ; i*dim
4           mov     edi, eax            ; i*size
5           mov     ebx, eax            ; i*size
6           add     ebx, ecx            ; i*size+k
7           shl     edi, 2              ; (i*size)*sizeof(float)
8           shl     ebx, 2              ; (i*size+k)*sizeof(float)
9           add     ebx, [ebp+8]        ; a[i*size+k]
10          add     edi, [ebp+16]       ; c[i*size]
11
12          movss   xmm0, [ebx]         ; xmm0 = a(i,k)
13          pshufd  xmm0, xmm0, 0       ; recopie dans xmm0
14
15          mov     eax, ecx            ; k
16          mul     dword [ebp+20]      ; k*size
17          shl     eax, 2              ; k*size*sizeof(float)
18          mov     esi, [ebp+12]       ; b
19          add     esi, eax            ; b[k*size]
20          <<<3>>>
```

Enfin, on écrit la boucle `for j` :

```
1   <<<3>>>
2           xor     ecx, ecx            ; j = 0
3           mov     edx, [ebp+20]
4   .for_j:
5           cmp     ecx, edx            ; fin de boucle si j >= dim
6           jge     .endfor_j
7
8           movdqu  xmm1, [esi + ecx*4] ; xmm1 = <b(k,j+3), ... , b(k,j)>
9           movdqu  xmm2, [edi + ecx*4] ; xmm2 = <c(i,j+3), ... , c(i,j)>
10          mulps   xmm1, xmm0          ; xmm1 = <a(i,k)*b(k,j+3), ... >
11          addps   xmm2, xmm1
12          movdqu  [edi + ecx*4], xmm2
13
14          add     ecx, 4              ; j += 4
15          jmp     .for_j
16  .endfor_j:
```

Amélioration inversion de boucles j et k + SSE

En inversant les boucles `j` et `k` et en combinant avec la technologie vectorielle SSE, on ne met plus que $3,70$ secondes, on va donc environ 19 fois plus vite. Cela est moins performant que la seule inversion des boucles `j` et `k`, car il n'y a pas de dépliage de boucle.

11.8 Tuilage

Une autre technique d'amélioration, évoquée Chapitre 5, consiste à réaliser le tuilage (ou *tiling*) en ne travaillant que sur une petite partie des données. Dans ce cas les matrices sont découpées en carrés qui correspondent à de petites zones mémoires ce qui permet de les charger dans le cache et les réutiliser.

11.8.1 Tuilage 4×4 avec SSE

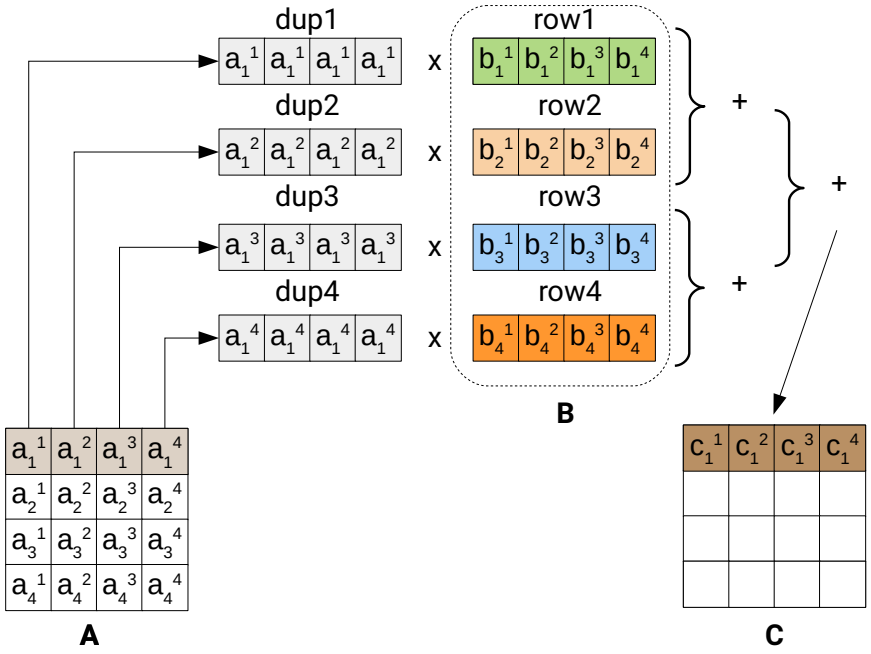


FIGURE 11.5 – Produit de matrices 4 par 4 en SSE Intrinsics

On peut concevoir un sous-programme de calcul du produit de deux matrices $A(4,4)$ par $B(4,4)$ en chargeant les données dans les registres SSE. Le principe est simple, on charge les lignes de B dans des registres SSE puis on réalise les multiplications pour chaque ligne de A.

Nous donnons listing 11.8.1 le code intrinsics qui permet de réaliser le produit

```
void M4x4_SSE(float *A, float *B, float *C, u32 dim) {
    __m128 row1 = _mm_load_ps(&B[0]);
    __m128 row2 = _mm_load_ps(&B[dim]);
    __m128 row3 = _mm_load_ps(&B[2*dim]);
    __m128 row4 = _mm_load_ps(&B[3*dim]);

    for(int i=0; i<4; i++) {
        __m128 dup1 = _mm_set1_ps(A[dim*i + 0]);
        __m128 dup2 = _mm_set1_ps(A[dim*i + 1]);
        __m128 dup3 = _mm_set1_ps(A[dim*i + 2]);
        __m128 dup4 = _mm_set1_ps(A[dim*i + 3]);

        __m128 sum1 = _mm_add_ps(   _mm_mul_ps(dup1, row1),
                                    _mm_mul_ps(dup2, row2)  );
        __m128 sum2 = _mm_add_ps(   _mm_mul_ps(dup3, row3),
                                    _mm_mul_ps(dup4, row4)  );

        __m128 row = _mm_add_ps(sum1, sum2);
        __m128 old_row = _mm_load_ps(&C[dim*i]);

        row = _mm_add_ps(row, old_row);
        _mm_store_ps(&C[dim*i], row);
    }
}
```

Listing 11.8.1 – Produit de matrice - Tuilage 4x4

efficacement. Si on utilise l'option `-funroll-loops` de `gcc`, la boucle sera dépliée et normalement plus efficace. L'intrinsics `_mm_set1_ps` qui charge quatre fois la valeur de son argument dans un registre SSE est traduite par deux instructions :

- `movss` qui charge l'argument dans la partie basse d'un registre SSE
- `shufps` avec un masque égal à 0 afin de recopier trois fois la valeur en partie basse dans les quatre emplacements de 32 bits du registre SSE

On peut par la suite réaliser un sous-programme qui effectue le produit de deux matrices de dimensions multiple de 4 et réutiliser le sous-programme précédent.

Amélioration avec tuilage 4x4

En utilisant le tuilage 4×4 on met 9 secondes, on va donc environ 8 fois plus vite. Ce n'est pas très efficace.

11.8.2 Tuilage $b \times b$ de manière générale

On peut réaliser le tuilage de plusieurs manières différentes certaines étant plus efficaces que d'autres. Nous avons implanté plusieurs versions du tuilage (versions 1 à 4) :

- la version 1 réalise le blocage de boucle sur j et k, puis on fait varier i de 0 à $dim - 1$
- la version 2 réalise le blocage de boucle sur i, j puis k, puis à l'intérieur de chaque bloc on fait varier i, j puis k
- la version 3 réalise le blocage de boucle sur i et j, puis k varie de 0 à $dim - 1$
- la version 4 réalise le blocage de boucle sur i, j puis k et à l'intérieur de chaque bloc on fait varier i, k puis j

La dernière version (version 4) semble être la plus efficace. Son code est donné Listing 11.8.2.

```
1  void mp_tile_nxn_v4(f32 *A, f32 *B, f32 *C, u32 size) {
2
3      for (u32 i=0; i<size; i += BLOCK_DIM) {
4          for (u32 j=0; j<size; j += BLOCK_DIM) {
5              for (u32 k=0; k<size; k += BLOCK_DIM) {
6                  for (u32 ib=i; ib<min(i+BLOCK_DIM,size); ++ib) {
7                      for (u32 kb=k; kb<min(k+BLOCK_DIM,size); ++kb) {
8                          f32 *aib = &a(ib,0);
9                          f32 *cib = &c(ib,0);
10                         for (u32 jb=j; jb<min(j+BLOCK_DIM,size); ++jb) {
11                             cib[jb] += aib[kb] * b(kb,jb);
12                         }
13                     }
14                 }
15             }
16         }
17     }
18 }
```

Listing 11.8.2 – Produit de matrice - Tuilage

On utilise la variable `BLOCK_DIM` qui représente le facteur de blocage. Ce facteur de blocage (b) influe sur le temps de calcul. Cette valeur est généralement égale à 16, 32, 64 ou 128, et il est préférable de prendre une valeur multiple de 4 ou 8 afin que le compilateur vectorise le code en utilisant les technologies SSE ou AVX. La Table 11.3, ci-après, présente les résultats obtenus en utilisant le script `blocking_factor_test.php` pour la fonction `mp_tile_bxb_v4` en faisant varier la variable `BLOCK_DIM`.

Facteur	Temps	Facteur	Temps	Facteur	Temps
8	5.64	88	0.74	948	0.62
12	4.31	92	0.95	952	0.55
16	2.39	96	0.75	956	0.61
20	2.32	100	0.92	960	0.55
24	1.63	104	0.70	964	0.61
28	1.86	108	0.89	968	0.53
32	1.25	112	0.72	972	0.60
36	1.53	116	0.90	976	0.53
40	1.09	120	0.71	980	0.61
44	1.36	124	0.88	984	0.53
48	0.97	128	0.72	988	0.60
52	1.26	132	0.92	992	0.53
56	0.93	136	0.72	996	0.60
60	1.19	140	0.87	1000	0.52
64	0.83	144	0.71	1004	0.60
68	1.08	148	0.87	1008	0.52
72	0.76	152	0.70	1012	0.60
76	1.00	156	0.88	1016	0.53
80	0.74	160	0.70	1020	0.60
84	0.98	164	0.85	1024	0.54

TABLE 11.3 – Temps d'exécution en secondes sur AMD Ryzen 5 3600 du produit de matrices 2048×2048 avec tuilage et influence du facteur de blocage b entre 8 et 1024.

On note que plus la valeur du facteur de blocage augmente, plus le temps de calcul diminue jusqu'à atteindre une valeur minimale, puis au-delà, le temps de calcul augmente légèrement.

Amélioration tuilage $b \times b$

Avec un tuilage de $b = 64$ on ne met plus que $0,83$ secondes. Dans le test que nous avons effectué, nous avons fait varier le facteur de blocage entre 8 et 1024 et le temps minimum de $0,52$ secondes est atteint pour $b = 1000$ ou $b = 1008$, soit un facteur d'amélioration d'environ $43,62/0,52 \simeq 84$ par rapport à la méthode de référence.

11.9 Tests de performance

Afin de tester les différentes versions que nous avons écrites, nous allons examiner les résultats obtenus pour les méthodes suivantes :

1. méthode de référence (référence)
2. traduction directe de la méthode de référence en assembleur avec utilisation du coprocesseur pour les calculs sur les flottants
3. amélioration de la méthode précédente avec dépliage par 4 de la boucle k
4. méthode avec inversion des boucles j et k, optimisée par le compilateur GCC avec option `-O2` (`inv_jk_O2`)
5. méthode avec inversion des boucles j et k, optimisée par le compilateur GCC avec les options `-O3` et `-funroll-loops` (`inv_jk`)
6. méthode avec inversion des boucles j et k, optimisée avec SSE (`inv_jk_sse`)
7. méthode avec inversion des boucles j et k, optimisée avec AVX (`inv_jk_avx`)
8. méthode avec tuilage 4 par 4 (tile 4x4)
9. méthode avec tuilage b par b, version 1 (tile bxb v1)
10. méthode avec tuilage b par b, version 2 (tile bxb v2)
11. méthode avec tuilage b par b, version 3 (tile bxb v3)
12. méthode avec tuilage b par b, version 3 (tile bxb v4)

Pour ce qui est des méthodes de tuilage, nous avons choisi $b = 64$, bien que comme évoqué précédemment, ce ne soit pas le facteur de blocage qui donne les meilleurs résultats.

La colonne *ratio* représente le rapport entre le temps de la méthode de référence et celui de la méthode d'inversion de boucle (méthode 3).

N°	Marque Gamme Modèle Année	Intel Pentium D 925 2006	Intel Core 2 Q9300 2008	Intel Core i7 860 2009	AMD Phenom II 1090T 2010	Intel Core i5 3570k 2012	Intel Core i7 4790 2014
1	`mp_reference`	363.91	257.74	90.53	172.35	78.62	64.46
2	`mp_asm_fpu`	380.06	190.83	74.86	180.55	78.33	63.05
3	`mp_asm_fpu_ur4`	379.90	197.35	74.40	179.28	79.18	63.02
4	`mp_inv_jk_O2`	23.07	14.94	8.25	10.35	6.27	5.00
5	`mp_inv_jk`	**8.51**	8.09	**2.63**	**4.79**	**2.16**	2.07
6	`mp_inv_jk_sse`	85.23	32.28	4.36	17.06	4.32	4.96
7	`mp_inv_jk_avx`	-	-	-	-	2.33	1.93
8	`mp_tile_4x4`	35.77	28.71	10.19	21.24	7.98	6.76
9	`mp_tile_bxb_v1`	28.19	178.39	23.82	60.46	19.25	16.82
10	`mp_tile_bxb_v2`	25.74	33.68	23.15	49.14	15.20	11.67
11	`mp_tile_bxb_v3`	12.62	7.99	3.98	7.62	2.97	1.86
12	`mp_tile_bxb_v4`	14.11	**7.70**	3.74	6.36	2.82	**1.55**
	ratio (1/5)	42.76	31.85	34.42	35.98	36.39	31.14

TABLE 11.4 – Architectures anciennes : temps d'exécution en secondes du produit de matrices 2048×2048 en 32 bits

11.9.1 Architectures anciennes (avant 2015)

Nous avons fait figurer Table 11.4, les résultats obtenus pour les architectures anciennes, c'est à dire les processeurs conçus avant 2015. Nous considérons ici le produit de deux matrices carrées de dimension 2048.

Le codage en assembleur avec utilisation de la FPU (méthode 2) n'est pas probant : il peut améliorer le temps de calcul comme dans le cas de l'Intel Q9300 ou l'i7 860, ou alors, le dégrader (Pentium D, AMD Phenom II 1090T), ou alors être équivalent à la fonction de référence (i5 3570k, i7 4790). Le dépliage par 4 de la boucle k (méthode 3) n'apporte aucune amélioration et a parfois tendance à être moins efficace (Q9300).

On note que le fait d'inverser les boucles j et k apporte un gain de performance non négligeable même si le code n'est optimisé qu'avec `-O2` (méthode 4). On passe par exemple pour un Pentium D 925 de 363 à 23 secondes, soit un facteur d'amélioration de $15,78$. On va donc presque 16 fois plus vite. En utilisant les options de compilation comme `-O3` (méthode 5), on gagne encore un facteur $\times 2$ à $\times 2,5$ suivant le microprocesseur.

Les méthodes 6 qui consiste à faire le calcul principal en utilisant les registres SSE dégrade les performances par rapport à la méthode 5. Par contre la méthode 7

tend à se rapprocher des résultats de la méthode 5. Le compilateur C a probablement utilisé l'AVX lors de la traduction de la méthode 5.

Le tuilage 4×4 n'est pas très intéressant et les résultats sont moins bons que la méthode 4.

Le tuilage est intéressant sauf pour les méthodes 9 et 10, mais le fait de se limiter à une taille de 64×64 est pénalisant.

11.9.2 Architectures modernes (2015 à 2019)

En ce qui concerne les architectures modernes (cf. Table 11.5), on observe le même phénomène que noté précédemment pour l'inversion des boucles j et k. Mais dans ce cas, le tuilage dans sa version 3 ou 4 avec facteur de blocage de 64 est parfois plus performant que l'inversion des boucles.

N°	Marque Gamme Modèle	Intel Core i3 6100 2015	AMD Ryzen 7 1700X 2017	Intel Core i5 7400 2017	Intel Core i7 8700 2017	AMD Ryzen 5 3600 2019	Intel Xeon 4208 2019
1	`mp_reference`	60.08	57.66	49.10	50.13	43.62	53.63
2	`mp_asm_fpu`	59.77	52.46	49.28	46.85	34.42	53.23
3	`mp_asm_fpu_ur4`	60.63	55.26	48.36	42.57	45.58	52.22
4	`mp_inv_jk_O2`	10.22	6.84	10.32	8.01	6.16	11.28
5	`mp_inv_jk`	2.11	1.53	**1.56**	**1.29**	0.92	2.22
6	`mp_inv_jk_sse`	5.84	5.53	5.81	4.53	4.39	4.33
7	`mp_inv_jk_avx`	2.18	1.76	1.63	1.40	1.03	2.51
8	`mp_tile_4x4`	6.89	6.35	5.27	4.71	4.62	7.74
9	`mp_tile_bxb_v1`	17.31	122.13	18.05	14.03	37.54	21.31
10	`mp_tile_bxb_v2`	12.49	27.39	12.70	10.57	27.60	13.40
11	`mp_tile_bxb_v3`	1.84	1.78	1.85	2.48	0.97	1.86
12	`mp_tile_bxb_v4`	**1.48**	**1.46**	1.72	2.48	**0.81**	**1.62**
	ratio (1/5)	28.47	37.68	31.47	38.86	47.41	24.15

TABLE 11.5 – Architectures modernes : temps d'exécution en secondes du produit de matrices 2048×2048 en 32 bits

11.9.3 Architectures récentes (2020 et après)

Pour les architectures récentes (cf. Table 11.6), on note les mêmes tendances que pour les architectures modernes. C'est la version avec tuilage qui est la plus efficace.

N°	Marque Gamme Modèle	Intel Core i7 10850H 2020	AMD Ryzen 5 5600g 2021
1	`mp_reference`	39.18	46.48
2	`mp_asm_fpu`	39.70	39.27
3	`mp_asm_fpu_ur4`	37.56	46.59
4	`mp_inv_jk_O2`	7.71	4.19
5	`mp_inv_jk`	1.26	0.80
6	`mp_inv_jk_sse`	2.76	2.68
7	`mp_inv_jk_avx`	1.33	0.90
8	`mp_tile_4x4`	4.59	4.91
9	`mp_tile_bxb_v1`	13.18	28.61
10	`mp_tile_bxb_v2`	9.76	21.36
11	`mp_tile_bxb_v3`	1.47	0.97
12	`mp_tile_bxb_v4`	**1.23**	**0.77**
	ratio (1/5)	31.09	58.10

TABLE 11.6 – Architectures récentes : temps d'exécution en secondes du produit de matrices 2048×2048 en 32 bits

11.9.4 Analyse des versions liées au tuilage

La Table 11.7 donne, pour les différentes implantations du blocage de boucle, les temps d'exécution obtenus sur différents processeurs. Les deux premières versions sont très mauvaises puisque plus le facteur augmente, plus le temps d'exécution augmente. Or, ce devrait être l'inverse, bien que pour les processeurs Intel i5 7400 et i7 8700, on observe une diminution du temps de calcul.

Etrangement, c'est sur les processeurs AMD que cette tendance, inverse de la normale, s'observe de manière flagrante et de manière moins prononcée sur l'Intel Xeon 4208. Avec l'utilitaire `perf` on peut déterminer que c'est le nombre de références au cache qui augmente passant de 3 à 12, 21 puis 37 milliards. A partir de là, c'est le nombre de défauts de cache qui augmente alors que le nombre de références au cache diminue. Le cas de l'AMD Ryzen 1700X est emblématique puisqu'on atteint des temps de calcul de plus de 120 secondes pour la version 1.

Les versions 3 et 4 sont conformes à ce qui est attendu, l'augmentation du facteur de blocage a pour effet de diminuer le temps d'exécution. La méthode 4 étant la plus efficace de ces deux méthodes. On remarque que passer d'un facteur de blocage de 8 à 16 puis 32 et enfin 64 améliore sensiblement le temps de calcul. Ensuite, à partir de 128 et 256 le gain est faible. On note cependant que pour l'Intel Xeon 4208, pour la valeur $b = 256$, le temps de calcul augmente. Il faudrait

Méthode	blocage	i7 4790	i5-7400	i7 8700	Ryzen 3600	Ryzen 1700	Xeon 4208
version 1	8	21,19	22,10	17,77	**15,99**	**20,46**	**18,72**
version 1	16	17,02	19,43	15,58	**29,13**	**59,53**	**18,01**
version 1	32	16,48	18,44	14,67	**37,84**	**120,75**	**19,50**
version 1	64	15,38	18,03	14,10	**38,33**	**125,88**	**20,73**
version 1	128	15,17	18,00	14,10	**44,20**	**127,45**	**21,68**
version 1	256	16,03	18,38	13,86	**43,85**	**128,80**	**32,50**
version 2	8	9,62	**8,78**	**7,54**	**8,44**	**10,70**	**11,88**
version 2	16	8,83	**7,04**	**5,86**	**13,60**	**14,83**	**10,71**
version 2	32	13,17	**8,97**	**7,62**	**22,96**	**57,23**	**11,80**
version 2	64	12,50	**12,76**	**10,42**	**28,06**	**75,01**	**13,11**
version 2	128	12,20	**13,74**	**10,80**	**40,00**	**80,63**	**14,69**
version 2	256	12,08	**14,21**	**10,63**	**47,77**	**79,55**	**30,53**
version 3	8	11,15	11,49	17,57	4,30	5,56	12,67
version 3	16	4,94	5,55	6,43	2,35	3,22	7,17
version 3	32	2,67	2,76	3,90	1,46	2,11	3,05
version 3	64	1,83	1,83	2,37	0,99	1,78	1,95
version 3	128	1,37	1,44	1,71	0,83	1,52	1,33
version 3	256	1,45	1,37	1,32	0,76	1,40	2,89
version 4	8	7,10	7,63	13,02	4,50	6,31	9,23
version 4	16	3,57	3,54	4,77	2,19	2,95	4,60
version 4	32	2,28	2,21	3,08	1,25	1,72	2,56
version 4	64	1,54	1,55	2,51	0,82	1,46	1,66
version 4	128	1,13	1,42	1,83	0,73	1,25	1,14
version 4	256	1,08	1,32	1,46	0,66	1,22	1,92

TABLE 11.7 – Tuilage : influence de l'implantation et de la dimension du facteur de blocage pour le produit de matrices 2048×2048

investiguer afin de déterminer si c'est le cache du Xeon qui est la source de cette augmentation. Il est pourtant de 11 ko pour le cache L3 alors que celui d'un i5 7400 est de 6 ko, mais est 11-*way set associative* sur Xeon, alors qu'il est entre 12 et 16-*way set associative* sur les autres machines. Le problème ne proviendrait-il pas de là ?

En fonction de la taille des caches, on obtiendra des temps de calcul plus ou moins importants en faisant varier le facteur de blocage. Ainsi, sur un Intel Q9300, voici pour différentes dimensions n de la matrice, le facteur b donnant le temps de calcul minimal :

- pour $n = 1024$, $b = 776$ ou 1024
- pour $n = 2048$, $b = 256$
- pour $n = 3072$, $b = 512$
- pour $n = 4096$, $b = 192$

11.10 Conclusion

Comme le montre cette étude de cas, **l'ordre dans lequel on accède les données possède une grande influence sur le temps de calcul** et en particulier pour le produit de matrice. Nous avons mis en exergue le fait que l'implantation directe de la formule mathématique produisait des temps de calcul très fluctuants et prohibitifs pour certaines tailles de matrices. L'implantation avec inversion des boucles j et k corrige ce défaut. Enfin, l'utilisation d'un facteur de blocage lié au tuilage apporte une amélioration très importante et intéressante, mais il faut être en mesure de bien implanter le blocage en le combinant par exemple avec l'inversion des boucles j et k. On note également beaucoup de comportements spécifiques en fonction de l'implantation et du microprocesseur utilisé au niveau des résultats. Tout ceci montre qu'**il peut être nécessaire de modifier des algorithmes de base afin de gagner en efficacité**. Comme nous l'avons montrer on peut atteindre sur AMD Ryzen 5 3600, un facteur d'amélioration de 84 entre la version de référence et la version avec tuilage pour laquelle on utilise un facteur de blocage $b = 1000$.

11.11 Exercices

Exercice 45 - A titre d'exercice vous pouvez réaliser un dépliage de la version SSE par 2, puis par 4 et incorporer à l'étude de cas ces fonctions afin de les tester.

Exercice 46 - Il serait intéressant d'étudier le facteur de blocage b afin de déterminer en fonction de la dimension de la matrice n ainsi que de la taille des caches L1, L2, L3, quelle valeur est la plus intéressante. Par exemple, sur un Intel Core i5 7400, le facteur de blocage qui donne le meilleur temps de calcul pour $n = 4096$ est obtenu pour $b = 512$. En réalisant un échantillonage des dimensions de la matrice, réaliser des tests en faisant varier le facteur de blocage et déterminer :

- quel facteur de blocage est le pus intéressant en moyenne
- quels facteurs de blocage sont les pus intéressants en fonction de la dimension de la matrice

Chapitre 12

Etude de cas POPCNT

12.1 Introduction

Compter le nombre de bits à 1 dans un registre est une opération que l'on rencontre dans de nombreux traitements. Par exemple imaginons que l'on dispose d'un tableau de booléens qui indique si un élément d'un tableau d'enregistrements doit être traité ou non. La question se posera probablement de savoir combien d'enregistrements doivent être traités afin d'allouer l'espace juste nécessaire avant de manipuler les données. Si on utilise un tableau de booléens, on aura la définition de données suivante :

```
#include <stdint.h>
typedef uint8_t u8;
typedef unsigned uint32_t u32;
// nombre d'enregistrements
const u32 MAX_RECORDS = 100000;

// Enregistrement
typedef struct {
....
} Record;

// tableau d'enregistrements
Record tab_records[ MAX_RECORDS ];

// tableau qui indique les enregistrements à traiter
bool tab_process[ MAX_RECORDS ];
```

La variable `tab_records` est un tableau d'enregistrements et `tab_process` un tableau de booléens. Si la variable `tab_process[i]` est à `true` cela signifie que l'enregistrement correspondant devra être pris en compte dans un traitement ultérieur.

On pourra donc définir plusieurs méthodes liées au traitement du tableau

`tab_process` :

- `void set(u32 n)` qui met à jour le tableau `tab_process` afin d'indiquer que l'enregistrement `n` doit être traité
- `void unset(u32 n)` qui met à jour le tableau `tab_process` afin d'indiquer que l'enregistrement `n` ne doit pas être traité
- `bool use(u32 n)` qui retourne `true` si on doit traiter l'enregistrement `n`
- `u32 count()` qui retourne le nombre d'enregistrements qui doivent être traités

Le code de ces sous-programmes est très simple et ressemble à ce qui suit :

```
void set(u32 n) {
    tab_process[ n ] = true;
}

void unset(u32 n) {
    tab_process[ n ] = false;
}

bool use(u32 n) {
    return tab_process[ n ];
}

u32 count() {
    u32 total = 0;

    for (u32 i = 0; i < MAX_RECORDS; ++i)
        total += (u32) tab_process[i];
    return total;
}
```

La variable `tab_process` étant un tableau de booléens elle occupe en mémoire 100_000 octets car un booléen possède une taille d'un octet. On utilise donc $100\_000/1024 \simeq 98$ ko. Cependant sur ces 100_000 octets, seuls 100_000 bits sont vraiment utiles car la constante `true` est en fait égale à 1 et `false` vaut 0. En d'autres termes, 7 bits sur 8, soit $87,5$ % sont inutiles car non utilisés, seul le bit de poids faible code pour `true` ou `false`.

Il est donc plus intéressant de ne pas perdre de mémoire et de coder chaque valeur booléenne non pas par un octet mais par un bit. On parle alors de **compactage des données**. Dans ce cas le tabeau `tab_process` que nous renommons alors `tab_process_bits` aura une taille de $(100\_000 + 7)/8 \simeq 12500 \simeq 12,2$ ko. L'expression $100\_000 + 7$ permet d'arrondir la taille à l'octet supérieur.

```
u32 MAX_RECORDS_IN_BYTES = (MAX_RECORDS + 7) / 8;
u8 tab_process_bits[ MAX_RECORDS_IN_BYTES ];
```

Les méthodes que nous avons définies précédemment doivent être réécrites afin de prendre en compte les spécificités du nouveau tableau `tab_process_bits` :

```
void set(u32 n) {
    tab_process_bits[ n / 8 ] |= 1 << (n % 8);
}

void unset(u32 n) {
    tab_process_bits[ n / 8 ] &= ~(1 << (n % 8));
}

bool use(u32 n) {
    return (tab_process_bits[n / 8] & (1 << (n % 8))) != 0;
}

u32 count() {
    u32 total = 0;

    for (u32 i = 0; i < MAX_RECORDS_IN_BYTES; ++i) {
        total += pop_count_8( tab_process_bits[i] );
    }
    return total;
}
```

Par exemple pour la méthode `set`, l'élément `n` se trouve à l'indice `n / 8` du tableau `tab_process_bits` et occupe le bit à la position `n % 8`. La traduction de ce sous-programme en assembleur x86 32 bits est la suivante :

```
set:
        push    ebp
        mov     ebp, esp
        mov     ecx, [ebp + 8] ; ecx = n
        mov     edx, ecx       ; edx = n
        shr     edx, 3         ; edx = n / 8
        and     ecx, 7         ; ecx = n % 8
        mov     eax, 1         ; eax = 1
        shl     eax, cl        ; eax = 1 << (n % 8)
        or      [tab_process_bits + edx], al
        mov     esp, ebp
        pop     ebp
        ret
```

La fonction `count` doit être réécrite en utilisant la fonction `pop_count_8` qui compte le nombre de bits à 1 dans un octet. Une version simple de cette fonction qui nous servira de fonction de référence, est par exemple :

```
u32 pop_count_8(u8 n) {
    u32 count_bits = 0;

    while (n) {
        if ((n & 1) != 0) ++count_bits;
        n = n >> 1;
    }
    return count_bits;
```

```
}
```

On réalise une boucle et tant que la variable `n` n'est pas égale à 0, on regarde si le bit de poids faible est égal à 1 et dans ce cas on incrémente le compteur `count_bits`, puis on décale de 1 bit vers la droite la valeur de `n` et on recommence. Vous pouvez essayer, à titre d'exercice, d'écrire cette fonction en assembleur 32 bits.

Temps de référence

Le test de référence consiste à réaliser 30_000 fois le calcul de la somme du nombre de bits d'un tableau de 262_207 octets. Initialement chaque octet du tableau se voit assigner une valeur aléatoire.
Les tests sont réalisés sur un AMD Ryzen 5 3600. Pour l'implantation par le compilateur `gcc` de la fonction de référence, l'exécution dure environ $54,96$ secondes.

L'efficacité de la fonction est biaisée par le `if` qui n'est pas prédictible. On peut cependant éliminer le `if` en écrivant la fonction comme suit :

```
u32 pop_count_8(u8 n) {
    u32 count_bits = 0;

    while (n) {
        count_bits += (n & 1);
        n = n >> 1;
    }
    return count_bits;
}
```

Version de référence, élimination du if

La version de référence améliorée en supprimant le `if` s'exécute en $48,78$ secondes ce qui constitue une faible mais notable amélioration.

12.2 Améliorations simples

Malheureusement la fonction de référence n'est pas très efficace et on peut l'améliorer en utilisant trois techniques pour compter le nombre de bits à 1 dans un octet :

- en utilisant une table de conversion,
- en comptant les bits par paires, quartets, octets,
- en utilisant l'instruction assembleur `popcnt`.

12.2.1 Table de conversion

On peut utiliser une table de 256 octets, chaque octet contenant le nombre de bits de la valeur correspondant à l'indice du tableau. Ainsi, la valeur pour l'indice du tableau égal à 189 est 6 car $189_{10} = 1011\_1101_2$, soit 6 bits à 1 :

```
u8 bits_table[256] = { 0, 1, 1, 2, 1, 2, 2, 3, ..., 8 };

u32 pop_count_8(u8 n) {
    return bits_table[n]
}
```

Cette version est relativement courte mais pour qu'elle soit efficace il faut que la table `bits_table` tienne en mémoire cache L1.

Amélioration table de conversion

En utilisant une table de conversion (résultats non présentés par la suite), on ne met plus que $2,76$ secondes, on va donc environ 20 fois plus vite.

12.2.2 Compter les bits

On désire redéfinir une fonction `pop_count_8` qui compte le nombre de bits à 1 dans un octet. La première étape consiste à compter le nombre de bits à 1 dans une paire de bits. On a alors quatre cas possibles :

- `11` : 2 bits
- `10` : 1 bit
- `01` : 1 bit
- `00` : 0 bit

Cela est relativement simple à réaliser. Considérons une valeur a sur 8 bits. Il nous suffit de calculer les expressions suivantes :

```
b0 = (a & 0x55);
b1 = (a >> 1) & 0x55;
c = b0 + b1;
```

En fait la valeur 55_{16} représente un masque de sélection qui ne prend en compte que le bit de poids faible de chaque paire : $55_{16} = 01010101_2$. On sélectionne les bits de poids faible dans **`b0`** et les bits de poids fort que l'on a décalé vers la droite dans **`b1`**. On additionne ensuite les deux valeurs **`b0`** et **`b1`**.

Voyons ce que cela donne sur un exemple (voir Figure 12.1) pour la valeur $a = 87_{16} = 1000\_0111_2$:

- $b0 = 0000\_0101_2$
- $b1 = 0100\_0001_2$
- $c = 0100\_0110_2$

On obtient bien le résultat escompté.

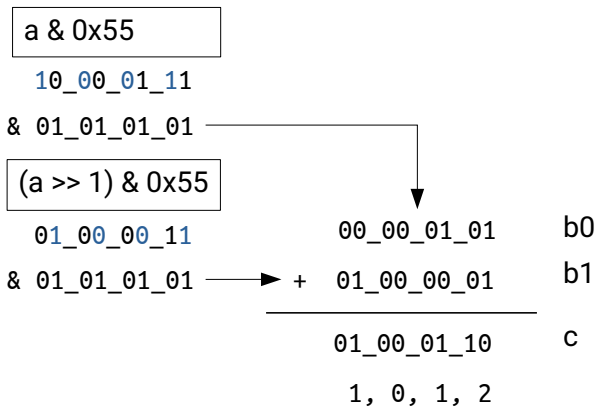


FIGURE 12.1 – Masques appliqués à la valeur $a = 87_{16}$

On notera cependant que l'expression n'est pas factorisable :

$$(a\ and\ 55_{16}) + ((a >> 1)\ and\ 55_{16}) \neq ((a + (a >> 1))\ and\ 55_{16}$$

On réitère ensuite le processus pour s'intéresser aux quartets, octets puis aux mots. Le masque évolue comme présenté Table 12.1 ainsi que le décalage appliqué.

# bits	Décalage	Masque Binaire	Masque Hexa
paire	1	$0101\_0101\_0101\_0101_2$	5555_{16}
quartet	2	$0011\_0011\_0011\_0011_2$	3333_{16}
octet	4	$0000\_1111\_0000\_1111_2$	$0F0F_{16}$
mot	8	$0000\_0000\_1111\_1111_2$	$00FF_{16}$

TABLE 12.1 – Masques en fonction du nombre de bits

Le code de la fonction `pop_count_8` est alors :

```
1 const u32 m1  = 0x55555555;
2 const u32 m2  = 0x33333333;
3 const u32 m4  = 0x0f0f0f0f;
4
5 u32 pop_count_8(u8 n) {
6     u8 x;
```

```
    x = (x & m1 ) + ((x >>  1) & m1 ); // compte les paires de bits
    x = (x & m2 ) + ((x >>  2) & m2 ); // compte les quartets
    x = (x & m4 ) + ((x >>  4) & m4 ); // compte les octets

    return x;
}
```

Si nous reprenons notre exemple avec la valeur 87_{16}, on obtient successivement :

- pour la première étape `x = 01_00_01_10` en base 2
- pour la deuxième étape `x = 00_01_00_11`
- pour la troisième étape `x = 00_00_01_00`, soit la valeur 4 en décimal ce qui signifie que initialement $87_{16} = 135_{10}$ possède 4 bits à 1

Amélioration en comptant les bits par paires, quartets, etc

En utilisant des décalages et additions, le temps d'exécution est de $2,83$ secondes, on va donc environ 19 fois plus vite.

Aussi étrange que cela puisse paraître, le calcul du premier terme :

```
// version 1
x = (x & m1 ) + ((x >>  1) & m1 );
```

peut être remplacé par le code suivant :

```
// version 2
x = x - ((x >>  1) & m1 );
```

En fait, cela est tout à fait naturel puisque d'après le tableau suivant on a :

paire de bits	00	01	10	11
décalage à droite	00	00	01	01
résultat soustraction	00	01	01	10

En conséquence, le code de la deuxième expression se traduit par 5 instructions assembleur alors que le premier en utilise 6 puisque l'on ne réalise le *et binaire* avec `m1` qu'une seule fois (et non deux fois dans la première version).

```
; x = x - ((x >>  1) & m1 );
    mov eax, [x]
    mov edx, eax
    shr edx
    and edx, 1431655765 ; 0x55555555
    sub eax, edx
```

Le code de la version 2 sera donc normalement plus efficace.

12.2.3 Utilisation de l'instruction popcnt

L'instruction `popcnt` que nous avons déjà évoquée plusieurs fois au cours des chapitres précédents a pour but de compter le nombre de bits à 1 dans un registre. Elle est de la forme :

popcnt r(16/32/64), r/m(16,32,64)

En d'autres termes, elle accepte en opérande destination un registre et en opérande source un registre ou une adresse mémoire. Dans notre code il suffit de remplacer la fonction `pop_count_8` par cette instruction.

Amélioration utilisation de l'instruction assembleur popcnt

En utilisant l'instruction popcnt combiné au dépliage de boucle, le temps d'exécution est de $3,09$ secondes, on va donc environ 18 fois plus vite. Cela est moins efficace que la méthode précédente.

12.3 Traitements par 32 bits

Plutôt que de traiter le tableau `tab_process_bits` octet par octet, on peut le traiter en considérant qu'il s'agit d'un tableau d'entiers non signés de 32 bits. Cela revient à faire une sorte de dépliage par 4.

Il suffit alors de modifier les fonctions en conséquence, par exemple, pour les fonctions liées à la table de conversion, on obtient :

```
1  u32 u32_popcnt_table_32_v1(u8 *x, u32 size) {
2      u32 count = 0, i;
3      // convertir x en un tableau d'entiers 32 bits
4      u32 *y = (u32 *) x;
5
6      // compter par groupe de 4 octets (dépliage par 4)
7      for (i = 0; i < (size & ~3); i+=4) {
8          count += popcnt_table_u32(*y++);
9      }
10
11     // compter les derniers octets restants
12     while (i < size) {
13         count += popcnt_table[ x[i] ];
14         ++i;
15     }
16
17     return count;
18 }
```

Cependant, la fonction `popcnt_table_32` peut être écrite au moins de deux manières différentes dont l'une est plus efficace que l'autre.

Voici la version qui est la moins efficace car elle utilise une boucle :

```
u32 popcnt_table_u32(u32 x) {
    u32 total = 0;

    do {
        total += (u32) popcnt_table [x & 0xFF];
    } while ((x >>= 8)!=0);

    return total;
}
```

Et la version la plus efficace, car dépliée :

```
u32 popcnt_table_u32(u32 x) {
    u32 total = popcnt_table[ x & 0xFF ];
    x >>= 8;
    total +=  popcnt_table[x & 0xFF];
    x >>= 8;
    total +=  popcnt_table[x & 0xFF];
    x >>= 8;
    total +=  popcnt_table[x & 0xFF];

    return total;
}
```

Il en résulte des temps d'exécution bien plus intéressants comme indiqué ci-après :

Amélioration traitement par 32 bits

- fonction de référence : $36,07$ s
- fonction de référence optimisée sans if : $23,34$ s
- fonction avec table de conversion : $4,69$ s
- fonction avec table de conversion dépliée : $2,25$ s
- fonction avec utilisation de `popcnt` : 0.62 s
- fonction avec utilisation de `popcnt`, dépliée par 2 : 0.53 s
- fonction avec utilisation de `popcnt`, dépliée par 4 : 0.47 s

On note que la version qui utilise l'instruction assembleur `popcnt` est plus efficace qu'en 8 bits : on passe de $3,09$ s avec un traitement par 8 bits à $0,62$ s en traitant 32 bits.

12.4 Vectorisation SSE et AVX

La vectorisation avec SSE permet de traiter les données par groupe de 16 octets. Malheureusement, il n'existe pas d'instruction vectorielle qui s'applique sur un registre SSE et qui réalise le décompte des bits. La seule solution qui s'offre à nous, a priori, est de charger les données par groupe de 16 octets puis d'extraire chaque double mot, et enfin d'en compter le nombre de bits avec `popcnt`. Voici un aperçu de la boucle principale de ce traitement :

```
.for_u16:
    movdqa  xmm0, [esi + ecx]   ; charge x[i] à x[i+15] dans xmm0
    pshufd  xmm1, xmm0, 0x01    ; extrait le 2nd   mot dans xmm1
    pshufd  xmm2, xmm0, 0x02    ; extrait le 3ieme mot dans xmm2
    pshufd  xmm3, xmm0, 0x03    ; extrait le 4ieme mot dans xmm3

    movd    edi, xmm0           ; compte les bits
    popcnt  edi, edi            ; de la partie basse de xmm0
    add     eax, edi

    movd    ebx, xmm1           ; compte les bits
    popcnt  ebx, ebx            ; de la partie basse de xmm1
    add     eax, ebx

    movd    edi, xmm2           ; compte les bits
    popcnt  edi, edi            ; de la partie basse de xmm2
    add     eax, edi

    movd    ebx, xmm3           ; compte les bits
    popcnt  ebx, ebx            ; de la partie basse de xmm3
    add     eax, ebx

    add     ecx, 16
    cmp     ecx, edx
    jne     .for_u16
```

Une seconde version consiste à utiliser le même principe que lorsque nous avons compté les bits dans un registre général (voir Section 12.2.2). Voici le code AVX mais qui s'applique sur les registres SSE. On commence par charger les masques dans les registres `xmm4` à `xmm7`

```
    mov     eax, 0x55555555
    movd    xmm4, eax
    vpshufd xmm4, xmm4, 0

    mov     eax, 0x33333333
    movd    xmm5, eax
    vpshufd xmm5, xmm5, 0

    mov     eax, 0x0f0f0f0f
    movd    xmm6, eax
```

```
11      vpshufd xmm6, xmm6, 0
12
13      mov     eax, 0x00ff00ff
14      movd    xmm7, eax
15      vpshufd xmm7, xmm7, 0
```

Puis dans la boucle principale, on réalise les décalages de bits et on additionne le résultat à `xmm3` qui fait office de somme. Le registre `xmm3` contiendra au final deux valeurs 64 bits mais on ne prendra en compte que les 32 bits de chaque *quad word* pour faire la somme du nombre de bits à 1 (lignes 34 à 37) :

```
1   for_ur16:
2       movdqa  xmm0, [esi + ecx]   ; load x[i] to x[i+15] into xmm0
3
4       ; x = x - (x >> 1) & 0x5555... 5 (version 2)
5       vpsrlw  xmm1, xmm0, 1
6       vpand   xmm1, xmm4
7       vpsubw  xmm0, xmm1
8
9       ; x = (x & m2) + ((x >> 2) & m2);
10      vpand   xmm1, xmm0, xmm5
11      vpsrlw  xmm2, xmm0, 2
12      vpand   xmm2, xmm5
13      vpaddw  xmm0, xmm1, xmm2
14
15      ; x = (x + (x >> 4)) & m4;
16      vpsrlw  xmm1, xmm0, 4
17      vpaddw  xmm0, xmm1
18      vpand   xmm0, xmm6
19
20      ; x += x >> 8
21      vpsrlw  xmm1, xmm0, 8
22      vpaddw  xmm0, xmm1
23      vpand   xmm0, xmm7
24
25      vpxor   xmm2, xmm2
26      vpsadbw xmm1, xmm0, xmm2
27      vpaddq  xmm2, xmm1, xmm3
28      vmovdqu xmm3, xmm2
29
30      add     ecx, 16
31      dec     edx
32      jnz     .for_ur16
33
34      vmovd   ebx, xmm3
35      vpunpckhqdq xmm1, xmm3, xmm3
36      vmovd   eax, xmm1
37      add     eax, ebx
```

L'instruction `vpsrlw` (*Shift Packed Data Right Logical*) réalise un décalage à droite dans chacun des mots de `xmm1` par la quantité donnée en troisième opérande.

Les instructions `vpsubw` (*Subtract Packed Integers*) et `vpaddw` (*Add Packed Integers*) réalisent respectivement la soustraction et l'addition des 8 mots de chaque registre SSE qui leur sont passé en paramètres.

Enfin l'instruction en ligne 26, `vpsadbw` (*Compute Sum of Absolute Differences*), calcule la somme des valeurs absolues des différences entre opérande destination et opérande source pour chaque mot du registre SSE. Le mot en partie basse reçoit cette somme, les autres mots sont mis à 0.

On notera, ligne 15, que l'on calcule :

```
1  x = (x + (x >> 4)) & m4;
```

alors que nous avons indiqué Section 12.2.2 que ce n'était pas équivalent à :

```
1  x = (x & m4)  + ((x >> 4) & m4);
```

Cependant, dans le cas de la vectorisation on travaille sur des mots (16 bits) et les bits les plus à droite qui sont décalés sont éliminés de chaque mot, ils ne sont pas répercutés sur le mot suivant, ce qui permet de réaliser la simplification.

12.5 Implantations

Pour cette étude de cas nous avons vu les principales technniques pour compter le nombre de bits à 1 d'un tableau d'octets. On peut compter octet par octet ou alors tenter de prendre en considération des quantités plus grandes et compter par groupe de 4 octets (*double word*). On peut également en architecture 64 bits compter par groupe de 8 octets (*quad word*).

Dans les sources de l'étude de cas, j'ai réalisé 18 implantations différentes afin de trouver les variantes éventuelles qui seraient les plus performantes possibles.

12.6 Résultats

Plusieurs solutions ont été implantées parmi lesquelles :

- `u8_reference` : fonction de référence qui travaille par octet
- `u8_reference_opt` :fonction de référence qui travaille par octet optimisée en supprimant le `if`
- `u32_reference` : fonction de référence qui travaille par double mot
- `u32_reference_opt` :fonction de référence qui travaille par double mot optimisée en supprimant le `if`
- `u8_shift_v1` : fonction avec décalage pour le calcul par octet

- `u8_shift_v2` : amélioration de la fonction précédente
- `u32_shift_v1` : fonction avec décalage pour le calcul par double mot
- `u32_shift_v2` : amélioration de la fonction précédente
- `u8_asm` : fonction assembleur qui fait appel à l'instruction `popcnt` et qui travaille par octet
- `u32_asm` : fonction assembleur qui fait appel à l'instruction `popcnt` et qui travaille par double mot
- `u32_asm_ur4` : dépliage par 4 de la fonction précédente
- `u32_sse_v1` : version SSE avec utilisation de `popcnt` sur chaque double mot contenu dans le registre
- `u32_sse_v2` : version SSE avec décalages
- `u32_avx2_v1` : version AVX avec décalages mais qui travaille sur les registres SSE
- `u32_intrinsics` : version intrinsics qui est la traduction de la méthode `u32_sse_v2`

Le test de performance consiste à réaliser 30_000 appels aux fonctions sur des vecteurs de 262_207 octets.

12.6.1 Architectures anciennes (avant 2015)

Les résultats pour les architectures anciennes sont présentés Table 12.2.

On notera que la fonction de référence prend énormément de temps par rapport à sa version SSE (méthode 12) ou la version avec utilisation de l'instruction `popcnt` (méthodes 9, 10 et 11). Le fait de traiter les données par double mot (32 bits) et d'optimiser le `if` apporte un gain non négligeable (méthodes 2, 3 et 4).

Sur les processeurs ne disposant pas de l'instruction `popcnt`, l'amélioration est faible (facteur 33 pour le Pentium D et 75 pour le Q9300) comparativement aux autres processeurs pour lesquels le facteur d'amélioration est supérieur à 95.

La version par décalage (*shift*, méthode 8) en 32 bits donne des temps d'exécution très intéressants en fonction de l'augmentation de l'année de production des processeurs.

Mais c'est au final la version intrinsics qui est la plus optimisée et qui donne les meilleurs résultats sauf pour l'Intel i7 860, l'Intel i7 4790 ou l'AMD 1090 T.

12.6.2 Architectures modernes (2015 à 2019)

Pour les architectures modernes (Table 12.3), on observe les mêmes tendances. Cependant, les méthodes 8 et 11 donnent les meilleurs résultats et sont un peu plus

n°	Méthode	Intel Pentium D 925 2006	Intel Core 2 Q9300 2008	Intel i7 860 2009	AMD X6 1090T 2010	Intel i5 3570K 2012	Intel i7 4790 2014
1	u8_reference	150.22	170.20	110.19	80.21	72.58	54.99
2	u8_reference_opt	111.73	161.63	102.39	64.68	65.12	47.38
3	u32_reference	126.41	105.45	60.91	43.46	53.34	39.42
4	u32_reference_opt	73.51	88.36	60.81	43.31	48.68	33.40
5	u8_shift_v1	16.50	7.77	4.52	4.31	3.23	1.98
6	u8_shift_v2	16.45	7.66	4.76	4.62	3.20	2.02
7	u32_shift_v1	8.01	4.25	2.05	2.50	1.29	0.56
8	u32_shift_v2	7.76	3.13	1.82	1.86	1.06	**0.49**
9	u8_asm	-	-	6.88	10.94	4.65	2.96
10	u32_asm	-	-	1.27	1.65	1.16	0.71
11	u32_asm_ur4	-	-	**0.89**	**0.84**	0.88	**0.49**
12	u32_sse_v1	-	-	0.96	1.52	0.92	0.74
13	u32_sse_v2	-	-	-	-	1.06	0.81
14	u32_avx2_v1	-	-	-	-	-	0.71
15	u8_intrinsics	**4.55**	**2.26**	0.92	1.35	**0.76**	0.61
	ratio 1 / (11 ou15)	33.01	75.30	123.80	95.48	95.50	112.22

TABLE 12.2 – Architectures anciennes : temps d'exécution en secondes sur 30_000 exécutions de la fonction popcnt sur des tableaux de 262_207 octets

performantes que la version intrinsics. Traiter les données sous format 32 bits est donc bénéfique dans ce cas.

La méthode 8 avec décalage de bits est souvent la plus performante, talonnée par la méthode 11 qui réalise un dépliage par 4 de la boucle.

On peut alors se demander si le passage au 64 bits améliorera encore les performances ? La réponse est oui à en croire les tests effectués en traitant les données par groupe de 32 ou de 64 bits sous une architecture 64 bits avec un AMD Ryzen 5 3600 :

- architecture 32 bits, traitement par 32 bits : $0,43$ s
- architecture 32 bits, traitement par 32 bits et dépliage par 4 de la boucle : $0,31$ s
- architecture 64 bits,traitement par 64 bits : $0,21$ s
- architecture 64 bits,traitement par 64 bits et dépliage par 4 de la boucle : $0,15$ s

N°	Marque Gamme Modèle	Intel Core i3 6100 2015	AMD Ryzen 7 1700X 2017	Intel Core i5 7400 2017	Intel Core i7 8700 2017	AMD Ryzen 5 3600 2019	Intel Xeon 4208 2019
1	`u8_reference`	57.99	59.13	64.81	47.28	54.96	71.73
2	`u8_reference_opt`	53.14	54.11	58.58	43.03	48.78	74.70
3	`u32_reference`	36.39	44.72	39.89	29.53	35.36	54.49
4	`u32_reference_opt`	32.65	28.77	35.83	26.47	23.67	46.18
5	`u8_shift_v1`	2.17	2.51	2.32	1.75	2.83	2.49
6	`u8_shift_v2`	2.11	2.51	2.23	1.70	2.84	2.50
7	`u32_shift_v1`	0.61	1.00	0.64	0.49	0.44	0.72
8	`u32_shift_v2`	**0.50**	0.80	**0.54**	**0.41**	**0.35**	**0.62**
9	`u8_asm`	6.39	3.24	6.83	5.20	3.06	7.53
10	`u32_asm`	1.07	0.68	1.14	0.86	0.62	0.91
11	`u32_asm_ur4`	0.54	**0.51**	0.59	0.45	0.47	0.74
12	`u32_sse_v1`	0.90	0.90	0.96	0.72	0.83	1.05
13	`u32_sse_v2`	0.87	0.74	0.92	0.70	0.64	1.05
14	`u32_avx2_v1`	0.87	0.83	0.95	0.70	0.69	1.02
15	`u8_intrinsics`	0.64	0.64	0.68	0.52	0.56	0.76
	ratio 1 / 11	107.38	115.94	109.84	N/A	116.93	96.93

TABLE 12.3 – Architectures récentes : temps d'exécution en secondes sur 30_000 exécutions de la fonction popcnt sur des tableaux de 262_207 octets

On divise donc le temps d'exécution par deux en passant au 64 bits et en traitant les données par des registres 64 bits.

12.6.3 Architectures récentes (2020 et après)

Pour les architectures récentes (Table 12.4), on observe encore les mêmes tendances que précédemment. La méthode 8 est la plus efficace.

12.7 Conclusion

Ce problème révèle deux choses importantes. La première est que **le traitement des données par groupe de 32 bits** (voire de 64 bits) au lieu de 8 bits **permet de gagner en efficacité**, cela semble normal puisqu'on traite les données en une seule fois plutôt qu'en 4 fois (ou 8 fois). Le seconde leçon que l'on peut tirer montre que **la vectorisation va se révéler complexe** car on ne dispose pas d'instruction

N°	Marque Gamme Modèle	Intel Core i5 10850H 2020	AMD Ryzen 5 5600g 2021	Intel Core i5 12400F 2022
1	`u8_reference`	43.82	46.19	51.75
2	`u8_reference_opt`	40.06	42.72	49.10
3	`u32_reference`	27.42	25.91	35.64
4	`u32_reference_opt`	24.50	21.25	28.58
5	`u8_shift_v1`	1.65	2.28	0.73
6	`u8_shift_v2`	1.61	2.29	0.73
7	`u32_shift_v1`	0.48	0.37	0.31
8	`u32_shift_v2`	**0.39**	**0.30**	**0.25**
9	`u8_asm`	2.45	3.60	1.79
10	`u32_asm`	0.80	0.45	0.50
11	`u32_asm_ur4`	0.41	0.44	0.44
12	`u32_sse_v1`	0.67	0.79	0.65
13	`u32_sse_v2`	0.66	0.53	0.63
14	`u32_avx2_v1`	0.65	0.73	0.69
15	`u8_intrinsics`	0.49	0.47	0.44
	ratio 1 / 11	106.87	104.97	117.61

TABLE 12.4 – Architectures actuelles : temps d'exécution en secondes sur 30_000 exécutions de la fonction popcnt sur des tableaux de 262_207 octets

vectorielle qui réaliserait ce que fait l'instruction `popcnt` sur les registres généraux. L'introduction d'une telle instruction permetrait probablement de gagner encore en efficacité.

Chapitre 13

Etude de cas Variante de SAXPY

13.1 Introduction

Ce chapitre traite de l'implantation de la fonction *saxpy* et permet de mettre en pratique ce qui a été vu au Chapitre 7 concernant le coprocesseur arithmétique et au Chapitre 8 pour le calcul vectoriel avec unités SSE et AVX.

Pour rappel, la fonction saxpy (*Single-Precision* $A \times X$ *Plus* Y) consiste, étant donné deux vecteurs de n réels appelés X et Y, à calculer :

$$Y = a \times X + Y$$
$$y_i = a \times x_i + y_i,\ \forall i \in [1..n]$$

où a est une constante réelle. Afin de complexifier la traduction pour le compilateur, nous avons décidé de modifier quelque peu la fonction en lui faisant retourner la somme des y_i :

$$sum = \sum_{i=1}^{n} y_i$$

Nous allons nous intéresser à des vecteurs de taille quelconque et programmer en architecture 32 bits. Les types C que nous allons utiliser sont :

- pour les entiers non signés `typedef uint32_t u32;`
- pour les nombres flottants en simple précision `typedef float f32;`

13.2 Fonction de référence

La fonction à implanter est donnée Listing 13.2.1. Elle comporte quatre paramètres qui sont les vecteurs x et y, la constante a ainsi que la taille des vecteurs ($size$).

```
1 f32 saxpy_c(f32 *x, f32 *y, f32 a, u32 size) {
2     f32 sum = 0.0;
3     for (u32 i = 0; i < size; ++i) {
4         y[i] = a * x[i] + y[i];
5         sum += y[i];
6     }
7     return sum;
8 }
```

Listing 13.2.1 – SAXPY modifiée - fonction de référence

13.3 Version FPU

La première implantation en assembleur que nous allons réaliser (voir Listing 13.3.1) est la traduction de la fonction de référence en utilisant le coprocesseur arithmétique. Etant donné qu'il s'agit d'une fonction qui retourne un `float` en architecture 32 bits, c'est le premier registre du coprocesseur qui contient le résultat de la fonction, c'est à dire `st0`. Nous avons choisi de réaliser l'association variables / registres de la Table 13.1.

Cste/Param/Var	Type	Paramètre	Registre	Description
`x`	`f32 []`	`[ebp+8]`	`esi`	adresse du vecteur x
`y`	`f32 []`	`[ebp+12]`	`edi`	adresse du vecteur y
`a`	`f32`	`[ebp+16]`	pile	constante a
`size`	`u32`	`[ebp+20]`	`edx`	taille des vecteurs
`i`	`u32`		`ecx`	variable de boucle
`sum`	`f32`		`st0`	somme

TABLE 13.1 – Association entre variables et registres pour la fonction de référence de la variante de SAXPY

Les registres `esi` et `edi` devront être sauvegardés car ils ne doivent pas être modifiés pour le sous-programme appelant d'après les conventions d'appel du C en 32 bits.

La traduction est assez simple, elle est présentée Listing 13.3.1. En ligne 4, on initialise `sum` à 0 grâce à l'instruction `fldz`. Comme indiqué précédemment, `sum` sera en `st0`, puis décalé en `st1` lors du calcul $a \times x_i + y_i$. En ligne 5, on vérifie que la taille des vecteurs n'est pas nulle, auquel cas il faut sortir du sous-programme. On sauvegarde ensuite les registres `esi` et `edi` (lignes 7 et 8), puis on charge les paramètres du sous-programme dans les registres appropriés (lignes 9 à 11).

On débute la boucle `for` en ligne 13, puis lignes 17 à 22, on exécute le calcul du corps de la boucle `for`. On peut voir Table 13.2 comment sont utilisés les registres du coprocesseur lors du calcul.

```
1  saxpy_fpu:
2      push    ebp
3      mov     ebp, esp
4      fldz                            ; sum = 0
5      cmp     dword [ebp + 20], 0     ; si size == 0 alors retourne 0
6      jz      .end
7      push    esi                     ; sauve les registres
8      push    edi                     ;
9      mov     esi, [ebp + 8]          ; charge les paramètres
10     mov     edi, [ebp + 12]         ;
11     mov     edx, [ebp + 20]         ;
12
13     xor     ecx, ecx                ; i = 0
14 .for:
15     cmp     ecx, edx                ; fin de boucle si i >= size
16     jge     .endfor
17     fld     dword [esi + ecx * 4]   ; x[i]
18     fmul    dword [ebp + 16]        ; a*x[i]
19     fadd    dword [edi + ecx * 4]   ; a*x[i]+y[i]
20     fst     dword [edi + ecx * 4]   ; y[i] = a*x[i]+y[i]
21     faddp   st1, st0                ; sum += y[i]
22     inc     ecx                     ; ++i
23     jmp     .for
24 .endfor:
25     pop     edi                     ; restaure les registres
26     pop     esi                     ;
27 .end:
28     mov     esp, ebp
29     pop     ebp
30     ret
```

Listing 13.3.1 – SAXPY modifiée - implantation FPU

On commence par placer x_i au sommet de la pile du coprocesseur (ligne 17). La variable `sum` initialement dans `st0` est alors déplacé en `st1`. On multiplie ensuite x_i par la constante a (ligne 18), puis on ajoute y_i (ligne 19). Finalement on stocke le résultat dabs `y[i]` et on ajoute ce résultat à `st1` qui contient `sum` et on dépile `st0`.

Afin de donner un ordre d'idée du temps d'exécution pour les différentes implantations que nous allons réaliser, nous reportons par la suite, les résultats obtenus sur un ordinateur doté d'un Core i5 7400. Le test effectué consiste à calculer 50_000 fois la fonction de référence appliquée sur des vecteurs initialisés aléatoirement de 524_287 éléments.

Temps de référence

Pour l'implantation que nous venons de donner, l'exécution dure environ $22,95$ secondes.

ligne	instruction	st0	st1
16	`jge .endfor`	sum	?
17	`fld dword [esi + ecx * 4]`	x_i	sum
18	`fmul dword [ebp + 16]`	$a \times x_i$	sum
19	`fadd dword [edi + ecx * 4]`	$a \times x_i + y_i$	sum
20	`fst dword [edi + ecx * 4]`	$a \times x_i + y_i$	sum
21	`faddp st1, st0`	$sum + (a \times x_i + y_i)$	?

TABLE 13.2 – Calculs du coprocesseur

13.4 Version FPU dépliée par 4

Une fois que l'on dispose de la version FPU, on peut la modifier afin d'introduire une amélioration liée à la boucle de calcul. On a vu précédemment (cf. Section 5.4.11.1) qu'il peut être intéressant de déplier les boucles. On va donc réaliser un dépliage par 4 du corps de la boucle. Le code devra ressembler à celui du Listing 13.4.1. Nous avons fait usage d'une macro instruction du langage C afin rendre le code plus lisible.

```
#define SAXPY_BODY(i) \
    y[i] = a * x[i] + y[i]; \
    sum += y[i];

f32 saxpy_c(f32 *x, f32 *y, f32 a, u32 size) {
    f32 sum = 0.0;
    u32 i;
    // dépliage par 4
    for (i = 0; i < size; i += 4) {
        SAXPY_BODY(i);
        SAXPY_BODY(i+1);
        SAXPY_BODY(i+2);
        SAXPY_BODY(i+3);
    }
    // dernières itérations
    while (i < size) {
        SAXPY_BODY(i);
        ++i;
    }
    return sum;
}
```

Listing 13.4.1 – SAXPY modifiée - fonction de référence dépliée par 4

Nous avons, par souci de clareté et pour ne pas produire de listing assembleur trop volumineux, supprimé les parties qui sont identiques à la version précédente.

```
1 %macro fpu_body 1
2     fld     dword [esi + ecx * 4 + %1] ; x[i]
3     fmul    dword [ebp + 16]           ; a*x[i]
4     fadd    dword [edi + ecx * 4 + %1] ; a*x[i]+y[i]
5     fst     dword [edi + ecx * 4 + %1]
6     faddp   st1, st0                   ; sum += y[i]
7 %endmacro
```

Listing 13.4.2 – Macro instruction nasm

Tout comme en C, afin de simplifier l'écriture de la fonction dépliée, on utilise une macro instruction `nasm` afin de ne pas réécrire entièrement les 5 lignes de code qui constituent le corps de la boucle `for` (cf. Listing 13.4.2).

Cette macro instruction comprend un paramètre figuré par le chiffre 1 en fin de ligne 1 du listing. Il va correspondre à un décalage de l'adresse qui correspond à `x[i + k]`, où `k` varie entre 0 et 3, ce qui, en assembleur, nous contraint à utiliser `4 * k` car on manipule des `float` qui occupent 4 octets en mémoire.

On réutilise cette macro instruction au niveau du listing 13.4.3. On fait appel à une fonctionnalité de `nasm` lignes 8 à 12 qui consiste à écrire une boucle qui génère au final les quatre lignes suivantes :

```
1 fpu_body  0
2 fpu_body  4
3 fpu_body  8
4 fpu_body 12
```

En ligne 8 du Listing 13.4.3, ci-après, on définit une variable `k` que l'on initialise à 0. En ligne 9, on répète 4 fois l'ensemble des lignes 10 et 11. On génère la macro instruction `fpu_body` avec `k` comme paramètre, puis on augmente `k` de 4 pour passer au réel simple précision suivant.

Amélioration dépliage par 4

En dépliant la fonction de référence par 4, on n'obtient aucune amélioration puisque l'exécution dure 22, 97 secondes.

13.5 Version SSE

Après avoir déplié le corps de la boucle `for` par 4, il est relativement aisé d'écrire la version utilisant les instructions SSE pour obtenir un code vectorisé. On doit réaliser les calculs en parallèle dans les registres SSE dans la partie dépliée par 4 de la boucle.

On a choisi les conventions suivantes :

```
1  saxpy_fpu_u4:
2      ...
3      xor     ecx, ecx
4      and     edx, ~3                 ; edx multiple de 4
5  .for_u4:
6      cmp     ecx, edx
7      jge     .endfor_u4
8      %assign k 0                     ; équivalent à
9      %rep 4                          ; fpu_body 0
10        fpu_body k                   ; fpu_body 4
11        %assign k k+4                ; fpu_body 8
12     %endrep                         ; fpu_body 12
13     add     ecx, 4
14     jmp     .for_u4
15 .endfor_u4:
16                                     ; dernières itérations
17                                     ; ecx = (size / 4) * 4
18                                     ; recharger edx avec size
19     mov     edx, [ebp + 20]
20 .for:
21     cmp     ecx, edx
22     jge     .endfor
23     fpu_body  0
24     inc     ecx
25     jmp     .for
26 .endfor:
27     ...
28     ret
```

Listing 13.4.3 – SAXPY modifiée - implantation partielle avec FPU et dépliage par 4

- **xmm0** contient les sommes partielles
- **xmm1** contient x_i, x_{i+1}, x_{i+2}, x_{i+3}
- **xmm2** contient y_i, y_{i+1}, y_{i+2}, y_{i+3}
- **xmm3** contient quatre fois la valeur de la constante a.

Le calcul est alors simple à réaliser, il suffit de multiplier **xmm1** par **xmm3** puis d'additionner ce résultat à **xmm2**. Une fois ce calcul terminé on peut ajouter **xmm2** à **xmm0**.

En sortie de la boucle dépliée, le registre **xmm0** contient :

- **xmm0.ps[0]** $= (a \times x_0 + y_0) + (a \times x_4 + y_4) + \ldots$
- **xmm0.ps[1]** $= (a \times x_1 + y_1) + (a \times x_5 + y_5) + \ldots$
- **xmm0.ps[2]** $= (a \times x_2 + y_2) + (a \times x_6 + y_6) + \ldots$
- **xmm0.ps[3]** $= (a \times x_3 + y_3) + (a \times x_7 + y_7) + \ldots$

Le code de la version SSE est donné Listing 13.5.1. On commence par mettre **xmm0** à 0 (ligne 3), puis on charge quatre fois la constante **a** dans **xmm3** (lignes 4

```
1  saxpy_sse:
2      ...
3      xorps   xmm0, xmm0
4      movss   xmm3, [ebp + 16]        ; xmm3 = [a, a, a, a]
5      shufps  xmm3, xmm3, 0           ;
6      xor     ecx, ecx                ; i = 0
7      and     edx, ~3                 ; dépliage par 4
8  .for_u4:
9      cmp     ecx, edx
10     jge     .endfor_u4
11     movaps  xmm1, [esi + ecx * 4]   ; xmm1 = x[i+3]:x[i]
12     movaps  xmm2, [edi + ecx * 4]   ; xmm2 = y[i+3]:y[i]
13     mulps   xmm1, xmm3
14     addps   xmm2, xmm1
15     addps   xmm0, xmm2
16     movaps  [edi + ecx * 4], xmm2   ; stocke résultat
17     add     ecx, 4
18     jmp     .for_u4
19 .endfor_u4:
20     haddps  xmm0, xmm0              ; somme des valeurs
21     haddps  xmm0, xmm0              ; du registre xmm0
22     sub     esp, 4                  ; et addition au
23     movss   [esp], xmm0             ; registre st0
24     fadd    dword [esp]             ;
25     add     esp, 4
26                                     ; dernières itérations
27     ...
28     ret
```

Listing 13.5.1 – SAXPY version SSE

et 5). Ligne 6, on initialise `i` (`ecx`) à 0, puis on calcule dans `edx` le plus proche multiple de 4 inférieur ou égal à size `size`. On aura bien évidemment chargé `size` au préalable dans le registre `edx`. Le corps de la boucle (lignes 9 à 18) ne comporte aucune difficulté.

En sortie de boucle, celle-ci effectuant les calculs en parallèle dans les registres SSE, on doit terminer les dernières itérations de la boucle dans le coprocesseur. On pourrait bien évidemment continuer les calculs avec les registres SSE grâce aux instructions de type `addss`, `mulss` mais en 32 bits on utilise par convention la FPU.

Il faut alors calculer la somme des quatre valeurs flottantes contenues dans le registre `xmm0` puis la stocker dans `st0`. L'instruction `haddps` permet de faire cela (cf. Section 8.2.2) et nous avons vu qu'il fallait l'exécuter deux fois (lignes 20 et 21).

Sachant que les données sont chargées dans le coprocesseur à partir d'un emplacement mémoire, on décide de réserver dans la pile un emplacement temporaire de 32 bits (ligne 22) et on y place la somme des valeurs du registres `xmm0` (ligne 23). Cette somme est alors ajoutée à `st0`, car comme pour la version non dépliée, on commence par réaliser `fldz` en début de sous-programme.

Amélioration SSE

En vectorisant le code on abaisse le temps d'exécution à $7,93$ secondes, on est donc $2,89$ fois plus rapide par rapport à la fonction de référence.

13.6 Version AVX

La version AVX impose de déplier la boucle par 8, puisqu'un registre AVX peut contenir 8 `float`. En conséquence, il faut déplier la boucle principale par 8. L'utilisation de l'AVX apporte une légère amélioration par rapport au SSE, voire dans certains cas, une forte amélioration, c'est le cas du AMD Ryzen 5 3600 pour lequel on divise par $1,8$ le temps d'exécution. On consultera la section des résultats ci-après pour vérifier cette affirmation.

La seule difficulté résiduelle concerne le calcul de la somme des 8 `float` de `ymm0`. Comme indiqué 8.3.3, il n'est pas possible de procéder comme avec le SSE. On peut voir comment on a procédé dans la version FMA (Listing 13.7.1) qui est très proche de la version AVX.

Amélioration AVX

En vectorisant le code avec l'AVX on abaisse le temps d'exécution à $5,26$ secondes, on est donc $4,36$ fois plus rapide par rapport à la fonction de référence. Le gain en dépliant la boucle par 2 ou 4 est infime, on obtient respectivement des temps d'exécution de $5,13$ et $5,09$ secondes. L'utilisation des *intrinsics* permet d'obtenir un temps d'exécution de $4,93$ secondes.

13.7 Version FMA

Le sigle FMA signifie *Fused Multiply–Add* et permet étant donné trois flottants a, b et c de calculer $d = round(a + b \times c)$, c'est à dire qu'on n'utilise qu'un seul calcul d'arrondi. Cette technique est supposée être plus rapide qu'une multiplication suivie d'une addition mais peu parfois conduire à des erreurs de précision.

Les microprocesseurs AMD et Intel implantent les instructions du jeu d'instructions FMA3 depuis 2012 et 2013 respectivement.

Nous utilisons ici l'instruction `\gls {vfmadd321ps} xmm1, xmm2, xmm3`[1] qui multiplie `xmm2` par `xmm3` et ajoute le résultat à `xmm1`.

Le code de la version FMA est donné Listing 13.7.1. On utilise les registres AVX comme `ymm0` pour contenir les sommes partielles. Lignes 23 à 27, on réalise le

1. qui est une variante de vfmadd231ps et donne le même résultat.

calcul du corps de la boucle. On charge `x[i:i+7]` dans `ymm1`, puis `y[i:i+7]` dans `ymm2`. On réalise ensuite le produit suivi de la somme en utilisant `vfmadd321ps`. On ajoute le résultat à `ymm0` et on le stocke en mémoire.

```
saxpy_fma:
    ...
    vxorps  ymm0, ymm0              ; ymm0 = [0.0 x 8 fois]
    movss   xmm3, [ebp + 16]        ; ymm3 = [a x 8 fois]
    shufps  xmm3, xmm3, 0
    vinsertf128 ymm3, xmm3, 1

    xor     ecx, ecx                ; i = 0
    and     edx, ~7                 ; edx multiple de 8 de size
.for_u8:
    cmp     ecx, edx                ; sortie de boucle si ecx >= edx
    jge     .endfor_u8
    vmovaps ymm1, [esi + ecx * 4]   ; ymm1 = x[i:i+7]
    vmovaps ymm2, [edi + ecx * 4]   ; ymm2 = y[i:i+7]
    vfmadd321ps ymm2, ymm1, ymm3    ;
    vaddps  ymm0, ymm2
    vmovaps [edi + ecx *4], ymm2    ; y[i:i+7] = ymm2
    add     ecx, 8                  ; i += 8
    jmp     .for_u8
.endfor_u8:
    vhaddps ymm0, ymm0              ; somme des valeurs
    vhaddps ymm0, ymm0              ; de ymm0
    vextractf128    xmm1, ymm0, 1
    vaddps  xmm0, xmm1
    sub     esp, 4                  ; stockage du résultat dans st0
    vmovss  [esp], xmm0
    fadd    dword [esp]
    add     esp, 4

    mov     edx, [ebp + 16]         ; dernières itérations
.for:
    cmp     ecx, edx
    jge     .endfor
    fld     dword [esi + ecx * 4]
    fmul    dword [edi + ecx * 4]
    faddp   st1, st0
    inc     ecx
    jmp     .for
.endfor:
    ...
```

Listing 13.7.1 – SAXPY version FMA

Amélioration FMA

L'amélioration FMA donne le meilleur temps d'exécution à $4,90$ secondes si on utilise les *intrinsics*.

13.8 Résultats

Plusieurs solutions ont été implantées parmi lesquelles :

- `C` : fonction de référence
- `asm fpu` : fonction de référence codée en assembleur en utilisant la FPU
- `asm fpu ur2` : version précédente avec dépliage par 4 de la boucle
- `asm sse` : version assembleur utilisant la vectorisation avec registres SSE
- `asm sse` : fonction avec décalage pour le calcul par octet
- `C sse intrin.` : version C utilisant les intrinsics sur des registres SSE
- `asm avx` : version assembleur utilisant la vectorisation avec registres AVX
- `asm avx2 ur2` : version AVX avec dépliage de la boucle par 2
- `asm avx2 ur4` : version AVX avec dépliage de la boucle par 4
- `C avx2 intrin.` : version C utilisant les intrinsics sur des registres AVX
- `C avx2 ez_ii` : version C utilisant les intrinsics sur des registres AVX avec l'interface ez_ii
- `asm fma` : version assembleur avec instructions FMA
- `C fma ez_ii` : version C avec *ez intrinsics interface* (voir ci-après)
- `C avx512 intrin.` : version intrisics avec instructions AVX512
- `C avx512 fma intrin` : version intrisics avec instructions AVX512 et FMA

13.8.1 Un mot sur l'interface ez_ii

L'interface ez_ii fait partie d'un projet plus vaste que j'ai initié il y a quelques années qui a pour but de **simplifier** l'utilisation de la STL, la bibliothèque standard du C++. Quelques autres projets en découlent comme ez_cuda qui se base sur des classes spécifiques afin de ne gérer qu'une seule instance d'un tableau dont l'allocation mémoire est réalisée à la fois sur le CPU et le GPU. On simplifie également la définition des paramètres des kernels.

Le terme **ez** vient de l'anglais et se lit *easy* qui signifie facile en français.

L'ensemble de ces bibliothèques et interfaces est intégré dans un projet plus vaste en cours de développement qui a pour but de mettre au point un langage dont la syntaxe est proche du langage Pascal et qui a pour objectif de simplifier l'écriture des programmes C++. Le code du *ez language* est traduit en C++ et il doit permettre à terme d'inclure de l'assembleur, de pouvoir intégrer des requêtes SQL ou des programmes en logique.

L'interface ez_ii, où le terme **ii** signifie *Intrinsics Interface* tente de simplifier l'utilisation des Intrinsics tout en procurant des fonctions d'affichage des registres SSE et AVX, utiles au débogage.

13.8.2 Architectures anciennes (avant 2015)

Les résultats pour les architectures de processeurs pré 2015 figurent Table 13.3.

N°	Marque Gamme Modèle	Intel Pentium D 925 2006	Intel Core 2 Q9300 2008	Intel Core i7 860 2009	AMD Phenom II 1090T 2010	Intel Core i5 3570k 2012	Intel Core i7 4790 2014
1	C	97.29	75.68	23.35	32.31	23.58	19.70
2	asm fpu	101.91	62.52	24.69	30.75	23.74	19.70
3	asm fpu ur4	91.77	58.92	25.36	30.47	23.64	19.68
4	asm sse	86.84	52.38	11.16	24.74	8.70	6.00
5	C sse intrin.	**84.11**	**51.89**	**10.12**	**24.27**	8.33	5.64
6	asm avx	-	-	-	-	8.21	5.42
7	asm avx2 ur2	-	-	-	-	8.42	5.42
8	asm avx2 ur4	-	-	-	-	**8.11**	5.48
9	C avx2 intrin.	-	-	-	-	-	5.43
10	C avx2 ez_ii	-	-	-	-	-	**5.41**
11	asm fma	-	-	-	-	-	5.43
12	C fma ez_ii	-	-	-	-	-	5.44
	ratio 1 / 5	1.15	1.45	2.30	1.33	2.83	3.49

TABLE 13.3 – Résultats comparatifs des méthodes SAXPY : architectures anciennes

Le gain apporté par une traduction directe de la fonction de référence en assembleur est faible par rapport à sa traduction par le compilateur. On note parfois une dégradation qui peut être corrigée si on déplie la boucle par 4 (sauf pour l'Intel i7 860). L'utilisation du SSE apporte un faible gain sur les architecture avant 2012. Sur l'Intel i5 3570k et l'Intel i7 4790, la technologie SSE permet de diviser respectivement par un facteur 2.83 et 3.49 le temps d'exécution par rapport à la méthode de référence. A noter également que l'Intel i7 860 possède un facteur d'amélioration de 2.3 en utilisant le SSE. Sur ces mêmes processeurs (Intel i5 3570k et i7 4790) passer à l'AVX ou au FMA (seulement pour l'i7 4790) n'apporte rien.

13.8.3 Architectures modernes (2015 à 2019)

Pour les architectures modernes (voir Table 13.4), l'utilisation de l'AVX par rapport au SSE apporte un gain substantiel. L'utilisation des instructions FMA n'apporte pas d'amélioration majeure dans le cas du traitement SAXPY modifié.

On note que les processeurs AMD ont une FPU peu performante, plus de 30 secondes pour les calculs des méthodes 1 à 3 alors que les autres processeurs

N°	Marque Gamme Modèle	Intel Core i3 6100 2015	AMD Ryzen 7 1700X 2017	Intel Core i5 7400 2017	Intel Core i7 8700 2017	AMD Ryzen 5 3600 2019	Intel Xeon 4208 2019
1	C	22.35	33.89	22.95	17.30	31.36	25.98
2	asm fpu	23.01	33.76	23.02	17.33	31.30	25.62
3	asm fpu ur4	22.96	33.84	22.97	17.28	31.33	25.93
4	asm sse	9.41	5.16	7.93	5.79	4.90	9.90
5	C sse intrin.	9.02	5.38	7.79	5.74	5.04	9.40
6	asm avx	7.56	3.53	5.26	3.75	2.62	9.83
7	asm avx2 ur2	7.49	3.49	5.13	3.72	2.58	10.08
8	asm avx2 ur4	7.46	3.43	5.09	3.72	**2.56**	9.82
9	C avx2 intrin.	**7.17**	3.29	4.93	3.67	2.61	9.88
10	C avx2 ez_ii	7.18	3.33	4.95	3.68	2.62	**9.13**
11	asm fma	7.27	3.37	5.03	3.68	2.67	9.75
12	C fma ez_ii	7.22	**3.28**	**4.90**	**3.66**	2.65	9.98
13	C avx512 intrin.	-	-	-	-	-	11.59
14	C avx512 fma intr.	-	-	-	-	-	11.47
	ratio 1 / 5	2.47	6.29	2.94	3.01	6.22	2.76
	ratio 1 / 12	3.09	10.33	4.68	4.72	11.83	2.60

TABLE 13.4 – Résultats comparatifs des méthodes SAXPY : architectures modernes

sont très en dessous de cette valeur. En revanche les unités AVX de ces mêmes processeurs sont plutôt performantes.

Le passage à l'AVX512 sur le Xeon Silver 4208 semble détériorer les performances. On note d'ailleurs pour ce processeur que le passage du SSE à l'AVX n'apporte qu'un gain très faible.

13.8.4 Architectures récentes (2020 et après)

Pour les architectures récentes (voir Table 13.5), la version FMA3 est la plus efficace ainsi que les versions intrinsics utilisant l'AVX2 sur Intel 10850H. Pour l'AMD 5600g, c'est également la version AVX2 intrinsics qui est la plus efficace.

On remarque encore que la FPU est fortement pénalisante (méthodes 1, 2 et 3) sur AMD.

N°	Marque Gamme Modèle	Intel Core i7 10850H 2020	AMD Ryzen 5 5600g 2021	Intel Core i5 12400F 2022
1	C	16.05	38.86	18.50
2	asm fpu	16.22	39.81	18.35
3	asm fpu ur4	16.14	39.95	18.62
4	asm sse	5.49	5.36	4.51
5	C sse intrin.	5.35	5.31	4.60
6	asm avx	3.79	3.75	4.41
7	asm avx2 ur2	3.75	3.47	4.42
8	asm avx2 ur4	3.72	3.44	4.49
9	C avx2 intrin.	3.62	**3.26**	4.79
10	C avx2 ez_ii	3.62	3.39	4.75
11	asm fma	3.66	3.30	**4.40**
12	C fma ez_ii	**3.61**	3.38	4.81
13	C avx512 intrin.	-	-	-
14	C avx512 fma intr.	-	-	-
	ratio 1 / 5	3.00	7.31	4.02
	ratio 1 / 12	4.44	11.49	3.84

TABLE 13.5 – Résultats comparatifs des méthodes SAXPY : architectures récentes

13.9 Conclusion

Nous avons vu dans ce chapitre comment utiliser les instructions de la FPU pour un calcul simple. On note que c'est le passage à la vectorisation qui apporte une amélioration significative du temps de calcul. L'utilisation du FMA qui est censé apporter une diminution du temps de calcul est généralement minime pour ce traitement. On retiendra que la FPU des microprocesseurs AMD n'est pas du tout performante. Heureusement les calculs effectués avec des registres vectoriels sont, quant à eux, bien plus efficaces.

13.10 Exercices

Exercice 47 - A titre d'exercice vous pouvez réaliser un dépliage de la version AVX par 2 puis par 4 et intégrer les nouveaux sous-programmes au code existant afin de tester leur efficacité.

Exercice 48 - Utiliser `vpbroadcastd` afin de charger huit fois a dans `ymm3` pour la version FMA.

Chapitre 14

Etude de cas
Maximum de Parcimonie

14.1 Introduction

Ce chapitre traite de l'implantation de la fonction de *Fitch* dans le cadre de la résolution du problème de la recherche du Maximum de Parcimonie en Bioinformatique. Il permet d'introduire plusieurs instructions assembleur liées au calcul vectoriel avec unités SSE sur les entiers.

Le problème de recherche du Maximum de Parcimonie consiste étant donné un ensemble de n séquences d'ADN de même longueur k à trouver un arbre binaire dont le coût est minimum étant donné un critère d'optimisation.

Pour calculer ce coût qui correspond au nombre de mutations entre séquences, chaque feuille de l'arbre contient une des séquences initiale du problème et les noeuds internes contiennent des séquences dites *hypothétiques* qui sont calculées en utilisant la fonction de *Fitch*. Celle-ci sera notre fonction de référence à améliorer et pour laquelle toute mutation engendre un coût d'une unité.

Le coût total d'un arbre est égal à la somme des coûts de chaque séquence hypothétique. Pour le calculer, on part de la racine et on descend jusqu'aux feuilles, puis on remonte vers la racine en calculant les séquences hypothétiques tout en sommant leurs coûts.

Prenons un exemple avec les quatre séquences d'ADN suivantes :

- `S1 = AAAAA`
- `S2 = AAAAC`
- `S3 = CCCTT`
- `S4 = CCCAT`

On rappelle que les acides nucléiques qui composent les séquence d'ADN [1] sont l'adénine (A), la cytosine (C), la guanine (G) et la thymine (T). La séquence S1 est donc composée de 5 adénines, S2 de quatre adénines suivies d'une cytosine, etc.

On peut voir, Figure 14.1, deux arbres binaires ainsi que le coût de parcimonie selon *Fitch*.

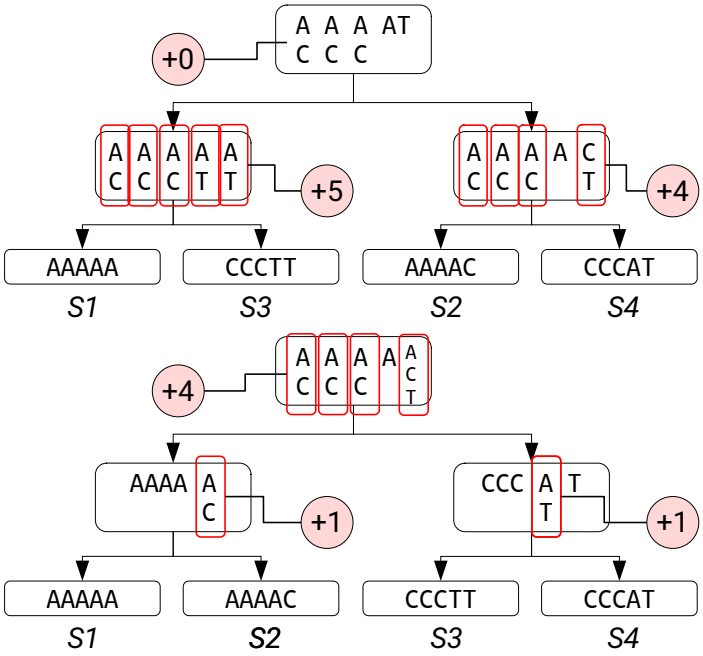


FIGURE 14.1 – Arbres et coûts de parcimonie selon Fitch

Le premier arbre, qui dans la notation *Newick* [2] est $T_1 = ((S1, S3), (S2, S4))$, possède un coût total de $4 + 5 + 0 = 9$. En effet, lorsque l'on calcule la séquence hypothétique qui résulte de $(S1, S3)$, on a cinq mutations : trois modifications où A est muté en C et deux où A subit une mutation en T. Sur la branche de droite qui correspond à séquence hypothétique qui résulte de $(S2, S4)$, on a quatre mutations : trois modifications où A est muté en C, puis A en regard de A, donc aucune mutation, et enfin, une mutation de C en T. Au niveau de la racine, on n'aura aucune mutation car on a à chaque fois des sous-ensembles de caractères communs. On trouve trois fois le sous-ensemble $\{A, C\}$ en regard de lui-même, puis $\{A, T\}$ en regard de $\{A\}$ dont l'intersection est $\{A\}$ et enfin $\{A, T\}$ en regard de $\{C, T\}$ dont l'intersection est $\{T\}$.(

Le second arbre $T_2 = ((S1, S2), (S3, S4))$ possède un coût inférieur à T_1, égal à $4 + 1 + 1 = 6$. C'est donc T_2 qui est le plus *parcimonieux*, c'est à dire qui possède le moins de mutations et qui doit être gardé comme solution au problème.

Le problème de recherche de l'arbre de parcimonie maximum est un problème d'optimisation combinatoire et est NP-Complet ce qui signifie, en simplifiant à

1. Acide désoxyribonucléique.
2. Notation parenthésée utilisée pour décrire des arbres.

outrance, qu'il n'existe pas d'algorithme qui nous permette de trouver la solution autre qu'une recherche exhaustive.

Malheureusement, le nombre d'arbres binaires à examiner est exponentiel, par exemple pour 50 séquences, il y a $2,8\ 10^{74}$ arbres possibles. On ne peut donc envisager une recherche exhaustive et il faut faire appel à des métaheuristiques qui sont des techniques de recherche liées à la résolution de ce genre de problème.

Nous n'irons pas plus avant dans l'explication des techniques de résolution qui font partie de l'*Optimisation Combinatoire* car nous allons nous focaliser sur l'amélioration de la fonction de Fitch.

14.2 Fonction de référence

La fonction à implanter est donnée Listing 14.2.1. Elle comporte quatre paramètres qui sont les séquences en entrée `x` et `y`, la séquence hypothétyque `z` qui est calculée par la fonction ainsi que la longueur des séquences (`size`). En retour on donne le nombre de mutations trouvées.

```
1  u32 maxpars_reference(u8 *x, u8 *y, u8 *z, u32 size) {
2      u32 mutations = 0;
3
4      for (u32 i = 0; i < size; ++i) {
5          z[i] = x[i] & y[i];
6          if (z[i] == 0) {
7              z[i] = x[i] | y[i];
8              ++mutations;
9          }
10     }
11     return mutations;
12 }
```

Listing 14.2.1 – Maximum de Parcimonie fonction de référence en C

Les séquences sont modélisées sous forme de tableaux d'**octets** et pour coder efficacement la possibilité d'avoir des combinaisons d'acides nucléiques comme 'A ou C', 'A ou C ou T', 'A ou C ou G ou T', etc, on utilise des puissances de 2 :

- A = 2^1
- C = 2^2
- G = 2^3
- T = 2^4

Ainsi 'A ou C' qui en notation ensembliste se note $\{A\} \cup \{C\}$ ou encore $\{A, C\}$ est codé par $2^1 + 2^2 = 2 + 4 = 6$. La fonction de Fitch vérifie que pour chacun des sous-ensembles de caractères en regard des deux séquences en entrée, il existe

un caractère ou un sous-ensemble commun en réalisant une intersection des sous-ensembles de caractères de `x[i]` avec `y[i]`. Si cet ensemble est vide c'est qu'il n'existe aucun caractère en commun : on a une mutation et il faut alors prendre l'union des ensembles de caractères.

Union et intersection sont très simple à réaliser si on code les caractères sous forme de puissances de 2. L'union est alors le OU-binaire (`|`) et l'intersection est le ET-binaire (`&`).

Au delà de la signification qui peut sembler complexe, on peut simplement considérer la fonction à implanter comme manipulant des tableaux d'octets.

14.3 Implantation en assembleur

La difficulté de l'implantation réside dans le fait que l'on ne dispose en 32 bits que de 6 registres et que la fonction fait apparaître 6 variables :

- les séquences `x`, `y`, `z`,
- la taille `size` des séquences,
- la variable de boucle `i`,
- le nombre de `mutations`

Cependant il faudra réaliser des calculs comme `x[i] & y[i]`, il est donc nécessaire de disposer d'au minimum un registre pour les calculs.

L'association variables / registres est celle de la Table 14.1. Nous avons fait le choix de ne pas stocker la variable `size` dans un registre et de la laisser dans la pile ce qui nous permet de garder le registre `edx` afin de réaliser les calculs `x[i] & y[i]` et `x[i] | y[i]`.

Cste/Param/Var	Type	Paramètre	Registre	Description
`x`	`u8 []`	`[ebp+8]`	`esi`	séquence `x`
`y`	`u8 []`	`[ebp+12]`	`edi`	séquence `y`
`z`	`u8 []`	`[ebp+16]`	`ebx`	séquence `z`
`size`	`u32`	`[ebp+20]`	pile	taille des séquences
`i`	`u32`		`ecx`	variable de boucle
`mutations`	`u32`		`eax`	nombre de `mutations`
	`u32`		`edx`	calculs

TABLE 14.1 – Association entre variables et registres pour la fonction de référence de Fitch

Le code ressemble donc à ce qui suit et est la traduction directe de la fonction de référence :

```
    push    ebp                 ; entrée dans la fonction
    mov     ebp, esp

    mov     eax, [ebp + 20]     ; si size == 0 alors retourne 0
    test    eax, eax
    jz      .end

    push    esi                 ; sauvegarde des registres
    push    edi                 ; qui seront modifiés mais
    push    ebx                 ; doivent être préservés

                                ; chargement des paramètres
    mov     esi, [ebp + 8]      ; x dans esi
    mov     edi, [ebp + 12]     ; y dans edi
    mov     ebx, [ebp + 16]     ; z dans ebx

    xor     eax, eax            ; mutations = 0
    xor     ecx, ecx            ; i = 0
.for:
    cmp     ecx, [ebp + 20]     ; fin de boucle si i >= size
    jge     .endfor
    mov     dl, [esi + ecx]     ; dl = x[i]
    and     dl, [edi + ecx]     ; dl = x[i] & y[i]
    jnz     .endif              ; si dl != 0 alors aller en .endif
    mov     dl, [esi + ecx]     ; dl = x[i]
    or      dl, [edi + ecx]     ; dl = x[i] | y[i]
    inc     eax                 ; ++mutations
.endif:
    mov     [ebx + ecx], dl     ; z[i] = dl
    inc     ecx                 ; ++i
    jmp     .for
.endfor:

    pop     ebx                 ; restauration des registres
    pop     edi
    pop     esi
.end:
    mov     esp, ebp            ; sortie de fonction
    pop     ebp
    ret
```

Listing 14.3.1 – Maximum de Parcimonie fonction de référence en assembleur

Afin de donner un ordre d'idée du temps d'exécution pour les différentes implantations que nous allons réaliser, nous reportons par la suite, les résultats obtenus sur un Core i7 8700. Le test effectué consiste à calculer 50_000 fois la fonction de référence appliquée sur des séquences initialisées aléatoirement de 524_287 éléments.

Temps de référence

Pour l'implantation que nous venons de donner, l'exécution dure environ $84,37$ secondes.

14.4 Amélioration de la fonction de référence

L'implantation précédente se révèle inefficace pour une simple raison que nous avons déjà évoquée. La présence d'une conditionnelle (`if`) non prédictible à l'intérieur d'une boucle (`for`). Si on a choisi le mauvais chemin d'exécution il faudra vider le pipeline ce qui nous ralenti.

Afin d'améliorer l'efficacité du traitement il est nécessaire de supprimer le (`if`). On peut, dans ce cas précis, choisir d'utiliser des instructions comme `setCC` ou `cmovCC` qui s'exécutent en fonction de la condition, modélisée ici par les deux lettres `CC`.

On va donc modifier l'association variables / registres afin de libérer les registres `eax`, `ebx` et `edx`. Ainsi `esi` sera utilisé pour contenir soit `x`, soit `z` et la variable `mutations` sera placée dans la pile. Le code est alors celui du Listing 14.4.1.

On charge `x[i]` en partie basse de `eax` et `ebx` puis `y[i]` en partie basse de `edx`. On calcule ensuite :

- `x[i] | y[i]` dans `ebx`
- `x[i] & y[i]` dans `eax`

L'instruction `and eax, edx` met à jour le registre `eflags`, dès lors on peut utiliser deux instructions conditionnelles :

- `setz dl` fixe `dl` (donc `edx`) à 1, si `x[i] & y[i]` est égal à 0, sinon `dl` sera égal à 0
- `cmovz eax, ebx` remplace `x[i] & y[i]` par `x[i] | y[i]` dans `eax`

Ces deux instructions réalisent la conditionnelle `if` de la fonction de référence et on met ensuite à jour le nombre de mutations stockées dans la pile en `[ebp-4]` ainsi que la séquence `z`.

Amélioration sans if

En éliminant le `if` on ne met plus que $34,68$ secondes, on va donc environ $2,43$ fois plus vite.

On note l'utilisation de l'instruction `movzx` (lignes 20 et 21) afin de charger respectivement `x[i]` et `y[i]` dans `edx` et `eax`. Cette instruction permet de charger un octet en partie basse d'un registre tout en positionnant à 0 les bits de la partie haute. Elle est généralement plus performante qu'un `mov` qui mettrait `y[i]` dans `dl`.

```
    push    ebp                     ; entrée dans la fonction
    mov     ebp, esp
    sub     esp, 4                  ; mutations = 0
    mov     dword [ebp-4], 0        ;
    mov     eax, [ebp + 20]         ; si size == 0 alors retourne 0
    test    eax, eax
    jz      .end

    push    esi                     ; sauvegarde des registres
    push    edi                     ; qui seront modifiés mais
    push    ebx                     ; doivent être préservés

    mov     edi, [ebp + 12]         ; y dans edi

    xor     ecx, ecx                ; i = 0
.for:
    mov     esi, [ebp+8]            ; esi = x
    cmp     ecx, [ebp + 20]         ; fin de boucle si i >= size
    jge     .endfor
    movzx   edx, byte [edi + ecx]   ; dl = y[i]
    movzx   eax, byte [esi + ecx]   ; al = x[i]
    mov     ebx, eax                ; bl = x[i]
    mov     esi, [ebp+16]           ; esi = z
    or      ebx, edx                ; bl = x[i] | y[i]
    and     eax, edx                ; al = x[i] & y[i]
    setz    dl                      ; dl = (al == 0) ? 1 : 0
    cmovz   eax, ebx                ; al = (al == 0) ? bl : al
    add     [ebp-4], edx            ; mutations += edx
    mov     [esi + ecx], al         ; z[i] = al

    add     ecx, 1                  ; ++i
    jmp     .for

.endfor:
    mov     eax, [ebp-4]            ; eax = nombre de mutations

    pop     ebx                     ; restauration des registres
    pop     edi
    pop     esi
.end:
    mov     esp, ebp                ; sortie de fonction
    pop     ebp
    ret
```

Listing 14.4.1 – Maximum de Parcimonie fonction de référence sans if

14.5 Optimisation de la version sans if

La version sans `if` peut encore être améliorée, pour cela il faut procéder à plusieurs changements :

- la variable `mutations` sera remplacée par le registre `eax`, cela évite des accès à la mémoire
- l'accès au paramètres se fera au travers de `esp` et non de `ebp` qui va être utilisé pour stocker `z`, ce qui fait qu'on aura à notre disposition 7 registres
- les calculs seront réalisés en utilisant `ebx`, `edx` et `edi`
- le registre `edi` sera également utilisé pour stocker `y`, après avoir réalisé les calculs il faudra donc le recharger avec `y`

Le code est donné Listing 14.5.1.

Amélioration sans if et optimisation

Avec cette version optimisée, on ne met plus que $5,24$ secondes, on va donc $3,88$ fois plus vite que la fonction de référence.

14.6 Version SSE

L'implantation de la fonction de référence en utilisant des instructions SSE est relativement simple dès lors que l'on connait les instructions adéquates. On traitera 16 octets en parallèle ce qui apporte un gain substantiel dans l'amélioration de la fonction. Pour réaliser le OU-binaire, on utilisera l'instruction `por` (*Parallel OR*) et pour le ET-binaire, on dispose de l'instruction `pand` (*Parallel AND*)

La difficulté repose sur l'extraction de l'information liée au résultat du ET-binaire. On dispose heureusement pour celà de deux instructions :

- `pcmpeqb xmm1, xmm2` (*Parallel CoMPare EQual Bytes*) qui compare les octets du registre `xmm1` à ceux du du registre `xmm2`, si `xmm1.b[i] == xmm2.b[i]` alors `xmm1.b[i] = 0xFF`, sinon `xmm1.b[i] = 0`
- `pmovmskb eax, xmm1` (*Parallel MOVe MaSK of Bytes*) est utilisée pour récupérer le résultat de la comparaison précédente, on extrait le bit de poids fort de chaque octet du registre `xmm1` et on le place dans `eax`

On est donc en mesure de comparer deux vecteurs d'octets et si deux valeurs au même indice sont égales on positionnera dans le registre destination l'octet correspondant à `0xFF`, c'est à dire *true*, alors que si elles sont différentes, l'octet sera positionné à `0x00`, c'est à dire *false*.

14.6.1 Association variables registres

Comme les calculs se feront dans les registres SSE on va pouvoir disposer des 6 registres généraux (cf. Table 14.2) afin de gérer les adresses des vecteurs, leur taille, la variable de boucle et le nombre de mutations.

```
    mov     eax, [esp + 16]         ; si size == 0 alors retourne 0
    test    eax, eax
    jz      .end

    xor     eax, eax                ; mutations = 0

    push    ebp                     ; sauvegarde des registres
    push    esi
    push    edi
    push    ebx
                                    ; chargement des paramètres
    mov     esi, [esp + 20]         ; x dans esi
    mov     edi, [esp + 24]         ; y dans edi
    mov     ebp, [esp + 28]         ; z dans ebp

    xor     ecx, ecx                ; i = 0
.for:
    movzx   ebx, byte [esi + ecx]   ; bl = x[i]
    movzx   edx, byte [edi + ecx]   ; dl = y|i]
    mov     edi, ebx                ; edi = x[i]

    or      edi, edx                ; edi = x[i] | y[i]
    and     ebx, edx                ; ebx = x[i] & y[i]
    sete    dl                      ; dl = (ebx == 0) ? 1 : 0
    cmovz   ebx, edi                ; ebx = (ebx == 0) ? edi : ebx
    add     eax, edx                ; mutations += edx
    mov     [ebp + ecx], bl         ; z[i] = bl

    mov     edi, [esp + 24]         ; edi = y
    add     ecx, 1                  ; ++i
    cmp     ecx, [esp + 32]         ; fin de boucle si i >= size
    jl      .for

    pop     ebx                     ; restauration des registres
    pop     edi
    pop     esi
    pop     ebp
.end:                               ; sortie de fonction
    ret
```

Listing 14.5.1 – Maximum de Parcimonie fonction de référence sans if optimisée

Le registre **xmm0** sera utilisé pour stocker `x[i:i+15]`, puis le résultat du ET-binaire avec **xmm2** qui lui, contiendra `y[i:i+15]`.

Le registre **xmm1** sera utilisé pour stocker une copie de **xmm0**, puis le résultat du OU-binaire avec **xmm2**

Le registre **xmm6** est mis à 0 car il nous servira à comparer en parallèle le résultat du ET-binaire et permettra de déterminer quels éléments sont à 0 et pour lesquels il faudra prendre le résultat du OU-binaire.

Cste/Param/Var	Type	Paramètre	Registre	Description
x	u8 []	[ebp+8]	esi	séquence x
y	u8 []	[ebp+12]	edi	séquence y
z	u8 []	[ebp+16]	ebx	séquence z
size	u32	[ebp+20]	pile	taille des séquences
i	u32		ecx	variable de boucle
	u32		edx	nombre de répétitions
mutations	u32		eax	nombre de mutations
	u32		ebp	calculs temporaires
	u8[16]		xmm6	$[0, ..., 0]$
	u8[16]		xmm0	x[i:i+15] and(x[i:i+15], y[i:i+15])
	u8[16]		xmm2	y[i:i+15]
	u8[16]		xmm1	y[i:i+15] or(x[i:i+15], x[i:i+15])

TABLE 14.2 – Associations entre variables et registres pour l'implantation SSE

Le registre edx contient le nombre d'itérations de 16 octets. Par exemple si la taille des séquences est de 263 acides nucléiques alors `edx = 16` et il restera 7 itérations à traiter.

Le code de la version SSE étant assez conséquent, nous nous focalisons sur la boucle principale qui ressemble à ceci :

On commence par charger les registres avec les données puis à calculer le ET-binaire et le OU-binaire (lignes 2 à 6). On compare ensuite (lignes 8 et 9) le résultat du ET-binaire, déplacé dans xmm4 au registre xmm6 afin de déterminer quels octets sont à 0, le registre xmm4 va servir par la suite de masque de sélection.

Lignes 9, 15, 16, 18, on extrait l'information sur le nombre de mutations que l'on additionne à eax qui contient le nombre total de mutations.

Enfin, lignes 20 à 24, on calcul la séquence hypothétique z[i:i+15] en sélectionnant soit le résultat du OU-binaire si le ET-binaire a produit un résultat égal à 0, soit le résultat du ET-binaire s'il n'est pas nul.

Notons qu'il ne s'agit pas d'une véritable boucle for mais plutôt un while car le registre edx, comme indiqué précédemment, contient le nombre de répétitions de 16 octets à effectuer. Il est donc décrémenté en ligne 27 et s'il est égal à 0, on sortira de la boucle.

```
.for_u16:
    movdqa  xmm0, [esi + ecx]   ; xmm0 = x[i:i+15]
    movdqa  xmm2, [edi + ecx]   ; xmm2 = y[i:i+15]
    movdqa  xmm1, xmm0          ; xmm1 = xmm0
    pand    xmm0, xmm2          ; xmm0 = x[i:i+15] & y[i:i+15]
    por     xmm1, xmm2          ; xmm1 = x[i:i+15] | y[i:i+15]

    movdqa  xmm4, xmm0          ; xmm4 = x[i:i+15] & y[i:i+15]
    pcmpeqb xmm4, xmm6          ; xmm4 est le masque
                                ; si (x[i] & y[i] == 0) alors
                                ;   xmm4[i] = 0xFF
                                ; sinon
                                ;   xmm4[i] = 0x00

    pmovmskb ebp, xmm4          ; obtenir les bits
    popcnt  ebp, ebp            ; compte le nombre de mutations

    add     eax, ebp            ; ajouter à eax
                                ; calcul de la séquence
    pand    xmm1, xmm4          ; not(xmm4) & (x[i] | y[i])
    pandn   xmm4, xmm0
    por     xmm0, xmm1

    movdqa  [ebx+ecx],xmm0      ; résultat dans z[i:i+15]
    add     ecx, 16             ;  i+= 16

    dec     edx
    jnz     .for_u16
```

Listing 14.6.1 – Maximum de Parcimonie version SSE2

Amélioration SSE2

Grâce à la vectorisation avec jeu d'instruction SSE2, on abaisse le temps d'exécution à $2,70$ secondes soit un facteur d'amélioration de $84,37/2,70 \simeq 31$.

14.7 Version SSE 4.1

Un variante de la version SSE2 va consister à utiliser l'instruction appelée **pblendvb** (*Variable Blend Packed Bytes*) apparue avec le SSE4.1.

Elle permet de sélectionner des octets de chacune de ses deux opérandes suivant le masque de sélection contenu (par convention) dans le registre **xmm0**. Le code s'écrit alors :

Le registre **xmm0** ne contiendra donc plus `x[i:i+15]` mais permettra de vérifier quels octets de l'intersection `x[i:i+15] & y[i:i+15]` sont à 0.

```
.for_ur16:
    cmp     ecx, edx
    jge     .endfor_ur16
    pxor    xmm0, xmm0              ; xmm0 = [0, 0, ..., 0]
    movdqu  xmm1, [esi + ecx]       ; xmm1 = x[i:i+15]
    movdqu  xmm3, [edi + ecx]       ; xmm3 = y[i:i+15]
    movdqa  xmm2, xmm1              ; xmm2 = x[i:i+15]
    movdqa  xmm4, xmm3              ; xmm4 = y[i:i+15]
    pand    xmm1, xmm3              ; xmm1 = x[i:i+15] & y[i:i+15]
    por     xmm2, xmm4              ; xmm1 = x[i:i+15] | y[i:i+15]
    pcmpeqb xmm0, xmm1              ; si xmm1.b[i] == 0
                                    ;    xmm0.b[i] = 0xFF
                                    ; alors
                                    ;    xmm0.b[i] = 0x00
    pmovmskb    ebp, xmm0           ; obtenir le nombre de mutations locales
    popcnt  ebp, ebp                ; compter
    add     eax, ebp                ; ajouter aux mutations totales
    pblendvb    xmm1, xmm2, xmm0    ; calculer la séquence hypothétique
    movdqa  [ebx + ecx], xmm1       ; stocker le résultat en z[i:i+15]
    add     ecx, 16                 ; i+= 16
    jmp     .for_ur16
.endfor_ur16
```

Listing 14.7.1 – Maximum de Parcimonie version SSE4.1

Lorsque l'on exécute l'instruction `pblendvb xmm1, xmm2, xmm0` on sélectionne les octets de `xmm2.b[i]` pour lesquels `xmm0.b[i]` vaut FF_{16}. Dans le cas contraire on garde `xmm1.b[i]`. Etant donné que :

- `xmm2` contient `x[i:i+15] | y[i:i+15]`
- `xmm1` contient `x[i:i+15] & y[i:i+15]`
- chaque octet de `xmm0` vaut FF_{16} si `x[i] & y[i] == 0`

On obtient bien le résultat escompté.

Amélioration SSE4.1

On passe alors à un temps d'exécution à $2,60$ secondes soit une légère amélioration facteur d'amélioration de $\simeq 32$.

14.8 Version AVX / AVX2

La version AVX utilise les registres `ymm` qui ont une taille de 32 octets (soit 256 bits). Le code est similaire à celui de la version SSE 4.1. On utilise le préfixe v afin de signaler qu'il s'agit d'instructions AVX.

On notera que :

```
1  .for_ur32:
2      vpxor       ymm0, ymm0          ; ymm0 = [0, 0, ..., 0]
3      vmovdqa     ymm1, [esi + ecx]   ; ymm1 = x[i:i+15]
4      vmovdqa     ymm3, [edi + ecx]   ; ymm3 = y[i:i+15]
5      vmovdqa     ymm2, ymm1          ; ymm2 = ymm1
6      vmovdqa     ymm4, ymm3          ; ymm4 = ymm3
7
8      vpand       ymm1, ymm3          ; ymm1 = ymm1 & ymm3
9      vpor        ymm2, ymm4          ; ymm2 = ymm2 | ymm3
10
11     vpcmpeqb    ymm0, ymm1          ; si ymm1.b[i] == 0
12                                     ;    ymm0.b[i] = 0xFF
13                                     ; alors
14                                     ;    ymm0.b[i] = 0x00
15     vpmovmskb   ebp, ymm0           ; obtenir le nombre de mutations locales
16     popcnt      ebp, ebp            ; compter
17     add         eax, ebp            ; ajouter aux mutations totales
18
19     vpblendvb   ymm1, ymm2, ymm0    ; calculer la séquence hypothétique
20
21     vmovdqa     [ebx + ecx], ymm1   ; stocker le résultat en z[i:i+15]
22
23     add         ecx, 32             ; i += 32
24     dec         edx
25     jnz         .for_ur32
```

Listing 14.8.1 – Maximum de Parcimonie version AVX2

- l'instruction `vpcmpeqb ymm0, ymm1` compare les 32 octets de `ymm0` à ceux de `ymm1`.
- l'instruction `vpmovmskb ebp, ymm0` extrait 32 bits de chaque octet de poids fort de `ymm0`

On aurait pu également remplacer les lignes 8 et 9 du Listing 14.8.1 par :

```
1  vpand       ymm1, ymm1, ymm3        ; ymm1 = ymm1 & ymm3
2  vpor        ymm2, ymm2, ymm4        ; ymm2 = ymm2 | ymm3
```

ou bien stocker dans d'autres registres les résultats des unions et intersections afin d'éviter les dépendances :

```
1  vpand       ymm4, ymm1, ymm3        ; ymm1 = ymm1 & ymm3
2  vpor        ymm5, ymm2, ymm4        ; ymm2 = ymm2 | ymm3
3  vpcmpeqb    ymm0, ymm4
4  ...
```

Amélioration AVX2

Le temps d'exécution est alors de $1,35$ secondes soit un facteur d'amélioration d'environ 62, soit presque deux fois plus rapide que le SSE.

14.9 Fonction de référence et compilateur

Notons que la fonction de référence optimisée par le compilateur en utilisant par exemple avec `gcc` les options d'optimisation `-O3 -mavx2` et le dépliage de boucle donne un temps d'exécution de l'ordre de $1,85$ secondes soit proche de la version AVX2 assembleur.

Le code vectorisé avec AVX2 et généré par le compilateur (version *intrinsics*) est nettement plus complexe que ce que nous avons écrit mais le temps d'exécution est très proche de la version AVX2 assembleur : $1,39$ s. On peut alors se demander si le compilateur produit un code plus complexe car il sait que le code sera bien plus optimisé, ou si il fait cela car il ne parvient pas à traduire le code.

14.10 Version intrinsics

La version intrinsics en AVX2 nécessite pour être efficacement traduite par le compilateur de fournir quelques informations à ce dernier.

Notamment le mot clé `__restrict__`[3], indique que pour la durée de vie du pointeur, seul le pointeur sera utilisé pour accéder à l'objet vers lequel il pointe. L'objectif est de limiter les effets de l'*aliasing de pointeur*[4] ce qui permet au compilateur d'optimiser le code.

En outre, il est préférable d'indiquer au compilateur que les adresses des tableaux `x`,`y` et `z` sont alignées sur un multiple de 16 ou 32 octets grâce à la fonction (ou directive) `__builtin_assume_aligned`. En conséquence le compilateur utilisera les instructions de type `movdqa` plutôt que `movdqu` et pourra procéder à quelques optimisations.

```
1 u32 maxpars_avx2_intrinsics(u8 * __restrict__ x, u8 * __restrict__ y,
2                              u8 * __restrict__ z, u32 size) {
3     u32 i, mutations=0;
4
5     x = (u8 *) __builtin_assume_aligned(x, CPU_MEMORY_ALIGNMENT);
6     y = (u8 *) __builtin_assume_aligned(y, CPU_MEMORY_ALIGNMENT);
7     z = (u8 *) __builtin_assume_aligned(z, CPU_MEMORY_ALIGNMENT);
8
```

3. Pour d'autre compilateurs, comme le compilateur Intel, il faut utiliser `restrict`.
4. Le fait qu'un objet soit accédé par plusieurs pointeurs différents.

```
    __m256i v_x, v_y, v_z, v_x_and_y, v_x_or_y,
            v_zero, v_cmp __attribute__((aligned(32)));

    v_zero = _mm256_set_epi8(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                             0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);

    for (i = 0; i < (size & (~31)); i+=32) {
        v_x = _mm256_load_si256((__m256i *) &x[i]);
        v_y = _mm256_load_si256((__m256i *) &y[i]);
        v_x_and_y = _mm256_and_si256( v_x, v_y);
        v_x_or_y = _mm256_or_si256( v_x, v_y );
        v_cmp = _mm256_cmpeq_epi8( v_zero, v_x_and_y );
        u32 r = _mm256_movemask_epi8( v_cmp );
        mutations += _mm_popcnt_u32(r);
        v_x = _mm256_andnot_si256( v_cmp, v_x_and_y );
        v_y = _mm256_and_si256( v_cmp, v_x_or_y );
        v_z = _mm256_or_si256( v_x, v_y );
        _mm256_store_si256( (__m256i *) &z[i], v_z);
    }

    // dernières itérations
    for ( ; i<size; ++i) {
        z[i] = x[i] & y[i];
        if (z[i] == 0) {
            z[i] = x[i] | y[i];
            ++mutations;
        }
    }

    return mutations;
}
```

On notera que l'appel à `_mm256_set_epi8` peut être remplacé par un `vpxor` (dont l'intrinsics est `_mm256_setzero_si256`), puisqu'elle a pour but de mettre le vecteur `v_zero` à 0).

14.11 Version AVX512

Pour l'AVX512, on utilise les registres `zmm` d'une capacité de 64 octets. Le nombre total de mutations est stocké dans `eax` alors que le registre `ebp` permet de compter (en deux fois 32 bits) les mutations pour le vecteur courant `zmm1`.

```
    push    ebp             ; sauvegarde d'ebp

.for_u64:
    vmovdqa64   zmm3, [esi + ecx]       ; charge x[i:i+63]
    vpandd      zmm1, zmm3, [edi + ecx] ; zmm1 = x[i:i+63] & y[i:i+63]
    vpord       zmm2, zmm3, [edi + ecx] ; zmm2 = x[i:i+63] | y[i:i+63]

```

```
8       vpcmpeqb    k1, zmm0, zmm1      ; comparaison
9
10      vmovdqu8    zmm1 {k1}, zmm2     ; remplacement des éléments de zmm1
11                                      ; par les éléments de zmm2
12                                      ; suivant k1
13      vmovdqa64   [ebx + ecx], zmm1
14
15      kmovd       ebp, k1             ; partie basse de k1 dans ebp
16      kshiftrq    k2, k1, 32
17      popcnt      ebp, ebp
18      add         eax, ebp
19      kmovd       ebp, k2             ; partie haute de k2 dans ebp
20      popcnt      ebp, ebp
21      add         eax, ebp
22
23      add     ecx, 64
24      dec     edx
25      jnz     .for_u64
26
27      pop     ebp
28      vzeroupper
29  .last_63:
```

L'AVX512 apporte un léger gain par rapport à l'AVX2 (cf. Résultats ci-après).

14.12 Tests de performance

Afin de tester les différentes versions que nous avons écrites, nous allons examiner les résultats obtenus pour les méthodes suivantes :

1. méthode de référence optimisée par le compilateur GCC avec option `-O2`
2. méthode de référence optimisée par le compilateur GCC avec option `-O3` et options de vectorisation en fonction de l'architecture
3. traduction en assembleur de la méthode de référence
4. traduction en assembleur de la méthode qui supprime le `if`
5. amélioration de la version précédente
6. amélioration de la version précédente avec dépliage par 4 de la boucle
7. amélioration de la version précédente avec dépliage par 8 de la boucle
8. traduction en assembleur avec vectorisation en SSE2
9. traduction en assembleur avec vectorisation en SSE4.1
10. traduction en assembleur avec vectorisation en AVX2
11. version intrinsics AVX2 optimisée par le compilateur

14.12.1 Architectures anciennes (avant 2015)

Les résultats pour les architectures anciennes sont présentés Table 14.3. On note que le compilateur g++ est capable, grâce à l'option -O3, d'optimiser la fonction de référence de manière très importante. On avoisine, ou parfois on est en dessus de la méthode SSE (méthode 8).

Les méthodes 4 à 7 qui suppriment le `if` diminuent le temps de calcul par un facteur 2 même si elles restent loin de ce que peut apporter la vectorisation avec SSE.

n°	Méthode	Intel Pentium D 925 2006	Intel Core 2 Q9300 2008	Intel i7 860 2009	AMD X6 1090T 2010	Intel i5 3570K 2012	Intel i7 4790 2014
1	ref_v1_O2	224.64	170.22	131.86	110.89	140.27	103.54
2	ref_v1_O3	29.86	11.80	**5.47**	**9.11**	**4.23**	2.56
3	asm	235.32	185.40	126.40	118.98	114.89	96.59
4	no_if_asm	133.48	98.36	67.72	52.67	48.67	41.62
5	no_if_opt_asm	94.97	85.60	65.33	45.64	33.35	25.95
6	no_if_opt_ur4_asm	70.15	87.67	51.68	38.16	28.95	22.52
7	no_if_opt_ur8_asm	70.29	86.91	50.25	36.73	28.32	21.89
8	sse2_v1	**27.64**	9.37	5.84	11.75	4.49	3.66
9	sse41	-	**9.25**	5.72	-	4.42	3.73
10	avx2	-	-	-	-	-	**1.86**
11	avx2_intrinsics	-	-	-	-	-	**1.89**
	ratio (1 / 8 ou 10)	8.12	18.16	22.57	9.43	31.24	55.66

TABLE 14.3 – Architectures anciennes : temps d'exécution en secondes de la fonction de Fitch avec 50_000 répétitions sur des chaînes de 524_287 bases

On note enfin, sur l'Intel 4790, que le passage à l'AVX permet de diminuer par deux le temps de calcul par rapport au SSE.

14.12.2 Architectures modernes (2015 à 2019)

Pour les architectures modernes (Table 14.4), on observe les mêmes tendances. On notera que la version AVX2 (méthode 10) est souvent plus performante que la version intrinsics traduite par le compilateur (méthode 11).

Dans le cas de l'AMD Ryzen 5 3600, la traduction en assembleur avec jeu d'instruction AVX2 permet d'atteindre $0,79$ secondes soit un facteur d'amélioration de presque 131.

N°	Marque Gamme Modèle	Intel Core i3 6100 2015	AMD Ryzen 7 1700X 2017	Intel Core i5 7400 2017	Intel Core i7 8700 2017	AMD Ryzen 5 3600 2019	Intel Xeon 4208 2019
1	`ref_v1_02`	139.36	111.55	148.59	96.72	103.41	118.36
2	`ref_v1_03`	2.20	2.50	2.45	1.85	2.02	3.61
3	`asm`	104.56	106.52	110.79	84.37	101.56	123.00
4	`no_if_asm`	42.58	53.86	45.42	34.68	47.03	42.45
5	`no_if_opt_asm`	21.43	19.58	22.82	17.40	17.24	49.80
6	`no_if_opt_ur4_asm`	19.67	18.85	20.95	15.92	16.75	23.04
7	`no_if_opt_ur8_asm`	18.75	17.17	19.99	15.21	14.68	22.56
8	`sse2_v1`	3.58	3.59	3.52	2.70	3.16	4.30
9	`sse41`	3.58	3.61	3.46	2.60	3.19	4.28
10	`avx2`	**1.54**	**1.14**	**1.76**	**1.35**	**0.79**	**2.87**
11	`avx2_intrinsics`	1.81	1.39	1.82	1.39	1.09	3.03
	ratio (1 / 10)	90.49	97.85	84.42	71.64	130.89	41.24

TABLE 14.4 – Architectures modernes : temps d'exécution en secondes de la fonction de Fitch avec 50_000 répétitions sur des chaînes de 524_287 bases.

L'Intel Xeon Silver 4208 dispose du jeu d'instructions AVX512. Les résultats obtenus sont les suivants :

- version de base AVX512 : 2.95 s
- amélioration avec dépliage par 8 et élimination des dépendances : 2.43 s
- version intrinsics : 2.34 s

Dans ce cas c'est la version *intrinsics* qui est la plus efficace.

14.12.3 Architectures récentes (2020 et après)

Les résultats concernant les architectures récentes figurent Table 14.5). Là également, on note que l'AVX2 est très bénéfique qu'il soit codé à la main ou écrit sous une version intrinsics et permet de diviser par un facteur 2 le temps obtenu avec le SSE.

Le cas de l'Intel 12400F est également remarquable. Avec une compilation en `-O2`, il se montre bien moins performant que lIntel i7 10850H et l'AMD Ryzen 5 5600G. Néanmoins, par la suite les résultats obtenus sont très compétitifs pour les méthodes 5,6 et 7 ainsi que les méthodes vectorielles (8 à 11).

N°	Marque Gamme Modèle	Intel i7 10850H 2020	Intel i7 12400f 2022	AMD Ryzen 5 5600g 2021
1	`ref_v1_02`	83.50	104.50	91.38
2	`ref_v1_03`	1.85	1.69	1.98
3	`asm`	79.75	100.53	82.38
4	`no_if_asm`	27.44	44.54	29.31
5	`no_if_opt_asm`	16.17	11.98	17.23
6	`no_if_opt_ur4_asm`	14.81	11.06	14.96
7	`no_if_opt_ur8_asm`	14.20	10.49	13.99
8	`sse2_v1`	2.56	1.82	1.74
9	`sse41`	2.53	1.81	1.75
10	`avx2`	**1.36**	**0.96**	**0.68**
11	`avx2_intrinsics`	**1.36**	**0.96**	**0.70**
	ratio (1 / 10)	61.39	108.85	97.85

TABLE 14.5 – Architectures récentes : temps d'exécution en secondes de la fonction de Fitch avec 50_000 répétitions sur des chaînes de 524_287 bases.

14.13 Conclusion

Dans cette étude de cas, la fonction de référence peut être optimisée de manière spectaculaire par le compilateur en utilisant les options de compilation liées à la vectorisation et le dépliage de boucle. Cependant nous voyons que la version assembleur avec jeu d'instruction AVX2 écrite à la main (ou la version intrinsics) sont les plus efficaces. Nous sommes bien entendu tributaires de la disponibilité d'instructions liées à ce traitement comme `pcmpeqb` et `pmovmskb`. Enfin, on remarquera que l'utilisation de l'AVX512 permet de grandement simplifier le codage du traitement en raison de l'utilisation des registres de masque `k1` et `k2`. Le traitement est également encore plus simple et plus efficace à traduire en architecture 64 bits, puisqu'on n'est pas contraint de réaliser le calcul du nombre de mutations en deux fois 32 bits mais en une fois 64 bits. Gageons que l'implantation de l'AVX512 deviendra plus efficace dans les années à venir et deviendra deux fois plus performante que l'AVX2.

14.14 Exercices

Exercice 49 - Réalisez un dépliage de la version SSE 4.1 par 4, puis par 8. Intégrez votre code au projet et comparez les résultats obtenus aux méthodes existantes.

Exercice 50 - Réalisez un dépliage de la version AVX par 4, puis par 8. Intégrez votre code au projet et comparez les résultats obtenus aux méthodes existantes.

Chapitre 15

Etude de cas
Compter les voyelles

15.1 Introduction

Nous allons dans ce chapitre nous intéresser à un problème simple qui sert d'exemple de démonstration et qui consiste à compter les voyelles dans une chaine de caractères. Ce problème bien qu'évident à implanter demande de posséder les connaissances que nous avons mises en avant dans les Chapitres 1 et 3. L'utilisation des instructions vectorielles apporte également une amélioration substantielle sous certaines architectures.

On considère pour ce problème des chaînes de caratères ne possédant que des caractères en minuscule sans accents ni signes de ponctuation afin de simplifier l'écriture du code. On ne prend en compte que les voyelles (a, e, i, o, u et y) et on ne considère que des chaînes de longueur multiple de 64 pour pouvoir utiliser l'AVX-512 et simplifier le code.

Nous allons coder en 64 bits afin de disposer de nombreux registres qui vont grandement nous simplifier la tâche.

15.2 Fonctions de référence

Nous pouvons concevoir trois fonctions de référence pour répondre au problème :

- la première fonction est écrite en utilisant des `if`, ce qui est normalement très pénalisant lorsqu'ils sont dans une boucle
- la seconde fonction utilise un `switch` qui est sensé pallier au problème de performance du `if`

- enfin la troisième fonction utilise un tableau afin d'éviter les branchements conditionnels induits par le `if` ou le `switch`

La fonction réalisée avec un `if` est présentée Listing 15.2.1. Les fonctions auront toutes la même signature à savoir un pointeur sur une chaine de caractères en C, la longueur de la chaine et un pointeur sur un tableau de six entiers qui sont les compteurs du nombre d'occurrences de chaque voyelle. Ici `v[0]` compte le nombre de `'a'`, `v[1]` le nombre de `'e'`, etc. Cette fonction est pénalisante car on ne peut pas prédire quel *then* sera exécuté et, de plus, si on trouve un `'y'` ou une consonne on devra réaliser six comparaisons.

```
1  void count_if(u8 *s, u32 size, u32 v[6]) {
2      for (u32 i=0; i<size; ++i) {
3          if (s[i] == 'a') {
4              ++v[0];
5          } else if (s[i] == 'e') {
6              ++v[1];
7          } else if (s[i] == 'i') {
8              ++v[2];
9          } else if (s[i] == 'o') {
10             ++v[3];
11         } else if (s[i] == 'u') {
12             ++v[4];
13         } else if (s[i] == 'y') {
14             ++v[5];
15         }
16     }
17 }
```

Listing 15.2.1 – Compter les voyelles avec un if

La fonction implantée avec un `switch` tente de remédier au problème du `if`, elle est présentée Listing 15.2.2. Son codage en assembleur par un compilateur C/C++ génère un tableau de 25 adresses qui correspondent aux lettres `'a'` à `'y'`. Ces adresses sont utilisées pour se brancher sur une partie du sous-programme qui incrémente `v[i]` pour la voyelle correspondante ou qui incrémente la variable de boucle s'il s'agit d'une consonne.

La fonction implantée avec un tableau (cf. Listing 15.2.3) consiste à compter tous les caractères. Etant donné qu'il y a 26 lettres dans l'alphabet on crée un tableau temporaire (`letters`) de 26 entiers que l'on initialise à 0. L'écriture de la boucle est donc simplifiée puisqu'on n'a plus qu'une seule instruction et que le dépliage de la boucle sera facilement réalisé par le compilateur. Le tableau temporaire composé de 26 entiers de 32 bits tient aisément dans la mémoire cache et permettra d'accélérer le traitement. En fin de sous-programme, on recopiera dans `v` le nombre d'occurrences de chaque voyelles.

```
void count_switch(u8 *s, u32 size, u32 v[6]) {
    for (u32 i=0; i<size; ++i) {
        switch(s[i]) {
            case 'a': ++v[0]; break;
            case 'e': ++v[1]; break;
            case 'i': ++v[2]; break;
            case 'o': ++v[3]; break;
            case 'u': ++v[4]; break;
            case 'y': ++v[5]; break;
        }
    }
}
```

Listing 15.2.2 – Compter les voyelles avec un switch

```
void count_letters(u8 *s, u32 size, u32 v[6]) {
    u32 letters[26];
    for (u32 i=0; i<26; ++i) letters[i] = 0;

    for (u32 i=0; i<size; ++i) {
        ++letters[s[i]-'a'];
    }
    v[0] = letters['a'-'a'];
    v[1] = letters['e'-'a'];
    v[2] = letters['i'-'a'];
    v[3] = letters['o'-'a'];
    v[4] = letters['u'-'a'];
    v[5] = letters['y'-'a'];

}
```

Listing 15.2.3 – Compter les voyelles avec un tableau

Temps de référence

Le test de référence consiste à exécuter 50_000 fois le dénombrement du nombre de voyelles pour une chaîne de 256_000 caractères, initialisée aléatoirement, avec environ 20 % de voyelles.
Sur un AMD Ryzen 5 3600, on obtient :

- pour la version `if` le test dure $57,09$ secondes
- pour la version `switch` le test s'effectue en $54,71$ secondes
- et pour la version avec tableau, le temps d'exécution est de $4,64$ secondes

La méthode qui consiste à compter toutes les lettres est donc la plus efficace car elle ne contient pas de conditionnelle et elle peut être dépliée simplement.

15.3 Traduction de la méthode du tableau en assembleur

On peut traduire la méthode du tableau directement en assembleur mais nous allons utiliser quelques améliorations liées à la vectorisation pour initialiser le tableau de lettres.

On stockera le tableau `letters` dans la pile à une adresse multiple de 32 afin de l'initialiser par la suite avec un registre AVX. Les conventions choisies sont celles de la Table 15.1.

Variable	Type	Registre	Description
`s`	`u8 *`	`rdi`	`&s[i]`
`size`	`u32/u64`	`rsi`	`size`
`v`	`u32 *`	`rdx`	`&v[0]`
`i`	`u32/u64`	`rcx`	`i`
	`u64`	`rbx`	adresse du tableau `letters`
	`u64`	`rax`	calculs temporaires
	`u32`	`r8`	stockage temporaire de `rdi`
	`u32`	`r9`	compteur pour 'e'
	`u256`	`ymm0`	stockage de [0,0,...,0]

TABLE 15.1 – Associations variables C et registres pour compter les voyelles pour l'implantation avec tableau

15.3.1 Initialisation du tableau

On commence par sauvegarder le registre `rbx` dans la *red zone* car il ne doit pas être modifié et sera donc préservé dans la pile car il va stocker l'adresse du tableau `letters`. Puis, on crée le tableau qui va occuper $26 \times 4 = 104$ octets, toujours dans la *red zone* en faisant en sorte que son adresse de début soit multiple de 32 afin d'améliorer l'accès mémoire. L'adresse stockée dans `rbx` doit être diminuée de $8 + 104$ octets puisqu'on sauvegarde le contenu de `rbx` en premier dans la pile. On met `eax` à 0 car on aura besoin d'utiliser la valeur 0 pour initialiser le tableau.

```
1     mov     [rsp - 8], rbx
2     lea     rbx, [rsp - (8 + 4*26) ]
3     and     rbx, ~(32-1)     ; rbx multiple de 32
4                              ; en bas de la red zone
5     xor     eax, eax
```

Par exemple si `rsp = 0xfdcf8`, alors `rsp - (8+4*26) = 0xfdc88`, puis si on

arrondi au multiple de 32 inférieur (ligne 3). On obtient alors dans `rbx` la valeur `0xfdc80`.

Il faut ensuite initialiser le tableau et on peut le faire de trois manières différentes, en utilisant :

- un registre 64 bits comme `eax`, affecté à 0, et en initialisant le tableau dans l'ordre des adresses croissantes, soit 13 affectations
- `rep stosq`
- un registre AVX de 32 octets affecté à 0

15.3.1.1 Initialisation par registre général

On utilise les macro-instructions de `nasm` pour initialiser l'ensemble du tableau soit 26×4 octets divisés par la taille d'un registre 64 bits, soit $26 \times 4/8 = 13$:

```
1     xor     rax, rax
2     %assign i 0                 ; variable i = 0
3     %rep 13                     ; répète 13 fois
4     mov     [rbx + i], rax
5     %assign i i + 8             ; i = i + 8
6     %endrep
```

15.3.1.2 Initialisation rep stosq

C'est le même principe que précédemment mais on utilise `stosq`, il faut donc fixer `rax` à 0, sauvegarder `rdi` temporairement (on utilise ici `r8`) et mettre dans `rcx` ou `ecx` le nombre de répétitions :

```
1     xor     rax, rax    ; rax = 0
2     mov     r8, rdi     ; sauvegarde rdi dans r8
3     mov     rdi, rbx    ; rdi = &letters[0]
4     mov     ecx, 13     ; faire 13 fois
5     rep     stosq       ;
6     movd    rdi, r8     ; restaure rdi depuis r8
```

15.3.1.3 Initialisation par registre vectoriel AVX

On utilise `ymm0` qui permet de stocker 8 entiers 32 bits. On doit donc stocker ce registre vectoriel trois fois, puis les deux derniers entiers sont mis à 0 grâce au registre `rax` :

```
1     vpxor   ymm0, ymm0
2     vmovdqa [rbx], ymm0          ; letters[ 0: 7] = 0
3     vmovdqa [rbx + 32], ymm0     ; letters[ 8:15] = 0
4     vmovdqa [rbx + 64], ymm0     ; letters[16:23] = 0
```

```
5       vzeroupper
6       mov     [rbx + 96], rax     ; letters[24:25] = 0
```

15.3.2 Boucle principale

On commence par vérifier si `size` est égale à 0, sinon on exécute la boucle. On charge la lettre `s[i]` dans `eax` en la transformant en une valeur 32 bits grâce à `movzx`. On retranche alors la valeur de `'a'` pour obtenir l'indice de la lettre stockée dans le tableau `letters`. Il peut-être nécessaire de transformer cet indice en 64 bits en utilisant l'instruction `cdqe` mais normalement ce n'est pas nécessaire puisque nous avons mis `eax` à 0 précédemment.

```
1       test    rsi, rsi        ; si size == 0 alors
2       jz      .end_while      ;   aller en .end_while
3
4       xor     rcx, rcx        ; i = 0
5   .while:
6       movzx   eax, byte [rdi + rcx] ; eax = s[i]
7       inc     rcx                   ; ++i
8       sub     eax, 'a'              ; eax = s[i] - 'a'
9       cdqe                          ; convertir en 64 bits
10      inc     dword [rbx + rax * 4] ; ++letters[ s[i]-'a' ]
11      cmp     rcx, rsi              ; si i < size alors
12      jne     .while                ;   aller en .while
13  .end_while:
```

On incrémente finalement `letters[i]` (ligne 10) puis on continue la boucle si `i` est inférieur à `size`.

15.3.3 Sortie de fonction

La sortie de la fonction consiste à recopier le nombre d'occurrences de chaque voyelle dans le tableau `v`.

```
1       mov     r9, 'a'-'a'             ; stocke le nombre d'occurrences
2       mov     eax, [rbx + r9 * 4]     ; de 'a' dans v[0]
3       mov     [rdx], eax
4
5       mov     r9, 'e'-'a'             ; stocke le nombre d'occurrences
6       mov     eax, [rbx + r9 * 4]     ; de 'e' dans v[1]
7       mov     [rdx + 4], eax
8
9       mov     r9, 'i'-'a'             ; stocke le nombre d'occurrences
10      mov     eax, [rbx + r9 * 4]     ; de 'i' dans v[2]
11      mov     [rdx + 8], eax
12
13      mov     r9, 'o'-'a'             ; stocke le nombre d'occurrences
```

```
14      mov     eax, [rbx + r9 * 4]     ; de 'o' dans v[3]
15      mov     [rdx + 12], eax
16
17      mov     r9, 'u'-'a'             ; stocke le nombre d'occurrences
18      mov     eax, [rbx + r9 * 4]     ; de 'u' dans v[4]
19      mov     [rdx + 16], eax
20
21      mov     r9, 'y'-'a'             ; stocke le nombre d'occurrences
22      mov     eax, [rbx + r9 * 4]     ; de 'y' dans v[5]
23      mov     [rdx + 20], eax
24
25      mov     rbx, [rsp - 8]          ; restaure rbx
26      ret
```

On utilise ici le registre `r9` pour représenter l'indice de chaque voyelle dans le tableau et on récupère le nombre d'occurrences dans le registre `eax`. Le registre `rdx` contient l'adresse du tableau `v`. A la fin de la fonction on restaure le registre `rbx` qui avait été sauvegardé.

On notera qu'il est possible d'améliorer la fonction en remplaçant `rbx` par `r10` qui n'a pas besoin d'être sauvegardé.

15.3.4 Dépliage par 4

La boucle principale peut être dépliée par 4 ou 8. Deux possibilités s'offrent à nous :

- soit on recopie le code de la boucle précédente quatre ou huit fois
- soit on charge `s[i:i+3]` dans `eax` puis on traite chacun des octets du registre indépendamment comme suit :

```
1   .while_ur4:
2       mov     eax,  dword [rdi + rcx]   ; charge s[i:i+3]
3       add     ecx, 4                    ; i += 4
4
5       movzx   r8, al          ; r8 = s[i]
6       shr     eax, 8
7       movzx   r9, al          ; r9 = s[i+1]
8       shr     eax, 8
9       movzx   r10, al         ; r10 = s[i+2]
10      shr     eax, 8          ; eax = s[i+3]
11
12      sub     eax, 'a'        ; calcul des indices
13      sub     r8,  'a'
14      sub     r9,  'a'
15      sub     r10, 'a'
16
17      inc     dword [rbx + rax * 4]   ; incrément de occurrences
18      inc     dword [rbx + r8  * 4]   ; de chaque lettre
```

```
19      inc     dword [rbx + r9  * 4]
20      inc     dword [rbx + r10 * 4]
21
22      cmp     ecx, esi
23      jl      .while_ur4
24  .end_while_ur4:
```

Il faut bien évidemment déplier la boucle principale par quatre puis gérer les dernières itérations.

Amélioration traduction en assembleur

Pour la version basée sur un tableau des occurrences de lettres traduite en assembleur le test s'exécute en $5,6$ secondes en moyenne (4,85 s pour le temps minimum, voir encadré ci-après). Le meilleur temps de calcul sur AMD Ryzen 5 3600 est donné par la version dépliée par 4.

Attention

Sur les microprocesseurs Intel le temps de calcul sur 10 itérations est sensiblement le même. Par contre sur un AMD Ryzen 5 3600, on observe environ 7 à 9 exécutions de l'ordre de $4,8$ à 5 secondes et 1 à 3 exécutions entre 7, 8, voire 14 secondes ! Ce bug est également présent sur un Ryzen 7 1700X, mais semble réglé sur Ryzen 5 5600G. Le problème vient de l'utilisation d'une instruction `inc` qu'il est préférable de remplacer par un `add`.

15.4 Vectorisation avec SSE

Il est possible d'envisager une version utilisant les registres SSE car le corps du Listing 15.2.3 est facilement vectorisable.

L'association variables / registres est présentée Table 15.2. Nous utilisons 11 registres généraux (12 si on considère également `rax` pour des calculs temporaires) et 12 registres SSE.

Les registres `xmm8` à `xmm13` contiendront `s[i:i+15]` et seront modifiés par les calculs. Les registres `xmm2` à `xmm7` contiennent des vecteurs composés de voyelles et ne seront pas modifiés. On comparera en parallèle `xmm8` avec `xmm2`, puis `xmm9` avec `xmm3`, etc.

La première partie du code consiste à sauvegarder les registres `rbx`, `r13`, `r14` qui vont être modifiés et qui par convention doivent être préservés pour le sous-programme appelant. On sauvegarde ces registres dans la *red zone*.

```
1   mov     [rsp - 16], rbx     ; sauvegarde des registres
2   mov     [rsp - 24], r12
3   mov     [rsp - 32], r13
```

Variable	Type	Registre	Description
s	u8 *	rdi	&s[i]
size	u32/u64	rsi	size
v	u32 *	rdx	&v[0]
i	u32/u64	rcx	i
	u64	rbx	résultat popcnt
	u32/u64	r8	compteur pour 'a'
	u32/u64	r9	compteur pour 'e'
	u32/u64	r10	compteur pour 'i'
	u32/u64	r11	compteur pour 'o'
	u32/u64	r12	compteur pour 'u'
	u32/u64	r13	compteur pour 'y'
	u128	xmm8	xmm13 à s[i:i+15]
	u128	xmm2	['a',...,'a']
	u128	xmm3	['e',...,'e']
	u128	xmm4	['i',...,'i']
	u128	xmm5	['o',...,'o']
	u128	xmm6	['u',...,'u']
	u128	xmm7	['y',...,'y']

TABLE 15.2 – Associations variables C et registres pour compter les voyelles pour la version SSE

```
4     xor       r8, r8              ; nb_occ('a') = 0
5     xor       r9, r9              ; nb_occ('e') = 0
6     xor       r10, r10            ; nb_occ('i') = 0
7     xor       r11, r11            ; nb_occ('o') = 0
8     xor       r12, r12            ; nb_occ('u') = 0
9     xor       r13, r13            ; nb_occ('y') = 0
```

On remplit ensuite chacun des registres xmm2 à xmm7 avec respectivement des 'a', des 'e', etc.

```
1     mov       eax, 0x61616161  ; # ASCII(a) = 0x61
2     movd      xmm2, eax
3     pshufd    xmm2, xmm2, 0
4
5     mov       eax, 0x65656565  ; # ASCII(e) = 0x65
6     movd      xmm3, eax
7     pshufd    xmm3, xmm3, 0
8     ...
9
10    mov       eax, 0x79797979  ; # ASCII(y) = 0x79
11    movd      xmm7, eax
```

```
12      pshufd    xmm7, xmm7, 0
```

On passe ensuite à la boucle `for`. On lit les 16 octets à partir de `&s[i]` et on les place dans `xmm8`. On recopie ensuite `xmm8` dans `xmm9` à `xmm13` (lignes 5 à 10), puis on passe aux comparaisons.

```
1   .for:
2       cmp       ecx, esi            ; fin de boucle si i >= size
3       jge       .end_for
4
5       movdqu    xmm8, [rdi + rcx]   ; xmm8 = s[i:i+15]
6       movdqu    xmm9, xmm8          ; xmm9 = xmm8
7       movdqu    xmm10, xmm8         ; ...
8       movdqu    xmm11, xmm8
9       movdqu    xmm12, xmm8
10      movdqu    xmm13, xmm8         ; xmm13 = xmm8
11
12      ; 'a'                          ; trouver 'a'
13      pcmpeqb   xmm8, xmm2          ; comparer s[i:i+15] à ['a',...,'a']
14      pmovmskb  ebx, xmm8           ; xmm8.b[i] = 0xFF si s[i] == 'a'
15      popcnt    ebx, ebx            ; compter le nombre de 'a'
16      add       r8, rbx             ; ajouter au compteur de 'a'
17
18      ; 'e'
19      pcmpeqb   xmm9, xmm3
20      pmovmskb  ebx, xmm9
21      popcnt    ebx, ebx
22      add       r9, rbx
23
24      ...
25
26      add       ecx, 16     ; 16 prochains caractères
27      jmp       .for
28  .end_for:
```

On compare `xmm8` qui contient `s[i:i+15]` à `xmm2` qui contient 16 fois le caractère `'a'`. Si `xmm8b[i]` == `xmm2b[i]` alors `xmm8b[i]` prendra la valeur `0xFF`, sinon il prendra la valeur `0x00`. On utilise ensuite l'instruction `pmovmskb` pour remplir le registre `ebx` avec soit 0, soit 1 en fonction de `xmm8b[i]`. Il suffit ensuite de compter le nombre de bits à 1 dans `ebx` qui correspond au nombre de `'a'` dans `s[i:i+15]`. On réitère l'opération pour les autres voyelles.

Enfin on met à jour le tableau `v` avec les valeurs des registres `r8` à `r13` puis on restaure les registres sauvegardés dans la red zone et on quitte le sous-programme.

```
1       mov       [rdx], r8d          ; v[0] = r8d
2       mov       [rdx + 4], r9d      ; v[1] = r9d
3       mov       [rdx + 8], r10d     ; ...
4       mov       [rdx + 12], r11d
5       mov       [rdx + 16], r12d
6       mov       [rdx + 20], r13d    ; r[5] = r13d
```

```
7
8       mov     r12, [rsp - 32]     ; restauration des registres
9       mov     r13, [rsp - 24]
10      mov     rbx, [rsp - 16]
11      ret
```

Amélioration SSE

Pour la version SSE le test s'exécute en $1,59$ secondes.

15.5 Vectorisation avec AVX2

On suit le même principe que pour le SSE mais on va traiter la chaîne par paquets de 32 caractères.

Pour remplir les registres AVX `ymm2` à `ymm7` avec les voyelles, on peut utiliser dans ce cas l'instruction `vpbroadcastd`, comme suit :

```
1       ; remplir le vecteur ymm2 avec [a,a,a,a,...]
2       mov     eax, 0x61616161     ; 4 fois le code ASCII de 'a'
3       movd    xmm2, eax           ; charger dans xmm2.d[0]
4       vpbroadcastd ymm2, xmm2     ; recopier dans ymm2.d[1] à ymm2.d[7]
```

Le code de la boucle `for` est pratiquement identique :

```
1   .for:
2       cmp     ecx, esi            ; fin de boucle si i >= size
3       jge     .end_for
4
5       vmovdqu ymm8, [rdi + rcx]
6       vmovdqa ymm9, ymm8
7       vmovdqa ymm10, ymm8
8       vmovdqa ymm11, ymm8
9       vmovdqa ymm12, ymm8
10      vmovdqa ymm13, ymm8
11
12      ; 'a'
13      vpcmpeqb    ymm8, ymm2      ; comparaison
14      vpmovmskb   ebx, ymm8       ; extraction
15      popcnt      eax, ebx        ; compter les bits = compter les 'a'
16      add         r8d, eax        ; ajouter au compteur de 'a'
17
18          ...
19      add     ecx, 32             ; i += 32
20      jmp     .for
21  .end_for:
```

Amélioration AVX2

Que l'on utilise l'AVX, l'AVX2 ou que l'on ajoute un dépliage par deux, le temps d'exécution est de $0,79$ secondes.

15.6 Vectorisation AVX2 avec intrinsics

Il est nécessaire d'aider le compilateur afin d'optimiser le code en lui fournissant des informations quant à l'utilisation des pointeurs (`restrict`) et l'alignement des données. En fonction du compilateur, nous devons utiliser des fonctions ou des mots-clés différents :

```
#include <xmmintrin.h>
#include <immintrin.h> // AVX
#include <smmintrin.h>

#ifdef __INTEL_COMPILER
void cv_avx2_intrinsics(u8 * restrict s, u32 size, u32 v[6]) {
#else
void cv_avx2_intrinsics(u8 * __restrict__ s, u32 size, u32 v[6]) {
#endif

    u32 i = 0;
#ifdef __INTEL_COMPILER
    __assume_aligned(s, CPU_MEMORY_ALIGNMENT);
    __assume(i%CPU_MEMORY_ALIGNMENT==0);
#endif
#if __GNUC__ > 3
    s = (u8 *) __builtin_assume_aligned(s, CPU_MEMORY_ALIGNMENT);
#endif
```

On déclare ensuite les registres AVX nécessaires au calculs :

- les registres `y2` à `y7` contiennent les voyelles
- les registres `y8` à `y13` sont la copie de `s[i:i+15]`

```
   __m256i y2, y3, y4, y5, y6, y7;

    y2 = _mm256_set1_epi32(0x61616161);
    y3 = _mm256_set1_epi32(0x65656565);
    y4 = _mm256_set1_epi32(0x69696969);
    y5 = _mm256_set1_epi32(0x6F6F6F6F);
    y6 = _mm256_set1_epi32(0x75757575);
    y7 = _mm256_set1_epi32(0x79797979);

    for ( ; i < (size & (~31)); i += 32) {
        __m256i y8, y9, y10, y11, y12, y13;

        y8 = _mm256_loadu_si256( (__m256i *) &s[i]);
```

```
        y9 = y8;
        y10 = y8;
        y11 = y8;
        y12 = y8;
        y13 = y8;

        y8 = _mm256_cmpeq_epi8(y8, y2);
        v[0] += _mm_popcnt_u32( _mm256_movemask_epi8(y8) );

        y9 = _mm256_cmpeq_epi8(y9, y3);
        v[1] += _mm_popcnt_u32( _mm256_movemask_epi8(y9) );

        y10 = _mm256_cmpeq_epi8(y10, y4);
        v[2] += _mm_popcnt_u32( _mm256_movemask_epi8(y10) );

        y11 = _mm256_cmpeq_epi8(y11, y5);
        v[3] += _mm_popcnt_u32( _mm256_movemask_epi8(y11) );

        y12 = _mm256_cmpeq_epi8(y12, y6);
        v[4] += _mm_popcnt_u32( _mm256_movemask_epi8(y12) );

        y13 = _mm256_cmpeq_epi8(y13, y7);
        v[5] += _mm_popcnt_u32( _mm256_movemask_epi8(y13) );

    }
```

Enfin, il reste à traiter les éventuels derniers 31 octets qui peuvent résulter du dépliage par 32 caractères de la boucle :

```
    u32 letters[26];
    memset(letters, 0, 26*sizeof(u32));

    // last iterations
    for ( ; i<size; ++i) {
        ++letters[ s[i] - 'a' ];
    }

    v[0] += letters['a'-'a'];
    v[1] += letters['e'-'a'];
    v[2] += letters['i'-'a'];
    v[3] += letters['o'-'a'];
    v[4] += letters['u'-'a'];
    v[5] += letters['y'-'a'];
```

Cette fonction intrinsics est celle qui est en moyenne la plus performante à l'exécution.

15.7 Vectorisation avec AVX512

Avec l'AVX512 on est en mesure de traiter 64 octets en une seule fois en stockant les données dans les registres `zmm`. Le code est assez conséquent. On commence par déplier la boucle par 64 et on traite les données grâce aux masques après avoir réalisé la comparaison entre registres grâce à `vpcmpeqb` :

```
    ; fill xmm2 vector with 'a,a,a,a,...'
    mov     eax, 0x61616161
    movd    xmm1, eax
    vpbroadcastd zmm1, xmm1     ; AVX512

    ...

    ; unroll by 64
    mov     r14d, esi
    shr     r14d, 6
    test    r14d, r14d
    jz      .last_63

.for_u64:
    vmovdqa32   zmm8, [rdi + rcx]
    add         ecx, 64
    vpcmpeqb    k1, zmm1, zmm8
    vpcmpeqb    k2, zmm2, zmm8
    vpcmpeqb    k3, zmm3, zmm8
    vpcmpeqb    k4, zmm4, zmm8
    vpcmpeqb    k5, zmm5, zmm8
    vpcmpeqb    k6, zmm6, zmm8

    ; 'a'
    kmovq       rbx, k1
    popcnt      rbx, rbx
    add         r8d, ebx

    ...

    dec     r14d
    jnz     .for_u64

    ...
```

Reste ensuite à traiter les 63 derniers octets potentiels. S'il y a plus de 32 octets à traiter, on traitera les 32 premiers octets en utilisant les registres AVX, puis s'il reste plus de 16 octets à traiter on utiliser les registres SSE. Enfin, pour les 15 derniers caractères on utilise la méthode du tableau (voir Section 15.2).

15.8 Résultats

Nous donnons Table 15.3 les résultats comparatifs des méthodes que nous avons évoquées pour un Intel i7 4900MQ ainsi que l'amélioration par rapport à la méthode de référence qui correspond à l'implantation avec le `if`.

Méthode	Temps (s)	Amélioration
`if`	13,22	1
`switch`	17,22	$\times \simeq 0.8$
tableau	2,79	$\times \simeq 4.7$
vectorisation SSE	1,59	$\times \simeq 8.3$
vectorisation AVX2	0,79	$\times \simeq 16.7$

TABLE 15.3 – Résultats comparatifs des méthodes pour compter les voyelles

15.8.1 Architectures anciennes (avant 2015)

Pour certaines de ces architectures, l'AVX n'est pas disponible on se contente donc du SSE pour la vectorisation. Nous avons rapporter les temps d'exécution des méthodes suivantes :

1. implantation en langage C utilisant un `if`
2. implantation en langage C utilisant un `switch`
3. implantation en langage C utilisant un tableau
4. implantation assembleur de la méthode avec tableau
5. implantation assembleur de la méthode avec tableau avec dépliage de la boucle par 4, version 1, on charge chaque octet dans `eax`
6. implantation assembleur de la méthode avec tableau avec dépliage de la boucle par 4, version 2, on charge quatre octets dans `eax` en une seule fois puis on les répartit dans `r8`, `r9`, `r10`
7. dépliage par 8 de la version 5
8. dépliage par 8 de la version 6
9. dépliage par 8 de la version 5 mais en supprimant les dépendances sur `eax`
10. vectorisation en assembleur avec jeu d'instructions SSE2
11. vectorisation en assembleur avec jeu d'instructions AVX
12. vectorisation en assembleur avec jeu d'instructions AVX2 (version 1) avec dépliage par 2 de la boucle principale
13. vectorisation en assembleur avec jeu d'instructions AVX2 (version 3) avec élimintaion des dépendances

n°	Méthode	Intel Pentium D 925 2006	Intel Core 2 Q9300 2008	Intel i7 860 2009	AMD X6 1090T 2010	Intel i5 3570K 2012	Intel i7 4790 2014
1	C if	79.19	62.44	43.63	39.43	38.04	27.49
2	C switch	126.15	102.51	57.28	47.88	54.41	39.72
3	C tableau	**12.57**	14.80	29.91	21.91	8.37	6.36
4	tableau asm	32.95	16.02	26.16	18.35	8.37	6.59
5	tableau asm ur4 v1	37.17	17.54	29.89	21.92	8.53	6.69
6	tableau asm ur4 v2	19.26	19.78	13.16	8.51	11.55	8.93
7	tableau asm ur8 v1	32.48	16.61	27.50	18.38	7.37	5.64
8	tableau asm ur8 v2	19.25	21.60	13.04	8.37	11.48	8.89
9	tableau asm ur8 v3	32.49	16.64	27.40	18.44	7.34	5.65
10	SSE2	17.58	**8.95**	**3.59**	**4.27**	**4.25**	3.61
11	AVX	-	-	-	-	-	1.82
12	AVX2 v1	-	-	-	-	-	1.82
13	AVX2 v3	-	-	-	-	-	1.82
14	AVX2 v2 ur8	-	-	-	-	-	1.82
15	AVX2 intrinsics	-	-	-	-	-	**1.14**
	ratio (1 / 10)	4.50	6.97	12.15	9.23	8.95	7.61
	ratio (1 / 15)	-	-	-	-	-	24.11

TABLE 15.4 – Architectures anciennes : temps d'exécution en secondes pour 100_000 itérations sur des chaînes de 256_000 caractères.

14. implantation en langage C et intrinsics AVX2

Concernant les architectures anciennes (voir Table 15.4), on note que l'utilisation du `switch` (méthode 2) est pénalisante car elle dégrade les performances par rapport à la méthode de référence. L'utilisation d'un tableau pour compter les lettres (méthode 3) permet de fortement diminuer le temps d'exécution.

La traduction en assembleur de la méthode utilisant un tableau suscite quelques commentaires. On en donne deux versions : la première estampillée `v1` traite chaque octet dans `eax` et la seconde `v2` commence par charger quatre octets consécutifs dans `eax` puis les répartit dans `r8`, `r9`, `r10` par décalage de `eax`, puis on effectue la conversion et l'incrémentation du nombre d'occurrences de la lettre correspondante en utilisant ces registres.

Pour les architectures anciennes c'est la méthode `v2` qui est la plus performante (Pentium D, Q9300, i7 860, X6 1090T), puis à partir des Intel i5 3570k et i7 4790, c'est la méthode `v1` qui prend le dessus. On note également que le dépliage par 4 ou

par 8 de la boucle n'a que peu d'influence de manière générale sur la performance.

Cependant, c'est l'utilisation du SSE (méthode 10) qui apporte une amélioration conséquente ainsi que l'utilisation de l'AVX pour l'Intel i7 4790 (méthodes 11 à 15). Le compilateur C est d'ailleurs en mesure de produire un code bien plus optimisé que celui écrit en assembleur (méthode 9).

15.8.2 Architectures modernes (2015 à 2019)

Pour les architecture modernes (Table 15.5), la tendance observée pour les achitectures anciennes se confirme. L'utilisation de l'AVX et notamment sous sa forme intrinsics (méthode 15) produit généralement les meilleurs résultats.

On note que l'Intel i3 6100, l'i5 7400 et le Xeon Silver 4208 sont bien moins performants que leurs concurrents pour la méthode 1. Autre fait notable, les processeurs AMD sont bien moins performants que les processeurs Intel quand on passe à l'utilisation du `switch` (méthode 2). Sur l'AMD Ryzen 7 1700X, on passe de 31 s à 56 s soit une augmentation de 80 % proche des 84 % d'augmentation du AMD Ryzen 5 3600. Elle est inférieure à 30 % pour les processeurs Intel. On peut donc supposer que la prédiction de branchement est moins bonne sur les processeurs AMD que sur les processeurs Intel.

Pour les méthodes 5 à 9, on observe le fait que la version 1 est plus performante que la version 2 et que le dépliage n'apporte qu'une très légère amélioration. On observe cependant pour l'AMD Ryzen 7 1700X un comportement assez étrange.

La vectorisation avec SSE ou AVX est plus performante que sur les architectures anciennes. Pour l'AVX on obtient un facteur moyen d'amélioration d'environ 31 sur toutes les architectures. Par contre, l'utilisation du SSE est plus intéressante sur les processeurs AMD de type Ryzen que sur les processeurs Intel.

Le cas du Xeon Silver est particulier car il fait partie d'un cluster pour lequel on ne dispose que de `gcc` 8.4. On peut donc légitimement se demander si le codage assembleur réalisé par le compilateur `gcc` est aussi efficace que dans la version 10. L'utilisation de l'AVX2 permet d'obtenir une amélioration d'un facteur de près de 33 par rapport à la méthode de référence.

Amélioration AVX512

L'utilisation de l'AVX512 (cf. Table 15.6) sous forme intrinsics (voir le code du projet) permet alors d'atteindre un temps d'exécution de 0,76 secondes, soit un facteur d'amélioration d'environ 60.
Différentes implantations AVX512 ont été réalisées et la plus efficace est la version écrite en assembleur avec un dépliage de la boucle par 8. Elle permet d'atteindre 0,62 secondes, soit un facteur d'amélioration de 73.

N°	Marque Gamme Modèle	Intel Core i3 6100 2015	AMD Ryzen 7 1700X 2017	Intel Core i5 7400 2017	Intel Core i7 8700 2017	AMD Ryzen 5 3600 2019	Intel Xeon 4208 2019
1	C if	35.89	31.33	38.19	29.33	29.95	45.47
2	C switch	47.23	56.33	50.35	39.72	55.31	55.75
3	C tableau	6.12	4.26	6.51	5.46	4.94	7.83
4	tableau asm	7.06	11.26	7.52	5.78	5.93	8.84
5	tableau asm ur4 v1	6.11	16.91	6.51	5.05	5.03	7.55
6	tableau asm ur4 v2	10.28	6.76	10.98	8.34	5.05	12.44
7	tableau asm ur8 v1	5.96	4.18	6.32	4.72	4.84	7.26
8	tableau asm ur8 v2	10.20	4.44	10.87	8.28	4.58	12.44
9	tableau asm ur8 v3	5.87	4.21	6.23	4.76	4.92	7.19
10	SSE2	3.89	2.12	4.16	3.16	2.08	4.86
11	AVX	1.95	1.20	2.08	1.58	1.07	2.41
12	AVX2 v1	1.95	1.19	2.09	1.58	1.08	2.43
13	AVX2 v3	1.95	1.17	2.08	1.58	**0.91**	2.42
14	AVX2 v2 ur8	1.96	**0.94**	2.08	1.58	1.00	2.44
15	AVX2 intrinsics	**1.13**	0.98	**1.20**	**0.91**	1.08	**1.36**
	ratio (1 / 10)	9.22	14.77	9.18	9.28	14.39	9.35
	ratio (1 / 15)	31.76	31.96	31.82	32.23	29.95	33.43

TABLE 15.5 – Architectures modernes : temps d'exécution en secondes pour 100_000 itérations sur des chaînes de 256_000 caractères.

Méthode	Temps (s)
AVX512 asm v1	2.56
AVX512 asm v2	1.45
AVX512 asm v2 (dépliage par 8)	1.22
AVX512 C intrinsics	1.56

TABLE 15.6 – Intel Xeon Silver 4208 et AVX512 : temps d'exécution en secondes pour 100_000 itérations sur des chaînes de 256_000 caractères.

15.8.3 Architectures récentes (2020 et après)

N°	Marque Gamme Modèle	Intel Core i7 10850H 2020	AMD Ryzen 5 5600g 2021	Intel Core i5 12400f 2022
1	C if	24.37	23.87	27.97
2	C switch	35.14	43.75	35.87
3	C tableau	4.71	3.90	3.66
4	tableau asm	5.42	4.02	4.93
5	tableau asm ur4 v1	4.72	3.97	4.19
6	tableau asm ur4 v2	7.78	4.07	7.17
7	tableau asm ur8 v1	4.52	3.55	3.52
8	tableau asm ur8 v2	7.74	4.23	7.61
9	tableau asm ur8 v3	4.45	3.50	3.48
10	SSE2	2.97	1.81	1.64
11	AVX	1.52	0.90	1.64
12	AVX2 v1	1.50	0.91	1.61
13	AVX2 v3	1.53	0.90	0.85
14	AVX2 v2 ur8	1.55	0.89	0.85
15	AVX2 intrinsics	**0.92**	**0.84**	**0.84**
	ratio (1 / 10)	8.20	13.18	17.05
	ratio (1 / 15)	26.48	28.41	33.29

TABLE 15.7 – Architectures récentes : temps d'exécution en secondes pour 100_000 itérations sur des chaînes de 256_000 caractères.

Pour les architectures récentes dont les résultats figurent Table 15.7, la méthode 15 (version AVX2 intrinsics) est la plus efficace. On observe que les méthodes 6 et 8 donnent de mauvais résultats sur les microprocesseurs Intel. C'est aussi le cas sur les microprocesseurs AMD mais de manière moins significative.

Si le traitement initial durait une heure (méthode 1), le fait de passer à une version vectorisée (méthode 15) sur un AMD Ryzen 5 5600g, permettrait d'abaisser le temps d'exécution à $3600/28,41 = 126,71$, soit un peu plus de deux minutes. Cela représenterait une amélioration drastique.

15.8.4 Influence du nombre de voyelles

On notera également que le nombre de voyelles influe sur le temps d'exécution. Nous avons réalisé une étude simple qui consiste à faire varier le pourcentage de

voyelles que contient la chaine pour laquelle on compte les voyelles. On fait alors varier ce pourcentage de 10 à 100 par pas de 10. Les résultats obtenus sur un AMD Ryzen 5 3600 sont présentés Table 15.8. On travaille toujours sur une chaîne de 256_000 caractères et on réalise ici 100_000 fois le calcul.

Pourcentage	Méthode 1	Méthode 2	Méthode 3
10	43.83	83.65	9.41
20	57.17	110.53	9.46
30	70.17	127.40	9.51
40	83.74	140.92	9.52
50	95.87	154.01	9.45
60	108.27	162.11	9.39
70	121.03	167.08	9.44
80	133.43	169.29	9.68
90	137.77	169.07	10.01
100	134.95	165.68	10.43

TABLE 15.8 – Influence du pourcentage du nombre de voyelles sur le temps d'exécution : AMD Ryzen 5 3600

Pour l'ensemble des méthodes, le temps d'exécution augmente à mesure que le nombre de voyelles augmente. Cela paraît normal car au début, avec par exemple 10 % de voyelles, le code le plus souvent exécuté est celui lié aux consonnes. A mesure que l'on augmente le nombre de voyelles on exécute moins souvent ce code et plus souvent le code lié aux différentes voyelles. On est face à un problème lié à la prédiction de branchement. Cependant, pour la méthode 3 qui utilise un tableau, on ne devrait pas voir le temps augmenter

15.9 Conclusion

Comme le montre cette étude de cas, un traitement banal, peut être, s'il est mal implanté, source de grands ralentissments pour le reste d'un programme. La vectorisation apporte un facteur d'amélioration important en raison, d'une part, du traitement de plusieurs octets simultanément, et d'autre part, de l'élimination du `if`.

Cet exemple est très emblématique car le compilateur est incapable de vectoriser le code. Or, c'est la vectorisation qui donne un gain substantiel, la version intrinsics n'étant que la retraduction en C de la version assembleur. Passer du SSE à l'AVX, puis à l'AVX512 permet à chaque fois de diminuer le temps d'exécution.

Chapitre 16

Etude de cas
Suite de Fibonacci

16.1 Introduction

Ce chapitre traite de l'implantation de fonctions qui permettent de calculer les termes de la suite de *Fibonacci*. *Léonardo Fibonacci* était un mathématicien italien (1175 - 1250) qui a contribué notamment à démocratiser la numérotation indo-arabe. Il semblerait que la suite ait été découverte par des mathématiciens indiens (*Gopala* 1133, *Hemachandra* 1150, *Fibonacci* 1202) et qu'elle fut ensuite attribuée à Fibonacci dans le monde occidental. La suite qui prend donc son nom est une suite d'entiers naturels construite en calculant la somme des deux termes précédents et est définie de manière récursive comme suit :

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

Les premiers termes de la suite sont donc : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181.

La suite de Fibonacci possède de nombreuses propriétés mathématiques singulières ainsi que des ramifications dans le domaine du vivant (choux romanesco, pomme de pin), ou de la dynamique des populations. La suite de Fibonacci est également liée au nombre d'or ϕ qui s'exprime par $(1 + \sqrt{5})/2 \simeq 1,618033$. Le nombre d'or est qualifié de *divine proportion* car nombre de choses dans la nature sont liées à cette valeur. A mesure que l'on calcule les termes de la suite de Fibonacci, le ratio F_{n+1}/F_n tend vers ϕ.

16.1.1 Dynamique des populations

On considére des couples de lapins qui sont à maturité sexuelle, que l'on notera S, et qui peuvent se reproduire pour générer un nouveau couple de lapins qui, lui, n'est pas à maturité sexuelle et que l'on notera N. Les couples qui ne sont pas à maturité sexuelle doivent attendre avant de parvenir à maturité et pourrons alors se reproduire. L'évolution d'une population est donc la suivante :

- au temps $t = 0$, il n'y a aucun couple
- au temps $t = 1$, on a un couple qui n'est pas à maturité sexuelle
- au temps $t = 2$, on a un couple parvenu à maturité sexuelle
- au temps $t = 3$, on a deux couples : un couple à maturité sexuelle, un couple issu de la reproduction du couple à maturité et qui n'est donc pas à maturité sexuelle
- au temps $t = 4$, on a trois couples : deux couples à maturité sexuelle et un couple non mature
- etc

On a reproduit l'évolution de la population Table 16.1.

Temps	Couples	Nbr. Couples
$t = 0$	$\emptyset$	0
$t = 1$	N	1
$t = 2$	S	1
$t = 3$	SN	2
$t = 4$	SNS	3
$t = 5$	$SNSSN$	5
$t = 6$	$SNSSNSNS$	8

TABLE 16.1 – Evolution d'une population de lapins selon Fibonacci

Du point de vue de l'informatique, on peut voir cette évolution comme un système de réécriture :

$$\begin{aligned} N &\longrightarrow S \\ S &\longrightarrow SN \end{aligned}$$

Du point de vue de la biologie, la suite de Fibonacci est en rapport avec la phyllotaxie des plantes, c'est à dire, l'organisation en spirale des organes autour d'une tige. On remarque que pour un nombre important de plantes, estimé à 90%, le nombre d'organes dans une spirale suit très souvent une progression proche des rapports de la séquence de Fibonacci. Il en résulte que ces organes émergent

souvent à des angles de $137,5$ degrés ce qui permet à chaque organe de recevoir une quantité optimale de lumière du soleil en évitant les chevauchements ce qui conduit à favoriser la photosynthèse.

Nous n'entrerons pas dans de plus amples considérations puisque ce qui nous intéresse est l'implantation de cette fonction sous une forme particulière. Pour terminer, nous dirons que la suite de Fibonacci fait partie de l'encyclopédie en ligne des suites de nombres entiers OEIS (*On-Line Encyclopedia of Integer Sequences*). Elle est présente sous l'identifiant A000045. En informatique, la suite de Fibonacci possède des applications liées à la génération des nombres aléatoires, aux arbres AVL [1] (qui sont des arbres de recherche dits automatiquement équilibrés) ou aux structures de données du même nom (*Fibonacci Heap*).

16.2 Récursivité

Le code de la fonction de Fibonacci récursive est donné Listing 16.2.1.

```
1 u32 fib_rec( u32 n ) {
2     if (n <= 1) {
3         return n;
4     } else {
5         return fib_rec(n-1) + fib_rec(n-2);
6     }
7 }
```

Listing 16.2.1 – Fibonacci - fonction récursive

Si on réalise quelques tests de performance, on s'aperçoit rapidement que le nombre d'appels récursifs devient prohibitif (voir Table 16.2, ci-après) et la fonction devient de moins en moins efficace. Par exemple sur un AMD Ryzen 5 5600g dont la fréquence de fonctionnement en mode *boost* est de 4440 MHz, F_{44}, F_{45} et F_{46} s'exécutent respectivement en 1.685, 2.559 et 4.365 secondes.

Comme on peut s'en rendre compte Table 16.2, colonne `fib_rec`, le nombre d'appels récursifs est proportionnel à ϕ, le nombre d'or. Il tend vers $1,447214 \times \phi^n$ à mesure que n tend vers ∞. Notons que $1,447214 \simeq 1 + 1/\sqrt{5}$.

Il existe cependant une autre implantation récursive mais qui se base sur les propriétés suivantes. Elle permet de *casser* la complexité initiale du problème. Le code correspondant est donné Listing 16.2.2 :

$$F_n = \begin{cases} \text{si } n \text{ est pair}, & k = n/2, \quad F_n = (2 \times F_{k-1} + F_k) \times F_k \\ \text{si } n \text{ est impair}, & k = (n+1)/2, \quad F_n = (F_{k-1}^2 + F_k^2) \end{cases}$$

1. Adelson-Velsky and Landis

F_i	fib_rec	fib_rec_improved
F_{10}	177	15
F_{20}	21891	31
F_{25}	242785	41
F_{30}	2692537	45
F_{35}	29860703	55
F_{40}	331160281	63
F_{45}	3672623805	75

TABLE 16.2 – Nombre d'appels des fonctions récursives de Fibonacci

A chaque étape on calcule $k = n/2$, puis il faut évaluer F_k et F_{k-1}. On termine la récursion dès que n vaut 0 ou 1.

```
u32 fib_rec_improved( u32 n ) {
    if (n <= 1) return n;
    if (n % 2 == 1) {
        int k = (n+1) >> 1;
        int f1 = fib_rec_improved( k - 1 );
        int f2 = fib_rec_improved( k );
        return (f1 * f1 + f2 * f2);

    } else {
        int k = n >> 1;
        int f1 = fib_rec_improved( k - 1 );
        int f2 = fib_rec_improved( k );
        return (2 * f1 + f2) * f2;

    }
}
```

Listing 16.2.2 – Fibonacci - fonction récursive améliorée

Pour la version récursive améliorée, le nombre d'appels récursifs est nettement plus petit et d'une complexité en $O(n)$ comme on peut le constater Table 16.2, colonne `fib_rec_improved`.

Nous donnons, Figure 16.1, un exemple du calcul de F_{13}, en utilisant la formule de réccurrence améliorée.

16.3 Formule avec nombres flottants

On peut calculer F_n en utilisant l'arrondi de l'expression suivante :

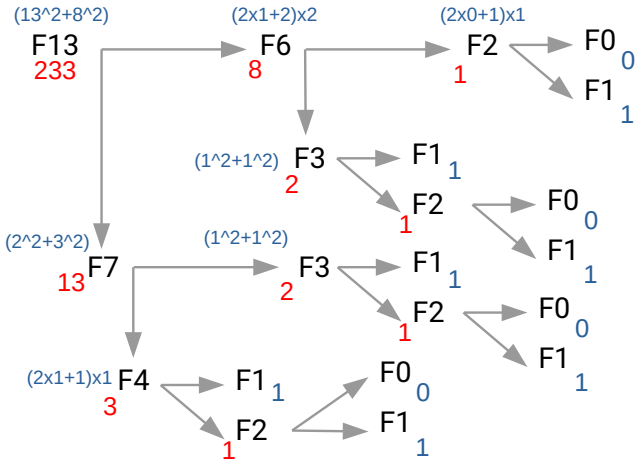


FIGURE 16.1 – Calcul de F_{13} avec la version récursive améliorée.

$$F_n \simeq \phi^n \times \frac{1}{\sqrt{5}}$$

Le temps de calcul du test que nous réaliserons et qui est décrit ci-après est de l'ordre de $22,58$ secondes. Nous ne nous intéresserons donc pas aux temps de calculs obtenus grâce à cette formule même s'il sont parfois inférieurs à d'autres implantations notamment la traduction directe (sans optimisation) en assembleur de la fonction de référence qui s'exécute en plus de 50 secondes.

16.4 Version de référence en C

La version de référence à laquelle nous allons nous intéresser (voir Listing 16.4.1), est une variante de la version récursive. Elle est à la fois récursive puisqu'elle s'appelle, mais également linéaire puisque seulement n appels seront réalisés.

Elle prend en paramètres la valeur du nombre à calculer `n` ainsi que F_0 et F_1 représentés respectivement par `f0` et `f1`. L'appel est réalisé en prenant `f0` $= 0$ et `f1` $= 1$.

Cette fonction de référence est traduite par le compilateur `g++` en utilisant les options de compilation suivantes :

- `-O3` (Optimisation niveau 3)
- `-funroll-loops` (dépliage de boucle)

Le compilateur parvient à éliminer la récursivité et produit une version très performante. Il transforme la récursivité en une boucle `while` avec un dépliage par 8. De plus, il utilise `esp` plutôt que `ebp` pour récupérer les paramètres de la fonction.

```
1  u32 fib_ref( u32 f0, u32 f1, u32 n ) {
2      if (n == 0) {
3          return f0;
4      } else {
5          return fib_ref( f1, f0 + f1, n - 1 );
6      }
7  }
8
9  u32 r = fib_ref( 0, 1, n);
```

Listing 16.4.1 – Fibonacci - fonction de référence

Temps de référence

Le test de référence consiste à réaliser 500_000_000 fois le calcul de F_{43}. Les tests sont réalisés sur un AMD Ryzen 5 5600g. Pour l'implantation par le compilateur g++, l'exécution dure environ $2,853$ secondes, ce qui représente un temps d'exécution très performant qui va se révéler difficile à battre, mais à cœur vaillant, rien d'impossible.

16.5 Versions assembleur de la fonction de référence

La traduction de la fonction de référence en assembleur est facile à réaliser. Un rapide examen permet de déterminer qu'il est intéressant de charger `f0` dans `eax` puisque c'est la valeur qui sera retournée dans le cas où la variable `n` est égale à 0. On utilise également `edx` pour stocker `f1` et `ecx` pour `n` (voir Listing 16.5.1). Ainsi, nous n'utilisons que des registres modifiables d'après les conventions d'appel en 32 bits.

On peut mettre en commentaire la ligne 19 qui remonte le sommet de pile en libérant les paramètres passés lors de l'appel récursif car la ligne 23 rétablit `esp` à sa valeur d'origine.

Version assembleur de la fonction de référence

La version de référence implantée en assembleur (il s'agit d'une traduction directe) s'exécute en $50,630$ secondes ce qui est énorme par rapport à la version optimisée par le compilateur.

On peut donc se demander si c'est l'accès mémoire qui est la cause de ce ralentissement (cf. partie résultats pour l'explication) .

```
fib_v1:
    push    ebp
    mov     ebp, esp

    mov     eax, [ebp +  8] ; eax <- f0
    mov     edx, [ebp + 12] ; edx <- f1
    mov     ecx, [ebp + 16] ; ecx <- n
    test    ecx, ecx
    jz      .endif

    dec     ecx
    push    ecx             ; n-1

    add     eax, edx        ; eax <- f0+f1
    push    eax

    push    edx             ; f1
    call    fib_v1
    ; add esp    12

.endif:

    mov     esp, ebp
    pop     ebp
    ret
```

Listing 16.5.1 – Fibonacci - fonction de référence en assembleur

16.6 Versions axées sur les tableaux

Pour calculer les nombres de la suite de Fibonacci, on peut utiliser un tableau de $n+1$ entiers et on commence par remplir les deux premiers éléments par les valeurs 0 et 1 qui correspondent à F_0 et F_1. Chaque élément d'indice `i` du tableau correspond à la valeur de F_i. On applique ensuite la formule de récurrence sur les éléments du tableau. Le Listing 16.6.1 montre comment procéder.

Version tableau

La version basée sur un tableau dynamique alloué à chaque appel de la fonction prend $13,790$ secondes pour s'exécuter. Alors que si on utilise un tableau statique le temps d'exécution passe à $6,380$ secondes.

La version avec tableau dynamique est moins performante car on fait de nombreux appels aux fonctions système `malloc` et `free`. Le temps est doublé par rapport à la fonction avec tableau statique.

```
u32 fib_array( u32 f0, u32 f1, u32 n ) {

    if (n <= 1) return n;
    data[ 0 ] = f0;
    data[ 1 ] = f1;

    u32 i = 2;
    while (i <= n) {
        data[ i ] = data[ i-1 ] + data[ i-2 ];
        ++i;
    }

    return data[ i-1 ];

}
```

Listing 16.6.1 – Fibonacci - fonction itérative avec tableau

16.7 Versions itératives

Nous allons à présent nous concentrer sur des versions itératives et remplacer l'appel récursif par une boucle `while` mais nous allons nous passer de l'utilisation d'un tableau pour stocker les calculs intermédiaires. Ces calculs seront réalisés dans les registres `eax` et `edx` principalement. Le code C correspondant à cette modification figure Listing 16.7.1.

```
u32 fib_iterative( u32 f0, u32 f1, u32 n ) {
    while (n != 0) {
        int tmp = f1;
        f1 += f0;
        f0 = f1;
        --n;
    }
    return f0;
}
```

Listing 16.7.1 – Fibonacci - fonction itérative avec boucle while

Si `n` est égal à 0, on retourne `f0`, donc 0. Sinon on calcule F_1, puis F_2, jusqu'à F_n. La variable `f0` devient `f1` et `f1` devient `f1 + f0`, c'est à dire F_2. Pour faire ce transfert de valeurs il est nécessaire d'utiliser une variable temporaire `tmp`.

On peut traduire ce code directement en assembleur (voir Listing 16.7.2) en utilisant le registre `ebx` pour stocker temporairement la valeur de `f1`. On se doit alors de sauvegarder `ebx` dans la pile, puis de le restaurer par la suite.

Chaque itération de la boucle `while` est alors composée de trois instructions assembleur (lignes 14 à 16) du Listing 16.7.2.

```asm
fib_v3:
    push    ebp
    mov     ebp, esp
    push    ebx

    mov     eax, [ebp +  8] ; f0
    mov     edx, [ebp + 12] ; f1
    mov     ecx, [ebp + 16] ; n

    test    ecx, ecx
    jz      .end

.while:
    mov     ebx, edx        ; tmp = f1
    add     edx, eax        ; f1 = f1 + f0  => f2, f3, ...
    mov     eax, ebx        ; f0 = tmp        => f1, f2, ...

    dec     ecx             ; --n
    jnz     .while
.endwhile:

.end:
    pop     ebx
    mov     esp, ebp
    pop     ebp
    ret
```

Listing 16.7.2 – Fibonacci - fonction itérative avec while en assembleur

Version itérative avec while

La version itérative avec `while` ne prend alors plus que $6,30$ secondes pour s'exécuter. Il s'agit d'une bonne amélioration mais qui reste encore loin de la version traduite par le compilateur. On peut alors déplier la boucle par 2, 4 ou 8. On obtient les temps d'exécution suivants :

- dépliage par 2 : $5,26$ s
- dépliage par 4 : $3,54$ s
- dépliage par 8 : $3,47$ s

Les dépliages par un facteur 4 ou 8 semblent donc les plus performants.

16.7.1 Astuce

On peut s'apercevoir que le corps de la boucle `while` peut être optimisé. Au lieu de coder :

```
1     mov     ebx, edx
2     add     edx, eax
3     mov     eax, ebx
```

on peut utiliser les deux instructions suivantes :

```
1     add     eax, edx    ; eax = f0 + f1, edx = f1
2     xchg    eax, edx    ; échange des registres
```

Initialement **eax** contient F_0 et **edx** F_1. La première addition revient à mettre F_2 dans **eax**, puis on échange les valeurs contenues dans les registres. Au final **eax** contient F_1 et **edx** contient F_2.

On gagne alors une instruction et on n'est pas forcé d'utiliser un registre comme **ebx** qui nécessitait d'être préservé.

Version itérative avec while et astuce

Malheureusement, cette amélioration ne s'avère pas forcément très bénéfique pour notre traitement par rapport à la solution précédente puisqu'elle donne à peu près les mêmes temps d'exécution, sauf pour le dépliage par 2.

- sans dépliage : $6,304$ s
- dépliage par 2 : $3,608$
- dépliage par 4 : $3,500$
- dépliage par 8 : $3,497$

16.7.2 Amélioration lors du dépliage

Lors du dépliage, on répète plusieurs fois le code du corps de la boucle **while**, ce qui va représenter 4 instructions avec le code de base ou 2 instructions avec l'astuce évoquée précédemment.

En fait, on peut trouver une amélioration qui consiste à écrire :

```
1     add     eax, edx
2     add     edx, eax
```

En effet, si on considère que **eax** contient F_i et **edx** F_{i+1} alors le résultat de la première instruction d'addition est F_{i+2} dans **eax**, puis F_{i+3} dans **edx**

Instruction / Registre	eax	edx
initialement	F_i	F_{i+1}
`add eax, edx`	F_{i+2}	F_{i+1}
`add eax, edx`	F_{i+2}	F_{i+3}

TABLE 16.3 – Astuce dépliage par 2

Version itérative avec while et amélioration du dépliage

En utilisant cette amélioration, on diminue le temps d'exécution :

- dépliage par 2 : $3,755$
- dépliage par 4 : $3,544$
- dépliage par 8 : $2,453$

Le code correspondant est donné Listing 16.7.3 pour un dépliage par 8.

On commence par définir deux macro-instructions afin de ne pas réécrire le code entièrement. La première nommée `swap_1` calcule F_{i+1} alors que la seconde `swap_2` calcule F_{i+2}.

On charge les paramètres dans les registres (lignes 17 à 19), puis on teste si `n` est égal à 0. Dans ce cas on sort de la fonction avec `eax` qui contient F_0.

On calcule ensuite le nombre d'itérations du dépliage par 8 (ligne 24). Si celui-ci est égal à 0, c'est que la valeur de `n` est comprise entre 1 et 7. On se déplace donc (ligne 25) vers une boucle `while` qui traite ces dernières itérations.

Après la boucle de dépliage par 8 (Lignes 27 à 32), on recharge dans `ecx` la valeur de `n` et on recalcule le nombre d'itérations restantes (lignes 35 et 36). Si ce nombre est égal à 0, on sort de la fonction car `eax` contient F_n. Sinon on traite comme indiqué précédemment les dernières itérations.

16.7.3 Amélioration des dernières itérations

Le dépliage par 8 semble être le plus efficace, mais lors des dernières itérations qui peuvent varier entre 1 et 7, il est préférable d'utiliser un `switch` afin d'améliorer l'efficacité de la fonction plutôt que faire une boucle `while`. La partie de code qui diffère est présentée ci-après.

```
1 align 16
2 switch_jumps_table: dd ..@case_0, ..@case_1, ..@case_2
3                     dd ..@case_3, ..@case_4, ..@case_5
4                     dd ..@case_6, ..@case_7
5 ; code précédent
6 ; ....
```

```
%macro swap_1 0
                            ; eax=f0        edx=f1
    add     eax, edx        ; eax=f2=f0+f1
    xchg    eax, edx        ; eax=f1        edx=f2
%endmacro

%macro swap_2 0
                            ; eax=f0        edx=f1
    add     eax, edx        ; eax=f0+f1=f2
    add     edx, eax        ; edx=f2+f1=f3
%endmacro

fib_v13:
    push    ebp
    mov     ebp, esp

    mov     eax, [ebp +  8] ; f0
    mov     edx, [ebp + 12] ; f1
    mov     ecx, [ebp + 16] ; n

    test    ecx, ecx
    jz      .end

    shr     ecx, 3          ; / 8
    jz      .last_7

.while_ur8:     ; eax=f0   edx=f1
    swap_2      ; eax=f2   edx=f3
    swap_2      ; eax=f4   edx=f5
    swap_2      ; eax=f6   edx=f7
    swap_2      ; eax=f8   edx=f9
    loop    .while_ur8

.last_7:
    mov     ecx, [ebp + 16]
    and     ecx, 7
    jz      .end

.while:
    swap_1
    loop    .while

.end:
    mov     esp, ebp
    pop     ebp
    ret
```

Listing 16.7.3 – Fibonacci - fonction itérative avec while et amélioration du dépliage

```
.while_ur8:
    swap_2
    swap_2
```

```
10      swap_2
11      swap_2
12      loop    .while_ur8
13
14  .last_7:
15      mov     ecx, [ebp + 16] ; n
16      and     ecx, 7
17      jmp     [switch_jumps_table + ecx * 4]
18
19  align 16    ; switch
20  ..@case_7:  swap_1
21  ..@case_6:  swap_1
22  ..@case_5:  swap_1
23  ..@case_4:  swap_1
24  ..@case_3:  swap_1
25  ..@case_2:  swap_1
26  ..@case_1:  swap_1
27  ..@case_0:
28
29  .end:
```

Il est nécessaire de définir une table d'adresses qui correspond aux différents `case` (cf. Section 5.4.10).

Version itérative avec while, amélioration du dépliage et switch

En ajoutant un `switch`, on obtient un temps d'exécution de $2,58$ secondes donc un peu moins performant que la version précédente.

16.7.4 Amélioration avec `esp`

Plutôt que d'utiliser `ebp` pour récupérer les arguments de la fonction, on utilise directement `esp`. Le premier paramètre `f0` est donc en `[esp+4]` car `[esp]` contient l'adresse de retour du sous-programme.

```
1   fib:
2
3       mov     eax, [esp +  4] ; f0
4       mov     edx, [esp +  8] ; f1
5       mov     ecx, [esp + 12] ; n
6
7       test    ecx, ecx
8       jz      .end
9
10      shr     ecx, 3          ; / 8
11      jz      .last_7
12
13      ; suite de la fonction
14      ; ...
```

Version itérative avec while, amélioration du dépliage, switch, esp

En éliminant tout ce qui touche à **ebp**, notamment l'entrée et la sortie de la fonction, on exécute la fonction en $2,298$ secondes.

16.7.5 Amélioration du dépliage par 8

Lors du dépliage par 8, lorsque l'on exécute 4 fois les instructions

```
1     add     eax, edx
2     add     edx, eax
```

on génère de nombreuses dépendances. On peut alors tenter de supprimer des dépendances en utilisant l'instruction **lea** comme sur le code suivant :

```
1  .while_ur8:
2                                        ;   eax       edx       ebx
3                                        ;    x
4                                        ;              y         ?
5     add     eax, edx                   ;   x+y        y
6     add     edx, eax                   ;   x+y       x+2y
7
8     lea     ebx, [eax + edx]           ;   x+y       x+2y     2x+3y
9     lea     eax, [ebx + edx]           ;  3x+5y      x+2y     2x+3y
10
11    lea     edx, [eax + ebx]           ;  3x+5y      5x+8y    2x+3y
12    lea     ebx, [eax + edx]           ;  3x+5y      5x+8y    8x+13y
13
14    lea     eax, [ebx + edx]           ; 13x+21y     5x+8y    8x+13y
15
16    lea     edx, [eax + ebx]           ; 13x+21y    21x+34y   8x+13y
17
18    loop    .while_ur8
```

Version itérative avec while, amélioration du calcul interne, switch, esp

Le test d'efficacité ne met alors plus que $2,229$ secondes pour s'exécuter. Même si on a encore des dépendances entre les registres, il se peut que l'instruction **lea** soit plus performante que **add**.

16.8 Versions vectorielles

16.8.1 Version SSE

On peut reprendre le même principe que la boucle `while` en utilisant les registres vectoriels. On commence par charger dans le registre SSE `xmm0` les premières valeurs de la suite de Fibonacci et on fait de même avec `xmm1` avec un décalage d'un élement. Il s'agit des tableaux de données `sse_vect0` et `sse_vect1` du Listing 16.8.1. On n'aura donc plus besoin des paramètres `f0` et `f1`.

Il est généralement préférable d'aligner ces données sur une adresse multiple de 16 octets, c'est à dire la taille d'un registre SSE. On peut alors utiliser `movdqa` pour charger les données dans `xmm0` et `xmm1`. On peut stocker ces données au niveau de la section des données (`.data`) ou de la section de code (`.text`).

Puis pour effectuer un dépliage par 4, on réalise la série d'opérations des lignes 15 à 18 du Listing 16.8.1.

Instruction	xmm0	xmm1
initialement	$F3, F_2, F_1, F_0$	$F4, F_3, F_2, F_1$
`paddd xmm0, xmm1`	$F5, F_4, F_3, F_2$	$F4, F_3, F_2, F_1$
`paddd xmm1, xmm0`	$F5, F_4, F_3, F_2$	$F6, F_5, F_4, F_3$
`paddd xmm0, xmm1`	$F7, F_6, F_5, F_4$	$F6, F_5, F_4, F_3$
`paddd xmm1, xmm0`	$F7, F_6, F_5, F_4$	$F8, F_7, F_6, F_5$

TABLE 16.4 – Evolution du contenu des registres vectoriels

Il est nécessaire de réserver de l'espace dans la pile (ligne 11) afin de stocker le résultat final contenu dans `xmm0`. Les trois dernières itérations potentielles sont traitées à partir de la ligne 23. Ici, il n'est nul besoin de réaliser un `while` puisque le registre `xmm0` contient F_i, F_{i+1}, F_{i+2}, F_{i+3}. On stocke donc au niveau de la pile le contenu du registre et on récupère l'élément voulu en utilisant `eax` qui contient, depuis la ligne 10, le nombre d'itérations restantes après dépliage.

16.8.2 Version AVX

La version AVX consiste à faire un dépliage par 8 de la boucle `while` car un registre AVX stocke 8 entiers. Comme le montre le Listing 16.8.3, on utilise des instructions AVX qui commencent par la lettre **v**. On aligne également les données sur une adresse mémoire multiple de 32 octets.

```
align 16
sse_vect0:  dd 0, 1, 1, 2 ; F0, F1, F2, F3
sse_vect1:  dd 1, 1, 2, 3 ; F1, F2, F3, F4

fib_sse:
    mov     ecx, [esp+12]
    movdqa  xmm0, [sse_vect0]
    movdqa  xmm1, [sse_vect1]
    mov     eax, ecx
    and     eax, 3
    sub     esp, 16     ; espace pour stockage du résultat
    shr     ecx, 2      ; / 4
    jz      .last_3
.while_ur4:             ; dépliage par 4
    paddd   xmm0, xmm1
    paddd   xmm1, xmm0
    paddd   xmm0, xmm1
    paddd   xmm1, xmm0

    dec     ecx
    jnz     .while_ur4

.last_3:
    vmovdqu [esp], xmm0 ; stockage du résultat
    mov     eax, [esp + eax * 4]
    add     esp, 16
    ret
```

Listing 16.8.1 – Fibonacci - fonction itérative vectorielle

Version vectorielle SSE

En utilisant les registres vectoriels on obtient les temps d'exécution suivants :

- version SSE intrinsics (méthode 26) : $2,393$ s
- version SSE assembleur (méthode 27) : $2,008$ s
- version SSE assembleur améliorée avec dépliage par 8 (méthode 30) : $1,961$ s
- version AVX intrinsics (méthode 32) : $2,471$ s
- version AVX assembleur dépliage par 8 (méthode 33) : $1,982$ s
- version AVX assembleur dépliage par 16 (méthode 34) : $1,973$ s

16.9 Résultats

Le code qui correspond à l'ensemble des résultats produits dans cette section comporte 34 implantations différentes. Voici résumé les caractéristiques des implan-

```
#define ALIGN(x) __attribute__((aligned(x)))

const u32 sse_v0[4] ALIGN(16) = { 0, 1, 1, 2};
const u32 sse_v1[4] ALIGN(16) = { 1, 1, 2, 3};

u32 fib_iter_sse( u32 f0, u32 f1, u32 n ) {
    u32 v2[4] ALIGN(16);
    __m128i vf0, vf1;

    vf0 = _mm_load_si128((__m128i*)  &sse_v0 );
    vf1 = _mm_load_si128((__m128i*)  &sse_v1 );

    u32 r = n / 4;
    while (r) {
        vf0 = _mm_add_epi32( vf0, vf1 );
        vf1 = _mm_add_epi32( vf0, vf1 );
        vf0 = _mm_add_epi32( vf0, vf1 );
        vf1 = _mm_add_epi32( vf0, vf1 );
        --r;
    }

    _mm_store_si128( (__m128i*) &v2[0], vf0 );
    return v2[ n & 3 ];
}
```

Listing 16.8.2 – Fibonacci - fonction vectorielle SSE

tations qui ont été gardées pour présentation et analyse :

- **gcc** (1) : fonction de référence optimisée par `gcc`, le compilateur GNU
- **icc** (2) : fonction de référence optimisée par `icc`, le compilateur Intel
- **stat. array** (4) : tableau statique
- **dyna. array** (5) : tableau dynamique
- **n recursive** (6) : implantation directe en assembleur de la fonction de référence
- **while v1** (8) : version itérative avec un **while**
- **while v1 ur2** (9) : version itérative dépliée par 2
- **while v1 ur4** (10) : version itérative dépliée par 4
- **while v1 ur8** (11) : version itérative dépliée par 8
- **while v3 ur8** (21) : version itérative dépliée par 8 avec calcul utilisant **lea** et utilisation de **esp**
- **while v6 ur8** (24) : version itérative dépliée par 8 avec utilisation de l'astuce de dépliage par 2 et utilisation de **esp**
- **SSE intrin** (26) : version vectorielle SSE intrinsics

```asm
align 32
avx_vect0:  dd 0, 1, 1, 2, 3, 5,  8, 13 ; F0, ..., F7
avx_vect1:  dd 1, 1, 2, 3, 5, 8, 13, 21 ; F1, ..., F8

align 16
fib_avx:
    mov     ecx, [esp+12]
    vmovdqa ymm0, [avx_vect0]
    vmovdqa ymm1, [avx_vect1]

    sub     esp, 32
    mov     eax, ecx
    and     eax, 7

    shr     ecx, 3      ; / 8
    jz      .last_7

.while_ur8:
    vpaddd  ymm0, ymm1
    vpaddd  ymm1, ymm0
    vpaddd  ymm0, ymm1
    vpaddd  ymm1, ymm0
    vpaddd  ymm0, ymm1
    vpaddd  ymm1, ymm0
    vpaddd  ymm0, ymm1
    vpaddd  ymm1, ymm0

    dec     ecx
    jnz     .while_ur8

.last_7:
    vmovdqu [esp], ymm0
    mov     eax, [esp + eax * 4]
    add     esp, 32
    ret
```

Listing 16.8.3 – Fibonacci - fonction vectorielle AVX

- **SSE v4 ur8** (30) : version vectorielle SSE en assembleur dépliée par 8 et utilisation de **esp**
- **AVX intrin** (32) : version vectorielle AVX intrinsics
- **AVX ur8** (33) : version vectorielle AVX en assembleur dépliée par 8 et utilisation de **esp**

  Chaque méthode est exécutée 10 fois si son temps d'exécution est inférieur à 10 secondes. On prend alors ma moyenne des 10 exécutions. Par contre, si le temps d'exécution est supérieur à 10 secondes on reporte simplement ce temps.

16.9.1 Architectures modernes (2015 à 2019)

L'analyse des résultats montre à peu de choses près les mêmes tendances pour toutes les architectures.

La méthode 1 qui est la traduction optimisée par `gcc` du code de référence se montre très efficace par rapport aux autres méthodes. La méthode 2 qui est le code optimisé par `icc` est par contre bien moins performant alors que généralement `icc` produit un code plutôt bien optimisé.

Les méthodes basées sur les tableaux ne sont pas très performantes surtout la méthode 5 basée sur une allocation du tableau à chaque itération. Les nombreux appels systèmes engendrent un doublement du temps d'exécution.

La méthode la plus problématique est la méthode 6 qui correspond à la traduction directe de la méthode de référence. Son temps d'exécution est bien trop important. Est-ce dû aux accès répétés dans la pile des paramètres ou aux appels de sous-programmes ? En fait, le problème vient du nombre d'instructions exécutées. La méthode est appelée 500 millions de fois. On calcule F_{43} lors du test, la méthode s'appelle donc 43 fois et comporte 16 instructions, cela fait un total de 344 milliards d'instructions. En prenant en compte les instructions de la boucle qui réalise les 500 millions d'appels, cela représente, d'après `perf`, environ 352 milliards d'instructions. La méthode 1 par contre n'utilise que 45 milliards d'instructions. L'exécution du test avec la méthode 33 nécessite seulement 34 milliards d'instructions .

La méthode 8 qui consiste à remplacer les appels récursifs par un `while` se montre plus intéressante mais le dépliage par 2 ou par 4 sur les architectures Intel apparaît contre-productif. Le dépliage par 8 est plus intéressant mais ne permet pas de diminuer le temps d'exécution par rapport à la méthode non dépliée.

On observe par contre sur AMD Ryzen 1700X que le dépliage est intéressant puisqu'il diminue significativement le temps d'exécution par rapport à la méthode sans dépliage.

Le fait d'utiliser l'astuce mentionnée en section 16.7.5 permet de diminuer le temps d'exécution (cf. méthode 21) sur les processeurs AMD et Intel, de même pour la méthode 24.

Mais c'est l'implantation vectorielle SSE ou AVX (méthodes 30 et 33) qui permet de supplanter la version optimisée par le compilateur.

16.9.2 Architectures récentes (2020 et après)

Pour les architectures récentes (voir Table 16.6), on observe globalement les mêmes phénomènes.

Cependant, pour l'Intel 10850H le temps d'exécution de la méthode 6 est 30 fois plus lent que la méthode 1, ce qui est très étrange. Cela est-il dû à la mémoire qui serait extrêmement lente bien qu'étant de la DDR4-SDRAM fonctionnant à 3200

N°	Marque Gamme Modèle	Intel Core i3 6100 2015	AMD Ryzen 7 1700X 2017	Intel Core i5 7400 2017	Intel Core i7 8700 2017	Intel Core i5 8365U 2019
1	**gcc**	4,11	4,39	4,24	3,23	3,61
2	**icc**	7,97	11,31	8,56	6,49	7,39
4	**stat. array**	8,99	8,84	9,12	7,37	8,30
5	**dyna. array**	18,57	21,16	21,67	15,79	18,09
6	**n recursive**	63,39	82,48	67,71	51,53	**112,24**
8	**while v1**	7,52	11,47	8,99	5,78	7,44
9	**while v1 ur2**	18,45	5,48	22,26	15,01	18,88
10	**while v1 ur4**	12,56	5,25	16,07	10,21	12,79
11	**while v1 ur8**	9,65	5,14	11,85	7,86	9,95
21	**while v3 ur8**	5,37	4,39	5,74	4,45	4,96
24	**while v6 ur8**	4,35	4,51	4,66	3,60	3,95
26	**SSE intrin**	4,11	4,27	4,36	3,17	3,66
30	**SSE v4 ur8**	**3,48**	**3,22**	**3,36**	**2,57**	**2,95**
32	**AVX intrin v2**	4,20	5,73	4,38	3,21	3,93
33	**AVX asm ur8**	**3,21**	5,60	**3,34**	**2,57**	**3,06**
	ratio (1 / 30)	1,18	1,36	1,26	1,25	1,22
	ratio (1 / 33)	1,28	0,78	1,26	1,25	1,17

TABLE 16.5 – Résultats comparatifs de fib_iter : architectures modernes

MHz ?

Au final, la méthode qui semble la plus efficace est la méthode basée sur une implantation assembleur avec AVX, une boucle dépliée par 8, utilisant `esp` pour accéder à `n` (voir Listing 16.9.1).

16.9.3 Variation des fréquences de fonctionnement

J'ai créé un petit programme qui s'intitule `frequency.cpp` dont le but est d'enregistrer la fréquence de fonctionnement du core `0` sur lequel est exécuté le programme principal de test `fib.exe`. Ce programme permet de vérifier si la fréquence de fonctionnement reste stable ou varie.

On note que pour les processeurs AMD la fréquence reste stable alors que pour les processeurs Intel elle peut varier fortement. Cela se traduit par un écart type important.

N°	Marque Gamme Modèle	Intel Core i7 10850H 2020	Intel Core i7 1165G7 2020	Intel Core i5 10400 2020	AMD Ryzen 5 5600g 2021	Intel Core i5 12400F 2022
1	**gcc**	3,007	3,07	3,516	2,853	2,035
2	**icc**	5,365	5,81	5,610	3,990	4,489
4	**stat. array**	6,446	5,96	7,478	6,380	4,871
5	**dyna. array**	14,470	14,84	16,660	13,790	9,441
6	**n recursive**	93,390	59,53	**108,570**	50,630	55,870
8	**while v1**	6,038	11,94	7,322	6,304	5,570
9	**while v1 ur2**	13,860	16,09	18,210	**3,608**	14,930
10	**while v1 ur4**	9,510	11,07	13,140	**3,500**	11,250
11	**while v1 ur8**	7,254	8,69	9,583	**3,497**	8,197
21	**while v3 ur8**	4,121	5,02	4,823	**2,229**	4,795
24	**while v6 ur8**	3,286	3,41	3,832	2,271	2,807
26	**SSE intrin**	3,076	3,47	3,584	2,393	2,480
30	**SSE v4 ur8**	2,442	**2,70**	**2,816**	**1,961**	2,153
32	**AVX intrin v2**	3,235	3,71	3,837	2,471	2,566
33	**AVX asm ur8**	**2,410**	**2,70**	**2,768**	**1,982**	**2,074**
	ratio (1/33)	1,247	1,137	1,270	1,439	0,981

TABLE 16.6 – Résultats comparatifs pour fib_iter : architectures récentes

16.10 Remerciements

Merci à Mohamed Sylla et Matéo Grimaud, étudiants à l'Université d'Angers, pour les tests effectués respectivement sur Intel Core i7 1165G7 et Core i5 8365U.

```
align 32
avx_vect0:  dd 0, 1, 1, 2, 3, 5,  8, 13
avx_vect1:  dd 1, 1, 2, 3, 5, 8, 13, 21

align 16
fib:
    mov     ecx, [esp+12]
    vmovdqa ymm0, [avx_vect0]
    vmovdqa ymm1, [avx_vect1]

    sub     esp, 32
    mov     eax, ecx
    and     eax, 7

    shr     ecx, 3      ; / 8
    jz      .last_7

.while_ur8:
    vpaddd  ymm0, ymm1
    vpaddd  ymm1, ymm0
    vpaddd  ymm0, ymm1
    vpaddd  ymm1, ymm0
    vpaddd  ymm0, ymm1
    vpaddd  ymm1, ymm0
    vpaddd  ymm0, ymm1
    vpaddd  ymm1, ymm0

    dec     ecx
    jnz     .while_ur8

.last_7:
    vmovdqu [esp], ymm0
    mov     eax, [esp + eax * 4]
    add     esp, 32
    ret

```

Listing 16.9.1 – Fibonacci - fonction la plus performante

Marque Gamme Modèle	Intel Core i7 10850H 2020	Intel Core i5 7400 2017	AMD Ryzen 5 5600g 2021	AMD Ryzen 7 1700X 2017
Fréquence moyenne	4973,97	3475,01	4441,90	3492,71
Ecart type	24,20	13,84	0,00	0,12
Fréquence minimum	4945,02	3450,57	4441,90	3492,07
Fréquence maximum	5012,62	3499,39	4441,91	3493,04

TABLE 16.7 – Variation de la fréquence du microprocesseur lors des tests

Chapitre 17

Etude de cas
nombres auto-descriptifs

17.1 Introduction

Un nombre auto-descriptif se définit comme un entier naturel ayant pour propriété que chacun de ses chiffres repéré par son rang indique combien de fois ce rang apparaît en tant que chiffre dans l'écriture de ce nombre. On parle aussi de nombre autobiographique ou de nombre qui se décrit lui-même. Le premier nombre auto-descriptif est 1210. En effet :

- il contient **1** chiffre zéro
- il contient **2** chiffres un
- il contient **1** chiffre deux
- il contient **0** chiffre trois

Il en va de même pour 2020, 21200. Ces nombres sont très rares, on en compte 7 dont la liste figure Table 17.1 :

n	$a(n)$
1	1_210
2	2_020
3	21_200
4	3_211_000
5	42_101_000
6	521_001_000
7	6_210_001_000

TABLE 17.1 – Nombres auto-descriptifs

Formalisons tout cela. On considère x un nombre entier positif de k chiffres de

la forme :

$$x = d_0 \times 10^{k-1} + \ldots + d_{k-1} \times 10^0 = \sum_{i=0}^{k-1} d_i \times 10^{k-1-i}$$

On définit pour ce nombre une suite de valeurs c_0 à c_{k-1} où chaque c_i représente le nombre d'occurrences du chiffre i dans x :

$$c_i = Card(\{d_j \text{ tel que } d_j = i, \forall j \in [0, k-1]\})$$

On doit alors vérifier la contrainte suivante qui définit un nombre auto-descriptif :

$$\forall i \in [0, k-1],\ c_i = d_i$$

où $[0, k-1]$ représente l'intervalle de valeurs entre 0 et $k-1$. De ces définitions découlent les propriétés suivantes :

- **propriété 1** : un nombre auto-descriptif ne peut pas commencer par 0
- **propriété 2** : un nombre auto-descriptif contient au moins un 0
- **propriété 3** : la somme des c_i est égale à k, et donc, la somme des d_i est égale à k
- **propriété 4** : un nombre auto-descriptif contient au moins un 0 en position $k-1$
- **propriété 5** : un nombre auto-descriptif de k chiffres ne peut par avoir un chiffre supérieur à k

Théorème 17.1.1. Un nombre auto-descriptif ne peut pas commencer par 0.

Démonstration. En effet, si $d_0 = 0$ alors le nombre commence par 0 et par définition n'est pas valide. □

Théorème 17.1.2. Un nombre auto-descriptif contient au moins un 0.

Démonstration. En effet, si $c_0 = 0$ alors $d_0 = 0$ et le nombre commence par 0 et par définition n'est pas valide d'après le théorème précédent. □

Théorème 17.1.3.

$$\sum_{i=0}^{k-1} c_i = \sum_{i=0}^{k-1} d_i = k$$

Démonstration. Si par définition les c_i représentent le nombre d'occurrences de chaque chiffre, leur somme doit donc être égale à k. Etant donné que par définition $c_i = d_i$, on en déduit également que la somme des d_i est égale à k. □

Théorème 17.1.4. Un nombre auto-descriptif se termine par un 0.

Démonstration. Appelons $S(k) = \sum_{i=0}^{k-1} d_i$ et raisonnons pas l'absurde pour montrer qu'il n'est pas possible que d_{k-1} soit différent de 0. Par définition $S(k) = k$ et donc $S(k) = S(k-1) + d_{k-1}$. Supposons alors que x ne setermine pas par 0, alors $d_{k-1} > 0$, et on en déduit que $S(k-1) < k$ ou encore que $S(k-1) <= k-1$. Le fait qu'un nombre auto-descriptif ne commence pas par un 0 mais contient au moins un 0, implique que $d_0 \neq 0$ et donc qu'il existe au moins un 0 parmi les d_1 à d_{k-1}. En conséquence $S(k) >= k-1$. Sachant que $S(k) = S(k-1) + d_{k-1}$, on a donc $S(k) - S(k-1) = d_{k-1}$. Mais comme $S(k) >= k-1$ et que $S(k-1) <= k-1$, on en déduit que $S(k) - S(k-1) = 0$ et donc que $d_{k-1} = 0$. □

Théorème 17.1.5. Un nombre auto-descriptif de k chiffres ne peut par avoir un chiffre supérieur à k, i.e. $d_i < k, \forall i \in [0, k-1]$

Démonstration. Si x possède k chiffres alors d_0 à d_{k-1} sont définis. Si $d_i \geq k$ cela implique que $c_i \geq k$ ce qui est en contradiction avec la définition du nombre. □

17.2 Fonction de référence

La fonction de référence à implanter est donnée Listing 17.2.1. Elle comporte un paramètre qui est le nombre entier non signé `x` pour lequel on veut déterminer s'il est auto-descriptif ou non. Nous n'allons pas utiliser certaines propriétés énoncées préalablement afin de garder un temps de calcul qui nous permettra de comparer les différentes améliorations proposées. En effet, si on utilise le fait qu'un nombre auto-descriptif se termine par 0, il suffit de calculer le premier reste de la division par 10 pour trouver le premier chiffre et la recherche des nombres auto-descriptifs, en utilisant cette propriété, s'exécute en moins de 2 secondes.

Nous allons donc nous focaliser sur les trois points suivants :

- conversion du paramètre x en base 10, en d'autres termes : détermination des d_i
- détermination du nombre d'occurrences de chaque chiffre, soit la détermination des c_i
- comparaison du nombre d'occurrences avec le i ième chiffre, en fait, comparaison des d_i avec les c_i

Dans cette première version, on utilise les fonctionnalités du C++ pour transformer le nombre `x` en caractères et déterminer les d_i.

La variable `counts` représente le nombre d'occurrences de chaque chiffre, soit les c_i. On comptabilise donc le nombre d'occurrences de chaque chiffre en parcourant la chaîne obtenue après conversion de x de la base 2 vers la base 10. Finalement,

```cpp
typedef uint8_t   u8;
typedef int32_t  i32;
typedef uint32_t u32;

bool ad_cpp_32( u32 x ) {
    // nombre d'occurrences de chaque chiffre
    u32 counts[ 10 ] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

    // convertir le nombre en chaîne de caractères
    string s = std::to_string( x );

    // compter les occurrences de chaque chiffre
    for (u32 i = 0; i < s.length(); ++i) {
        ++counts[ (u32) (s[i] - '0') ];
    }

    // comparer les occurrences aux chiffres
    for (u32 i = 0; i < s.length(); ++i) {
        if (static_cast<u32>(s[i] - 48) != counts[ i ]) return false;
    }

    return true;
}
```

Listing 17.2.1 – Nombre auto-descriptif, fonction de référence

on compare le nombre d'occurrences de chaque chiffre aux chiffres du nombre représenté sous forme de chaîne.

Fonction de référence

Le temps d'exécution sur AMD Ryzen 5 5600g est de 7,34 secondes.

Attention

Attention cependant ! Je me suis aperçu que le compilateur gcc dans sa version 10 générait une fonction dont le temps d'exécution est d'environ 18 secondes alors que le même compilateur, dans sa version 12, génère du code qui ne s'exécute plus qu'en 7,6 secondes, soit près de deux fois plus rapide. La différence d'exécution tient à la conversion de x en chaîne. Nous utilisons donc par la suite la version 12 de gcc quand cela est possible.

17.3 Première amélioration

Plutôt que d'utiliser un tableau d'entiers pour compter les occurrences des chiffres, on utilise un tableau d'octets. En effet, diminuer la taille des données utilisées peut parfois apporter une amélioration.

Fonction de référence avec 8 bits

Le temps d'exécution sur AMD Ryzen 5 5600g est de 7,20 secondes en utilisant des octets pour représenter le nombre d'occurrences de chaque chiffre.

On en déduit donc qu'il vaut mieux travailler sur un tableau d'octets que sur un tableau d'entiers en général (cf. Section 17.6) même si l'amélioration est faible.

17.4 Convertir en chiffres et non en chaîne

Plutôt que de convertir `x` en chaîne de caractères, nous allons la convertir directement en chiffres que l'on va stocker dans un tableau (d'entiers ou d'octets) appelé `digits` en faisant des divisions par 10 afin d'obtenir chaque chiffre (lignes 7 à 12 du Listing 17.4.1). A chaque nouveau reste trouvé on incrémente le tableau `counts` en conséquence et on stocke le reste dans le tableau `digits`.

Au final on obtient le remplissage des tableaux suivants :

Indice	0	1	2	3	4	5	6	7	8	9	10	...	15
counts	**2**	**1**	**2**	**0**	**0**	0	0	0	0	0	0	0	0
digits	**0**	**0**	**2**	**1**	**2**	0	0	0	0	0	0	0	0

Fonction avec conversion 8 bits

C'est en fait cette version qui est la plus efficace lorsque traduite par le compilateur car elle ne prend que 4,22 secondes pour s'exécuter et c'est elle qu'il va falloir tenter de battre. La version utilisant des entiers prend quant à elle 5,83 secondes.

La raison de l'efficacité est assez simple : la conversion est rapide car elle est optimisée et la division par 10 est remplacée par une multiplication par un invariant, puis le calcul du reste de la division est effectué par multiplication du dividende et soustraction. En outre, la boucle de conversion est dépliée.

```
bool ad_chiffres( u32 x ) {

    u8 counts[ 10 ] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
    u8 digits[ 10 ];

    // convertir x et compter les occurrences
    i32 i = 0;
    while (x) {
        u32 u = x % 10;
        digits[ i++ ] = u;
        ++counts[ u ];
        x = x / 10;
    }

    // comparer le nombre d'occurrences avec les chiffres
    i32 j;
    for (j = 0, --i; i >= 0; --i, ++j) {
        if (digits[ i ] != counts[ j ]) return false;
    }

    return true;

}
```

Listing 17.4.1 – Nombre auto-descriptif, fonction de conversion en chiffres

17.5 Versions assembleur

Etant données les différentes version précédentes, on peut se demander si la traduction assembleur ne serait pas plus performante.

17.5.1 Version 1 - Traduction

Dans cette première version, on se contente de traduire la version C de l'Algorithme 17.4.1 en assembleur. On choisit de réaliser l'association variables / registres de la Table 17.2 et on va garder le même schéma de traduction pour les autres fonctions.

Les registres `ebx`, `esi` et `edi` devront être sauvegardés car ils ne doivent pas être modifiés pour le sous-programme appelant d'après les conventions d'appel du C en 32 bits et nous allons les utiliser. Le registre `edx` sera mis à 0 et ne contiendra aucune donnée au début du sous-programme. Après chaque division, `edx` contient le reste de la division par 10, c'est à dire la variable `u` du Listing 17.4.1.

Les tableaux `counts` et `digits` sont stockés dans la pile en réservant (au moins)

Cste/Param/Var	Type	Paramètre	Registre	Description
x	u32	[ebp+8]	eax	nombre x
counts	u8 []	Pile	esp	occurences
digits	u8 []	Pile	esp	conversion
i	u32		ecx	indice

TABLE 17.2 – Association entre variables et registres pour les fonctions assembleur recherchant les nombres auto-descriptifs

```
1  ad_asm_v1:
2      push    ebp
3      mov     ebp, esp
4      mov     eax, [ebp + 8]
5      sub     esp, 44          ; on réserve l'espace pour stocker
6      and     esp, ~31         ; counts, digits et les registres
7      mov     [esp+32], ebx    ; ebx, esi, edi
8      mov     [esp+36], edi
9      mov     [esp+40], esi
10     vpxor   ymm0, ymm0       ; on met counts et digits à 0
11     vmovdqa [esp], ymm0
```

Listing 17.5.1 – Fonction assembleur - version 1 - début

48 octets et en alignant l'adresse du sommet de pile (esp) sur un segment de 32 octets (lignes 5 et 6 du Listing 17.5.1). Ces 44 octets se décomposent ainsi :

- 12 octets pour sauvegarder ebx, edi et esi
- 16 octets pour les d_i pour aligner le tableau digits sur une adresse multiple de 16
- 16 octets pour les c_i pour aligner le tableau digits sur une adresse multiple de 16

On sauvegarde ensuite les registres à préserver et on on accèdera counts et digits par l'intermédiaire de esp. En effet, on peut stocker counts de esp à esp+15 et digits de esp+16 à esp+26. Le tableau counts doit être initialisé à 0, on utilise ici le registre ymm0 pour faire cela qui va permettre d'initialiser en une seule fois les 32 octets qui stockent counts et digits .

Pour réaliser la conversion de x en base 10 (voir Listing 17.5.2), on réalise des divisions successives par 10. En utilisant l'instruction div, on obtient le quotient dans eax et le reste dans edx, ce qui est très pratique.

A la différence du code C++, on va stocker les chiffres (restes de la division) dans l'ordre décroissant du tableau digits en commençant à l'indice 9 qui sera stocké dans ecx, puisque un entier non signé contient au maximum 10 chiffres.

```
1  ; ad_asm_v1 conversion
2      mov     edi, 10         ; constante 10 pour division par 10
3      mov     ecx, 9          ; i = 9, indice pour tableau digits
4      cmp     eax, 10         ; si x < 10 alors aller en .x_lt_10
5      jl      .x_lt_10        ;
6  .while_x_ge_10:             ; tantque x >= 10 faire
7      xor     edx, edx
8      div     edi             ;    u, x = x % 10, x / 10
9      inc     byte [esp+edx]  ;    ++counts[ u ]
10     dec     ecx             ;     digits[ --i ] = u
11     mov     [esp+ecx+16],dl
12     cmp     eax, 10
13     jge     .while_x_ge_10  ; fin tantque
14     ; si x < 10, il n'est pas nécessaire de diviser
15 .x_lt_10:
16     cmp     eax, 0          ; si x == 0 alors aller en .next
17     jz      .next
18     inc     byte [esp+eax]  ; ++counts[ x ]
19     dec     ecx             ; digits[ --i ] = x
20     mov     [esp+ecx+16], al
```

Listing 17.5.2 – Fonction assembleur - version 1 - conversion

Un fois `x` converti et les tableaux `counts` et `digits` remplis, il suffit de comparer `counts[j]` à `digits[i]` en partant de `ecx` (voir Listing 17.5.3). En effet, pour $x = 21200$, on obtient le remplissage suivant :

Indice	0	1	2	3	4	5	6	7	8	9	10	...	15
counts	**2**	**1**	**2**	**0**	**0**	0	0	0	0	0	0	0	0
digits	0	0	0	0	0	**2**	**1**	**2**	**0**	**0**	0	0	0

On utilise alors `edi` pour connaître le nombre de chiffres en calculant dans ce registre la différence entre 9 et `ecx`. Le registre `edi` contient alors le fameux k, défini dans l'introduction, qui correspond au nombre de chiffres de la traduction en base 10. Puis, on stocke dans `esi` l'adresse de début du tableau `digits` à laquelle on ajoute l'indice `ecx`. On n'a plus alors qu'à effectuer une simple boucle `for` et comparer `counts[i]` soit `esp+ecx` à `digits[j]`, soit `esi+ecx`.

Si les deux tableaux sont égaux, on sortira de la fonction avec la valeur 1 (*true*) dans `eax`, ligne 15 du Listing 17.5.3.

Assembleur version 1 avec conversion 8 bits

Cette version assembleur s'exécute en 5,59 secondes.

```
; ad_asm_v1 comparaison et sortie de la fonction
.next:
    xor     eax, eax            ; valeur de retour = faux
    lea     esi, [esp+ecx+16]   ; esi = début de digits
    mov     edi, 9              ; longueur = nombre de chiffres
    sub     edi, ecx            ;
    xor     ecx, ecx            ; i = 0
.while_eq:                      ; faire
    movzx   ebx, byte [esp+ecx] ;    si counts[i] != digits[i] alors
    cmp     bl, [esi+ecx]       ;        retourner faux
    jne     .end
    inc     ecx                 ;    ++i
    sub     edi, 1
    jnz     .while_eq           ; tantque i < longueur
    inc     eax                 ; valeur de retour = vrai
.end:
    mov     ebx, [esp+32]
    mov     edi, [esp+36]
    mov     esi, [esp+40]
    mov     esp, ebp
    pop     ebp
    ret
```

Listing 17.5.3 – Fonction assembleur - version 1 - comparaison et sortie

17.5.2 Version 2 - Remplacement de la division

Une première amélioration consiste à remplacer la division par une multiplication. On le sait, la division est très pénalisante par rapport à la multiplication. Cela est possible ici car on divise par une constante (10), on peut donc remplacer la division par une multiplication par invariant (cf. Sous-section 2.4.7).

On va donc modifier l'utilisation des registres en conséquence. On va multiplier `eax`, en fait `x`, par `ebx` qui contiendra la constante `CC_CC_CC_CD` qui correspond à $0,1$. On obtient alors le résultat dans `edx:eax`. Cependant, c'est la partie haute du résultat que l'on doit garder, donc `edx` que l'on décale alors de 3 rangs vers la droite (cf. Section 2.4.7).

Le problème que l'on rencontre est qu'on ne dispose pas du reste de la division. Il va donc falloir le calculer en prenant le résultat de la division par 10, puis en le multipliant par 10 et en le soustrayant de `x`. Plutôt que de faire une multiplication par 10, on va utiliser l'instruction `lea` pour calculer $5 \times x$, puis on multipliera par 2 on faisant un décalage de bits grâce à l'instruction `shl`.

Malgré tout, il va nous manquer un registre pour calculer le reste de la division. On va en effet utiliser `ebx` pour calculer le reste de la division et éviter les dépendances liées à l'utilisation de `edx`. On va donc utiliser `edi` pour stocker temporairement la valeur de `x`. Cela est possible car au final on n'a besoin que d'un seul registre pour accéder à `counts` et `digits` puisque `counts` débute en `esi` et que `digits` débute en `esi` + 16.

```asm
ad_asm_v2:
    push    ebp
    mov     ebp, esp
    ; ... identique à ad_asm_v1 ...
    mov     edi, 0xCCCCCCCD     ; 0.1 dans edi
    mov     ecx, 9
    cmp     eax, 10
    jl      .x_lt_10
.while_x_ge_10:
    mov     ebx, eax            ; sauvegarde de x dans ebx
    mul     edi                 ; edx:eax <- x * 0.1
    shr     edx, 3              ; edx <- edx / 8 (éq. x/10)
    lea     esi, [edx+edx*4]    ; esi = 5*(x/10)
    shl     esi, 1              ; esi = 2*5*(x/10)
    sub     ebx, esi            ; calcul du reste u de la division
    dec     ecx                 ; --i
    inc     byte [esp+ebx]      ; ++counts[ u ]
    mov     [esp+ecx+16],bl     ; digits[ i ] = u
    mov     eax, edx            ; eax <- x/10
    cmp     eax, 10
    jge     .while_x_ge_10
.x_lt_10:
    ; ... identique à ad_asm_v1 ...
    mov     esp, ebp
    pop     ebp
    ret
```

Listing 17.5.4 – Fonction assembleur - version 2 - remplacement de la division par une multiplication

Le code correspondant est donné Listing 17.5.4.

Assembleur version 2 : multiplication au lieu d'une division

Cette version assembleur demande plus d'opérations s'exécute en 5,69 secondes soit une très légère dégradation par rapport à la version 1

17.5.3 Version 3 - Remplacement de la division et dépliage

On peut garder le remplacement de la division par une multiplication de la fonction précédente et ajouter le dépliage de la boucle de conversion en base 10.

Assembleur version 3 : multiplication et dépliage

Cette version assembleur s'exécute en 4,32 secondes, le dépliage est donc ici efficace et intéressant.

17.5.4 Version 4 - Comparaison vectorielle

Plutôt que de comparer les tableaux `digits` et `counts` élément par élément grâce à une boucle `for`, on peut le faire de manière vectorielle en chargeant `digits` dans un premier vecteur et `counts` dans un autre vecteur. Pour que cela fonctionne il faut que l'espace donné à `digits` soit plus grand que 16 octets, on va donc doubler la taille de `digits` et occuper 32 octets qui seront mis à 0.

Pour comparer les tableaux (voir ci-dessous), on charge `counts` dans `xmm1` et `digits` dans `xmm2`. On utilise l'instruction `pcmpeqb`, ici dans sa version AVX, qui compare chacun des octets des deux registres et remplace les octets de `xmm1` par `0xFF` si les deux octets sont égaux ou par `0x00` s'ils sont différents. Si les deux registres contiennent les mêmes valeurs chaque octet du registre `xmm1` aura la valeur `0xFF`. On utilise ensuite l'instruction `pmovmskb` tous les bits de poids fort de chaque octet de `xmm1`, le résultat étant placé dans `edx`. Au final, on obtient dans `edx` la valeur `0xFFFF` si les deux registres vectoriels sont égaux. L'utilisation de l'instruction `sete` permet de fixer `eax` à 0 si les registres vectoriels sont différents ou à 1 s'ils sont égaux.

```
1      xor      eax, eax                ; valeur de retour : false
2      vmovdqa  xmm1, [esp]             ; charger counts[0:15]
3      vmovdqu  xmm2, [esp + ecx + 16]  ; charger digits[0:15]
4      vpcmpeqb     xmm1, xmm2          ; comparer octet par octet
5      vpmovmskb edx, xmm1              ; récupérer le masque issu
6                                       ; de la comparaison
7      cmp      edx, 0xFFFF             ; s'il est égale à 0xFFFF alors
8                                       ; tous les octets sont identiques
9      sete     al                      ; dans ce cas positionner eax à true
```

La version précédente est écrite en AVX. On peut également l'écrire en SSE. Cependant, il n'est pas recommandé de mixer SSE et AVX notamment sur certains processeurs Intel. Travailler en AVX, sur `ymm0` par exemple, puis passer au SSE, et travailler sur `xmm0`, pose un problème relatif à la sauvegarde de la partie haute de `ymm0`. En fait cela ne devrait poser aucun problème puisque seule la partie basse doit être utilisée pour les calculs. Chez Intel, il est nécessaire de sauvegarder la partie haute du registre ce qui peut prendre jusqu'à 70 cycles. La seule explication plausible est probablement que le choix a été fait, chez Intel, de travailler sur le registre en totalité même lorsque l'on n'agit que sur le SSE. Dès lors, si on veut garder une certaine cohérence des valeurs contenues dans les registres, on se doit de sauvegarder la partie haute, réaliser le calcul, puis restaurer la partie haute.

Assembleur version 4 : comparaison finale vectorielle

Cette version assembleur s'exécute en 4,31 secondes, la comparaison vectorielle apporte un gain faible en général mais plus important sur d'autres architectures.

17.5.5 Versions 5 - Division par 100

Plutôt que de réaliser des divisions par 10, il peut être intéressant de réaliser des divisions par 100, on aura alors deux fois moins de divisions. De plus, nous allons remplacer la division par des multiplications par $0,01$.

```
1     mov     edi, 0x51EB851F     ; edi = 0.01
2     mov     ecx, 9              ; indice dans digits
3     cmp     eax, 10             ; si x < 10 alors traduire
4     jl      .x_lt_10            ; directement
5 .while_x_ge_10:
6     mov     ebx, eax            ; on sauvegarde x
7     mul     edi                 ; on multiplie par 0.01
8     shr     edx, 5              ; on décale edx de 5 rangs à droite
9     ; on calcule ensuite le reste de la division
10    lea     esi, [edx + edx * 4]; esi = 5 * (x/100)
11    lea     eax, [esi + esi * 4]; eax = 5 * (5 * (x/100))
12    shl     eax, 2              ; eax = 4 * 25 * (x/100)
13    sub     ebx, eax            ; obtenir le reste
14    mov     eax, edx            ; eax = x/100
15    movzx   edx, word [values_100 + ebx * 2]
16    sub     ecx, 2
17    mov     [esp + ecx + 16], dx
18    movzx   ebx, dh
19    xor     dh, dh
20    inc     byte [esp + ebx]
21    inc     byte [esp + edx]
22    cmp     eax, 10
23    jge     .while_x_ge_10
24 .x_lt_10:
25    cmp     eax, 0
26    jz      .next
27    inc     byte [esp + eax]
28    dec     ecx
29    mov     [esp + ecx + 16], al
```

Listing 17.5.5 – Fonction assembleur - versions 5 - remplacement de la division par une multiplication

Le code correspondant figure Listing 17.5.5. On calcule $q = x/100$ par multiplication et décalage (lignes 7 et 8). Il faut ensuite calculer le reste de la division en calculant $x - 100 \times q$, sachant que q est le résultat d'une division entière. On réalise le calcul grâce à deux instructions `lea` et un décalage qui permettent d'obtenir $4 \times 5 \times 5 \times q$ (lignes 10 à 12) et on retranche cette quantité à x (ligne 13) pour obtenir le reste `u`.

Une fois `u` obtenu, on utilise une table de conversion (`values_100`) qui permet de récupérer deux octets sous forme d'un mot correspondant aux deux chiffres décimaux du reste. Si `u` vaut 17, on récupère dabs `edx` (ligne 15) la valeur `0x0107`.

On sépare ensuite chacun des chiffres en en plaçant un dans `edx` et l'autre dans

`ebx` (lignes 18 et 19 du Listing 17.5.5). On n'a plus qu'à stocker les chiffres dans le tableau `digits` (lignes 16, 17) et incrémenter les éléments correspondants de `counts` (lignes 20 et 21).

La dernière partie du code (lignes 25 à 29) consiste à stocker le dernier reste éventuel qui sera inférieur à 10.

Assembleur version 4 : comparaison finale vectorielle

On a écrit trois versions différentes :

- version 100 (ou méthode 13 dans la section des résultats) : division par 100 (par multiplication par 0.01) : 3,62 secondes
- version 101, amélioration de la version 100 avec dépliage de la conversion : 3,56 secondes
- version 102, amélioration de la version 101 avec comparaison vectorielle AVX : 5,01 secondes

17.5.6 Versions 6 - Codage en BCD

Le *Binary Coded Decimal* ou Décimal Codé (en) Binaire est un ancien système de codage qui remonte aux années 1960. Il consiste à coder un nombre en plaçant deux chiffres décimaux par octet. Chaque quartet (ou *nibble* en anglais) représente donc 1 ou 2 chiffres. On dispose en assembleur d'une vieille instruction `fbstp` qui date du 8086 d'Intel. Elle permet de stocker au format BCD de 10 octets, un nombre, stocké au niveau de la FPU.

Par exemple, la valeur -1234567890 sera stockée au format BCD sous la forme :

Octet	9	8	7	6	5	4	3	2	1	0
	0x80	0x00	0x00	0x00	0x00	0x12	0x34	0x56	0x78	0x90

Le bit de poids fort indique ici le signe du nombre, s'il est à un, il s'agit d'un nombre négatif. On peut, par exemple, charger et stocker la valeur suivante : -123456789012345678. qui comprend 18 chiffres. Au delà, il se produit une erreur liée à la précision.

On va donc charger `x` comme un entier au niveau de la FPU et stocker le résultat au format BCD dans la pile. On aura donc besoin de 10 octets supplémentaires qui correspondent au format de stockage BCD que l'on va étdendre à 16 octets afin de garder l'alignement des données. Comme on ne traite que des entiers non signés qui comportent au maximum 10 chiffres et qui sont positifs ou nul, seuls 5 octets sont utilisés pour représenter le nombre.

```
1     fild      dword [ebp + 8] ; chargement de x dans la FPU
2     fbstp     [esp + 32]      ; stockage en [esp+32] au format BCD
```

Il faut ensuite relire le nombre au format BCD pour compter le nombre d'occurrences de chaque chiffres. J'ai mis au point deux méthodes :

- une première méthode qui décompose le nombre en utilisant les instructions assembleur classiques et qui fait appel également à des tables de conversion pour déterminer la longueur du nombre en terme de chiffres
- une seconde méthode qui se base sur des instructions spécifiques comme **pdep** et **movbe** afin d'extraire les chiffres BCD et stocker chacun dans un octet.

17.5.6.1 Décomposition avec les registres

Dans un premier temps, on détermine la longueur du nombre au format BCD : on part du dernier chiffre et on revient vers le premier (voir Listing 17.5.6). Dès qu'on a trouvé un chiffre différent de 0, on peut déterminer la longueur.

La méthode est assez complexe puisqu'elle s'intéresse en premier au 5ième octet qui représente les nombres de 9 ou 10 chiffres. Si cet octet est à 0 alors le nombre comporte 1 à 8 chiffres, le cas du 0 étant traité en amont, au tout début de la fonction en renvoyant la valeur *false*. Pour traiter les huit chiffres potentiels qui sont donc stockés sur 4 octets, on charge ce double mot dans **eax**, puis on calcule dans **ebx** le bit de poids fort de ce double mot en utilisant l'instruction **bsr**. Le résultat sera compris entre 0 et 31. On charge finalement à partir d'une table de 32 octets, la taille correspondante. Par exemple si le bit de poids fort est à l'indice 8, 9, 10 ou 11, il s'agit d'un nombre de 3 chiffres.

En fois la longueur déterminée, on passe à la conversion du nombre (cf. Listing 17.5.7).

Pour cela, en fonction de la longueur, on va se diriger vers un chemin de traduction spécifique. On définit à cet effet, une table d'adresses dont l'indice donne l'adresse du code qui correspond à la traduction. Il faut notamment distinguer les nombres dont le nombre de chiffres est pair de ceux qui ont un nombre de chiffres impair. Dans ce dernier cas, il ne faut traiter que les premiers 4 bits de l'octet que l'on aura chargé dans le registre **eax**. Les deux macro-instructions pour la conversion sont données Listing 17.5.8.

La macro instruction **cvt1** ne convertit qu'un chiffre en partie basse du registre **al**. La macro instruction **cvt2** convertit deux chiffres. Pour ce faire, on utilise une table nommée **bcd_table** de 200 octets organisée de manière à ce que deux octets consécutifs correspondent à deux chiffres codés chacun sur un octet. On charge donc ces deux octets dans **dx** (ligne 13) grâce à l'instruction **movzx** qui complète la partie haute de **edx** avec des 0. Puis on place le chiffre qui se trouve en **dh** dans le registre **ebx** et on met ensuite **dl** à 0. Les registres **ebx** et **edx** contiennent alors

```
; table des longueurs d'un nombre au format    BCD
; en fonction du bit de poids fort
bcd_tlengths:
    db  1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4,
    db  5, 5, 5, 5, 6, 6, 6, 6, 7, 7, 7, 7, 8, 8, 8, 8

    ...
    fild    dword [ebp + 8] ; chargement de x dans la FPU
    fbstp   [esp + 32]      ; conversion au format BCD

    mov     edi, 0

    ; détermination de la longueur
    ; on part du 10ième quartet en on descend pour trouver
    ; le premier quartet non égal à 0
.find_length:

    xor     ebx, ebx                ; ebx <- 0 valeur qui sera
                                    ; soutraite à ecx
    mov     ecx, 10                 ; taille initiale de 10
    movzx   eax, byte [esp + 36]    ; prendre le 5ième octet
    test    eax, eax                ; si il est à 0 alors considérer
                                    ; les 4 premiers octects
    jz      .length_1_to_8
    test    al, 0xF0                ; si le 10ième chiffre est à 0
    setz    bl                      ; alors mettre 1 dans bl
    sub     ecx, ebx                ; soustraire à ecx
    jmp     [bcd_cv_jumps + ecx * 4]; puis convertir

.length_1_to_8:
    mov     eax, [esp + 32]         ; mettre les 4 octets du nombre
                                    ; au format BCD dans eax
    bsr     ebx, eax                ; trouver le bit de poids fort
    movzx   ecx, byte [bcd_tlengths + ebx] ; charger la longueur
    jmp     [bcd_cv_jumps + ecx * 4]; convertir
```

Listing 17.5.6 – BCD - version 1 - Détermination de la longueur du nombre

les indices dans le tableau `counts` qui doivent être incrémentés. En dernier lieu, on place les chiffres dans le tableau `digits`.

Enfin, la dernière étape, figurant Listing 17.5.9, consiste à comparer le nombre d'occurrences de chaque chiffre et le nombre traduit au format un chiffre décimal par octet, et ressemble à ce que l'on a pu déjà faire.

```
bcd_cv_jumps:
    dd ..@cv1, ..@cv1, ..@cv2, ..@cv3, ..@cv4
    dd ..@cv5, ..@cv6, ..@cv7, ..@cv8, ..@cv9, ..@cv10

    ...
..@cv10:            ; convertir 10 chiffres
    cvt2    36
..@cv8:             ; convertir 8 chiffres
    cvt2    35
..@cv6:             ; convertir 6 chiffres
    cvt2    34
..@cv4:             ; convertir 4 chiffres
    cvt2    33
..@cv2:             ; convertir 2 chiffres
    cvt2    32
    jmp     .compare

..@cv9:             ; convertir 9 chiffres
    cvt1    36
    jmp     ..@cv8
..@cv7:             ; convertir 7 chiffres
    cvt1    35
    jmp     ..@cv6
..@cv5:             ; convertir 5 chiffres
    cvt1    34
    jmp     ..@cv4
..@cv3:             ; convertir 3 chiffres
    cvt1    33
    jmp     ..@cv2
..@cv1:             ; convertir 1 chiffres
    cvt1    32
```

Listing 17.5.7 – BCD - version 1 - Conversion du nombre

```
; convertir un octet qui ne contient qu'un chiffre
; au format BCD en partie basse
%macro cvt1 1
    movzx   eax, byte [esp + %1]    ; eax <- charger la valeur
    mov     [esp + 16 + edi], al    ; digits[ i ] = u
    inc     byte [esp + eax]        ; ++counts[ u ]
    add     edi, 1                      ; ++i
%endmacro

; convertir un octet qui contient deux chiffres
%macro cvt2 1
    movzx   eax, byte [esp + %1]            ;charge 2 chiffres
    movzx   edx, word [bcd_table + eax * 2] ; conversion en deux
                                            ; chiffres sur 2 octets
    movzx   ebx, dh                         ; dans edx
    xor     dh, dh                          ; et ebx
    inc     byte [esp + edx]                ; ++counts[ chiffre1 ]
    inc     byte [esp + ebx]                ; ++counts[ chiffre2 ]
    mov     [esp + 16 + edi], bl            ; stockage de chiffre2
    mov     [esp + 17 + edi], dl            ; stockage de chiffre1
    add     edi, 2
%endmacro
```

Listing 17.5.8 – BCD - version 1 - Macros instructions pour la conversion

```
.compare:
    xor     ecx, ecx
    xor     eax, eax
align 16
.do_while:
    movzx   edx, byte [esp + ecx]
    cmp     dl, [esp + 16 + ecx]
    jne     .end
    inc     ecx
    cmp     ecx, edi
    jl      .do_while
.end_while:

    mov     eax, 1

```

Listing 17.5.9 – BCD - version 1 - Comparaison du nombre d'occurrences des chiffres avec le nombre

17.5.6.2 Décomposition avec les instructions spécifiques

On utilise pour la conversion du nombre au format BCD deux instructions spécifiques. La première appelée `pdep` (*Parallel Bits Deposit*) fait partie du jeu d'instructions BMI2 (*Bit Manipulation Instructions*). Elle comporte trois opérandes sous la forme de trois registres 32 ou 64 bits et permet de sélectionner et copier les bits de la seconde opérande dans la première en utilisant un masque de sélection placé dans le troisième registre. Par exemple, le code suivant :

```
1    mov    ebx, 0xFEDC
2    mov    ecx, 0x0F0F
3    pdep   eax, ebx, ecx
```

donnera le résultat `0xD0C` dans `eax` comme le montre la Figure 17.1. Il permet de sélectionner les deux premiers quartets de `ebx` et les transformer en octets dans `eax`.

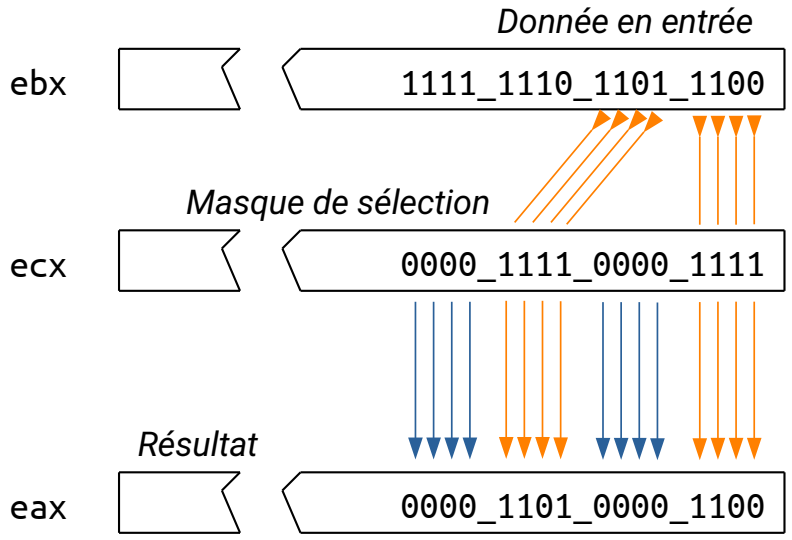


FIGURE 17.1 – Instruction PDEP

L'autre instruction utilisée est `movbe`, elle fait partie normalement du jeu d'instructions NNI (pour *Nehalem New Instructions*) et remonte à 2008, bien que je n'ai pas trouvé d'informations exactes à ce sujet. L'instruction est également appelée *Move Data After Swapping Bytes*. Elle inverse les octets d'un registre 16, 32 ou 64 bits. Elle possède une particularité qui tient à ses opérandes : l'une des opérandes est un registre et l'autre est une adresse mémoire. On ne peut malheureusement pas avoir deux opérandes de type registre comme c'est le cas de la plupart des instructions. Ainsi, le code suivant :

```
    mov     eax, 0x01020304
    movbe   [x], eax
    mov     eax, [x]
```

permet de récupérer la valeur `0x04030201` dans `eax`.

La traduction du nombre au format BCD vers une représentation par octet est donnée Listing 17.5.10.

```
                            ; ex. eax = 1234567890
    mov     eax, [esp + 36] ; chargement des 2 derniers
                            ; chiffres : 0x12
    mov     ecx, 0xF0F0F0F
    pdep    ebx, eax, ecx   ; décalage ebx = 0x0102
    movbe   [esp + 16], ebx ; stockage

    mov     eax, [esp + 32] ; chargement des chiffres 1 à 8
                            ; eax = 34567890

    pdep    ebx, eax, ecx   ; ebx = 0x07080900
    movbe   [esp + 24], ebx ; stockage 0x09080706
    shr     eax, 16         ; chiffres 5 à 8
    pdep    edx, eax, ecx   ; edx = 0x03040506
    movbe   [esp + 20], edx ; stockage 0X06070809
```

Listing 17.5.10 – BCD - version 2 - Conversion

Pour trouver la longueur du nombre, il suffit de parcourir le tableau jusqu'à trouver un chiffre non nul.

```
    lea     edi, [esp + 16]
.find_length:
    movzx   eax, byte [edi]
    test    al, al
    jnz     .end_find_length
    add     edi,1
    jmp     .find_length
.end_find_length:
```

Listing 17.5.11 – BCD - version 2 - Trouver la longueur du nombre

Assembleur avec codage BCD

Ces deux versions assembleur sont totalement inefficaces et prennent plus de 23 secondes pour s'exécuter sur AMD Ryzen 5 5600g.

17.5.7 Versions 7 - Division par 10000

Enfin, une dernière version consiste à diviser les nombres par 10_000 bien évidemment quand ceux-ci sont supérieurs ou égaux à cette valeur. Cela nécessite de créer une table de 10_000 entiers non signés sur 32 bits qui contiennent le reste de la division par 10_000 comme on l'a fait pour la division par 100.

```
1    mov     edi, 0xD1B71759 ; 0.0001
2    mov     ebx, eax        ; sauvegarde eax
3    mul     edi             ; multiplie par 0.0001
4    shr     edx, 13
5    imul    eax, edx, 10000 ; eax = 10000*(x/10000)
6    sub     ebx, eax        ; reste dans ebx
7    mov     esi, edx        ; sauvegarde de x/10000
8    ; chargement du reste sous forme de 4 octets
9    mov     edx, [values_10000_inv + ebx * 4]
10   sub     ecx, 4
11   mov     [esp + ecx + 16], edx
12   ; incrémentations de counts
13   movzx   ebx, dh
14   movzx   eax, dl
15   inc     byte [esp + ebx]
16   inc     byte [esp + eax]
17   shr     edx, 16
18   movzx   ebx, dh
19   movzx   eax, dl
20   inc     byte [esp + ebx]
21   inc     byte [esp + eax]
22   mov     eax, esi
```

Listing 17.5.12 – Division par 10000 - Conversion

Le code correspondant pour la conversion par 10_000 est présenté Listing 17.5.12. On commence par multiplier **eax** par $0,0001$ puis on calcule le reste. Ici, on utilise l'instruction **imul** avec trois opérandes au lieu d'un code n'utilisant que des additions et des décalages comme par exemple :

```
1     lea     esi, [edx + edx * 4]    ;    5 * (x/10000)
2     lea     eax, [esi + esi * 4]    ;   25 * (x/10000)
3     lea     esi, [eax + eax * 4]    ;  125 * (x/10000)
4     lea     eax, [esi + esi * 4]    ;  625 * (x/10000)
5     shl     eax, 4                  ; 10000 * (x/10000)
6     sub     ebx, eax
```

qui prend plus de temps à l'exécution.

17.6 Tests de performance

Afin de tester les différentes versions que nous avons écrites sur différentes architectures, nous allons examiner les résultats obtenus pour les méthodes suivantes :

- **1** : version C++ avec conversion de `x` grâce à `std::to_string` et tableaux d'entiers
- **2** : version C++ avec conversion de `x` grâce à `std::to_string` et tableaux d'octets
- **3** : version C++ avec conversion par divisions successives et tableaux d'entiers
- **4** : version C++ avec conversion par divisions successives et tableaux d'octets
- **6** : version assembleur traduction de la méthode 4
- **7** : amélioration de la méthode 6 avec remplacement de la division par une multiplication
- **8** : amélioration de la méthode 7 avec dépliage
- **9** : amélioration de la méthode 8 avec comparaison vectorielle
- **10** : méthode 9 avec mélange d'instructions SSE et AVX
- **11** : méthode 9 avec uniquement des instructions SSE
- **12** : méthode 9 avec uniquement des instructions AVX
- **13** : division par 100 mais pratiquée avec multiplication et table de valeurs
- **14** : amélioration de la méthode 13 avec dépliage de la conversion
- **15** : amélioration de la méthode 14 avec comparaison vectorielle
- **16** : utilisation du format BCD, instructions assembleur classiques
- **17** : utilisation du format BCD, instructions `pdep` et `movbe`
- **23** : division par 10000, version 4

Etant donné que nous travaillons avec des entiers non signés, nous ne pouvons trouver que les six premiers nombres auto-descriptifs, le septième nécessitant une représentation sur 64 bits.

Nous avons testé les fonctions sur différents matériels, cependant avec gcc 10 sur certaines machines et gcc 12 sur d'autres ce qui implique des temps de calculs supérieurs avec gcc 10 pour les méthodes 1 à 4. C'est le cas pour le Ryzen 7 1700X pour lequel on utilise gcc 10. On a donc fait figurer la version de gcc utilisée dans les tables D.1 et D.2 qui recensent les résultats d'exécution des méthodes pour les processeurs Intel et AMD respectivement.

L'analyse des résultats montre que la méthode 4, traduite par le compilateur est généralement performante. Cependant les méthodes 13 et 14 qui utilisent des divisions par 100 plutôt que par 10 se révèlent bien plus performantes. Visiblement

N°	Marque Gamme Modèle	Intel Pentium D	Intel Core i7 860	Intel Core i5 3570k	Intel Core i3 6100	Intel Core i5 7400	Intel Core i7 4790
1	cpp 32 bits	42.74	20.40	19.03	15.91	16.83	10.38
2	cpp 8 bits	40.36	19.39	15.04	11.58	13.52	11.06
3	tabs 32 bits	29.86	14.68	12.69	10.92	11.54	7.77
4	tabs 8 bits	26.44	14.22	10.60	7.08	7.53	7.95
6	asm v1	**95.46**	**29.58**	**23.28**	**19.03**	**19.78**	**20.40**
7	asm v2	**95.20**	**29.57**	**22.82**	**18.61**	**19.47**	**20.18**
8	asm v3	24.02	12.02	9.63	6.87	7.28	7.47
9	asm v4	–	–	–	6.71	7.10	7.33
10	asm v5 SSE/AVX	–	–	–	8.64	9.37	**31.74**
11	asm v6 SSE	37.36	12.09	10.23	8.64	9.40	8.35
12	asm v7 AVX	–	–	–	8.41	9.22	8.14
13	asm v100	21.79	9.94	7.86	6.01	6.50	6.36
14	asm v101	21.55	10.19	7.85	6.02	6.38	6.35
15	asm v102	–	–	9.01	7.32	7.96	7.23
16	asm bcd v1	**108.34**	**48.10**	**47.30**	**44.65**	**47.93**	**42.44**
17	asm bcd v2	–	–	–	**44.53**	**47.59**	**43.46**
23	asm v10004	**20.56**	**7.94**	**6.81**	**4.85**	**5.20**	**5.63**
	ratio (1 / 23)	2.07	2.56	2.79	3.28	3.23	1.84
	ratio (4 / 23)	1.28	1.79	1.13	1.45	1.44	1.41

TABLE 17.3 – Résultats comparatifs des méthodes pour la recherche des nombres auto-descriptifs pour les architectures anciennes Intel

la méthode 13 est la plus performante chez Intel alors que la méthode 14 l'est chez AMD.

Mais c'est la méthode des divisions par 10000 qui est au final la plus intéressante.

On notera que pour l'Intel i5 12400F, utiliser à la fois le SSE et l'AVX cause problème avec un temps d'exécution de plus de 50 secondes. Alors que pour les autres architectures ce problème n'existe pas.

N°	Marque Gamme Modèle	Intel Core i7 8700	Intel Core i5 10400f	Intel Core i7 10850h	Intel Core i5 12400
1	`cpp 32 bits`	13.35	14.98	8.88	6.25
2	`cpp 8 bits`	9.50	11.15	8.44	5.91
3	`tabs 32 bits`	9.90	10.22	5.47	3.82
4	`tabs 8 bits`	5.78	6.87	5.20	3.88
6	`asm v1`	**15.83**	**18.74**	**14.30**	**5.93**
7	`asm v2`	**15.39**	**17.69**	**14.03**	**5.77**
8	`asm v3`	5.73	6.56	5.29	3.51
9	`asm v4`	5.58	6.35	5.04	3.39
10	`asm v5 SSE/AVX`	7.06	8.22	6.49	**58.40**
11	`asm v6 SSE`	7.05	8.34	6.50	5.17
12	`asm v7 AVX`	6.87	8.07	6.29	5.17
13	`asm v100`	4.97	5.87	4.57	2.82
14	`asm v101`	4.86	5.76	4.60	2.87
15	`asm v102`	6.18	7.01	5.51	4.64
16	`asm bcd v1`	**36.89**	**42.69**	**33.55**	**35.14**
17	`asm bcd v2`	**38.07**	**43.27**	**33.37**	**35.81**
23	`asm v10004`	**4.00**	**4.56**	**3.96**	**2.43**
	ratio (1 / 23)	3.33	3.28	2.24	2.57
	ratio (4 / 23)	1.44	1.50	1.31	1.59

TABLE 17.4 – Résultats comparatifs des méthodes pour la recherche des nombres auto-descriptifs pour les architectures récentes Intel

N°	Marque Gamme Modèle	AMD Phenom 1090T 2009	AMD Athlon 200 GE 2018	AMD Ryzen 7 1700X 2017	AMD Ryzen 5 5600g 2020
	gcc	10	10	10	12
1	`cpp 32 bits`	22.33	17.20	15.30	12.34
2	`cpp 8 bits`	21.45	15.90	13.15	8.01
3	`tabs 32 bits`	15.54	11.24	10.05	9.19
4	`tabs 8 bits`	12.52	8.75	7.87	5.09
6	`asm v1`	**39.17**	**26.04**	**23.83**	5.81
7	`asm v2`	**39.29**	**26.03**	**23.70**	5.80
8	`asm v3`	12.01	9.24	8.36	4.44
9	`asm v4`	–	9.29	8.36	4.40
10	`asm v5 SSE/AVX`	–	11.89	10.71	5.52
11	`asm v6 SSE`	12.53	11.84	10.73	6.10
12	`asm v7 AVX`	–	11.90	10.71	5.48
13	`asm v100`	9.78	7.96	7.25	3.67
14	`asm v101`	9.75	7.55	6.29	3.66
15	`asm v102`	–	9.64	8.71	5.08
16	`asm bcd v1`	**35.83**	**33.12**	**29.67**	**23.53**
17	`asm bcd v2`	–	**67.41**	**61.35**	**24.56**
23	`asm v10004`	**8.77**	**5.63**	**5.57**	**3.12**
	ratio (1 / 23)	2.54	3.05	2.74	3.95
	ratio (4 / 23)	1.42	1.55	1.41	1.63

TABLE 17.5 – Résultats comparatifs des méthodes pour la recherche des nombres auto-descriptifs pour les architectures AMD

Annexe A

Conventions d'appel Linux

Nous récapitulons Table A.1 les conventions d'appel sous Linux en 32 et 64 bits. Pour une vision exhaustive des conventions d'appel on pourra se référer à [1].

Catégorie	**Linux 32 bits**	**Linux 64 bits**
Registres modifiables par l'appelé	`EAX`, `ECX`, `EDX`, `ST0-ST7`, `XMM0-XMM7`	`RAX`, `RCX`, `RDX`, `RSI`, `RDI` `R8-R11`, `ST0-ST7` `XMM0-XMM15`
Registres à préserver dans l'appelé si modifiés	`EBX`, `EBP`, `ESI`, `EDI`	`RBX`, `RBP` `R12-R15`
Paramètres	Pile `[ebp+8]`, `[ebp+12]`, ...	`RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`, (entiers) `XMM0-7` (flottants)

Valeur de retour
- entier
- flottant
	
	`EAX`, `EDX` :`EAX`
	`ST0`
		
		`RAX`, `RDX` :`RAX`
		`XMM0`

Appel rapide (fast call)	`ECX`, `EDX`	mode par défaut

TABLE A.1 – Conventions d'appel Linux 32 et 64 bits

Note : en architecture 64 bits si le sous-programme appelé possède plus de six paramètres entiers ou plus de huits paramètres flottants alors les paramètres restants seront placés dans la pile.

Annexe B

Le GNU Débogueur

GDB le GNU débogueur est un logiciel qui permet de déboguer, c'est à dire de trouver des bogues (ou *bugs* en anglais) dans un programme. La plupart des problèmes que l'on rencontre lors de la phase de débogage d'un programme concerne les pointeurs ou le débordement de pile losrque l'on appelle de manière récursive une fonction.

Pour analyser son programme il suffit de compiler les sources avec les options de débogage :

- pour **nasm** il s'agit de `-g -F dwarf` sous Linux
- pour les compilateurs **C/C++**, on utilise l'option `-g` ou `-ggdb`

Une fois l'exécutable obtenu, on lance **gdb** ou l'un des programmes basés sur **gdb** et qui dispose d'une interface graphique comme **xxgdb** ou **ddd**, le *Data Display Debugger*. Cependant, certaines commandes de **gdb** sont intéressantes à connaître pour être utilisées dans ces interfaces graphiques qui sont parfois un peu rudimentaires ou capricieuses lors de l'affichage.

B.1 Programme de démonstration

Le programme sur lequel nous allons travailler est celui du Listing B.1.1.

On notera deux bogues dans ce programme :

- en lignes 25 et 29, alors qu'on a déclaré un tableau de 10 entiers, on utilise une onzième valeur
- en ligne 34, l'appel récursif de la fonction va provoquer une saturation de la pile

```cpp
#include <iostream>
#include <string>
using namespace std;

int *table;

typedef struct Person {
    string nom, prenom, adresse;
    int t;
} Person;

int function( int n, Person p ) {
    if (n == 0) {
        return p.t;
    } else {
        return 1 + function( n-1, p );
    }
}

int main() {

    table = new int [ 10 ];

    for (int i = 0; i < 11; ++i) {
        table[ i ] = i;
    }

    for (int i = 0; i < 11; ++i) {
        cout << table[ i ] << " ";
    }

    Person person;
    person.t = 1;
    cout << function( 1000000, person );

    return EXIT_SUCCESS;
}
```

Listing B.1.1 – Programme comportant quelques bogues

B.2 Compilation et exécution

On commence par compiler le programme avec l'option `-ggdb` et sans options d'optimisation car celles-ci pourraient par exemple dérécursiver la fonction qui posera problème par la suite.

```
g++ -o test_gdb.exe test_gdb.cpp -Wall -std=c++11 -ggdb
```

Puis, on lance **gdb** sur l'exécutable :

```
gdb ./test_gdb.exe

GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./test_gdb.exe...
(gdb)
```

A partir du prompt de **gdb**, on tape la commande `run` afin de lancer l'exécution du programme :

```
(gdb) run
Starting program: test_gdb.exe
test_gdb.exe: malloc.c:2379: sysmalloc: Assertion `(old_top ==
initial_top (av) && old_size == 0) || ((unsigned long) (old_size) >= MINSIZE
&& prev_inuse (old_top) && ((unsigned long) old_end & (pagesize - 1)) == 0)'
failed.

Program received signal SIGABRT, Aborted.
__GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:50
50        ../sysdeps/unix/sysv/linux/raise.c: Aucun fichier ou dossier de ce type.
```

On lance ensuite la commande `backtrace` (`bt` en abrégé) afin de visualiser les appels de fonctions. Celle-ci affiche des *frames*, c'est à dire les différents appels de sous-programmes qui sont réalisés.

```
(gdb) bt
#0  __GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:50
#1  0x00007ffff7bcc859 in __GI_abort () at abort.c:79
#2  0x00007ffff7c3f30a in __malloc_assert (...) at malloc.c:298
```

```
#3  0x00007ffff7c4196f in sysmalloc (nb=nb@entry=1040, av=av@entry=0x7ffff7d96b80
<main_arena>) at malloc.c:2379
#4  0x00007ffff7c427c3 in _int_malloc (av=av@entry=0x7ffff7d96b80 <main_arena>,
bytes=bytes@entry=1024) at malloc.c:4141
#5  0x00007ffff7c44184 in __GI___libc_malloc (bytes=1024) at malloc.c:3058
#6  0x00007ffff7c2bd34 in __GI__IO_file_doallocate (...) at filedoalloc.c:101
#7  0x00007ffff7c3bf00 in __GI__IO_doallocbuf (...) at libioP.h:948
#8  0x00007ffff7c3af60 in _IO_new_file_overflow (...) at fileops.c:745
#9  0x00007ffff7c396e5 in _IO_new_file_xsputn (...) at libioP.h:948
#10 _IO_new_file_xsputn (...) at fileops.c:1197
#11 0x00007ffff7c2d3f1 in __GI__IO_fwrite (...) at libioP.h:948
#12 0x00007ffff7edc4a8 in ... from /lib/x86_64-linux-gnu/libstdc++.so.6
#13 0x00007ffff7eead5f in std::ostream& std::ostream::_M_insert<long>(long) ()
from /lib/x86_64-linux-gnu/libstdc++.so.6
#14 0x000055555555535c in main () at test_gdb.cpp:29
(gdb)
```

Dans le cas présent c'est la dernière frame (la frame 14) qui nous indique qu'à partir de la ligne 29 de notre programme, dans la fonction `main`, une série d'appels à provoquer l'erreur.

On se redirige donc vers le code source de l'erreur en tapant :

```
(gdb) frame 14
#14 0x000055555555535c in main () at test_gdb.cpp:29
29                      cout << table[ i ] << " ";
(gdb)
```

C'est donc l'instruction d'affichage qui a mené à l'erreur.

B.3 Afficher les données

On peut donc se demander ce que contient le tableau d'entiers **`table`**. Pour l'afficher on utilise la commande `x/` suivie du nombre d'éléments à afficher, ainsi que le format d'affichage et le type de données à afficher. On peut se référer à la Table B.1 pour connaître la manière d'afficher les données.

On affiche par exemple les 20 double mots à partir du tableau **`table`** :

```
(gdb) x/20d table
0x55555556aeb0:        0       1       2       3
0x55555556aec0:        4       5       6       7
0x55555556aed0:        8       9       10       0
0x55555556aee0:        0       0       0       0
0x55555556aef0:        0       0       0       0
```

Format	Type
d (décimal)	b (octet - 8 bits)
u (décimal non signé)	h (mot - 16 bits)
t (binaire)	w (double mot - 32 bits)
o (octal)	g (giant - 64 bits)
x (hexadécimal)	
f (float)	
a (address)	
i (instruction)	
c (char)	
s (string)	

TABLE B.1 – Format et type d'affichage de gdb

On peut également afficher la variable `i` de deux manières différentes :

```
(gdb) x/1d &i
0x7fffffffdb0c:        0
(gdb) print i
\$1 = 0
```

Néanmoins, on comprend mal pourquoi le programme aurait provoqué une erreur lors de l'affichage de la première valeur. En fait, la génération de cette erreur provient de la boucle précédente et de l'affectation de la onzième valeur à la ligne 25 du programme.

B.4 Electric Fence

Pour détecter cette erreur il existe un utilitaire appelé *Electric Fence* dont le but est de se concentrer sur deux types d'erreurs :

- l'accès en dehors d'un espace mémoire alloué dynamiquement
- l'accès à une zone mémoire désallouée par `free()`

Pour installer *Electric Fence*, il suffit d'installer le paquet du même nom :

```
sudo apt install electric-fence
dpkg -L electric-fence
/.
/usr
/usr/lib
```

```
/usr/lib/libefence.a
/usr/lib/libefence.so.0.0
...
/usr/lib/libefence.so
/usr/lib/libefence.so.0
```

On voit que les librairies sont installées dans `/usr/lib`. On lance alors `gdb`, puis dans la console de `gdb`, on saisit la ligne suivante avant de lancer l'exécution du programme ce qui permet de charger la librairie :

```
(gdb) set environment LD_PRELOAD=/usr/lib/libefence.so
(gdb) run
Starting program: test_gdb.exe

  Electric Fence 2.2 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

  Electric Fence 2.2 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>

Program received signal SIGSEGV, Segmentation fault.
0x0000555555555317 in main () at test_gdb.cpp:25
25                          table[ i ] = i;
(gdb) print i
\$1 = 10
```

Electric Fence indique que l'erreur se situe sur la ligne 25, on affiche alors `i` qui vaut 10. Or le tableau est de 10 entiers et on ne peut donc manipuler que les indices 0 à 9 du tableau. On corrige alors les erreurs dans les deux boucles `for` en lignes 24 et 28 en remplaçant 11 par 10.

B.5 Erreur liée au débordement de pile

Après correction des erreurs, on recompile le programme et on relance `gdb` :

```
(gdb) run
Starting program: test_gdb.exe

Program received signal SIGSEGV, Segmentation fault.
0x000055555555547d in Person::Person (this=<error reading variable:
Cannot access memory at address 0x7fffff7feff8>) at test_gdb.cpp:7
7       typedef struct Person {
```

L'erreur n'est pas facile à comprendre car elle masque la saturation de la pile. Dans ce cas il faut utiliser la commande `backtrace` avec un argument de -1 afin de connaître le nombre total de frames :

```
(gdb) bt -1
#52396 0x0000555555555398 in main () at test_gdb.cpp:34
```

Il existe donc 52396 appels de sous-programmes dans la pile. Il suffit alors de relancer la commande `backtrace` sans argument :

```
#0  0x000055555555547d in Person::Person (this=<error reading variable:
Cannot access memory at address 0x7fffff7feff8>) at test_gdb.cpp:7
#1  0x000055555555523d in function (n=947606, p=...) at test_gdb.cpp:16
#2  0x0000555555555254 in function (n=947607, p=...) at test_gdb.cpp:16
#3  0x0000555555555254 in function (n=947608, p=...) at test_gdb.cpp:16
...
#52394 0x0000555555555254 in function (n=999999, p=...) at test_gdb.cpp:16
#52395 0x0000555555555254 in function (n=1000000, p=...) at test_gdb.cpp:16
#52396 0x0000555555555398 in main () at test_gdb.cpp:34
```

On voit qu'il existe 52395 appels récursifs au sous-programme appelé **`fonction`**.

On peut obtenir la valeur du sommet de pile grâce à la commande `info registers rsp` en se plaçant dans la frame 0 et la frame 52396 :

```
(gdb) frame 0
#0  0x000055555555547d in Person::Person (this=<error reading
variable: Cannot access memory at address 0x7fffff7feff8>)
at test_gdb.cpp:7
7        typedef struct Person {
(gdb) info registers rsp
rsp            0x7fffff7feff0      0x7fffff7feff0
(gdb) frame 52396
#52396 0x0000555555555398 in main () at test_gdb.cpp:34
34                cout << function( 1000000, person );
(gdb) info registers rsp
rsp            0x7fffffffdb00      0x7fffffffdb00
```

On a donc une différence entre la frame 0 et la frame 52396 de $\text{ffdb00}_{16} - \text{7feff0}_{16} = \text{7feb10}_{16}$, soit 8_383_248 octets. Sachant que la pile possède une taille de 8 Mo $= 8\_388\_608$ octets, on a saturé la pile car lors du lancement du programme on a utilisé une partie de la pile.

Deux possibilités s'offrent à nous :

- soit modifier la taille de la pile en utilisant la command `ulimit` dans un terminal, mais cela n'est généralement pas recommandé
- soit revoir notre programme en dérécursivant le sous-programme

B.6 Autres commandes

B.6.1 Afficher le programme

Il s'agit de la commande `list` éventuellement suivie de la ligne du programme :

```
(gdb) list
6
7        typedef struct Person {
8                string nom, prenom, adresse;
9                int t;
10        } Person;
11
12        int function( int n, Person p ) {
13                if (n == 0) {
14                        return p.t;
15                } else {
(gdb) list 24
19
20        int main() {
21
22                table = new int [ 10 ];
23
24                for (int i = 0; i < 10; ++i) {
25                        table[ i ] = i;
26                }
27
28                for (int i = 0; i < 10; ++i) {
```

B.7 Afficher le contenu des registres

On utilise la commande `info registers` qui peut être suivie du nom du registre ou alors de la commande `print` :

```
(gdb) info registers
rax            0x555555558040      93824992247872
rbx            0x555555555590      93824992236944
rcx            0x20                32
rdx            0x7ffff7f903d0      140737353679824
rsi            0x555555556005      93824992239621
rdi            0x7ffff7d987e0      140737351616480
rbp            0x7fffffffdc70      0x7fffffffdc70
rsp            0x7fffffffdb70      0x7fffffffdb70
r8             0x1                 1
```

```
r9             0x0                0
r10            0x7ffff7de7cbc     140737351941308
r11            0x7ffff7eea690     140737353000592
r12            0x5555555550e0     93824992235744
r13            0x7fffffffdd60     140737488346464
r14            0x0                0
r15            0x0                0
rip            0x555555555319     0x555555555319 <main()+122>
eflags         0x293              [ CF AF SF IF ]
cs             0x33               51
ss             0x2b               43
ds             0x0                0
es             0x0                0
fs             0x0                0
gs             0x0                0
(gdb) info registers xmm0
xmm0           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0},
  v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0,
  0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0},
  uint128 = 0x0}
(gdb) print $xmm0.v8_int16
$4 = {0, 0, 0, 0, 0, 0, 0, 0}
(gdb) print $eax
$5 = 1431666752
```

B.8 Afficher le contenu des variables

Pour afficher le contenu d'une variable on utilise `print` :

```
(gdb) print i
$1 = 2
(gdb) print table[i]
$2 = 2
(gdb) print table[0]
$3 = 0
```

B.9 Modifier le contenu des registres ou des variables

Quand on désire modifier le contenu d'une variable ou d'un registre on utiliser `set` :

```
(gdb) set variable i=4
(gdb) set $eax=1
(gdb) print $eax
$4 = 1
```

B.10 Points d'arrêt

Les points d'arrêt (ou *breakpoints* en anglais) permettent d'arrêter l'exécution du programme sur une instruction particulière. Il faut pour cela spécifier le nom du fichier et/ou la ligne :

```
(gdb) break 24
Breakpoint 1 at 0x12cb: file test_gdb.cpp, line 24.
(gdb) break test_gdb.cpp:25
Breakpoint 3 at 0x5555555552fa: file test_gdb.cpp, line 25.
```

Il est également possible de déclencher un point d'arrêt si une condition est réalisée. Par exemple, on veut arrêter l'exécution du programme dans une boucle, si la variable de boucle a pour valeur 5 :

```
(gdb) break test_gdb.exe:29 if (i==5)
Breakpoint 1 at 0x1319: file test_gdb.cpp, line 29.
(gdb) run
Starting program: test_gdb.exe

Breakpoint 1, main () at test_gdb.cpp:29
29                          cout << table[ i ] << " ";
(gdb) print i
$1 = 5
```

B.11 Surveiller un changement de valeur

Une autre fonctionnalité intéressante est le *watchpoint* qui permet de stopper l'exécution du programme dès lors qu'une valeur change. On peut définir des points de surveillance lors de la lecture ou de l'écriture d'une variable (`watch`, `rwatch`, `awatch`) :

```
(gdb) rwatch table
Hardware read watchpoint 1: table
(gdb) run
Starting program: test_gdb.exe
```

```
5
6  Hardware read watchpoint 1: table
7
8  Value = (int *) 0x55555556aeb0
9  0x0000555555555301 in main () at test_gdb.cpp:25
10 25                        table[ i ] = i;
```

Il existe beaucoup d'autres commandes à utiliser avec **gdb** comme les commandes de base :

- `continue` (abrégé en `c`) pour continuer l'exécution du programme jusqu'à ce qu'il se termine ou qu'un point d'arrêt ou de surveillance soit déclenché
- `next` (abrégé en `n`) qui exécute la prochaine instruction et passe à la suivante
- `step` (abrégé en `s`) qui exécute la prochaine instruction mais s'il s'agit d'un appel de fonction on s'arrêtera à l'intérieur de la fonction

On pourra également se référer au Wiki de **gdb** : https://sourceware.org/gdb/wiki/.

Annexe C

Travail sur bsr

C.1 Introduction

Dans cette annexe nous investiguons de manière plus détaillée les résultats liés à l'utilisation de l'instruction assembleur `bsr` et de son implantation en comparaison également avec l'instruction `lzcnt`.

Nous rappelons que l'instruction `bsr` (*Bit Scan Reverse*) recherche le bit le plus significatif (le plus à gauche) d'une opérande 16, 32 ou 64 bits et stocke le résultat dans un registre de même taille. Cependant si l'opérande source contient la valeur 0, le résultat de l'opération n'est pas défini et dans ce cas le bit `ZF` du registre `eflags` est positionné à 1, sinon il sera positionné à 0.

lzcnt

L'instruction `lzcnt`, quant à elle, compte le nombre de 0 significatifs. Elle est donc proche de `bsr` mais elle nécessite de soustraire à la taille de son opérande, le résultat qu'elle produit afin d'obtenir la même valeur que `bsr`. Contrairement à `bsr`, `lzcnt` produit toujours un résultat. Si l'opérande source est égale à 0, alors elle retournera la valeur 32 !

On notera que :

- l'instruction `lzcnt` fait normalement partie du jeu d'instruction `ABM` pour AMD et `BMI1` pour Intel.
- pour les microprocesseurs qui ne supportent pas l'instruction `lzcnt`, cette dernière est remplacée par l'instruction `bsr` ce qui risque de fausser les résultats. Par exemple l'Intel Q6600 ne supporte pas l'instruction `lzcnt`.

Pour résumer :

```
1 mov     ebx, 129
2 bsr     eax, ebx
```

et le code suivant

```
1 mov     ebx, 129
2 mov     eax, 31
3 lzcnt   ebx, ebx
4 sub     eax, ebx
```

donneront le même réslultat, soit 7 dans le registre `eax`.

C.2 Comparaison bsr et lzcnt

Un premier test préalable consiste à comparer deux implantations basiques afin de comparer l'efficacité des instructions assembleur `bsr` et `lzcnt`. Les résultats figurent Table C.1 : on donne trois valeurs.

	AMD Ryzen 7 1700X	AMD Ryzen 5 3600	Intel Core i3 6100	Intel Core i5 7400	Intel Core i7 8700	Intel Xeon 4208
bsr	10,51	10.35	15.01	13.50	10.62	10.35
lzcnt	2.63	2.59	15.00	18.93	10.58	10.16
ratio	3.99	3.99	1.00	0.71	1.00	1.00

TABLE C.1 – Nombre moyen de milliards de cycles sur 10 exécutions pour une boucle de 3 milliards d'appels à bsr ou 31-lzcnt.

La première correspond au nombre de milliards de cycles processeur pour l'éxécution d'une boucle qui réalise 3 milliards d'appels de l'instruction `bsr` appliquée sur un registre 32 bits. La seconde fait la même chose mais en calculant 31 moins le résultat de `lzcnt`. Enfin, la troisième valeur (*ratio*) est le rapport des deux valeurs précédentes.

On s'aperçoit que `lzcnt` est plus rapide que `bsr` sur certaines architectures, cela correspond au cas ou la valeur *ratio* est plus grande que 1.

C.3 Code à traduire

Le code à traduire est le suivant :

```
u32 method(u32 *t, u32 size) {
    u32 sum = 0;
    for (u32 i = 0; i < size; ++i) {
        sum += function_bsr(t[i]);
    }
    return sum;
}
```

Il consiste à réaliser la somme des résultats de la fonction `function_bsr` appliquée sur chacun des éléments d'un tableau de valeurs entières. La fonction `function_bsr` peut alors être implantée de différentes manières :

- soit sous la forme d'une fonction C qui reproduit le comportement de l'instruction assembleur `bsr` dans le cas où on ne pourrait avoir accès à `bsr` grâce au compilateur
- soit par un appel direct à `bsr`, si le compilateur le permet, c'est le cas de `gcc`
- soit par un appel à la fonction `_builtin_clz` du compilateur `gcc`, qui va généralement remplacer `bsr` par `lzcnt`

On codera également en assembleur le sous-programme `method` en utilisant `bsr` ou `lzcnt` et en applicant un dépliage de boucle ou en essayant de supprimer les dépendances lors des calculs.

Enfin, il existe une instruction vectorielle du jeu AVX512 appelée `vplzcntd` qui permet donc de vectoriser le code et elle se révèle très efficace comme le montre les résultats ci-après. Voici un aperçu du code vectorisé en utilisant des registres de 128 bits :

```
        mov     eax, 31
        movd    xmm7, eax
        pshufd  xmm7, xmm7, 0          ; xmm7 = [31,31,31,31]
        pxor    xmm0, xmm0             ; xmm0 = [0,0,0,0]

        ; boucle dépliée par 4
.for:
        movdqa  xmm1, xmm7             ; xmm1 = [31,31,31,31]
        vplzcntd xmm2, [ebx + ecx * 4] ; xmm2 = lzcnt(t[i:i+3])
        psubd   xmm1, xmm2             ;
        paddd   xmm0, xmm1
        add     ecx, 4
        cmp     ecx, edx
        jne     .for

        phaddd  xmm0, xmm0             ; calcul du résultat final
        phaddd  xmm0, xmm0
        movd    eax, xmm0
```

Le registre `xmm0` représente la variable `sum`. Le registre `xmm7` stocke quatre fois la valeur 31 et sera placé à chaque itération de la boucle dans `xmm1`. On soustraira à

`xmm1` le résultat de `vplzcntd` qui sera placé dans `xmm2`.

C.4 Résultats

Nous présentons Table C.2 les résultats obtenus lors de tests sur différentes machines. Les différentes implantation sont les suivantes :

1. Code C avec appel d'une fontion `bsr` écrite en C
2. Code C avec appel de la fonction `__builtin_clz`
3. Code C avec appel de l'instruction assembleur `bsr`
4. Code assembleur avec boucle simple utilisant `bsr`
5. Code assembleur avec boucle dépliée par 4 utilisant `bsr`
6. Code assembleur avec boucle dépliée par 4 utilisant `bsr` et élimination des dépendances entre registres
7. Code assembleur avec boucle simple utilisant de 31 -`lzcnt`
8. Code assembleur avec boucle dépliée par 4 utilisant 31 -`lzcnt` et élimination des dépendances entre registres
9. vectorisation en AVX512, seulement disponible sur Xeon Silver 4208

N°	Méthode	AMD Ryzen 7 1700X	AMD Ryzen 5 3600	Intel Core i3 6100	Intel Core i5 7400	Intel Core i7 8700	Intel Xeon 4208
1	C function bsr	10.12	8.79	13.80	14.59	10.79	12.80
2	C builtin clz	1.38	1.19	1.05	1.15	0.86	**0.10**
3	C bsr	1.13	0.95	0.29	**0.31**	**0.23**	0.53
4	asm bsr	1.08	0.95	0.56	0.60	0.45	0.73
5	asm bsr ur4	1.08	0.96	0.30	**0.31**	**0.23**	0.39
6	asm bsr ur4 nodep	1.06	0.96	**0.28**	**0.31**	**0.23**	0.36
7	asm lzcnt	0.50	**0.44**	0.59	0.62	0.47	0.73
8	asm lzcnt ur4 nodep	**0.49**	**0.44**	0.59	0.63	0.47	0.79
9	asm avx512 vplzcnt	N/A	N/A	N.A	N/A	N/A	**0.00**

TABLE C.2 – Résultats comparatifs d'implantation de la fonction bsr

Concernant l'AVX512, le temps d'exécution est très faible et donc n'affiche que 0 secondes, en fait il s'agit d'environ 45 μ secondes.

Annexe D

Implantation de la fonction signe

D.1 Introduction

On désire implanter de la manière la plus efficace possible, la fonction **signe** (*sign* en anglais) d'un entier signé, qui donne le signe de ce dernier :

```
i32 sign( i32 x ) {
    if (x < 0)
        return -1;
    else if (x > 1)
        return 1;
    else
        return 0;
}
```

Une traduction intelligente de cette fonction en assembleur x86 32 bits est la suivante :

```
sign:
        push        ebp
        mov         ebp, esp
        mov         eax, [ebp + 8]
        cmp         eax, 0          ; on sort avec 0 si x = 0
        jz          .end
        mov         eax, 1
        jg          .end
        mov         eax, -1
.end:
        mov         esp, ebp
        pop         ebp
        ret
```

Elle consiste à charger la valeur de la variable `x` dans **eax** puis à la comparer à la constante 0. Si le résultat de la comparaison indique 0 on se rend à la fin de la fonction et on sort avec la valeur 0. Sinon on met la valeur 1 dans **eax** et si la

comparaison précédente avec 0 nous indique que eax était supérieur à 0, on sort de la fonction. Finalement, si x n'est pas égal à 0 ou supérieur à 0, c'est qu'il est inférieur à 0 et on met la valeur -1 dans eax, qui sera la valeur de retour de la fonction. Dans cette implantation, on utilise donc deux sauts conditionnels ce qui n'est pas forcément une bonne chose.

Il est donc nécessaire de trouver une implantation plus performante de la fonction.

D.2 Amélioration sans passer par ebp

La première amélioration à réaliser consiste à ne pas passer par ebp pour récupérer x mais à passer par esp :

```
sign:
        mov     eax, [esp + 4]
        cmp     eax, 0          ; on sort avec 0 si x = 0
        jz      .end
        mov     eax, 1
        jg      .end
        mov     eax, -1
.end:
        ret
```

On pourrait également réaliser un appel de type *fastcall* ce qui implique que x serait placée dans ecx, mais nous allons nous cantonner à une récupération de la valeur de x par l'intermédiaire de esp.

D.3 Amélioration avec suppression d'un saut

Il est possible de commencer par supprimer un saut en utilisant l'instruction setCC. Plutôt que de mettre x dans eax, on le place dans edx. On traite le cas pour lequel x est positif en fixant les flags grâce à l'instruction test, puis les deux autres cas (négatif ou nul) grâce à setnz qui mettra la valeur 1 dans eax si x est non nulle et 0 si x est nulle. Il ne reste plus qu'à prendre le complémentaire de eax pour obtenir une valeur négative.

```
jmr3:
        mov     edx, [esp + 4]
        mov     eax, 1          ; valeur de retour 1
        test    edx, edx        ; si x > 0 sortir avec true
        jg      .L1
        setnz   al              ; si x != 0 al=1 sinon al=0
        neg     eax
align 16
```

```
.L1:
        ret
```

D.4 Améliorations sans saut

Une amélioration qui devrait être plus performante consiste à supprimer les deux sauts en combinant plusieurs séquences d'instructions de manière bien particulière. Voici dans ce qui suit, trois exemples de suppression des sauts conditionnels.

D.4.1 Conversion et négation

La première version sans saut combine trois instructions dans l'ordre suivant : **cdq**, **neg**, **adc**.

```
jmr4:
        mov     eax, [esp + 4]
        cdq                     ; edx:eax = x
                                ; si eax < 0, edx = -1 (0xFF_FF_FF_FF)
                                ; si eax >= 0, edx = 0
        neg     eax             ; si eax = 0  CF=0
                                ; sinon CF = 1
        adc     edx, edx        ; si eax = 0, edx = 0
                                ; sinon eax = 1
        mov     eax, edx
        ret
```

L'instruction **cdq** convertit la valeur contenue dans **eax** dans **edx:eax** en étendant le signe. Si **eax** contient une valeur positive ou nulle, **edx** sera égal à 0. Sinon, si **eax** contient une valeur négative (bit de poids fort à 1), **edx** contiendra -1, soit au format hexadécimal : $FF\_FF\_FF\_FF$.

On calcule ensuite le complémentaire (du point de vue de la notation binaire en complément à deux) de **eax**. Ici la valeur résultante ne sera pas utilisée mais ce qui est intéressant c'est que l'instruction **neg** fixe le *Carry* flag à 1 si **eax** est différent de 0, et à 0 si **eax** vaut 0.

Dès lors, il suffit d'appliquer **adc**, l'addition avec prise en compte de la retenue sur **edx** pour obtenir la valeur à retourner et la placer dans **eax**. On a donc l'évoluation du calcul suivant :

Instruction / cas	$eax < 0$	$eax = 0$	$eax > 0$
cdq (**edx:eax**)	-1 :x	0 :0	0 :x
neg eax	CF=1	CF=0	CF=1
adc edx, edx	$-1 + -1 + 1$	$0 + 0 + 0$	$0 + 0 + 1$
Résultat	-1	0	$+1$

D.4.2 Propagation du signe

Une autre possibilité consiste à utiliser le bit de signe en le recopiant grâce à l'instruction `sar`.

```
1  jmr6:
2          mov     eax, [esp + 4]  ; eax = x
3          mov     edx, eax        ; edx = x
4          sar     eax, 31         ; propagation du bit de signe
5          lea     eax, [eax * 2 + 1]
6          add     edx, edx
7          cmovz   eax, edx
8          ret
```

On obtient dans dans `eax` la valeur -1 si la valeur de `x` est négative ou alors la valeur 0 dans le cas contraire. L'utilisation de `lea` permet d'otenir dans `eax` la valeur -1 si `x` est négative ou la valeur 1 dans les autres cas. Il faut ensuite distinguer le cas où `x` est égale à 0, ce qui réalisé par les deux instructions qui suivent.

Instruction / cas	$eax < 0$	$eax = 0$	$eax > 0$
`sar eax`,31	-1	0	0
`lea eax`, [`eax` * 2 + 1]	-1	1	1
`add edx, edx`	$-2x$	0	$2x$
`cmovz eax, edx`	-1	0	1

D.4.3 Déplacements conditionnels

Enfin, la dernière possibilité que nous allons étudier est l'utilisation de deux `cmov`. On place 1 dans `edx` et -1 dans `ecx`. Puis on compare `eax` à 0. Si c'est la cas, on ne procédera à aucun changement et on sortira de la fonction avec la valeur 0. Par contre, si `x` est positive l'instruction `cmovg` va déplacer la valeur de `edx` dans `eax`, donc 1, ou alors si `x` est négative l'instruction `cmovl` va déplacer la valeur de `ecx` dans `eax`, donc -1 :

```
1  jmr9:
2          mov     edx, -1
3          mov     eax, [esp + 4] ; eax = x
4          mov     ecx, 1
5          cmp     eax, 0
6          cmovg   eax, ecx
7          cmovl   eax, edx
8          ret
```

D.5 Tests de performance

Il existe d'autres variantes de ces fonctions mais nous allons nous cantonner à celles exposées ci-avant.

Afin de tester les différentes versions que nous avons écrites sur différentes architectures, nous allons examiner les résultats obtenus pour les méthodes suivantes :

- **1** : version de base
- **2** : version de base améliorée avec utilisation de `esp`
- **3** : version avec suppression d'un saut
- **4** : version sans saut avec cdq neg et adc
- **5** : version sans saut cdq xchg neg adc variante de la version 4
- **6** : version sans sans avec lea sar
- **7** : version avec deux cmov

N°	Marque Gamme Modèle	Intel Core i7 860 2009	Intel Core i3 6100 2015	Intel Core i5 7400 2017	Intel Core i7 4790 2014	Intel Core i7 10850H 2020	Intel Core i5 12400F 2021
1	`jmr1`	.450	.040	15.994			
2	`jmr2`	.450	.040	16.736			
3	`jmr3`	.450	.040	18.456			
4	`jmr4`	.450	.040	13.944			
5	`jmr5`	.450	.040	13.742			
6	`jmr6`	.450	.040	10.052			
7	`jmr7`	.450	.040	10.052			

TABLE D.1 – Résultats comparatifs des méthodes pour la recherche des nombres autodescriptifs pour les architectures Intel

N°	Marque Gamme Modèle	AMD Phenom 1090T 2009	AMD Ryzen 7 1700X 2017	AMD Ryzen 5 5600g 2020
1	`jmr1`	.450	.040	11.668
2	`jmr2`	.450	.040	11.650
3	`jmr3`	.450	.040	11.636
4	`jmr4`	.450	.040	10.048
5	`jmr5`	.450	.040	10.046
6	`jmr6`	.450	.040	10.052
7	`jmr7`	.450	.040	10.052

TABLE D.2 – Résultats comparatifs des méthodes pour la recherche des nombres auto-descriptifs pour les architectures AMD

Annexe E

Code ASCII de 0 à 127

Bin.	Hex.	Dec.	ASCII Symbol	Description
0000000	0	0	NUL	NULL character
0000001	1	1	SOH	Start of Heading
0000010	2	2	STX	Start of TeXt
0000011	3	3	ETX	End of TeXt
0000100	4	4	EOT	End Of Transmission
0000101	5	5	ENQ	Enquiry
0000110	6	6	ACK	Acknowledge
0000111	7	7	BEL	Bell
0001000	8	8	BS	Backspace
0001001	9	9	TAB	Horizontal Tab
0001010	A	10	LF	Line Feed
0001011	B	11	VT	Vertical Tab
0001100	C	12	FF	Form Feed
0001101	D	13	CR	Carriage Return
0001110	E	14	SO	Shift Out
0001111	F	15	SI	Shift In
0010000	10	16	DLE	Data Link Escape
0010001	11	17	DC1	Device Control
0010010	12	18	DC2	Device Control
0010011	13	19	DC3	Device Control
0010100	14	20	DC4	Device Control
0010101	15	21	NAK	Negative Acknowledge
0010110	16	22	SYN	Synchronous Idle
0010111	17	23	ETB	End of Transmission Block
0011000	18	24	CAN	Cancel
0011001	19	25	EM	End of Medium
0011010	1A	26	SUB	Substitute

0011011	1B	27	ESC	Escape
0011100	1C	28	FS	File Separator
0011101	1D	29	GS	Group Separator
0011110	1E	30	RS	Record Separator
0011111	1F	31	US	Unit Separator
0100000	20	32	SP	Space
0100001	21	33	!	Exclamation mark
0100010	22	34		Only quotes above
0100011	23	35	#	Pound sign
0100100	24	36	$	Dollar sign
0100101	25	37	%	Percentage sign
0100110	26	38	&	Commericial and
0100111	27	39	"	Apostrophe
0101000	28	40	(	Left bracket
0101001	29	41	)	Right bracket
0101010	2A	42	*	Asterisk
0101011	2B	43	+	Plus symbol
0101100	2C	44	,	Comma
0101101	2D	45	-	Dash
0101110	2E	46	.	Full stop
0101111	2F	47	/	Forward slash
0110000	30	48	0	
0110001	31	49	1	
0110010	32	50	2	
0110011	33	51	3	
0110100	34	52	4	
0110101	35	53	5	
0110110	36	54	6	
0110111	37	55	7	
0111000	38	56	8	
0111001	39	57	9	
0111010	3A	58	:	Colon
0111011	3B	59	;	Semicolon
0111100	3C	60	<	Smaller than
0111101	3D	61	=	Equals sign
0111110	3E	62	>	Bigger than
0111111	3F	63	?	Question mark
1000000	40	64	@	At symbol
1000001	41	65	A	
1000010	42	66	B	
1000011	43	67	C	

1000100	44	68	D	
1000101	45	69	E	
1000110	46	70	F	
1000111	47	71	G	
1001000	48	72	H	
1001001	49	73	I	
1001010	4A	74	J	
1001011	4B	75	K	
1001100	4C	76	L	
1001101	4D	77	M	
1001110	4E	78	N	
1001111	4F	79	O	
1010000	50	80	P	
1010001	51	81	Q	
1010010	52	82	R	
1010011	53	83	S	
1010100	54	84	T	
1010101	55	85	U	
1010110	56	86	V	
1010111	57	87	W	
1011000	58	88	X	
1011001	59	89	Y	
1011010	5A	90	Z	
1011011	5B	91	[	Left square bracket
1011100	5C	92		Inverse/backward slash
1011101	5D	93	]	Right square bracket
1011110	5E	94	^	Circumflex
1011111	5F	95	_	Underscore
1100000	60	96	‘	Gravis (backtick)
1100001	61	97	a	
1100010	62	98	b	
1100011	63	99	c	
1100100	64	100	d	
1100101	65	101	e	
1100110	66	102	f	
1100111	67	103	g	
1101000	68	104	h	
1101001	69	105	i	
1101010	6A	106	j	
1101011	6B	107	k	
1101100	6C	108	l	

1101101	6D	109	m	
1101110	6E	110	n	
1101111	6F	111	o	
1110000	70	112	p	
1110001	71	113	q	
1110010	72	114	r	
1110011	73	115	s	
1110100	74	116	t	
1110101	75	117	u	
1110110	76	118	v	
1110111	77	119	w	
1111000	78	120	x	
1111001	79	121	y	
1111010	7A	122	z	
1111011	7B	123	{	Left curly bracket
1111100	7C	124	|	Vertical line
1111101	7D	125	}	Right curly brackets
1111110	7E	126		Tilde
1111111	7F	127	DEL	Deletes a character

Glossaire des Instructions

adc Addition de registres généraux en prenant en compte la retenue éventuelle du *Carry Flag* 153

add Addition de registres généraux 153

addps, addpd Additions entre deux registres vectoriels considérés comme 4 floats ou 2 double 233

addss, addsd Additions entre deux registres vectoriels considérés comme contenant un float ou un double en partie basse des registres 183

and Et binaire entre registres généraux ou un registre général et un emplacement mémoire 158

blendps Permet de remplacer les valeurs du registre vectroriel SSE de destination par des valeurs du registre vectoriel source en utilisant un masque sous forme d'une constante d'un octet 239

call Appel de sous programme 162

cdq Conversion de `eax` en `edx:eax`, si `eax` contient une valeur négative alors `edx` contiendra -1 156

cmovCC *Conditional MOVe*, déplacement conditionnel de la valeur source vers la valeur cible, en général deux registres généraux si la condition `CC` est vérifiée 180

cmp Comparaison entre registres généraux ou un registre général et un emplacement mémoire, les flags du registre `eflags` sont positionnés en conséquence 160

cvtss2sd, cvtps2pd, cvtss2si, cvtsi2sd Conversions de données contenu dans des registres vectoriels SSE : de float en double ou de float en entier, d'entier en float 237

dec Décrémentation d'un registre général, correspond à une soustraction de 1 153

div Division de registres généraux, ne prend qu'une opérande : le diviseur . . 155, 157

fadd, faddp *Floating point ADD*, addition de nombres en virgule flottante, le **p** en suffixe indique que la valeur est dépilée de la pile de registres de la FPU 217

fcomi, fcomip, fcomu, fcomup Comparaison de nombres en virgule flottante, le **p** en suffixe indique que la valeur est dépilée de la pile de registres de la FPU 221

fcos, fsin, fsincos, fptan, fpatan Calcul du cosinus, sinus, sinus et cosinus, tangente partielle, arctangente partielle de nombres en virgule flottante 218

fdiv, fdivp, fdivr, fprem Division, division inverse, reste de la division de nombres en virgule flottante, le **p** en suffixe indique que la valeur est dépilée de la pile de registres de la FPU 217

fild *Floating point Integer LoaD*, chargement d'un nombre entier qui sera converti en virgule flottante dans `st0` 215

fld *Floating point LoaD*, chargement d'un nombre en virgule flottante dans `st0` 215

fmul, fmulp *Floating point MULtiplication*, multiplication de nombres en virgule flottante, le **p** en suffixe indique que la valeur est dépilée de la pile de registres de la FPU 217

fst, fstp *Floating point STore*, stockage d'un nombre en virgule flottante vers la mémoire, le **p** en suffixe indique que la valeur est dépilée de la pile de registres de la FPU 216

fsub Soustraction de nombres en virgule flottante, le **p** en suffixe indique que la valeur est dépilée de la pile de registres de la FPU 217

haddps Addition horizontale de 4 float dans un registre vectoriel SSE : cette instruction permet en étant exécutée deux fois sur le même registre de calculer la somme des 4 float contenu dans le registre vectoriel ... 236, 242

idiv Division d'un registre général considéré comme un entier signé : même format que `div` 9, 157

imul Multiplication d'un registre général considéré comme un entier signé : elle possède 3 formats différents 9, 157

inc Incrémentation d'un registre général, correspond à une addition de 1 ... 153

je *Jump on Equal* 162

jg *Jump on Greater* 162

jge *Jump on Greater or Equal* 162

jl *Jump on Less* 162

jle *Jump on Less or Equal* 162

jne *Jump on Not Equal* 162

jnz *Jump on Not Zero* 162

jz *Jump on Zero* 162

lea *Load Effective Address*, calcule le résultat de son opérande définie sous forme d'adresse 157

loop Décrémente le registre `ecx` et se branche à l'adresse indiquée par l'opérande si celui-ci n'est pas égal à 0 161

mov Chargement et stockage de registres généraux 152

movaps, movups Chargement et stockage d'un registre vectoriel SSE avec des nombres en virgule flottante 235

movbe Inverse les octets d'un registre général 32 bits 424

movdqa, movdqu Chargement et stockage d'un registre vectoriel SSE avec des données entières 234

movss, movsd Chargement et stockage de la partie basse d'un registre vectoriel SSE avec un nombre à virgule flottante 235

movsx Chargement et stockage de registres généraux avec extension du signe d'une valeur 8 ou 16 bits vers une valeur 16, 32 ou 64 bits 153

movzx Chargement et stockage de registres généraux avec extension et remplissage avec 0 d'une valeur 8 ou 16 bits vers une valeur 16, 32 ou 64 bits 153

mul Multiplication de registres généraux, ne prend qu'une opérande : le multiplicateur 155

neg Complément à deux : converti 1 en -1 et inversement 157

not Complémente chaque bit d'une opérande 159

or Ou binaire entre registres généraux ou un registre général et un emplacement mémoire 158

paddb, paddw, paddd Additions entre deux registres vectoriels considérés comme contenant 16 octets, 8 mots ou 4 double mots 233

pand, por, pxor Réalise un et-binaire, un ou-binaire ou bien un ou-exclusif binaire entre deux registres vectoriels SSE 237

pdep *Parallel bits Deposit*, agit sur des registres généraux et permet de sélectionner des bits d'un registre 424

pshufd Réorganise les 4 entiers contenus dans un registre vectoriel SSE 238

ret Retour de sous-programme, voir call 162

sar *SHift Arithmetic Right*, permet de réaliser une division par une puissance de 2 tout en préservant le signe de la valeur divisée 160

setCC *Set Byte on Condition*, met à 0 ou 1 un registre 8 bits ou un emplacement mémoire 8 bits en fonction des valeurs des bits CF, SF, OF, ZF et PF du registre eflags : on remplacera `CC` par les lettres qui correcpondent aux sauts conditionnels, par exemple nz pour *Not Zero* 180

shl *SHift Left*, décalage à gauche d'un registre de n bits, correspond également à une multiplication par 2^n 159

shr *SHift Right*, décalage à droite d'un registre de n bits, correspond également à une division par 2^n 159

shufps Réorganise les 4 float contenus dans un registre vectoriel SSE 238

sub Soustraction de registres généraux 153

test Comparaison de valeurs en réalisant un **and** entre les deux opérandes . . 161

xor Ou eXclusif binaire entre registres généraux ou un registre général et un emplacement mémoire 159

Bibliographie

[1] AGNER, Fog : *Calling conventions for different C++ compilers and operating systems*, 2018

[2] AGNER, Fog : *The microarchitecture of Intel, AMD and VIA CPUs : An optimization guide for assembly programmers and compiler makers*, 2018

[3] CORMEN, Thomas H. ; LEISERSON, Charles E. ; RIVEST, Ronald L. ; STEIN, Clifford : *Introduction to Algorithms*. 3rd. MIT Press, 2009. – ISBN 978–0–262–03384–8

[4] HEINEMAN, George T. ; POLLICE, Gary ; SELKOW, Stanley : *Algorithms in a nutshell, A Desktop Quick Reference*. O'Reilly, 2008. – ISBN 978059651624–6

[5] HENNESSY, John L. ; PATTERSON, David A. : *Computer Architecture : A Quantitative Approach*. 5. Amsterdam : Morgan Kaufmann, 2012. – ISBN 978–0–12–383872–8

[6] HENTENRYCK, Pascal V. ; DEVILLE, Yves : The Cardinality Operator : A New Logical Connective for Constraint Logic Programming. In : BENHAMOU, Frédéric (Hrsg.) ; COLMERAUER, Alain (Hrsg.) : *Constraint Logic Programming, Selected Research. WCLP 1991, Marseilles, France*, MIT Press, 1991, S. 283–403

[7] INTEL : *Intel 64 and IA-32 architectures software developer's manual volume 1 : Basic architecture*, January 2019. – Order Number : 253665-069US

[8] INTEL : *Intel 64 and IA-32 architectures software developer's manual volume 2A : Instruction set reference, A-L*, January 2019. – Order Number : 253666-069US

[9] INTEL : *Intel 64 and IA-32 architectures software developer's manual volume 2B : Instruction set reference, M-U*, January 2019. – Order Number : 253667-069US

[10] INTEL : *Intel 64 and IA-32 architectures software developer's manual volume 2C : Instruction set reference, V-Z*, January 2019. – Order Number : 326018-069US

[11] INTEL : *Intel 64 and IA-32 architectures software developer's manual volume 2D : Instruction set reference*, January 2019. – Order Number : 334569-069US

[12] INTEL : *Intel 64 and IA-32 architectures software developer's manual volume 3A : System programming guide, part 1*, January 2019. – Order Number : 253668-069US

[13] INTEL : *Intel 64 and IA-32 architectures software developer's manual volume 3B : System programming guide, part 2*, January 2019. – Order Number : 253669-069US

[14] INTEL : *Intel 64 and IA-32 architectures software developer's manual volume 3C : System programming guide, part 3*, January 2019. – Order Number : 326019-069US

[15] INTEL : *Intel 64 and IA-32 architectures software developer's manual volume 3D : System programming guide, part 3*, January 2019. – Order Number : 332831-069US

[16] INTEL : *Intel 64 and IA-32 architectures software developer's manual volume 4 : Model-specific registers*, January 2019. – Order Number : 335592-069US

[17] JACQUES, Baudé : Le mariage du sciècle : éducation et informatique. In : *1024 - Bulletin de la société informatique de France* 13 (2019), avril, S. 71–78

[18] KNUTH, Donald E. : An empirical study of FORTRAN programs. In : *Software : Practice and Experience* 1 (1971), Nr. 2, 105-133. http://dx.doi.org/10.1002/spe.4380010203. – DOI 10.1002/spe.4380010203

[19] LEITERMAN, James : *32/64bitt 80x86 Assembly Language Architecture*. Plano, TX, USA : Wordware Publishing Inc., 2005. – ISBN 1598220020

[20] MCCUNE, W W. : OTTER (Organized Techniques for Theorem-proving and Effective Research) 2. 0 users guide. (1990), 3

[21] MICHEL, Benoît : *Le livre du 64*. 3. Banneux, Belgique : BCM, 1986. – ISBN 2–87111001–80

[22] R., Patterson D. A.and P. : Assessing RISC's in a High-Level Language Support. In : *IEEE Micro* 2 (1982), Nov.

[23] RICHER, Jean-Michel : *Une approche de résolution de problèmes en logique des prédicats fondée sur des techniques de satisfaction de contraintes*, Université de Bourgogne, Dijon, Diss., 1999. http://www.info.univ-angers.fr/~richer/pub/these.pdf

[24] ROBINSON, J. A. : A Machine-Oriented Logic Based on the Resolution Principle. In : *J. ACM* 12 (1965), Januar, Nr. 1, 23–41. http://dx.doi.org/10.1145/321250.321253. – DOI 10.1145/321250.321253. – ISSN 0004–5411

[25] SPÉRANZA, René : *Guide Silicium des micro-ordinateurs anciens*. COREP, 2006. – ISBN 9782951747241

[26] STALLINGS, W. : *Organisation et architecture de l'ordinateur*. Pearson Education, 2003 https://books.google.fr/books?id=mF3IPAAACAAJ. – ISBN 9782744070075

[27] STOKES, Jon : *Inside the Machine : An Illustrated Introduction to Microprocessors and Computer Architecture*. San Francisco, CA, USA : No Starch Press, 2006. – ISBN 1593271042

Apprendre à développer en assembleur x86

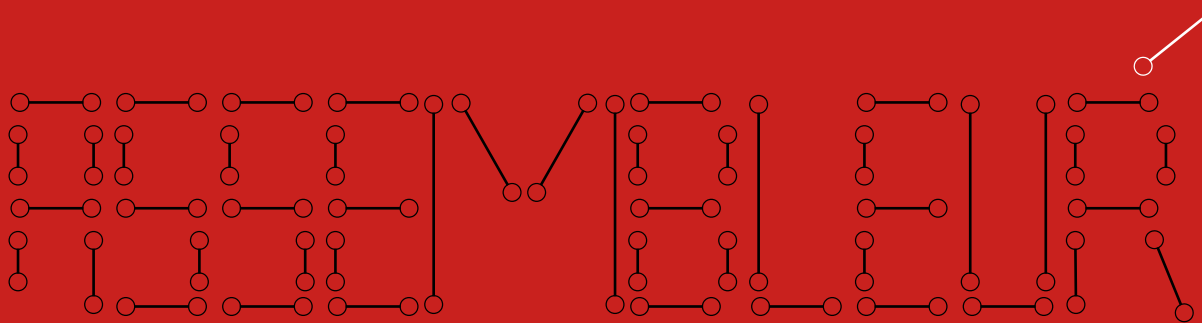
Grâce à cet ouvrage vous apprendrez les notions essentielles nécessaires pour programmer en assembleur x86. Les différents points abordés sont les suivants :

- représentation des entiers, des réels
- registres généraux 32 et 64 bits
- registres vectoriels et programmation vectorielle
- coprocesseur et calculs avec les réels
- appel de sous-programmes
- édition, compilation, édition de liens
- techniques de programmation : alignement mémoire, dépliage de boucle

La mise en application est réalisée au travers de plusieurs études de cas qui visent à améliorer le codage d'une fonction de base écrite en C.

A propos de l'auteur

L'auteur est maître de conférences en informatique à l'Université d'Angers. Il enseigne la programmation assembleur depuis L'année 2000 en Licence Informatique.



9 782957 316007