Table des matières

Table des figures

Chapitre 1

Introduction

1.1 Les microprocesseurs 64 bits

1.1.1 Définition

L'histoire des *microprocesseurs* est fortement liée à *Intel*, entreprise née en 1968 afin d'exploiter une technologie venant alors d'apparaître : les circuits intégrés. Elle débute comme *fondeur*, c'est-à-dire fabricant de circuits intégrés pas nécessairement conçus par elle, de la première application des circuits intégrés à connaître un succès certain : les mémoires à semi-conducteurs, appelées à remplacer les mémoires à tores de ferrite employées alors. Elle en devient très rapidement le numéro un. Le microprocesseur est inventé au sein de la société en 1971, dont elle demeurera le numéro un sur ce marché durant quarante ans.

Le premier microprocesseur, dénommé `4004`, est un microprocesseur dit 4 bits, dans la mesure où les entrées-sorties ainsi que les opérations (addition et multiplication) s'effectuent sur 4 bits en une seule instruction. Il a été conçu afin dêtre incorporé dans une nouvelle gamme de calculatrices de bureaux, pour lesquelles un tel microprocesseur est bien adapté : 4 bits permettent de manipuler les chiffres de '0' à '9'. Par contre, un tel microprocesseur ne peut pas servir de cœur d'un ordinateur, aussi petit soit-il. *Intel* sort ensuite le microprocesseur 8 bits *8008*, bientôt suivi des *8080* et *8085*. Les microprocesseurs 8 bits permettent l'émergence des micro-ordinateurs à usage domestique, mais *Intel* perd à ce moment-là la première place du marché au profit de concurrents. La société s'attelle alors au microprocesseur 16 bits *8086*, qui va connaître un énorme succès en permettant le déploiement des micro-ordinateurs à une très grande échelle dans les entreprises.

Sur sa lancée, *Intel* « innove » pour une question de prix de revient : elle crée le *8088*, permettant des entrées-sorties sur 8 bits mais des calculs sur 16 bits (le « cœur » du nouveau microprocesseur est le même que le *8086*). IBM choisit ce microprocesseur pour son PC, le micro-ordinateur adopté par les entreprises, destiné jusque-là surtout à un public d'« amateurs » ou pour des tâches spécialisées. Le nombre de bits n'est plus une caractéristique très précise de la puissance du microproceseur. *Intel* reprenant la première place des fondeurs, conçoit des microprocesseurs 32 bits (`i386`, `i486` puis *pentium*) qui lui permet de détenir la première place

sur le marché des micro-ordinateurs et crée le *mode protégé*, destiné à faciliter la conception des systèmes d'exploitation.

Il n'est plus alors facile de savoir ce qu'il faut entendre par *microprocesseur n bits* : le nombre de broches des données, le nombre de broches des adresses, la taille des registres, la taille des opérandes sur laquelle est effectuée une opération en peu de cycles machine ?

Définition.- *On parle de* **microprocesseur** n **bits** *lorsque ses registres généraux ont une taille de* n *bits.*

En 1982, *Intel* accorde à AMD (*Advanced Micro Devices*) une licence pour produire les microprocesseurs `8086` et `8088`, afin de renforcer la position de son architecture sur le marché. À la suite d'une bataille juridique, AMD obtient en 1995 le droit de produire des microprocesseurs fondés sur l'architecture IA-32, conçue par *Intel*. AMD conçoit en 2003 l'architecture 64 bits `AMD64`, totalement compatible avec l'IA-32, ce qui lui permet d'atteindre une part de près de 25 % sur le marché des microprocesseurs `x86`. Cette architecture est adoptée par *Intel* quelques années plus tard sous le nom de `EM64T` (*Extended Memory 64 bits Technology*).

Définition.- *On appelle* `x86-64`*, au lieu des dénominations* `AMD64` *ou* `EM64T` *propres aux fondeurs concernés, l'extension du jeu d'instructions* `x86` *pour une architecture 64 bits. Cette extension permet la gestion des entiers sur 64 bits, avec pour corollaire un adressage mémoire allant bien au-delà de la limite des 4 GiO du* `x86`*. À cela s'ajoute le doublement (de 8 à 16) du nombre de registres généralistes.*

1.1.2 Utilisation des microprocesseurs 64 bits

L'utilisation principale des microprocesseurs 64 bits est l'adressage de la mémoire : 32 bits ne peuvent normalement pas adresser plus de 4 Gio (2^{32} octets) de mémoire centrale, tandis que les processeurs 64 bits peuvent en adresser 16 Eio (2^{64} octets). C'est pourquoi dès qu'il y a plus de 4 Gio de RAM sur une machine, la mémoire située au-delà de ce seuil ne sera (directement) adressable qu'en mode 64 bits.

Une autre application serait d'effectuer les opérations sur 64 bits en peu de cycles machine de façon à améliorer la performance des calculs. Mais cela n'est pas le cas de l'architecture `x86-64`. *Intel* a essayé de le faire avec le microprocesseur *Itanium* (architecture **x64**), mais celle-ci n'a jamais connu de succès et a été abandonné en janvier 2019.

1.2 Aide matérielle au temps partagé

1.2.1 L'emploi des ordinateurs en temps partagé (1962)

Maurice WILKES, le concepteur du premier ordinateur opérationnel, écrit un petit livre [Wil-68] dans lequel il fait le point sur l'*emploi partagé des ordinateurs*, qui a commencé à être mis en place à partir de 1962 et est alors encore peu développé.

Il rappelle que, sur les premiers ordinateurs, l'utilisateur passait le temps qu'il voulait à tester son programme mais qu'« *on ne fut pas long à réaliser que c'était là une bien mauvaise façon d'employer une machine rare et coûteuse.* » On est donc passé au **traitement par lots** (*batch processing* en anglais) : « *Les travaux sont chargés par lots sur ruban magnétique, souvent à l'aide d'un petit ordinateur auxiliaire. Chaque lot de travaux est placé sur l'ordinateur principal ; les résultats sortent sur un autre ruban magnétique et sont ensuite imprimés.* [...] *Ces développements ont sans aucun doute amélioré le rendement des ordinateurs ; ils ont eu cependant l'effet regrettable d'éloigner l'utilisateur de l'ordinateur et de diminuer, en particulier dans le cas de programmes en développement, la rapidité du travail de mise au point.* » On est donc passé ensuite au **temps partagé** (*time-sharing* en anglais) : « *Il est maintenant possible aux utilisateurs d'être reliés par une paire de câbles à un ordinateur puissant qui peut être situé à proximité ou parfois à des kilomètres de distance. Tous les utilisateurs, quels qu'ils soient, ont accès instantanément à l'ordinateur et peuvent obtenir une réponse à leurs demandes sous la réserve que l'ordinateur doit partager son temps entre tous les utilisateurs. Le développement de tels systèmes est, cependant, encore en enfance.* »

1.2.2 Programmation des applications et programmation système

Maurice WILKES continue en distinguant ce que nous appelons maintenant **programmation des applications** et **programmation système** : « *En décrivant un système d'emploi partagé, une distinction doit être faite entre les programmes des utilisateurs et les programmes qui assurent diverses fonctions d'administration ou de commutation. Ces derniers sont plus ou moins interconnectés et désignés collectivement sous le nom de* superviseur. [...] *Au lieu de programme d'utilisateur, il est souvent préférable d'employer le terme programme-objet ; ceci inclut des programmes qui, bien que n'appartenant pas à un utilisateur donné, sont traités par le système exactement de la même manière que les programmes d'utilisateurs. L'une des plus importantes fonctions du superviseur est déviter que les programmes-objets interfèrent entre eux. Dans ce but, le superviseur doit avoir certains privilèges qui sont refusés aux programmes-objets. Parmi ceux-ci, on peut citer le contrôle des dispositifs d'entrée et de sortie, l'asservissement des interruptions, ainsi que l'établissement des limites de mémoire dans lesquelles les programmes-objets peuvent opérer. Beaucoup d'ordinateurs fonctionnent suivant deux modes opératoires, un mode ordinaire pour le déroulement du programme-objet, un mode privilégié réservé au superviseur ; quelques types d'instructions peuvent être exécutés seulement en mode privilégié.* »

Les concepts sont là, même si le vocabulaire a changé : on parle de *système d'exploitation*, *programme d'application* et *programme système* au lieu de *superviseur*, *programme-objet* et programme assurant diverses fonctions d'administration et de commutation. Un programme d'application est donc, par conséquent, un programme conçu pour un système d'exploitation donné, ce dernier étant en place. La programmation système concerne la programmation du système d'exploitation (y compris les pilotes de périphériques).

1.2.3 Aide matérielle au temps partagé

Au moment où Maurice WILKES écrit, en 1968, l'emploi partagé des ordinateurs repose entièrement sur du logiciel. Le but de son livre est d'inciter à répartir cette fonctionnalité entre logiciel et matériel.

Mais il va falloir un certain temps avant qu'il ne soit entendu.

Intel introduit le **mode protégé** dans ses microprocesseurs à partir du `80286` pour mettre en place la partie matérielle réclamée par Maurice WILKES. Nous aborderons donc la programmation des applications dans la première partie, réservant la programmation système à une seconde partie.

Comme le dit Maurice WILKES, la mise en place de l'emploi partagé des ordinateurs repose à la fois sur une partie matérielle (le mode protégé pour les microprocesseurs *Intel*) mais aussi sur une partie logicielle, une partie du système d'exploitation de nos jours. La programmation des applications ne peut donc pas faire complètement fi du système d'exploitation utilisé. Tous nos exemples concerneront *Linux*, pour une architecture x86-64.

1.3 Bibliographie

[Wil-68] WILKES, Maurice Vincent, **Time-sharing computer systems**, MacDonald and Co., London, 1968. Tr. fr. **Emploi partagé des ordinateurs**, Dunod, 1971, III + 124 p.

Première partie

Programmation d'applications

Chapitre 2

La programmation du x86-64 en langage d'assemblage

La façon la plus efficace, du point de vue du temps d'exécution des calculs en résultant, d'écrire un programme est de le faire en langage machine. Ceci présente cependant deux difficultés : comment écrire un tel programme et le lancer à partir d'un système d'exploitation ? De plus, cela devient très rapidement pénible car il faut « coder » chaque instruction élémentaire. La façon la plus conviviale, pour le programmeur, d'écrire un programme est de le faire en langage C puis d'utiliser un compilateur adéquat. Une façon intermédiaire entre langage machine et langage évolué est d'utiliser un *langage d'assemblage*.

John von Neumann affirmait que la seule façon de bien programmer se fait en langage machine. D'une part, tout le monde n'a pas les capacités intellectuelles de von Neumann, d'autre part lorsque les programmes deviennent imposants, il devient très pénible de « coder » à la main [1]. On écrit donc le programme en langage mnémonique et on désigne les emplacements mémoire par des étiquettes (*label* en anglais). On parle alors de **langage d'assemblage**. On se sert de l'ordinateur, c'est-à-dire d'un programme, pour traduire le programme en langage d'assemblage en un programme en langage machine. Un tel programme est appelé **assembleur**.

Pour pouvoir programmer en utilisant les instructions d'un microprocesseur, on doit en connaître l'architecture ou, plus exactement, un modèle de programmation de son architecture. C'est ce que nous allons étudier dans une première partie en ce qui concerne l'architecture x86-64. Puis nous verrons des outils permettant de programmer le x86-64 en langage d'assemblage.

1. Et même le grand von Neumann était capable de ne pas donner les bons programmes du premier coup, comme le montre l'analyse [**?**] de ses programmes publiés avant que l'ordinateur auquel ils étaient destinées ne soit disponible.

2.1 Le modèle utilisateur de programmation du x86-64

Du point de vue du programmeur, un microprocesseur est constitué de registres (internes) et on peut effectuer des opérations grâce à des instructions.

2.1.1 Les registres

2.1.1.1 Les registres généraux

Il y a 16 registres généraux (contre 8 pour le modèle de programmation précédent, appelé IA-32), chacun d'une capacité de 64 bits, appelés :

`rax`, `rbx`, `rcx`, `rdx`, `rsi`, `rdi`, `rbp`, `rsp`, puis `r8` à `r15`

par AMD, avec 'r' pour *Re-extended*.

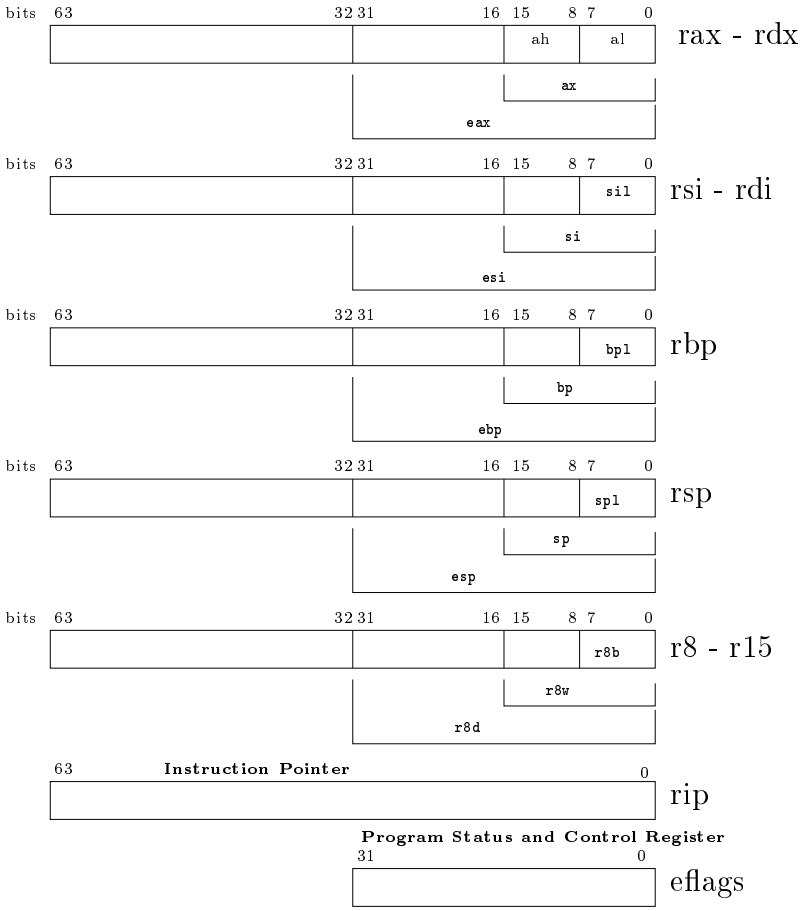

FIGURE 2.1 – Les registres généraux

Suivant en cela *Intel*, AMD permet, pour des raisons de compatibilité ascendante, d'accéder aussi à certaines parties de ces registres, de tailles 32, 16 et 8 bits, comme le montre la figure 2.1.

Remarque.- Les parties qui contiennent les bits `[15..8]`, permettant d'assurer la compatibilité avec une architecture 8 bits ne sont donc disponibles que pour `rax`, `rbx`, `rcx` et `rdx`.

2.1.1.2 Autres registres communs

Outre les registres généraux, le concepteur d'applications (que l'on oppose au concepteur du système d'exploitation) n'a à sa disposition que deux autres registres :

— le *registre d'instruction* `rip` (pour *Re-extended Instruction Pointer*), contenant l'adresse de l'instruction à exécuter, sur 64 bits,

— et le *registre des indicateurs* (*eflags* pour *Extended Flags*), contenantt un rapport sur la façon dont s'est déroulée la dernière instruction. Ce dernier registre, n'ayant pas évolué par rapport à IA-32, conserve une taille de 32 bits.

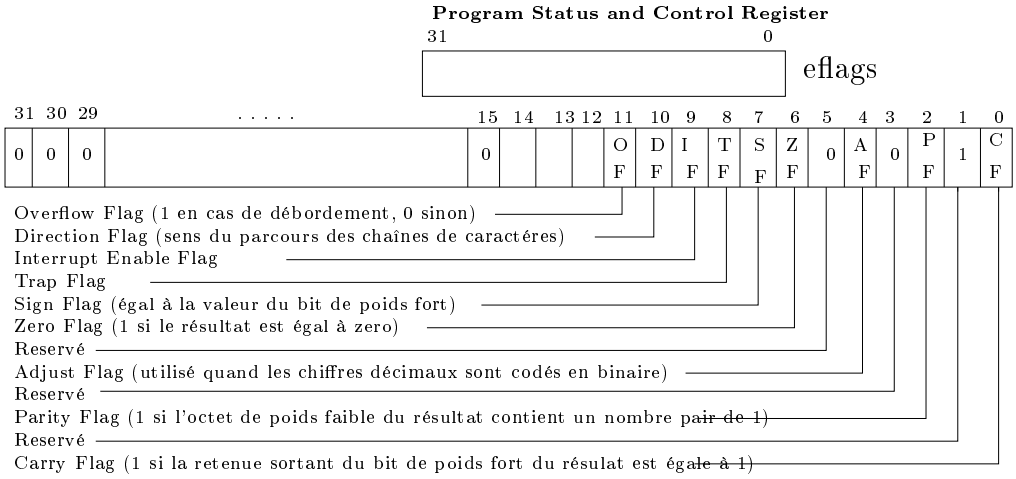

FIGURE 2.2 – Le registre des indicateurs

2.1.1.3 Registres spécifiques à un modèle de microprocesseur

Il existe également des registres spécifiques à chaque modèle de microprocesseur, appelés MSR pour *Model Specific Register*, dont nous parlerons à propos de TSC (*Time Stamp Counter*).

2.1.2 Accès à la mémoire

2.1.2.1 Principe de l'accès à la mémoire

Notion de mémoire.- La *mémoire* est un entrepôt dans lequel on stocke les données ainsi que les programmes (suites d'instructions), considérés eux aussi comme des données.

Chaque donnée stockée dans la mémoire est repérée par une **adresse**.

Du point de vue « conceptuel », la *mémoire* doit assurer deux fonctions principales :

— la *lecture* : étant donnée une adresse, fournir la donnée se trouvant à cette adresse ;

— l'*écriture* : étant données une adresse et une donnée, placer (enregistrer) la donnée dans la mémoire à cette adresse.

Une question se pose immédiatement : « la lecture (l'écriture) est-elle toujours possible (autorisée) ? » Nous nous occuperons de cette question un peu plus tard...

La mémoire dans l'architecture d'un ordinateur.- Du point de vue de l'architecture de l'ordinateur, la mémoire est un ensemble de circuits ayant deux « ports d'entrée » – un pour l'adresse et un pour la donnée à écrire, et un « port de sortie » – pour la donnée à lire, un « port » ét matérialisé par un ensemble de broches. Cet ensemble de circuits doit assurer les deux fonctions principales ci-dessus : la lecture et l'écriture, l'adresse ainsi que la donnée étant considérées comme des mots sur l'alphabet binaire dont les tailles sont fixées. On peut donc parler de la « taille des adresses » et de la « taille des données ».

La mémoire est vue comme une suite linéaire d'emplacements de taille fixe.

Après bien des vicissitudes sur cette taille au début de la conception des ordinateurs, la taille d'un tel emplacement s'est maintenant stabilisée à 8 bits (un *octet*[2]). La mémoire est donc une suite d'octets numérotés (par un entier naturel) à partir de 0. On parle de **modèle plat** ou **linéaire** de la mémoire. L'entier (représenté par un mot sur l'alphabet binaire) qu'on doit fournir aux broches du microprocesseur pour repérer un octet est appelé **adresse physique** de cet octet.

Un microprocesseur d'architecture `x86-64` utilise des adresses sur 64 bits et devrait donc être capable d'adresser 2^{64} octets, soit seize millions de téraoctets adressables. En fait, les microprocesseurs actuels ne disposent que de 36 ou 42 broches d'adresse (et non 64).

2. On parle de *byte* en anglais, bien qu'il ne s'agissait pas nécessairement de huit bits et on peut rencontrer des expressions comme « *7-bit byte* ». On est beaucoup plus clair dans le vocabulaire des réseaux, dans lequel on parle d'*octet* en anglais.

2.1.2.2 Hiérarchie de la mémoire

Suite aux technologies utilisées actuellement, et non pour des raisons intrinsèques, il existe plusieurs niveaux de mémoire, chaque niveau se distinguant des autres par les caractéristiques suivantes :

— le temps d'accès aux données (*latence*) ;
— la durée de l'opération (lecture ou écriture) ;
— la capacité de la mémoire ;
— la densité ;
— le prix.

Actuellement, les paramètres des principaux type de la mémoire des ordinateurs sont les suivants :

Type	Latence	Capacité	Prix
HDD	≈3 ms	1 - 4 TB	≈\$100/TB
DRAM	≈50 ns	1 - 4 GB	≈\$8/GB
SRAM	≈5 ns	1 - 4 KB	≈\$2/KB

Les échanges entre les niveaux différents de la mémoire s'effectuent par blocs d'octets. Les noms des blocs d'échange sont :

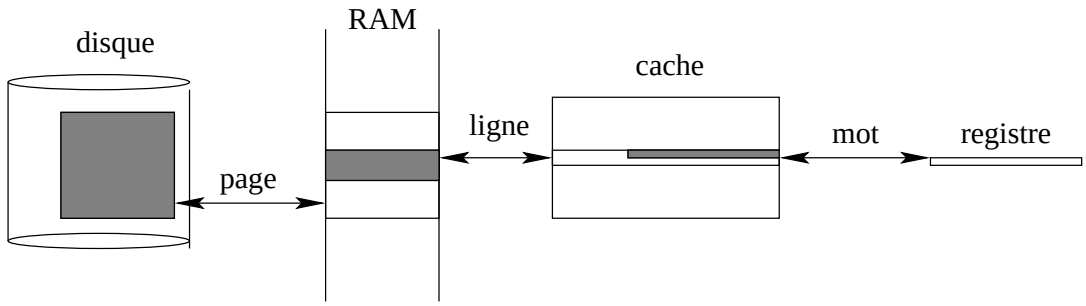

FIGURE 2.3 – Les unités d'échange

2.1.2.3 Adresse virtuelle

Notion et intérêt des adresses virtuelles.- Conformément à ce qu'a dit Maurice WILKES, comme nous l'avons vu, la philosophie du temps partagé a pour conséquence que le programmeur d'applications ne doit pas accéder directement aux adresses physiques, ni même qu'il sache où se situent physiquement les données. Sinon il pourrait, de bonne foi mais par maladresse, détruire des données utilisées par le système d'exploitation.

On lui fait donc utiliser ce qu'on appelle des **adresses virtuelles**, le système d'exploitation effectuant le lien entre adresse virtuelle et adresse physique, de façon transparente à l'utilisateur, c'est-à-dire que ce dernier n'a pas besoin de s'en préoccuper.

Les programmes d'application n'utilisent donc que des adresses virtuelles.

MMU.- Le dispositif, quel que soit sa nature, logicielle ou matérielle, permettant de traduire une adresse virtuelle en l'adresse physique correspondante est appelé **unité de gestion de la mémoire** (**MMU** pour *Memory Management Unit* en anglais).

De nos jours, la plupart de ses tâches sont dévolues à une partie du processeur, même si une partie de celles-ci reste encore à la charge du système d'exploitation (par exemple, et en des termes techniques, la récupération des pages sur le disque).

Taille des adresses physiques et virtuelles.- La taille d'une adresse physique n'est pas nécessairement la même que celle d'une adresse virtuelle. Voici, à titre d'exemple, les informations concernant le microprocesseur `Intel Core 2 Duo` (d'architecture **x86-64**), obtenues sous Linux :

```
$ cat /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
.........................
model name : Intel(R) Core(TM)2 CPU T7200  @ 2.00GHz
.........................
address sizes : 36 bits physical, 48 bits virtual
.........................
```

Historique.- Parmi les microprocesseurs INTEL, pour `i8086` et `i8088` l'utilisateur utilise les adresses physiques (il n'y a donc pas de MMU). La taille des adresses physiques est de 20 bits pour les deux. Syntaxiquement l'adresse est représentée par deux entiers de 16 bits : `0xYYYY:0xZZZZ`, l'adresse physique étant alors égale à `0x10*0xYYYY + 0xZZZZ`.

Le mécanisme des adresses virtuelles est implémentée pour la première fois par *Intel* sur le microprocesseur `i80286` en *mode protégé* mais l'utilisateur peut également utiliser les adresses physiques. La taille d'adresse de ce microprocesseur est de 24 bits.

Le microprocesseur `i80386` a des registres de 32 bits. La MMU permet à l'utilisateur de travailler avec des adresses virtuelles de 32 bits.

Adresse effective.- L'exemple du microprocesseur *Core 2 Duo* ci-dessus nous montre que, même lorsque la taille disponible des adresses est de 64 bits, on n'utilise pas, pour le moment, l'intégralité des 64 bits de l'adresse virtuelle : nous avons vu, par exemple, que des 64 bits possibles, seuls les 48 bits de poids faible sont pris en compte. Cette partie de 48 bits de poids faible de l'adresse virtuelle est appelée **adresse effective**.

Adresse canonique.- Par convention, la valeur de chacun des 16 bits de poids fort non significatifs doit être égale à la valeur du bit de poids fort (le 48-ième) de l'adresse effective. Les adresses sur 64 bits obtenues ainsi sont dites **canoniques**. Par exemple :

- les adresses `0xffffff00f500000`, `0x333333333000` et `0x7f4d85a89007` sont canoniques ;
- aucune des adresses `0xffff6fffff600000`, `0x333333333333333f` et `0x8f4d85a89007` n'est canonique.

2.1.3 Les constantes entières

2.1.3.1 Représentation des entiers

Supposons qu'on dispose de k bits pour représenter les entiers. Dans notre visualisation, les bits sont numérotés de droite à gauche à partir de `0` :

$$b_{k-1}b_{k-2}\dots b_1b_0$$

où chaque b_i est soit `0` soit `1`.

Entiers non signés (entiers naturels).- Le code ci-dessus représente le nombre :

$$a = \sum_{i=0}^{k-1} b_i * 2^i$$

Inversement, étant donné un entier naturel a, on l'écrit en base deux sur k chiffres (bits) pour obtenir sa représentation (ou bien un signal si c'est impossible à cause de manque de ressource). L'intervalle de nombres entiers non signés représentables sur k bits est ainsi $[0, 2^k - 1]$

Entiers signés (entiers relatifs).- Considérons toujours le code binaire $b_{k-1}b_{k-2}\dots b_1b_0$ sur k bits mais, cette fois ci, interprétons-le autrement. Le code peut aussi représenter l'entier relatif :

$$a = -b_{k-1} * 2^{k-1} + \sum_{i=0}^{k-2} b_i * 2^i$$

Exemples.- Prenons $k = 8$ et considérons un code binaire $m = b_7b_6\dots b_1b_0$.

1. Si $b_7 = 0$ alors m représente un nombre entier naturel de l'intervalle $[0, 127]$ (représentable sur 7 bits).
2. Le code `10000000` représente $-2^7 = -128$.
3. Le code `10000001` représente $-2^7 + 1 = -127$.
4. Le code `11111111` représente $-2^7 + 127 = -1$.

Extension signée.- Soit $m = b_{k-1}b_{k-2}\dots b_1b_0$ un code binaire sur k bits. L'**extention signée** de m sur $k + l$ $(l > 0)$ bits est le code :

$$\underbrace{b_{k-1}\dots b_{k-1}}_{l} b_{k-1}b_{k-2}\dots b_1b_0$$

Il est facile de montrer que le code binaire et son extention signée représentent le même entier (mais sur un nombre de bits différent).

Complément à deux.- Soit un code binaire $m = b_{k-1}b_{k-2}\dots b_1b_0$ sur k bits.

Le **complément à 2** de m sur k bits est le code $comp2_k(m) = c_{k-1}c_{k-2}\dots c_1c_0$ tel que :

$$\sum_{i=0}^{k-1} b_i * 2^i + \sum_{i=0}^{k-1} c_i * 2^i = 0 \pmod{2^k}$$

Les propriétés du complément à 2 qui nous intéressent sont :

1. $comp2_k(comp2_k(m)) = m$ pour chaque code m sur k bits.
2. $comp2_k(\overline{0\ldots0}) = \overline{0\ldots0}$.
3. $a - b = a + comp2_k(b)$ pour tous les codes a, b sur k bits (notez que les opérations arithmétiques sont effectuées sur *les codes*, pas sur *les nombres*.

Voyons comment calculer les chiffres du complément à 2 du m :

$$\begin{aligned}
\sum_{i=0}^{k-1} c_i * 2^i &= 2^k - \sum_{i=0}^{k-1} b_i * 2^i \pmod{2^k} \\
&= 2^k - 1 - \sum_{i=0}^{k-1} b_i * 2^i + 1 \pmod{2^k} \\
&= \sum_{i=0}^{k-1} 2^i - \sum_{i=0}^{k-1} b_i * 2^i + 1 \pmod{2^k} \\
&= \sum_{i=0}^{k-1} (1 - b_i) * 2^i + 1 \pmod{2^k}
\end{aligned}$$

Autrement dit, pour obtenir les chiffres de $comp2_k(m)$, il faut prendre le code m, inverser chaque chiffre (remplacer `0` par `1` et `1` par `0`), considérer le résultat comme un entier non-signé sur k bits, ajouter `1` à ce nombre et ne garder que les k bits de poids faible du résultat.

Exemples.- Prenons $k = 8$. Soit un code $m = b_7b_6\ldots b_1b_0$. Alors $\overline{b_7b_6\ldots b_1b_0}$ désigne l'entier non-signé représenté par ce code sur `8` bits.

1. $comp2_8(00000000) = \overline{11111111} + 1 = 00000000$ sur 8 bits.
2. $comp2_8(00000001) = \overline{11111110} + 1 = 11111111$ sur 8 bits.
3. $comp2_8(00000010) = \overline{11111101} + 1 = 11111110$ sur 8 bits.
4. $comp2_8(11111111) = \overline{00000000} + 1 = 00000001$ sur 8 bits.
5. $comp2_8(10000000) = \overline{01111111} + 1 = 10000000$ sur 8 bits.

Reprenons la représentation des entiers signés ci-dessus. Étant donné un code sur k bits, on sait calculer le nombre entier signé représenté par ce code. Comment résoudre le problème inverse : étant donné un nombre entier signé a, calculer le code qui le représente sur k bits ou bien conclure que le nombre en question n'est pas représentable sur k bits ?

Considérons les cas suivants.

1. $a \geq 0$ et a est représentable sur $k-1$ bits ($0 \leq a \leq 2^{k-1} - 1$) en tant qu'entier non-signé. Soit $a = \sum_{i=0}^{k-2} b_i * 2^i$. Alors il est évident que le code $0b_{k-2}\ldots b_1b_0$ représente a sur k bits.
2. $a < 0$ et $|a|$ est représentable sur $k-1$ bits ($-2^{k-1} + 1 \leq a < 0$) en tant qu'entier non-signé. Soit $|a| = \sum_{i=0}^{k-2} b_i * 2^i$. Considérons le code $m = 0b_{k-2}\ldots b_1b_0$ qui représente $|a|$. Son complément à 2 sur k bits est $comp2_k(m) = \overline{1(1-b_{k-2})\ldots(1-b_1)(1-b_0)} + 1$. Le nombre qu'il représente est :

$$-2^{k-1} + \sum_{i=0}^{k-2} (1 - b_i) * 2^i + 1 = -(2^{k-1} - 1) + \sum_{i=0}^{k-2} 2^i - \sum_{i=0}^{k-2} b_i * 2^i$$

$$= -(2^{k-1}-1) + (2^{k-1}-1) - \sum_{i=0}^{k-2} b_i * 2^i$$

$$= -\sum_{i=0}^{k-2} b_i * 2^i = -|a| = a$$

3. $a = -2^{k-1}$. Il est évident que le code $1\underbrace{0\ldots00}_{k-1}$ représente a sur k bits même si $|a|$ n'est pas représentable sur $k-1$ bits.

Résumé.- L'interprétation d'un code binaire sur k bits indiquée ci-dessus nous permet de représenter les entiers relatifs de l'intervalle $[-2^{k-1}, 2^{k-1}-1]$ de la façon suivante :

1. tout entier appartenantt à $[0, 2^{k-1}-1]$ est représenté sur k bits comme un entier non-signé (le bit de poids fort de cette représentation est toujours égal à `0`) ;
2. tout entier appartenant à $[-2^{k-1}+1, -1]$ est représenté sur k bits par le complément à 2 de sa valeur absolue (le bit de poids fort de cette représentation est toujours égal à `1`) ;
3. le nombre -2^{k-1} est représenté sur k bits par le code $1\underbrace{0\ldots0}_{k-1}$.

Cette représentation permet d'effectuer les additions et les soustractions d'une façon unifiée et systématique.

2.1.4 La notation AT&T

`Intel` définit un langage mnémonique dans la documentation de ses microprocesseurs [Int], notation qui est reprise et étendue pour les directives dans la plupart des assembleurs.

L'assembleur `gas`, dont l'équipe de développement faisait partie des laboratoires Bell (les laboratoires de recherche de la compagnie AT&T), peut utiliser la notation *Intel* mais utilise par défaut ce qui est appelé la **notation AT&T**.

2.1.4.1 Désignation des registres en notation AT&T

Les seize registres généraux de l'architecture x86-64 sont désignés par :

`%rax`, `%rbx`, `%rcx`, `%rdx`, `%rsi`, `%rdi`, `%rbp`, `%rsp`, puis `%r8` à `%r15`

en notation AT&T, c'est-à-dire en faisant précéder le nom choisi par *Intel* par '%'.

De même, les parties accessibles de ces registres sont désignées par :

`%eax`, `%ax`, `%al`, `%ah`, ..., `%r8d`, `%r8w`, `%r8b`, ...

2.1.4.2 Les constantes entières en notation AT&T

Représentation.- Un entier peut être représenté en numération décimale :

17

en utilisant les chiffres de '0' à '9' et ceci sans indication particulière, ou en numération hexadécimale :

0x11

en utilisant les chiffres de '0' à '9', puis 'a' (ou 'A') à 'f' (ou 'F') pour les chiffres dix à quinze. Dans ce cas l'entier est précédé du préfixe '0x' pour distinguer la représentation décimale de la représentation hexadécimale. On peut également utiliser la numération octale avec les chiffres '0' à '7' :

021

en faisant précéder le nombre du chiffre '0' (ce qui n'est pas permis pour la numération décimale sauf pour le nombre zéro, dont la représentation est heureusement la même en décimal et en octal).

Valeur.- La valeur `val(ent)` d'une constante entière `ent` est l'entier qu'elle représente.

2.1.4.3 Désignation des adresses

Syntaxe.- En notation AT&T, une adresse virtuelle est spécifiée par un entier naturel, un registre ou une expression un peu plus complexe, conformément à la syntaxe suivante :

Syntaxe	Exemples
<*entier non signé*>	`0x8061b4cf` `24`
(<*registre*>)	`(%rsp)` `(%rdx)`
displacement(base, index, scale)	`-4(%rsp,%rcx,2)`

où :

- le **déplacement** (*displacement* en anglais) est un entier relatif codé sur 32 bits (et non 64 bits comme on pourrait s'y attendre),
- la base *base* est l'un quelconque des registres généraux,
- l'index *index* est de même l'un quelconque des registres généraux hormis RSP,
- l'échelle *scale* est l'un des entiers 1, 2, 4 ou 8 avec 1 comme valeur par défaut si *scale* est omis.

Valeur de l'adresse.- Le contenu `Cont(reg)` d'un registre `reg` est l'entier qu'il contient. Dans le cas de la formule, on a :

$$adresse\ virtuelle = Cont(base) + scale \times Cont(index) + val(displacement)$$

2.1.5 Instructions du x86-64

2.1.5.1 Format d'une instruction

Cas du langage machine.- Une instruction en langage machine est constituée de 1 à 16 octets, répondant à un format sur lequel nous reviendrons lorsque nous verrons comment programmer en langage machine.

Abstraction en langage mnémonique.- Une instruction est représentée en langage mnémonique de l'une des trois façons suivantes :

```
op arg1,arg2
op arg
op
```

où `op` est un mnémonyme pour la nature de l'instruction (ou opération), `arg` l'argument lorsque celle-ci n'exige qu'un seul argument et `arg1` et `arg2` les arguments, séparés par une virgule, lorsqu'elle exige deux arguments.

Remarque.- Comme nous le verrons lorsque nous aborderons le code machine, un mnémonyme ne correspond pas à un seul code opération. Par exemple, le mnémonyme `add` correspond à plus de 2 000 variations de cette instruction, et donc codes opération, en langage machine.

Suffixe de taille en notation AT&T.- Pour certaines instructions, on peut choisir la taille des opérandes : octet, mot (de deux octets), mot double (de 32 bits) ou mot quadruple (de 64 bits). Le nom du mnémonyme `op` en notation AT&T est constitué de celui choisi par *Intel* (par exemple `add`, `sub`, `or`,...), suivi d'un suffixe précisant le nombre d'octets affectés par l'instruction :

`b` pour un octet (« *byte* »)
`w` pour un mot de deux octets (« *word* »)
`l` pour quatre octets (« *long* »)
`q` pour huit octets (« *quadro* »)

ce qui donne, par exemple, `addq` ou `orl`.

Remarque.- Les mnémonymes des instructions concernant la pile (à savoir `pop, push, call, ret, leave` et `enter`), étant toujours effectuées sur 64 bits, n'ont pas de suffixes. Les deux premières opérations incrémentent (décrémentent) le contenu du registre `%rsp` (*stack pointer*) de 8.

Nature des arguments.- Nous avons vu ci-dessus qu'une instruction peut avoir zéro, un ou deux arguments. Un argument peut être une constante entière, le contenu d'un registre ou une adresse. La syntaxe de la notation AT&T est la suivante :

Type	Syntaxe	Exemples
constante	$<*entier signé*>	`$0`, `$-8`, `$0x030a1b62`
registre	%<*nom du registre*>	`%rax`, `%rbp`, `%bx`, `%cl`
adresse	<*entier non signé*>	`0x8061b4cf`, `0x0`
	(<*registre*>)	`(%rsp)`, `(%rdx)`
	displacement(base,index,scale)	`-4(%rsp, %rcx, 2)`

reprenant la nature des arguments en code machine.

On y ajoute, à titre de directive, le cas d'une étiquette :

Type	Syntaxe	Exemples
étiquette (*label*)	comme en langage C	`_start, main, L5`

Valeur d'un argument.- Précisons ce qu'on appelle « la valeur » `Val()` d'un argument. La valeur d'une constante est la valeur de l'entier qu'on obtient en omettant le $ devant sa description. Le contenu d'une adresse est le mot binaire qu'on trouve dans la mémoire à cette adresse et qui s'étend sur 1, 2, 4 ou 8 octets selon le contexte.

Exemples.- Commentons quelques instructions :

- `xorq %rdx, %rdx` : les 64 bits du registre désigné sont affectés, comme le précisent, de façon redondante, le suffixe du mnémonyme mais aussi le nom du registre ;
- `xorl %edx, %edx` : les bits `[31..0]` sont affectés, pour les mêmes raisons ;
- `xorw $0xffff, %ax` : les bits `[15..0]` sont affectés, comme le précisent, de façon redondante, le suffixe du mnémonyme et le nom du seul registre ;
- `xorb -2(%rbp), %bh` : les bits `[15..8]` sont affectés, comme le précisent le suffixe du mnémonyme et le registre du deuxième argument ;
- `imulq %ebx` : la syntaxe n'est pas correcte puisqu'il y a incohérence entre le suffixe du mnémonyme, qui fait référence à un opérande de 64 bits, et l'opérande, qui est un registre de 32 bits ;
- `leave` ou `rdtsc` : instructions sans opérande ;
- `movw $64,1024` : aucun des opérandes, que ce soit la valeur de la constante ou l'adresse, ne permettent de préciser le nombre de bits affectés ; ici seul le suffixe spécifie qu'il s'agit de 16 bits.

Dissymétrie des arguments.- Dans le cas d'une instruction à deux arguments, les rôles de chacun des arguments ne sont pas symétrique. L'un des arguments est appelé *source* (`src`) et l'autre *destination* (`dst`).

Le type syntaxique de la source peut être une *constante*, un *registre* ou une *adresse*, le type syntaxique de la destination peut être un *registre* ou une *adresse*, avec la contrainte que `src` et `dst` ne peuvent pas simultanément avoir le type *adresse*.

La notation *Intel* est :

```
op dst,src
```

alors qu'en notation ATT on a :

```
op src,dst
```

2.1.5.2 Instructions de manipulatin des données

Donnons maintenant une liste (non exhaustive!) des instructions. Les noms des opérations ci-dessous ont tous le suffixe `q`. On peut évidemment changer ce suffixe en `l`, `w` ou `b` en fonction de la taille des opérandes.

Copie

Syntaxe	Sémantique	Exemple
`movq src,dst`	`Cont(dst) = Cont(src)`	`movq 8(%rbp),%rax`

Opérations arithmétiques

Les instructions arithmétiques sont l'incrémentation (ajout de un), la décrémentation (diminution de un), l'addition, la soustraction, la multiplication (en distinguant `mul`, multiplication sur les entiers naturels, de `imul`, multiplication sur les entiers relatifs), la division (en distinguant également `div`, division entre entiers naturels, et `idiv`, entre entiers relatifs) et le passage à l'opposé :

Syntaxe	Sémantique	Exemples
`incq dst`	`Cont(dst) = Cont(dst) + 1`	`incq (%rdx)`
`decq dst`	`Cont(dst) = Cont(dst) - 1`	`decq %rdx`
`addq src,dst`	`Cont(dst) = Cont(dst) + Cont(src)`	`addq (%rsp),%rax`
`subq src,dst`	`Cont(dst) = Cont(dst) - Cont(src)`	`subq $8,%rbx`
`mulq src`	`[Cont(%rdx)Cont(%rax)] = Cont(src)*Cont(%rax)`	`mulq (%rsp)`
`imulq src,dst`	`Cont(dst) = Cont(dst)*Cont(src)`	`imulq (%rsp),%rax`
`imulq src`	`[Cont(%rdx)Cont(%rax)] = Cont(src)*Cont(%rax)`	`imulq (%rsp)`
`divq src`	`Cont(%rax) = [Cont(%rdx)Cont(%rax)]/Cont(src)` `Cont(%rdx) = [Cont(%rdx)Cont(%rax)] % Cont(src)`	`divq (%rsp)`
`idivq src`	`Cont(%rax) = [Cont(%rdx)Cont(%rax)]/Cont(src)` `Cont(%rdx) = [Cont(%rdx)Cont(%rax)] % Cont(src)`	`idivq (%rsp)`
`negq dst`	`Cont(dst) = -Cont(dst)`	`negq 0xa(%rbx,%rsi,4)`

Commentaires.- 1^o) Comme nous l'avons déjà dit, les arguments des instructions `mulq` et `divq` sont des entiers **non signés**, autrement dit des entiers naturels, tandis que les arguments des instructions `imulq` et `idivq` sont des entiers **signés**, autrement dit des entiers relatifs.

2^o) Notons qu'il existe deux versions de l'instruction `imulq` : avec deux arguments (à utiliser si le produit ne dépasse pas 64 bits) et avec un seul argument (l'autre étant le contenu du registre `%rax`).

3^o) Notons aussi l'emplacement du dividende pour les instructions `divq` et `idivq`, qui peut avoir une taille de 128 bits. Par conséquent, on ne doit surtout pas oublier de placer `0` dans `%rdx` avant d'effectuer l'opération si le dividende n'occupe que 64 bits.

Opérations logiques

Les opérations logiques sont la conjonction, la disjonction, la disjonction exclusive et la négation :

Syntaxe	Sémantique	Exemples
`andq src,dst`	`Cont(dst) = Cont(dst) & Cont(src)`	`andq %rbx,%r11`
`orq src,dst`	`Cont(dst) = Cont(dst) | Cont(src)`	`orq %r8,-4(%rax)`
`xorq src,dst`	`Cont(dst) = Cont(dst) ^ Cont(src)`	`xorq %rdx,%rdx`
`notq dst`	`Cont(dst) = ~Cont(dst)`	`notq %r9`

Décalages

Syntaxe	Sémantique	Exemples
`shlq` *constante*`,dst`	`Cont(dst) = Cont(dst) < <` *constante*	`shlq $4,%rax`
`shrq` *constante*`,dst`	`Cont(dst) = Cont(dst) > >` *constante*	`shrq $4,%rax`
`sarq` *constante*`,dst`	`Cont(dst) = Cont(dst) > >` *constante*	`sarq $4,%rax`

Commentaire.- Il existe deux instructions de décalage vers la droite : `shr` (*SHift Right*) et `sar` (*Shift Arithmetical Right*), est la suivante : `shr` remplit par des `0` les places libres qui apparaissent tandis que `sar` remplit ces places par le bit de poids fort de son argument `dst` (et donc conserve le signe).

Si, par exemple, `Cont(%rax)=0x80000000000000ab` alors le registre `%rax` va contenir :

`0x080000000000000a` après avoir effectué `shrq 4,%rax` et

`0xf80000000000000a` après avoir effectué `sarq 4,%rax`.

Instructions sur la pile

Syntaxe	Sémantique	Exemples
`push src`	`Cont(%rsp) = Cont(%rsp)-8` `Cont((%rsp)) = Cont(src)`	`push %rbp`
`pop dst`	`Cont(dst) = Cont((%rsp))` `Cont(%rsp) = Cont(%rsp)+8`	`pop %rbx`

Chargement de l'adresse effective

Syntaxe	Sémantique	Exemples
`leaq` *étiquette*`, dst`	`Cont(dst) = Val(`*étiquette*`)`	`leaq Loop,%rcx`
`leaq` *adresse*`, dst`	`Cont(dst) = Val(`*adresse*`)`	`leaq 0xc(%rbp),%rcx`

Commentaire.- L'instruction `lea` (*Load Effective Address*) peut servir pour calculer des expressions arithmétiques simples : par exemple `leaq 3(%rax,%rax,4), %rcx` calcule `5*Cont(%rax)+3` et place cette valeur dans `%rcx`.

2.1.5.3 Instructions de contrôle

Saut inconditionnel

Le saut inconditionnel s'effectue par l'instruction :

`jmp` *étiquette*

Sauts conditionnels

L'exécution d'une instruction sur les données (sauf `mov`, `push`, `pop` et `lea`) change certains bits (*flags*) du registre de statut `%eflags`. Les valeurs de ces indicateurs peuvent ensuite être prises en compte pour changer l'ordre d'exécution des instructions et implémenter ainsi les sauts conditionnels.

Deux instructions particulières positionnent les indicateurs *sans modifier les autres registres ni le contenu de la mémoire* :

Syntaxe	Sémantique
`cmpq src,dst`	même changement que `subq src,dst`
`testq src,dst`	même changement que `andq src,dst`

Supposons que l'instruction `cmpq src,dst` soit exécutée. Les instructions de saut (*jump*) conditionnel transfèrent le contrôle à l'adresse qui est la valeur de l'étiquette, selon les conditions suivantes :

Syntaxe	Condition de saut
`je` *étiquette*	`ZF == 1 (Cont(src) == Cont(dst))`
`jne` *étiquette*	`ZF == 0 (Cont(src) != Cont(dst))`
`js` *étiquette*	`SF == 1 (Cont(dst) - Cont(src)` est négatif)
`jns` *étiquette*	`SF == 0 (Cont(dst) - Cont(src)` n'est pas négatif)
`jg` *étiquette*	`Cont(dst) > Cont(src)` pour les entiers relatifs
`jge` *étiquette*	`Cont(dst) >= Cont(src)` pour les entiers relatifs
`jl` *étiquette*	`Cont(dst) < Cont(src)` pour les entiers relatifs
`jle` *étiquette*	`Cont(dst) <= Cont(src)` pour les entiers relatifs
`ja` *étiquette*	`Cont(dst) > Cont(src)` pour les entiers naturels
`jae` *étiquette*	`Cont(dst) >= Cont(src)` pour les entiers naturels
`jb` *étiquette*	`Cont(dst) < Cont(src)` pour les entiers naturels
`jbe` *étiquette*	`Cont(dst) <= Cont(src)` pour les entiers naturels

Appel de sous-programme

Syntaxe	Sémantique non formelle
`call` *étiquette*	Met l'adresse de retour dans la pile et transmet le contrôle à l'*étiquette*
`leave`	Prépare la pile pour le retour
`ret`	Transmet le contrôle à l'adresse de retour

2.2 Les outils d'assemblage pour le x86-64

Illustrons nos propos par l'assembleur `gas` (pour *Gnu ASsembler*) de GNU pour un système d'exploitation GNU/Linux 64 bits pour l'architecture x86-64.

2.2.1 L'assembleur `gas`

Nous avons vu, dans la première section, le modèle du microprocesseur utile pour la programmation. Nous avons également vu comment noter les registres et les instructions de celui-ci dans la notation AT&T, utilisée par défaut par `gas`. Voyons maintenant comment écrire un programme en langage d'assemblage `gas`, ainsi que les premières **directives** de celui-ci, c'est-à-dire les commandes n'ayant pas d'équivalent en langage machine.

Faire exécuter un programme écrit en langage d'assemblage comprend quatre phases : l'écriture du programme source, l'assemblage, le liage et l'exécution proprement dite.

Première étape : écriture du programme source.- Un programme en langage d'assemblage est un texte, s'écrivant donc avec un éditeur de texte.

Instructions et directives

— Ce texte comprend des *instructions*, reflétant les instructions du microprocesseur écrites sous forme mnémonique, et des **directives**, n'ayant pas leur équivalent pour le microprocesseur.

Sections

— Un programme en langage d'assemblage `gas` est une suite de définitions de *sections*, en particulier de sections `.text`, contenant le code, et `.data`, contenant les données : cette différence entre la nature des sections est liée à l'idée, popularisée par VON NEUMANN en 1945, de placer non seulement les données en mémoire vive mais également le programme (qui était câblé jusque-là). Seule une section `.text` est obligatoire, puisqu'on n'a pas besoin de données pour un programme simple.

Un programme simple commencera donc par la directive :

```
.section .text
```

Étiquettes

— Le programme, se trouvant dans une section `.text`, est une suite d'instructions du microprocesseur (sous forme mnémonique, comme nous l'avons déjà dit, mais reflétant fidèlement celles-ci), éventuellement étiquetées.

Une **étiquette** est un identificateur du langage d'assemblage choisi, en général un mot qui n'est pas un mot réservé, c'est-à-dire ne représentant pas une instruction, une directive ou un opérande.

Point d'entrée dans un programme

— Pour un programme simple, il semble naturel de commencer l'exécution du programme par la première instruction que l'on veut voir exécuter. Cependant, ceci n'est pas toujours le cas, parce qu'on veut quelquefois placer avant le programme « principal » soit des sous-programmes, soit des données. L'assembleur `gas` lancera donc l'exécution du programme à partir de l'instruction étiquetée par `_start`. Un programme comprendra donc une seconde directive :

```
        .section .text
        .globl   _start
        [...]
_start:
```

spécifiant que l'étiquette `_start` est bien celle connue par l'assembleur de façon globale et non une nouvelle étiquette « locale ». Cette étiquette doit se trouver quelque part dans le programme.

La directive `.global`, ayant la même signification que `.globl`, existe également.

Exemple

— Écrivons un premier programme initialisant le registre `ah` à 3 (on n'a pas besoin d'un microprocesseur 64 bits pour cela, mais ce n'est pas le plus important pour l'instant) :

```
        .section .text
        .globl   _start
_start:
        movb     $3,%ah
```

Retour du programme

— C'est déjà assez compliqué pour un programme ne comportant qu'une seule instruction utile. Mais ce n'est pas terminé. Imaginons que nous lancions ce programme. L'instruction voulue est certes exécutée mais que se passe-t-il après ? L'instruction suivante est alors exécutée. Mais quelle est l'instruction suivante ? Celle qui se trouve en mémoire après celle qui nous intéresse, à un endroit de la mémoire vive que l'assembleur (dans un sens large) a choisi. Mais qu'est-ce qui se trouve à cet emplacement mémoire ? On n'en sait rien. Il faut donc « terminer proprement » le programme. En quoi cela consiste-t-il ? Cela dépend du système d'exploitation, dont un des rôles, rappelons-le, est de lancer le programme dont le nom est écrit sur la ligne de commande, puis d'attendre une nouvelle commande une fois celle-ci exécutée.

Comment dire au système d'exploitation qu'on veut lui rendre le contrôle ? Tous les systèmes d'exploitation prévoient une interruption logicielle pour cela, l'interruption `0x20` puis `0x21` pour MS-DOS. Sous *Linux*, il n'existe qu'une seule interruption (logicielle), celle de numéro `0x80` ; un numéro de « fonction » (placé dans le registre `eax`) permet de préciser la fonctionnalitée désirée de cette interruption lorsqu'elle est appelée ; la fonction 1 correspond au `exit()` du langage C, la valeur de retour étant placée dans le registre `ebx`.

Le programme complet s'écrira donc :

```
        .section .text
        .globl   _start
_start:
        movb     $3,%ah
        movl     $1,%eax      # function number: exit
        movl     $0,%ebx      # status: no error
        int      $0x80        # Linux interrupt
```

L'avant-dernière ligne permet de transmettre le code de statut du `exit()` du système d'exploitation *Linux*, ici 0 (par convention) puisque tout se termine bien.

Commentaires

— Remarquons au passage comment placer un commentaire de fin de ligne en `gas` : il commence par '#'.

Instruction `syscall`

— Le programme tel que nous venons de l'écrire est compatible avec une architecture IA-32. Pour x86-64, on peut remplacer la dernière ligne en utilisant la nouvelle instruction `syscall` (pour *fast SYStem CALL*), à condition que cette option ait été spécifiée (nous verrons comment lors de la programmation système ; en attendant, tester ce qu'il en est sur votre système !) :

```
        .section .text
        .globl   _start
_start:
        movb     $3,%ah
        movl     $1,%eax     # exit
        movl     $0,%ebx     # status
        syscall              # Linux interrupt
```

Sauvegarde du programme

— Puisque le programme source est écrit avec un éditeur de texte, on sait comment le sauvegarder dans un fichier (texte). On lui donne un nom, par exemple '`mov.s`'.

Remarquons le suffixe traditionnel pour un programme écrit en langage d'assemblage : `.s` pour *aSsembler*. Le lecteur aurait pu légitimemnt penser au suffixe `.a`, mais celui-ci est déjà utilisé par ailleurs pour *Archive*.

Deuxième étape : assemblage.- Il faut traduire ce programme en langage machine. Ceci s'effectue grâce à l'assembleur :

```
$ as mov.s -o ./mov.o
```

— L'appel de `gas` se fait grâce à la commande traditionnelle UNIX pour l'assembleur, c'est-à-dire par `as` (pour *ASsembler*).

— Le paramètre figurant sur la ligne de commande est évidemment le nom du fichier source.

— On utilise l'option `-o` (pour *Output*) pour spécifier le nom du fichier traduit.

— Si on regarde la taille du fichier obtenu, on s'aperçoit qu'il ne s'agit certainement pas simplement du programme en langage machine correspondant. Il s'agit d'un **fichier objet**, c'est-à-dire du programme en langage machine précédé d'un *préfixe*, propre au système d'exploitation, contenant des paramètres sur la façon de le placer au moment de l'exécution.

— Remarquons le suffixe traditionnel pour un programme objet : `.o` (pour *Object*).

Troisième étape : liage.- Le programme, ce qui n'est pas le cas ici, pourrait contenir des appels à des fonctions d'une bibliothèque. Il est, dans ce cas, nécessaire d'effectuer le lien avec celles-ci. Ceci est l'objet du **liage** (*linking* en anglais). Cette étape est indispensable, même pour notre petit programme : il faut faire le lien avec la variable globale `_start`. On doit donc transformer le programme objet en un programme exécutable :

```
$ ld mov.o
```

Le nom de l'exécutable obtenu est, par défaut, `a.out` (pour *Assembler OUTput*), comme nous le savons déjà par l'utilisation de `gcc`.

En fait il vaut mieux :

```
$ ld -O0 mov.o
```

avec l'option `-O` (pour *Optimisation*) à zéro pour éviter que l'assembleur essaie d'optimiser notre code, ce que nous ne voulons pas qu'il fasse pour l'instant.

Quatrième étape : exécution.- Si on fait exécuter le programme :

```
$ ./a.out
$
```

on n'obtient rien de spécial puisque ce programme n'a pas d'action visible à l'extérieur. On revient à la ligne de commande, ce qui n'est déjà pas si mal.

2.2.2 Le débogueur gdb

Comment voir ce que fait un programme ?

Pour cela, on peut utiliser un **débogueur**, c'est-à-dire un logiciel (plus précisément un *utilitaire*) permettant d'analyser le comportement d'un programme.

Prenons l'exemple du débogueur `gdb` (pour *Gnu DeBugger*), dont une documentation est [gdb-doc].

Lancement de `gdb`.- Contrairement au débogueur `debug` de MS-DOS, par exemple, il faut indiquer dès l'assemblage qu'on voudra utiliser `gdb` :

```
$ as mov.s -gstabs -o mov.o
```

en utilisant l'option `-gstabs`.

Après liage, on lance ensuite `gdb` de la façon suivante :

```
$ gdb ./a.out
GNU gdb (Ubuntu/Linaro 7.3-0ubuntu2) 7.3-2011.08
Copyright (C) 2011 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".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/patrick/a.out...done.
(gdb)
```

L'invite de commande `(gdb)` indiquant que `gdb` attend une commande de notre part.

Quitter le débogueur.- On quitte `gdb` par la commande `quit` (ou la commande abrégée `q`).

Listage du programme.- La commande `l(ist)` de `gdb` affiche le source du programme en numérotant les lignes :

```
(gdb) l
1               .section .text
2               .globl _start
3       _start:
4               movb $3,%ah
5               movl $1,%eax    # function number: exit
6               movl $0,%ebx    # status: no error
7               int  $0x80      # Linux interrupt
8
9
(gdb)
```

Documentation en ligne des commandes.- Pour obtenir des renseignements sur une commande, on utilise la commande `h(elp)` plus le nom de la commande comme second argument. Par exemple :

```
(gdb) h l
List specified function or line.
With no argument, lists ten more lines after or around previous listing.
"list -" lists the ten lines before a previous ten-line listing.
One argument specifies a line, and ten lines are listed around that line.
Two arguments with comma between specify starting and ending lines to list.
Lines can be specified in these ways:
  LINENUM, to list around that line in current file,
  FILE:LINENUM, to list around that line in that file,
  FUNCTION, to list around beginning of that function,
  FILE:FUNCTION, to distinguish among like-named static functions.
  *ADDRESS, to list around the line containing that address.
With two args if one is empty it stands for ten lines away from the other arg.
(gdb)
```

Placement d'un point d'arrêt.- L'intérêt d'un débogueur est de pouvoir faire exécuter le programme pas à pas. Il faut d'abord placer un point d'arrêt (*breakpoint*) quelque part, disons à la ligne 7 du code source :

```
(gdb) b 8
Aucune ligne 8 dans le fichier "mov.s".
(gdb) b 7
Breakpoint 1 at 0x400084: file mov.s, line 7.
(gdb)
```

On peut alors lancer le programme, qui s'exécutera jusqu'au point d'arrêt :

```
(gdb) run
Starting program: /home/patrick/a.out

Breakpoint 1, _start () at mov.s:7
7               int  $0x80   #
(gdb)
```

Attention ! Le programme s'arrête sans exécuter le code de la ligne où se trouve le point d'arrêt.

Informations sur les registres.- Dans cet état on peut, par exemple, afficher les informations concernant les registres :

```
(gdb) info registers
rax            0x1      1
rbx            0x0      0
rcx            0x0      0
rdx            0x0      0
rsi            0x0      0
rdi            0x0      0
rbp            0x0      0x0
rsp            0x7fffffffe070   0x7fffffffe070
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x200    512
r12            0x0      0
r13            0x0      0
r14            0x0      0
r15            0x0      0
rip            0x400084 0x400084 <_start+12>
eflags         0x202    [ IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb)
```

Que peut-on dire des valeurs affichées ? Presque tous les registres ont la valeur 0, ce qui est normal, vu ce que l'on a fait. Le registre `rax` a la valeur 1 puisqu'on l'impose dans le programme.

Les registres de segment (`cs`, `ss`, `ds`, `es`, `fs` et `gs`) sont également affichés. Il s'agit essentiellement d'une queue de comète de `gdb` écrit pour IA-32 puisque ces registres jouent peu de rôle en mode 64 bits ; de plus les registres `ds` à `gs` prennent systématiquement la valeur 0.

Exécution du programme pas à pas.- Quittons `gdb`, lançons-le à nouveau mais, cette fois-ci, en plaçant un point d'arrêt à la seconde ligne, en exécutant le programme (jusqu'à la première ligne) puis pas à pas grâce à la commande `s(tart)` :

```
(gdb) b 2
Breakpoint 1 at 0x400078: file mov.s, line 2.
(gdb) s
The program is not being run.
(gdb) run
Starting program: /home/patrick/a.out

Breakpoint 1, _start () at mov.s:4
4               movb $3,%ah
(gdb) s
5               movl $1,%eax    # exit
(gdb) s
6               movl $0,%ebx    # status
(gdb) s
7               int  $0x80      # Linux interrupt
(gdb) s
[Inferior 1 (process 2718) exited normally]
(gdb) s
The program is not being run.
(gdb)
```

Remarque.- On peut également utiliser une interface graphique pour `gdb`. Une interface graphique pratique est `ddd` (*Data Display Debugger*). Le programme à étudier doit être compilé/assemblé avec les options pour `gdb`. Essayez :

```
ddd -v
```

Si ce logiciel est installé, on obtient quelque chose de la forme :

```
GNU DDD 3.3.12 (x86_64-unknown-linux-gnu)
Copyright (C) 1995-1999 Technische Universität Braunschweig, Germany.
Copyright (C) 1999-2001 Universität Passau, Germany.
Copyright (C) 2001 Universität des Saarlandes, Germany.
Copyright (C) 2001-2009 Free Software Foundation, Inc.
```

2.2.3 Le désassembleur `objdump`

2.2.3.1 Mise en place

L'utilitaire `objdump` (qui fait partie du paquetage `binutils` de GNU) permet d'obtenir un certain nombre d'informations sur les fichiers objet. Il permet, entre autres, de désassembler un exécutable.

Exemple.- Considérons notre tout premier exemple et lançons `objdump` avec l'option `-d` (pour *Disassemble*) :

```
$ objdump -d ./a.out

./a.out:     file format elf64-x86-64


Disassembly of section .text:

0000000000400078 <_start>:
  400078:       b4 03                   mov    $0x3,%ah
  40007a:       b8 01 00 00 00          mov    $0x1,%eax
  40007f:       bb 00 00 00 00          mov    $0x0,%ebx
  400084:       cd 80                   int    $0x80
$
```

- Nous avons utilisé l'option `-d` de désassemblage appliquée au fichier exécutable.
- La première ligne nous rappelle le nom du fichier ainsi que son format : ELF (pour *Executable and Linkage Format*) pour l'architecture `x86-64`.
- Puis la section `.text` est désassemblée, en commençant par donner l'adresse de début sur 64 bits (en hexadécimal), puis le programme, instruction par instruction, avec trois champs par instruction :
  - l'adresse de l'instruction (les 24 derniers bits seulement, en hexadécimal) ;
  - l'instruction en langage machine ;
  - l'instruction en langage d'assemblage AT&T.

2.2.3.2 Incidence du liage

Voyons l'incidence de l'étape du liage.
Reprenons, pour cela, le programme source écrit ci-dessus :

```
        .section  .text
        .globl  _start
_start:
        movb $3,%ah
        movl $1,%eax
        movl $0,%ebx
        int  $0x80
```

Assemblons-le puis regardons le code objet :

```
$ as mov.s -o ./mov.o
$ objdump -d mov.o

mov.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_start>:
   0:   b4 03                   mov    $0x3,%ah
   2:   b8 01 00 00 00          mov    $0x1,%eax
   7:   bb 00 00 00 00          mov    $0x0,%ebx
   c:   cd 80                   int    $0x80
$
```

Effectuons maintenant l'opération de liage, ce qui donne le fichier exécutable `a.out`, et explorons celui-ci :

```
$ ld mov.o
$ objdump -d ./a.out

./a.out:     file format elf64-x86-64


Disassembly of section .text:

0000000000400078 <_start>:
  400078:       b4 03                   mov    $0x3,%ah
  40007a:       b8 01 00 00 00          mov    $0x1,%eax
  40007f:       bb 00 00 00 00          mov    $0x0,%ebx
  400084:       cd 80                   int    $0x80
$
```

On voit que ce qui a changé est que l'étiquette `_start` prend maintenant une valeur non nulle et que les instructions ont une adresse en conséquence.

2.2.4 Application des outils

Si on ne veut utiliser que les instructions du microprocesseur, il est difficile d'en vérifier le comportement : comment, en effet, visualiser le résultat, disons d'un calcul ? Ceci est possible, en théorie, avec les seules instructions du microprocesseur : il suffit de programmer les entrées-sorties. Mais il s'agit d'une tâche longue et délicate. Le programmeur « normal » compte sur le système d'exploitation pour l'aider à contourner cette étape. Nous verrons comment cela se fait en utilisant les appels système du système d'exploitation. En attendant, les outils présentés dans ce chapitre vont nous permettre de visualiser ce que l'on peut faire avec les instructions du microprocesseur.

2.2.4.1 Premier exemple : addition sur 64 bits

Écrivons un programme `add64.s` qui place un entier codé sur 64 bits dans chacun des registres `RAX` et `RBX`, en effectue la somme et place le résultat dans le registre `RBX` :

```
# add64.s
# adds two 64 bit constants
        .section  .text
        .globl    _start
_start:
        movq $0x4444444444444444,%rax
        movq $0x4444444444444444,%rbx
        addq %rax,%rbx                    # what does 'rbx' contain?
        nop
```

puis testons ce programme :

```
$ as -gstabs add64.s -o ./add64.o
$ ld add64.o
$ gdb ./a.out
[...]
(gdb) b 1
Breakpoint 1 at 0x400078: file add64.s, line 1.
(gdb) r
Starting program: /home/.../a.out

Breakpoint 1, _start () at add64.s:6
6               movq $0x4444444444444444,%rax
(gdb) info registers
rax            0x0      0
rbx            0x0      0
[...]
rip            0x400078 0x400078 <_start>
[...]
(gdb) s
7               movq $0x4444444444444444,%rbx
(gdb) info registers
rax            0x4444444444444444       4919131752989213764
rbx            0x0      0
[...]
rip            0x400082 0x400082 <_start+10>
eflags         0x202    [ IF ]
[...]
```

```
(gdb) s
8               addq %rax,%rbx       # what does 'rbx' contain?
(gdb) info registers
rax            0x4444444444444444       4919131752989213764
rbx            0x4444444444444444       4919131752989213764
[...]
rip            0x40008c 0x40008c <_start+20>
eflags         0x202    [ IF ]
[...]
(gdb) s
9               nop
(gdb) info registers
rax            0x4444444444444444       4919131752989213764
rbx            0x8888888888888888       -8608480567731124088
[...]
rip            0x40008f 0x40008f <_start+23>
eflags         0xa86    [ PF SF IF OF ]
[...]
(gdb) q
A debugging session is active.

        Inferior 1 [process 3297] will be killed.

Quit anyway? (y or n)  y
$
```

Nous voyons que l'addition a été effectuée.

2.2.4.2 Deuxième exemple : saut et adresse relative au RIP

Écrivons un programme `jmp64.s` qui effectue un saut à l'instruction suivant immédiatement l'instruction à partir de laquelle on effectue le saut :

```
# jmp64.s
        .section  .text
        .globl    _start
_start:
        jmp  end
end:    nop
```

Assemblons-le ! :

```
$ as -gstabs jmp64.s -o jmp64.o
$ ld jmp64.o
$ objdump -d ./a.out

./a.out:     file format elf64-x86-64


Disassembly of section .text:

0000000000400078 <_start>:
  400078:       eb 00                   jmp    40007a <end>

000000000040007a <end>:
  40007a:       90                      nop
$
```

Le point essentiel ici est le code machine de la première instruction, à savoir '`eb 00`' : l'adresse de saut n'est pas, contrairement à ce qu'on pourrait attendre, celle de l'instruction suivante (à savoir `40007a`) mais `0`. En effet, l'adresse est relative au RIP. La valeur de RIP, après avoir effectué une instruction, est celui de l'instruction suivante. Lui ajouter 0 permet donc d'exécuter l'instruction suivante.

2.3 Bibliographie

[AMD-02-1] Advanced Micro Devices, **AMD64 Architecture Programmer's Manual, volume 1 : Application Programming**, publication 24592, 2002, March 2012 for Revision 3.19, xxii + 352 p.

[AMD-02-2] Advanced Micro Devices, **AMD64 Architecture Programmer's Manual, volume 2 : System Programming**, publication 24593, 2002, September 2012 for Revision 3.22.

[AMD-02-3] Advanced Micro Devices, **AMD64 Architecture Programmer's Manual, volume 3 : General-Purpose and System Instructions**, publication 24594, 2002, September 2012 for Revision 3.19, xxxii + 538 p.

[AMD-02-4] Advanced Micro Devices, **AMD64 Architecture Programmer's Manual, volume 4 : 128-Bit and 256-Bit Media Instructions**, publication 26568, 2002, September 2012 for Revision 3.16, xl + 861 p.

[AMD-02-5] Advanced Micro Devices, **AMD64 Architecture Programmer's Manual, volume 5 : 64-Bit Media and x87 Floating-Point Instructions**, publication 26569, 2002, March 2012 for Revision 3.12, xxx + 338 p.

[Ces cinq volumes constituent la documentation officielle d'AMD sur la programmation de l'architecture x86-64.]

[Int] **Intel 64 and IA-32 Architectures Software Developper's**, 3 volumes set.

[À la fois la documentation officielle d'*Intel* en plus de 3 000 pages et la référence. La plupart des livres reprennent une partie de celle-ci. Tous les sujets sont abordés mais il n'y a pas d'exemple.]

`http://www.intel.com/products/processor/manuals/index.htm`

[gas] Manuel officiel de `gas` :

`http://sourceware.org/binutils/docs/as/`
`http://tigcc.ticalc.org/doc/gnuasm.html`
`http://www.gnu.org/software/gdb/documentation/`

[gdb-doc] Guide en ligne de `gdb` :

`http://www.unknownroad.com/rtfm/gdbtut/gdbtoc.html`

[objdump] Manuel officiel de `objdump` :
`http://sourceware.org/binutils/docs/binutils/objdump.html`