Programmation atomique

Découverte de l'assembleur *x86*

Enzo Calamia

26 juin 2012

Table des matières

Licence

Résumé

Ce document à pour sujet les processeurs de la famille *x86*, il aborde des généralités sur ces processeurs ainsi que la programmation assembleur. La partie sur les processeur n'est pas forcément indispensable pur programmer en assembleur. Ce document ne se veut pas être un cours, de très bons livres sur l'assembleur existent et rempliraient leur fonction mieux que le présent doucement. Ce document se veut plutôt être une *découverte*, néanmoins assez complète, à l'intention de ceux qui voudraient en savoir plus sur les mécanismes bas-niveau, voire matériels.

Ce document se veut accessible : le lecteur est juste supposé connaître la base hexadécimale (ou alors savoir se servir d'une calculatrice) et savoir qu'un octet est égal à huit bits. Des bases en programmation et en électronique élémentaire peuvent sûrement aider.

Chapitre 1

Processeurs

1.1 Langage machine et assembleur

Le processeur est capable d'exécuter diverses instructions simples. Ces instructions ne sont ni plus ni moins que de simples valeurs numériques que le processeur comprend nativement. Ces valeurs numériques décrivant les opérations à effectuer constituent le langage machine ou le code natif. Vous en conviendrez, il n'est absolument pas commode pour un humain d'écrire des programmes directement en langage machine, se résumant à une suite de nombres. Le langage assembleur se propose ainsi pour pour résoudre ce problème, nous pouvons voir l'assembleur comme étant la forme humanisée du langage machine : au lieu d'écrire les instructions par leur valeur numérique nous écrivons des instruction par des mnémotechniques qui correspondent au véritable langage machine.

Pour illustration, prenons l'exemple d'une instruction qui doit empiler la valeur 42 sur la pile – une zone mémoire dont nous parlerons prochainement. Voici l'instruction, sous sa forme numérique et en base hexadécimale : `6a2a`. Voici l'instruction assembleur qui lui correspond : `push 42`.

Le programme assembleur (à ne pas confondre avec la langage assembleur) est un traducteur chargé de générer le langage machine correspondant à suite d'instructions assembleur donnée. Un tel programme traduit donc notre `push 42` en `6a2a`.

1.2 Instructions

Nous l'avons vu, une instruction est une opération élémentaire qu'un processeur doit effectuer, mais toutes les instructions ne sont pas forcément compréhensibles par tous les processeurs. Chaque famille de processeurs peut exécuter un certain ensemble défini d'instructions, on appelle cet ensemble un *jeu d'instructions*. Un processeur prend en charge son jeu d'instructions, éventuellement plusieurs. Ce jeu d'instructions est généralement partagé par d'autre processeurs de la même famille, comme c'est le cas pour les processeurs de la famille

x86 qui compte plusieurs modèles de processeurs allant du Intel 8086 de 1978 jusqu'à des processeur plus récents comme les Core i7 de 2011 par exemple. L'intérêt est d'assurer une rétro-compatibilité, un programme écrit pour un 8086 doit fonctionner pour un processeur de la même famille plus récent, même si ce processeur récent est doté de nouveaux jeux d'instructions (plutôt appelées extensions).

On distingue plusieurs types de jeux d'instructions, dont notamment les RISC et les CISC ayant chacun leur paradigme.

Familles de jeux

Processeurs RISC

Les processeurs ayant un jeu de type RISC (*Reduced Instruction Set Computer*) ont la particularité suivante : les instructions sont très simples (élémentaires) et aussi très courtes (en taille). Il faut que les instructions soient toutes de la même taille (ou presque), c'est la clé du paradigme. L'enjeu est de pouvoir exécuter les instructions très rapidement, en utilisant le principe du pipeline : en résumé, c'est du taylorisme dans le processeur. Il faut savoir que l'exécution d'une instruction se fait en plusieurs étapes par le processeur ; sans pipeline on attend de finir totalement l'exécution d'une instruction (c'est-à-dire effectuer toutes les étapes aboutissant à l'exécution de l'instruction) avant d'en commencer une autre. On doit alors finir d'effectuer la dernière étape de l'exécution de l'instruction (plutôt appelée étage, dans le jargon), pour débuter une nouvelle instruction. Cela résulte qu'il y aura toujours des étages inactifs.

Le principe du pipeline est d'avoir toujours la totalité des étages actifs de telle sorte qu'on puisse débuter une nouvelle instruction à chaque étape effectué (et non à chaque instruction). Donc, pendant qu'on effectue l'étape Z d'une instruction i_1, on peut en même temps effectuer l'étape A d'une autre instruction i_2. Voici une petite analogie : pensez à un tuyau (le nom pipeline est bien explicite pour ça) dans lequel on introduit les instructions. Même si une instruction n'est pas encore complètement sortie du tuyau, on peut quand même commencer à en introduire une autre, ce qui est impossible sans pipeline (on serait obligé d'attendre que la première instruction soit totalement sortie du tuyau).

Souvent (mais pas toujours) une instruction de type RISC, grâce à sa sémantique légère, peut être exécutée en une seule période d'horloge (un tic), soit x^{-1} seconde si l'horloge en question est cadencée à x Hz. Ne croyez pas que votre super 3 GHz exécutera systématiquement 3 milliards d'instructions à la seconde, même si c'est presque le cas. Les processeurs RISC les plus connus sont certainement les processeurs de la famille POWER de chez IBM [**?**], ou encore les ARM, même si ces derniers ne sont pas pur RISC [**?**]. Vous comprendrez alors pourquoi il est préférable d'avoir des instructions de même taille. Cela évitera de trop déséquilibrer le pipeline.

Processeurs CISC

S'opposant au RISC, nous avons le CISC (*Complex Instruction Set Computer*). Le paradigme est tout autre, il s'agit ici d'avoir des instructions complexes, dans le sens où leur sémantique est assez lourde, l'instruction n'est plus si élémentaire que pour les RISC. Cela pourrait éventuellement rendre l'écriture d'un programme en assembleur plus simple (ou du moins, moins fastidieuse) pour le programmeur, mais pas pour un compilateur. Un code assembleur pour un processeur de type CISC sera dense, car chaque instruction aura une sémantique lourde. Les processeurs *x86* sont de type CISC, et ce document parle à propose de ceux-là.

Cependant, les processeurs CISC ont aussi un moteur de pipeline, ce n'est pas juste propre aux processeurs RISC, bien que ces dernier basent leur performance sur ce mécanisme principalement. La quatrième génération de Pentium Prescott (de la famille *x86*, de type CISC donc) possédait d'ailleurs une profondeur de pipeline de 31 étages. Le mythe qui dit « x Hz $\iff$ x instructions par seconde » est encore plus faux sur un processeur CISC, bien-entendu.

Processeurs EPIC

Comme nous sommes dans la nouvelle ère du calcul parallélisé, j'ai jugé bon de présenter aussi une dernière famille : EPIC (*Explicitly Parallel Instruction Computing*) qui est utilisé sur les processeurs Intel Itanium, notamment. Ce sont des machines destinées à la haute performance.

Autopsie d'une instruction *x86*

Une instruction *x86* se compose le plus souvent de deux parties. L'*opcode* (ou le code opération) et le ou les opérandes. l'opcode décrit l'action à exécuter, les opérandes sont des valeurs avec quoi l'action doit être exécutée. Bien-sûr, il ne peut y avoir qu'un seul opcode par instruction ; le nombre d'opérande peut en revanche varier de 0 à 2 en règles générales, selon l'instruction. Il est donc possible d'avoir des instructions étant uniquement composé d'un opcode. C'est le cas de `nop` et de `ret` par exemple.

L'opcode correspondant à un `push` d'un valeur immédiate d'un octet est `0x6a` [1], par exemple. Reprenons notre fameux `push 42` qui à l'avantage de rester une instruction simple, et regardons sa structure :

```
push 42
```

En rouge nous avons l'opcode et en bleu le seul opérande nécessaire qui peut être n'importe quoi du moment que ça tient sur le nombre de bits autorisé. Ici c'est 8 bits.

1.3 Expérimentations

C'est bien la culture générale, mais il est temps de passer un peu à la pratique. Nous allons nous munir d'un assembleur (j'utilise fasm, mais libre à vous d'en utiliser un autre, du moment que vous faites un binaire pur avec) et d'un éditeur hexadécimal, je me contenterais du hexdump sans prétention, ça suffira bien. On s'amusera à regarder le langage machine d'un petit programme écrit en assembleur. Nul besoin de connaître l'assembleur, on se focalise sur le langage machine pour l'instant.

Amusons-nous à assembler notre fameux `push 42`, ainsi qu'un petit `push "spin"`, nous allons peut-être être surpris.

```
; fichier : push.s

   use32

   push 42
   push "spin"
```

`use32` n'est pas une instruction, c'est juste une directive pour fasm, elle sert à lui indiquer qu'on travaille en 32 bits, car `"spin"` tient en 4 octets, soit 32 bits. Pour les détenteurs de processeurs 64 bits, n'essayez pas d'empiler une valeur immédiate de 8 octets, ça n'est pas possible. On assemble notre programme :

```
$ fasm push.s
```

ou pour NASM :

```
$ nasm -f bin push.s
```

Là, on a bel et bien un programme, mais il ne se lancera jamais. Et de toute façon on a pas à le lancer, c'est pas ce qui nous intéresse.

Bon, regardons les entrailles du programme :

```
$ hexdump push.bin

0000000 2a6a 7368 6970 006e
0000007
```

On ne regardera pas la première colonne, elle ne nous intéresse pas (elle indique juste les adresses). Le programme lui-même est le suivant (toujours en hexadécimal bien-sûr) : `2a6a 7368 6970 006e`.

À présente essayons de comprendre notre programme en langage machine, voici les informations dont nus aurons besoin :

- 0x6a = 42
- "spin" correspond aux octets suivants, dans l'ordre : 0x73 0x70 0x69 0x6e (simple conversion depuis la table ASCII)

– l'opcode d'un `push` d'un opérande de 8 bits est `0x6a`[**?**]
– l'opcode d'un `push` d'un opérande de 32 bits est `0x68`[**?**]

Remarquez qu'il existe en réalité plusieurs instructions `push` : une pour chaque famille d'opérandes : valeur immédiate de 8, 16 ou 32 bits (noté *imm8, imm16, imm32* dans la littérature Intel), registres de 8, 16, 32 ou 64 bits (noté *r8, r16, r32, r64*) et il existe même un `push` spécialement pour un registre en particulier qui est DS. Le set des instructions `push` n'est pas un cas isolé, il en est ainsi pour la grande majorité des autres set comme les instructions `add` par exemple, chaque instruction `add` est adaptée à une combinaison d'opérandes. Ainsi, quand nous introduirons de nouvelles instructions, nous indiquerons chaque combinaison d'opérandes possible. Par abus de langage nous disons « l'instruction `add` », mais il conviendrait de dire « les instructions `add` ».

Le programme assembleur génère automatiquement la bonne instruction à partir de la combinaison d'opérandes donnée, le programmeur ne s'en soucie pas en règles générales, sauf quelques exceptions.

Vous l'avez sans doute remarqué, c'est totalement le désordre ! Voici donc l'explication à tout cela qui devra servir de morale : les processeurs *x86* lisent tout en LSB (*Less Significant Byte*), c'est à dire de l'octet le plus faible à l'octet le plus fort. On parle aussi de l'*endianess* d'un processeur. En l'occurrence, notre *x86* est un processeur dit *little-endian* – en français, petit boutiste.

Pour avoir la suite d'instruction en MSB (*Most Significant Byte*), soit une façon plus humaine de la lire, nous avons alors simplement à permuter les octets deux à deux comme l'illustre la figure **??**. Je rappelle qu'un chiffre en hexadécimal correspond à un quartet – 4 bits –, donc deux chiffres correspondent à un octet. Il existe des processeurs qui lisent les octets à la mode MSB (les ARM par exemple), ce sont des processeurs dit *big-endian* – en français, gros boutiste.

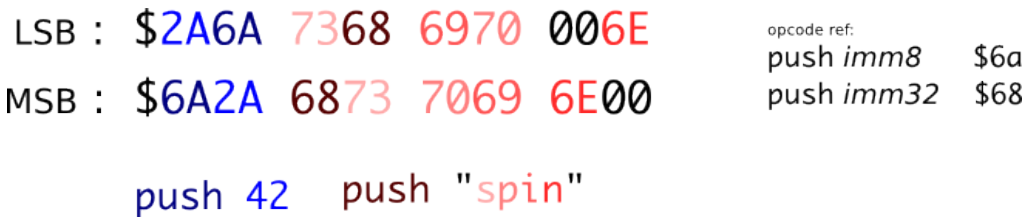

FIGURE 1.1 – Du little-endiann au big-endiann

On a donc tout bien dans l'ordre, et on voit bien ce qu'il se passe. Tous les opcodes ne sont pas sur 1 octet, les jeux CISC ont tellement d'instructions qu'il devient nécessaire de coder les opcode des dernières instructions sur plus d'un octet, comme c'est le cas pour `cpuid` qui a pour opcode `0x0fa2` [**?**] de 2 octets. Cela illustre la différence fondamentale entre les jeux RISC et les jeux CISC ayant beaucoup plus d'instructions de tailles très variables...

1.4 Registres

Je suppose que cette notion ne vous est pas parfaitement familière, mais dans le cas contraire sachez qu'un registre est un tout petit emplacement mémoire directement gravé sur le circuit du processeur, l'accès à un registre se fait donc de façon extrêmement rapide. Les processeurs *x86* disposent de beaucoup de registres – surtout les derniers – que nous n'allons pas énumérer, mais il faut dans un premier temps avoir un aperçu général de leur utilité.

La taille des registres peut varier de 16 bits à 256 bits pour les derniers modèles. En réalité, les registre d'usage général ne varient que de 16 bits à 64 bits. Nous comptons 4 registres à usage général : RAX, RBX, RCX et RDX qui ont une capacité de 64 bits chacun. Sur les modèles moins récents, nous comptons aussi ces quatre même registres, dans leur version 32 bits : EAX, EBX, ECX et EDX. Avec un processeur 64 bits, il est toujours possible d'accéder aux registres dans leur version 32 bits, ce derniers sont les parties basses – désignant les 32 bits de poids faible – des registres 64 bits. De même, il est possible d'accéder aux registre 16 bits qui sont : AX, BX, CX et DX. Là encore c'est le même principe, il s'agit juste des parties basses des registres EAX, EBX, ECX et EDX. À partir des registres 16 bits, il est à nouveau possible d'accéder aux registre de 8 bits, mais cette fois il est possible de spécifier la partie haute ou la partie basse : AH, AL, BH, BL, CH, CL, DH et DL. Les signifient *High* pour les parties hautes, et *L* signifie *Low* pour les parties basses. La figure **??** illustre la décomposition du registre RAX, aussi valable pour les autre registres d'usage général.

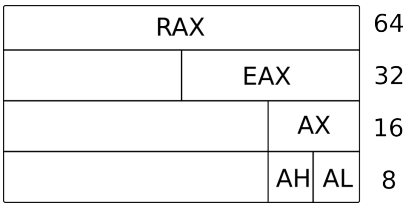

FIGURE 1.2 – Décomposition de RAX

Ces registres-là sont dit à usage général, il permettent de faire un peu ce qu'on veut mais le plus habituel est de stocker les résultats intermédiaires dedans, lors de calcul sur des valeurs contenues dans ceux-là. Nous manipulerons ces registres bientôt lors de l'implémentation de quelques algorithmes, dans un premier temps.

Chapitre 2

Premiers pas

Ce chapitre a pour but de se familiariser avec le paradigme de l'assembleur en douceur : utiliser les registres, effectuer des opérations avec. Le présent chapitre n'aborde donc pas la gestion de la mémoire.

2.1 Structure simplifiée du processeur

Il est important d'avoir une idée – même un vision vague – de la structure d'un processeur. Nous pouvons nous contenter d'une approximation si nous gardons bien à l'idée que c'est une structure simplifiée à outrance. Dans un processeur simple, nous pouvons compter trois éléments principaux : un bloc central qui sera le chef d'orchestre du processeur, un bloc de plusieurs registres et enfin une unité de calcul, qu'on appelle l'ALU (*Arithmetic and Logic Unit*).

Le bloc central peut communiquer avec les registre, avec l'ALU et avec la mémoire externe au processeur – la mémoire RAM (*Random Access Memory*). Nous savons bien qu'un programme est une suite d'instructions exécutables, mais cette suite doit bien est stocké quelque-part : dans la RAM. Ainsi, le bloc central ira récupérer une instruction dans la RAM, l'interpréter et enfin l'exécuter, puis elle recommence ; on appelle ça le cycle *fetch-decode-execute cycle*. Pour savoir où chercher dans la RAM, le bloc central (qui a aussi accès aux registres) a recours à un registre spécial : EIP (*Extended Instruction Pointer*). Ce registre contient toujours l'adresse de la prochain instruction à aller récupérer, il est donc incrémenté au fil des exécutions des instructions. Dans notre processeur schématique, c'est ce bloc contenant aussi l'étape *execute* qui fera exécuter les instructions.

L'ALU est l'unité de calcul arithmétique et logique, comme son nom l'indique. Il s'agit d'un circuit logique combinatoire complexe où les opérations sont implémentées directement en portes logique. L'ALU prend symboliquement deux entrées, effectue l'opération désirée dessus et renvoie en sortie le résultat. Cette unité implémente maintes opérations dont l'addition, la soustraction, la multiplication, la division, le ET logique, le OU logique, le XOR logique *etc.*

Enfin, le bloc des registres contient tous nos registres d'usage général, ou les pointeurs comme EIP par exemple, mais aussi un registre très important : il s'agit du registre de *flags*. Nous verrons son utilité d'ici peu.

2.2 Opérations arithmétiques et logique

Opération arithmétique

Le processeur nous fournit de base des instructions pour additionner, soustraire, multiplier, diviser et bien d'autres. Ce sont des instructions simples à utiliser comme nous le verrons dans les exemples. Comme nous l'avons vu,

Pour additionner, le processeur propose l'instruction `add` qui prend deux opérandes : la destination et la source. L'instruction `add` additionnera la source avec la destination, le résultat de l'addition sera placé dans la destination. Il existe une instruction `add` pour chacune de ces combinaisons suivantes :

- `add r/m8, r8`
- `add r/m16/32/64, r16/32/64`
- `add r8, r/m8`
- `add r16/32/64, r/m16/32/64`
- `add AL, imm8`
- `add RAX, imm16/32`

Voici quelque exemples de l'utilisation de `add` :

```
add ah, dl      ; place ah + dl dans ah
add rax, rcx    ; rax + rcx dans rax
add eax, 42     : eax + 42 dans eax
add al, 27      ; al + 27 dans al
```

L'instruction `sub` fonctionne de la même manière et propose les mêmes combinaisons d'opérande que l'instruction `add`, attention tout-de-même à l'ordre des opérandes cette fois-ci : l'instruction `sub` soustrait la source par la destination :

```
sub ah, dl      ; place ah - dl dans ah
sub rax, rcx    ; rax - rcx dans rax
sub eax, 42     : eax - 42 dans eax
sub al, 27      ; al - 27 dans al
```

Opération logiques

Les opérations logiques sont certainement la chose que le processeur sait le mieux faire

2.3 Sauts et étiquettes

L'assembleur génère un programme *statique*, chaque adresse de chaque instruction ou donnée est connue à l'exécution. Dès lors, il est possible de « sauter »

à une adresse en particulier où se trouvera une autre instruction ou une donnée (après tout, le processeur en fait pas la différence, dans tous les cas ce ne sont ni plus ni moins que de simple octets). Il existe une instruction pour ce faire : `jmp`, pour *JuMP*. Cette instruction prend un opérande : l'adresse de la prochaine instruction à exécuter. En d'autre termes, cette adresse sera chargé dans le registre IP, il en résulte ainsi que le processeur exécutera l'instruction de cette adresse juste après l'instruction `jmp`, sans revenir. Le code suivant présente un exemple. Les adresses sont ici prises au hasard, afin de faire office d'exemple :

```
jmp 0x539       ; adresse quelconque
add eax, 69
...
push 42         ; adresse 0x539
or al, al       ; adresse 0x539 + 2 = 0x541
```

Lors du `jmp 0x539`, la valeur $539_{(16)}$ est chargée dans IP (d'une façon simplifiée), la prochaine instruction est donc `push 42`, ensuite IP est incrémenté comme il faut – ici de 2 car `push 42` a une taille de deux octets – et on continue donc en exécutant `or al, al`. L'instruction `add eax, 69` ne sera jamais exécuté. Les instructions de saut tel que `jmp` sont des instruction dites de *branchement*.

Si l'assembleur génère un binaire pur comme nous en avons assemblé un dans la première partie, alors l'adresse de la première instruction est tout simplement 0. Nous pouvons le voir en reprenant notre exemple :

```
$ hexdump push.bin

0000000 2a6a 7368 6970 006e
0000007
```

Nous pouvons ainsi effectuer des branchements lors de l'exécution du programme, si nous connaissons l'adresse de la prochain instruction à exécuter. Considérons l'exemple suivant :

```
xor eax, eax
add eax, 44
sub eax, 2
jmp ?          ; on veut sauter...
xor ebx, ebx
add ebx, 1337
add ecx, edx   ; ...ici
push 42
push 1337
```

Nous souhaitons ici sauter à l'instruction `add ecx, edx`, pour cela nous devons connaître son adresse. Il nous est tout-à-fait possible de la calculer si on

spécifie au préalable une adresse de départ pour notre première instruction. Pour connaître cette adresse, il suffira ainsi d'additionner successivement la taille (en octet) de chaque instruction se trouvant entre le départ et l'instruction visée. En l'occurrence, il nous faudrait additionner la taille des instructions suivantes pour obtenir l'adresse de l'instruction visée :

```
xor eax, eax   ; adresse 0
add eax, 44    ; + taille(xor eax, eax)
sub eax, 2     ; + taille(add eax, 44)
jmp ?          ; + taille(sub eax, 2)
xor ebx, ebx   ; + taille(jmp ?)
add ebx, 1337  ; + taille(xor ebx, ebx)
               ; = adresse de l'instruction visée
```

C'est une tâche que nous nous donnerons pas la peine d'effectuer, c'est bien trop fastidieux. Il nous faudrait un manuel de référence du constructeur pour connaître la taille de chaque instruction et s'ajoute à cela le fait que le `jmp ?` lui-même n'a pas une taille dont nous pouvons être sûr si nous ne connaissons pas l'adresse.

Heureusement pour nous, les assembleurs calculent ces adresses pour nous. Pour spécifier un endroit précis dans le code où une adresse aura à être calculée par l'assembleur, nous disposons d'étiquettes, souvent appelés aussi *label*. Les labels sont des identificateurs associés à des adresses inconnues lors de l'écriture du code. Dans le suivant code, nous en faisons usage :

```
    xor eax, eax
    add eax, 44
    sub eax, 2
    jmp mon_label    ; on veut sauter...
    xor ebx, ebx
    add ebx, 1337
mon_label:
    add ecx, edx     ; ...ici
    push 42
    push 1337
```

Cette fois, nous plaçons un label du nom de `mon_label` correspondant à l'adresse où se trouve l'instruction (ou la donné) qui précède le label, soit `add ecx, edx` en l'occurrence. Un label est généralement une suite de lettres suffixée par un deux-point « : ». Ce label sera symbolisé simplement par son nom. Notez l'indentation, il est d'usage – pour des raison de lisibilité du code source – de placer les instructions à un, voire deux, niveaux d'indentations par rapport à la marge. Ainsi, il devient possible de placer les labels au bord de la marge en leur assurant d'être lisibles. Bien évidemment, il ne peut y avoir plus d'une adresse pour un seul label.

Il est possible d'écrire la suivante instruction, il vous reviendra de devinez ce à quoi elle correspond :

```
label:
    jmp label
```

Ce genre d'instruction peut éventuellement avoir un intérêt dans des cas très rares, et encore...

2.4 Branchements conditionnels

Le contrôle du flux d'instructions – ou conditions – permet aux programmes d'avoir un comportement non-linéaire, de lui permettre de faire des choix. En assembleur il n'y a pas de structure de contrôle comme on en trouverait dans un langage haut-niveaux tel que le C, pas de `if` ni de `while`.

En assembleur nous contrôlons le flux d'instructions grâce à un registre très spécial : le registre de *flags*. Ce registre contient des drapeaux – des bits – qui peuvent être levés ou non, ce qui correspond à un bit au niveau logique 1 ou 0. Ce registre est habituellement FLAGS dans sa version 16 bits, EFLAGS dans sa version 32 bits et enfin RFLAGS dans sa version 64 bits.

Le programmeur n'a pas à aller toucher directement au flags, chaque drapeau du registre est modifié en conséquence de la dernière instruction exécutée. Citons quelques flags des plus utiles afin d'illustrer cela :

- CF (*Carry Flag*), bit 0 : ce flag est levé (mis à 1) si jamais une opération arithmétique a générer une retenue. Sinon ce flag est baissé.
- ZF (*Zero Flag*), bit 6 : ce flag est levé si le résultat d'une instruction arithmétique a été 0. Utile pour les comparaisons.
- OF (*Overflow Flag*), bit 11 : ce flag est levé si une instruction arithmétique a déclenché un débordement de capacité

Voyons un code permettant d'illustrer les choses plus concrètement :

```
mov ax, 65534
add ax, 200    ; 65534 + 200 ne tient pas sur 16 bits, OF = 1

mov eax, 42
sub eax, 42    ; 42 - 42  = 0, ZF = 1
```

Vous vous demandez peut-être quel est intérêt de ces flags. Après tout, on ne contrôle pas le flux d'exécution plus qu'on ne le faisait avant, pour l'instant. Il nous manque en fait des instructions de branchement conditionnels, qui peuvent sauter à un endroit du programme si seulement le flag correspondant est levé. Pour nos trois flags précédents, voici les instructions de saut conditionnelles correspondantes :

- `jc` (*Jump if Carry*) : saute si CF = 1

– `jz` (*Jump if Zero*) : saute si ZF = 1, il existe un synonyme de cette instruction identique (même opcode) : `je` (*Jump if Equal*
– `jo` (*Jump if Overflow*) : saute si OF = 1

La raison de deux mnémotechniques `je` et `jz` pour une même instruction vient du fait qu'on utilise le Zero Flag aussi pour des comparaisons. Une comparaison se fait par l'instruction `cmp` qui est la même instruction que `sub` à l'exception qu'elle ne stocke pas le résultat de la soustraction dans la source. Elle se content juste de lever ou baisser le Zero Flag. Le fait d'utiliser le mnémotechnique `je` ou `jz` n'a vraiment aucune importance, c'est la même chose, mais par convention nous utiliserons le mnémotechnique qui correspondra le mieux au contexte dans le code afin de facilité la lecture de ce dernier. Voici un exemple de son utilisation :

```
mov eax, 42
sub eax, eax   ; ZF = 1, le résultat 0 est stocké dans EAX
jz label_1     ; si ZF = 1, redirection du flux

mov eax, 42
cmp eax, eax   ; ZF = 1, EAX contient toujours 42
je label_2     ; si ZF = 1, redirection du flux
```

Il existe des instruction de saut complémentaires à celles évoquées, elle redirigent le flux d'exécution seulement si le flag n'est pas levé : `jnz` (*Jump if Not Zero*) ou `jne` (*Jump if Not Equal*), `jnc` (*Jump if Not Carry*), `jno` (*Jump if Not Overflow*).

Il existe bien-entendu d'autres flags ainsi que d'autres instructions de saut conditionnel. Nous avons présenté ici les plus usuelles.

Chapitre 3

Notions élémentaires

Pour l'instant, nos programmes ne pouvaient pas répondre à des problématiques trop complexes. Nous illustrerons les concepts tout-au-long de ce chapitre par un petit programme qui doit effectuer des opérations vectorielles, pour des vecteurs tri-dimensionnel.

3.1 Mémoire et adressage

Données

Nous avions vu précédemment qu'il était possible de marquer des emplacements mémoire grâce au labels. Mais le programme assembleur est aussi un « éditeur d'octets » dans le sens o'u il en permet l'écriture directe dans le fichier final. Les assembleurs proposent en général une directive de déclaration de donnée. Déclarer des octets revient à les écrire directement dans le fichier final. Il est même possible d'écrire les instructions en langage machine directement :

```
db $6a, 2a  ; la même chose que push 42
```

Ici, la directive `db` (*Declare Byte*) permet d'écrire par octet. Le programme précédent écrit donc l'octet `6a`, puis l'octet `2a`. Notez qu'il n'est pas obligatoire d'écrire les octets en hexadécimal. Il existe aussi d'autre directives permettant de déclarer des mots de 16 bits, des double-mots de 32 bits, des quadruple-mots de 64 bits *etc.* Il sera possible d'en avoir une liste exhaustive dans les annexes.

Bien-entendu, la déclaration de donnée ne sert pas avant-tout à écrire des instructions en langage machine, mais plutôt à définir les données que le programme utilisera comme, par exemple, des chaînes de caractères, des constantes ou même des variables si l'on considère qu'il sera possible d'écrire sur les donnée durant l'exécution du programme [1]. Les lecteurs ayant déjà une expérience

1. Ce document se veut générique et ne traite pas de l'interaction avec un système d'exploitation, cependant il faut savoir que les systèmes modernes ne permettent pas aux programme de lire et écrire n'importe où dans la mémoire : les programmes sont scindés en plusieurs sections avec leurs droits d'accès. Plus de détail à l'annexe page **??**

dans un langage de haut-niveau pourront faire l'analogie avec les variables et les constante du C ou du C++.

Notre programme d'illustration doit satisfaire les suivantes conditions :

- des vecteurs à trois coordonnées
- chaque coordonnée à une taille d'un octet
- opérations vectorielles (addition, soustraction, scalaire et produit vectoriel)

Soient deux vecteurs $\vec{u}$ et $\vec{v}$:

$$\vec{u} \begin{pmatrix} 42 \\ 3 \\ 8 \end{pmatrix}$$

$$\vec{v} \begin{pmatrix} 4 \\ 6 \\ 1 \end{pmatrix}$$

Nous les représenterons ces données de la manière qui suivra, nous plaçons les labels `u` et `v` afin de nous en servir comme *pointeur* sur chacun des vecteurs, puisque nous ne pouvons pas manipuler 3 octets en une seule entité :

```
u: db 42, 3, 8  ; vecteur u
v: db  4, 6, 1  ; vecteur v
```

Adressage

Pour commencer, essayons d'implémenter un algorithme en assembleur qui prend en entrée la donnée d'un vecteur (disons le vecteur *vecu*) et rend en sortie un nombre – la somme des trois coordonnées. Nous ferons les calculs sur les registres d'usage général 8-bit. L'algorithme est simple : obtenir l'adresse d'un vecteur, placer sa première composante dans AL, additionner AL avec sa seconde composante, additionner AL avec sa troisième composante.

```
; fichier : vector.s

mov byte al, [u]   ; première composante placée
add byte al, [u+1] ; addition avec la seconde composante
add byte al, [u+2] ; addition avec la troisième composante


u: db 42, 3, 8
v: db  4, 6, 1
```

Certaines chose devraient vous intriguer, notamment les crochets. Nous savons bien que `u` est la valeur numérique représentant l'adresse du vecteur *vecu*. Mais ça ne nous intéresse pas, ici, d'additionner des adresses. Nous voulons le *contenu* des adresses. Pour l'indiquer, nous encadrons une référence au label par des crochets. Les lecteurs qui connaissent le langage C pourront faire l'analogie

suivante : soit `u` un pointeur, le `[u]` de l'assembleur désigne la même chose que le `*u` du langage C.

Vous remarquerez aussi que nous avons introduit un mot `byte`. Comme nous le savons, il n'existe pas un instruction `mov` et `add` mais *des* instructions `mov` et `add`. Il est parfois nécessaire de préciser la taille des opérandes de l'opération car l'assembleur ne peut pas toujours le deviner en fin de compte. En réalité, dans cet exemple fasm aurait su le deviner car il sait que AL est un registre d'un octet. Mais préciser la taille des opérandes lorsqu'on travaille avec des valeurs indirectes est une bonne habitude.

Quant aux additions dans les crochets, elle permettent simplement de passer à l'octet suivant. Il est important de savoir que les additions faites entre crochets sont uniquement à l'intention du programme assembleur. Le programme assembleur se contentera de calculer la nouvelle adresse définie par `u` + une constante, mais ça n'aura aucun incidence sur la valeur de `u` ni sur rien d'autre. Le dump suivant, obtenu par le désassembleur `ndisasm`, devrait vous en convaincre :

```
$ ndisasm vector.bin
00000000  A00B00            mov al,[0xb]
00000003  02060C00          add al,[0xc]
00000007  02060D00          add al,[0xd]
0000000B  2A03              sub al,[bp+di]
0000000D  0804              or [si],al
0000000F  06                push es
00000010  01                db 0x01
```

Les adresses sont à gauche, dès la première instruction nous chargerons bien le contenu de l'adresse $0b_{16}$, soit `2a`. Ensuite, nous l'additionnons avec l'octet contenu à l'adresse $0c_{16}$, soit `03` *etc.*

Vous remarquez donc que les instructions suivant le `add al,[0xd]` n'ont pas de sens pour nous. Et pourtant, nous les avons bel et bien écrites ! Il s'agit tout simplement de nos deux vecteurs *vecu* et *vecv* dont leur octet correspond à ce que vous voyez. Morale de cette histoire : des octets sont octets, le processeur ne fait *aucune* différence entre instruction et donnée, tout n'est ni plus ni moins que des octets.

À présent nous allons réaliser l'implémentation d'une opération plus complexe : la somme de deux vecteurs. Tout simplement, il s'agit d'additionner chaque composantes des deux vecteurs une-à-une. soient les vecteurs $\vec{u}$ et $\vec{v}$:

$$\vec{u} \begin{pmatrix} x_A \\ y_A \\ z_A \end{pmatrix}$$

$$\vec{v} \begin{pmatrix} x_B \\ y_B \\ z_B \end{pmatrix}$$

Le vecteur somme $\vec{w} = \vec{v} + \vec{u}$ est simplement l'addition de chaque composantes de $\vec{u}$ et $\vec{v}$:

$$\vec{w} = \vec{v} + \vec{u} = \begin{pmatrix} x_A \\ y_A \\ z_A \end{pmatrix} + \begin{pmatrix} x_B \\ y_B \\ z_B \end{pmatrix} = \begin{pmatrix} x_A + x_B \\ y_A + y_B \\ z_A + z_B \end{pmatrix}$$

À notre niveau, nous n'avons que peu de peine à imaginer la structure du code ; tout d'abord nos deux vecteurs habituels définis en tant que chaîne de trois octets. Puis, une autre chaîne de trois octets représentant le vecteur somme $\vec{w}$. Nous ne connaissons pas la valeur des composantes de $\vec{w}$, ainsi nous pouvons soit déclarer simplement trois octets à des valeurs quelconques, sachant que ces valeurs seront écrasées par l'opération, ou bien nous pouvons juste réserver des octets. La réservation d'octet est une alternative à la déclaration : elle permet de ne pas spécifier des valeurs. Avec fasm, la réservation se fait avec le mot-clé `rb` (*Reserve Byte*) suivi du nombre d'octets voulus.

```
u: db 42, 3, 8
v: db  4, 6, 1
w: rb 3           ; réservation de 3 octets
```

Plus rien ne nous empêche d'implémenter l'opération à présent, dont un algorithme simple pourrait être le suivant : charger x_A dans un registre, lui additionner x_B. Placer le contenu du registre dans la mémoire alloué pour le vecteur $\vec{w}$. En faire de même pour les autres composantes.

```
; fichier : vector2.s

mov al, [u]
add al, [v]
mov [w], al     ; 1re composante de w calculée

mov al, [u+1]
add al, [v+1]
mov [w+1], al   ; 2e composante de w calculée

mov al, [u+2]
add al, [v+2]
mov [w+2], al   ; 3e composante de w calculée


u: db 42, 3, 8
v: db  4, 6, 1
w: rb 3         ; réservation de 3 octets
```

Il est toujours intéressant de regarder le code désassemblé afin de s'assurer que l'adressage est correct :

```
$ ndisasm vector2.bin
```

```
00000000  A01E00            mov al,[0x1e]
00000003  02062100          add al,[0x21]
00000007  A22400            mov [0x24],al
0000000A  A01F00            mov al,[0x1f]
0000000D  02062200          add al,[0x22]
00000011  A22500            mov [0x25],al
00000014  A02000            mov al,[0x20]
00000017  02062300          add al,[0x23]
0000001B  A22600            mov [0x26],al
0000001E  2A03              sub al,[bp+di]
00000020  0804              or [si],al
00000022  06                push es
00000023  01                db 0x01
```

La raison pour laquelle nous passons par un registre intermédiaire, AL, est que nous ne pouvons pas directement charger une valeur depuis la mémoire, à la mémoire. Peut-être que vous vous demandez si un code comme le suivant est valide :

```
; code invalide

mov [w], [u]
add [w], [v]     ; 1re composante de w calculée
```

La réponse est non, aucune instruction `mov` ou `add` n'est prévue à cet effet, comme le manuel [**?**] nous l'indique. Nous aurions pu choisir à peu près n'importe quel registre pour l'intérimaire. Cependant, une instruction du type `mov al, [v]` est plus optimisée, son opcode est tout simplement `a0` comme nous le montre si bien `ndisasm`. Dans la littérature Intel, cette instruction est notée *MOV AL,moff8* [**?**].

3.2 Pile

La pile n'est ni plus ni moins qu'une section dans la mémoire.

3.3 Fonctions

Chapitre 4

Approfondissement

4.1 Extensions SIMD

Depuis l'ère du calcul vectoriel *maison*, les processeurs intègrent de nouvelles extensions, dont les extension SIMD (*Single Instruction Multiple Data*). Une extension est généralement le couple d'un jeu d'instructions et d'un jeu de registres, ces derniers ayant un usage en particulier. Les extensions SIMD furent dans un but de *vectorisations* des calculs. Le paradigme est le suivant : ayant une donnée multiple (un vecteur, une matrice), nous disposons d'instructions capable d'opérer sur la donnée multiple directement. Nous illustrerons bien-évidemment cette section par les même exemples précédents des vecteurs $\vec{u}$ et $\vec{v}$ afin d'en observer les contrastes entre l'ancien paradigme et le nouveau.

Les processeurs Intel et AMD se sont vus dotés de nouvelles extensions ces dernières années, notamment MMX (*MultiMedia eXtensions*) et SSE (*Streaming SIMD Extensions*) qui seront abordées dans le présent document.

Extensions MMX

L'extension MMX, introduite avec le Pentium II [**?**], est composée d'un jeu d'une cinquantaine d'instructions, ainsi que huit registres 64 bits portant les noms : MM0, MM1, MM2, MM3, MM4, MM5, MM6 et MM7. Un seul registre peut contenir une donnée multiple, soit deux composantes de 32 bits, quatre composantes de 16 bits ou bien huit composantes de 8 bits.

Reprenons notre premier programme : l'addition des deux vecteurs $\vec{u}$ et $\vec{v}$. MMX nous offre des instructions d'opération sur plusieurs composantes constituant la donnée d'un seul registre. En premier lieu, il nous faudrait avant tout pouvoir placer nos données dans ces fameux registre MMn. MMX inclut deux instructions de déplacement – des instructions `mov` spéciales – qui sont `movd` (*Move Doubleword* – double mot de 32 bits) ou `movq` (*Move Quadword* – quadruple mot de 64 bits). Nous utiliserons évidemment l'instruction `movd` qui déplace 4 octets. Nos vecteurs ayant seulement trois composante, il nous suffira juste de définir un quatrième composante étant égale à 0.

```
u: db 42, 3, 8, 0
v: db  4, 6, 1, 0
w: rd  1           ; réservation de 4 octets, idem que 'rb 4'
```

L'instruction déplacera les vecteurs ayant un taille de 32 bits dans les parties basse des registres MMn. La partie haute sera automatiquement mise à 0.

La directive `rd` (*Reserve Doubleword*) réserve un double mot, soit quatre octets. Ici l'algorithme serait le suivant : placer un vecteur dans un registre MMn, placer l'autre vecteur dans le registre MMm, puis additionner les deux vecteurs. Remarquez que cette fois-ci nous parlons d'un vecteur comme étant une entité à part entière, plus de pointeur sur une chaîne.

L'extension MMX fournit diverse instructions d'addition, pour chaque cas. Le cas intéressant ici est l'instruction d'addition sur des composantes d'un octet : `paddb`. Il est important de noter que cette instruction effectue des additions sur des entiers non-signés, soit des entiers naturels. [1]

```
movd mm0, [u]
movd mm1, [v]

paddb mm0, mm1

; mm0 contient le vecteur somme

movd [w], mm0

u: db 42, 3, 8, 0
v: db  4, 6, 1, 0
w: rd  1           ; réservation de 4 octets, idem que 'rb 4'
```

Souvenez-vous que nous manipulons ici des vecteurs comme étant une donné à part entière, ainsi nous mettons bel et bien le contenu de `u` et `v` dans les registres, plus des pointeurs ; d'où les crochets. De plus, nous n'avons pas besoin de spécifier la taille des opérandes, l'assembleur le devine tout seul étant donnée que l'instruction `movd` déplace justement quatre octets.

Il aurait été possible de réaliser le même programme de la façon suivante :

```
movd mm0, [u]
paddb mm0, [v]
movd [w], mm0

u: db 42, 3, 8, 0
v: db  4, 6, 1, 0
w: rd  1           ; réservation de 4 octets, idem que 'rb 4'
```

1. Formellement, un entier non-signé est un entier dans un ensemble $\mathbb{E}$ défini de la manière suivante : $\mathbb{E} = \{n \mid n \in \mathbb{N} \wedge n \leq 2^{n-1}\}$ où n est le nombre de bits sur lequel on représente le nombre.

Où nous additionnons le vecteur dans MM0 directement avec un vecteur en mémoire.

De toute évidence, les extensions SIMD sont extrêmement puissantes, nous avons réalisé ici une addition vectorielle en trois instructions tandis que notre ancien code séquentiel en comptais neuf :

```
; code séquentiel

mov al, [u]
add al, [v]
mov [w], al     ; 1re composante de w calculée

mov al, [u+1]
add al, [v+1]
mov [w+1], al   ; 2e composante de w calculée

mov al, [u+2]
add al, [v+2]
mov [w+2], al   ; 3e composante de w calculée


u: db 42, 3, 8
v: db  4, 6, 1
w: rb 3         ; réservation de 3 octets
```

Le lecteur désireux de connaître l'extension MMX dans les moindres détails peut se reporter au chapitre 9, *Programming With the Intel MMX Technology*, du manuel Intel, volume 1 [**?**].

4.2 Optimisations

4.3 Interface C

Convention `gcc`

Annexe A

Table ASCII

0 NUL	10 DLE	20	30 0	40 @	50 P	60 ‘	70 p
1 SOH	11 DC1	21 !	31 1	41 A	51 Q	61 a	71 q
2 STX	12 DC2	22 "	32 2	42 B	52 R	62 b	72 r
3 ETX	13 DC3	23 #	33 3	43 C	53 S	63 c	73 s
4 EOT	14 DC4	24 $	34 4	44 D	54 T	64 d	74 t
5 ENQ	15 NAK	25 %	35 5	45 E	55 U	65 e	75 u
6 ACK	16 SYN	26 &	36 6	46 F	56 V	66 f	76 v
7 BEL	17 ETB	27 ’	37 7	47 G	57 W	67 g	77 w
8 BS	18 CAN	28 (	38 8	48 H	58 X	68 h	78 x
9 HT	19 EM	29)	39 9	49 I	59 Y	69 i	79 y
A LF	1A SUB	2A *	3A :	4A J	5A Z	6A j	7A z
B VT	1B ESC	2B +	3B ;	4B K	5B [	6B k	7B {
C FF	1C FS	2C ,	3C <	4C L	5C \	6C l	7C |
D CR	1D GS	2D -	3D =	4D M	5D]	6D m	7D }
E SO	1E RS	2E .	3E >	4E N	5E ^	6E n	7E ~
F SI	1F US	2F /	3F ?	4F O	5F _	6F o	7F DEL

Annexe B

Aperçu des formats exécutables

B.1 Unix et ELF

Le format

B.2 Windows NT et PE

Annexe C

Utilitaires

C.1 fasm

L'assembleur fasm – ainsi que son code source, écrit en assembleur – est disponible gratuitement pour les plateformes Linux, Unix, Windows et DOS à l'adresse suivante : `http://flatassembler.net/`

C.2 NASM

L'assembleur NASM (*Netwide Assembler*) est disponible gratuitement – ainsi que son code source, écrit en C – à l'adresse suivante : `http://www.nasm.us/`. Il supporte Linux, Mac OS X, Unix, Windows et DOS.

Annexe D

Représentation des nombres

D.1 Entiers relatifs

heu...

D.2 Nombres à virgule flottante

Le comité IEEE (*Institute of Electrical and Electronics Engineers*) a défini une norme afin de décrire des nombres dit *flottants* (nombre à virgule). Il s'agit de a norme IEEE 745. Si vous avez du fric à dépenser vous pouvez vous procurer le document officiel de la norme en format PDF à l'adresse suivante `http://grouper.ieee.org/groups/754/`. Un tel document serait utile pour un ingénieur qui conçoit son processeur, mais nous sommes ici des programmeurs.

Soit $x \in \mathbb{R}$, x est représentable satisfaisant le standard IEEE 754 de la suivante manière : [1]

$$x = (-1)^s \cdot m \cdot \beta^p$$

où β est la base (10 ou 2), m est la mantisse du nombre, s est le bit de signe (1 pour $-$ et 0 pour $+$) et q est l'exposant.

D.3 Nombres complexes

Là vous vous démerdez à faire l'implémentation vous-même, définissez la norme que vous voulez. Techniquement, un nombre complexe pourrait simplement être représenté par un couple de deux réels a et b, après à vous d'implémenter les opérations.

1. En réalité, x n'est que dans une partie *finie* de $\mathbb{R}$. Les ordinateurs du futur auront peut-être un moyen de représenter n'importe quel $x \in \mathbb{R}$.

A propos

Ce document a été composé avec LaTeX, les images sont dessinées avec Inkscape 0.48, bien qu'elle soit matricialisées. Les images sont ré-utilisables à des fins non-commerciale, il en va de même pour le présente document.