Programmation en assembleur Gnu sur des microprocesseurs de la gamme Intel (du 80386 au Pentium-Pro)

ENSIMAG

Première année

Année 2009

X. Rousset de Pina

Sommaire

Remerciements

De nombreuses personnes ont collaboré à l'écriture de ces notes soit directement pour en relire et critiquer les premières versions, soit indirectement en m'autorisant à faire de nombreux emprunts à leurs propres notes de cours. Dans le premier groupe je tiens tout particulièrement à remercier : Luc Bellissard, Catherine Bellon, Guy Mazaré, Jacques Mossière, Simon Nieuviarts et Yann Rouzaud, et dans le second groupe Sacha Krakowiak, Yann Rouzaud et Xavier Nicollin auxquels j'ai fait de très nombreux emprunts.

1. Introduction

Pour pouvoir être exécuté un programme doit être placé dans la mémoire d'un ordinateur sous forme d'une suite d'instructions codées, soit en définitive une suite de bits. Cette forme est peu appropriée à une expression commode des algorithmes : aussi utilise-t-on communément des programmes écrits dans un langage plus adapté (Pascal, Java, Ada, C ou autres). Ces programmes sont dits **source**, par opposition aux programmes exécutables, dits **objets**.

Mais un programme source n'est pas directement exécutable par un processeur. Il existe deux manières de faire exécuter les actions qu'il décrit :

- Traduction : Un programme appelé **traducteur** convertit le programme source en un programme objet équivalent. Ce programme objet est ensuite exécuté par l'ordinateur. En fait, les compilateurs de la plus part des langages ne produisent pas directement des programmes objets, mais des programmes dans un langage intermédiaire. Ce programme, doit à son tour être traduit ou interprété. Le langage intermédiaire est souvent un langage dont les constituants sont très liés à ceux fournis par la machine cible (celle sur laquelle le programme doit s'exécuter), du point de vue des types de données aussi bien que de celui des instructions. De tels langages sont appelés langage d'assemblage ou **assembleurs** et leurs traducteurs des assembleurs. Ainsi, sous Unix la plus part des traducteurs (compilateurs) produisent de l'assembleur. Le langage Java est une des exceptions, il produit un code intermédiaire dit « code binaire » qui est interprété par la machine virtuelle Java (JVM).
- Interprétation : Un programme appelé **interpréteur** lit le programme source et exécute les actions décrites dans le programme au fur et à mesure. Tout se passe donc comme si l'ensemble (ordinateur, programme interpréteur) constituait une nouvelle machine adaptée à l'interprétation directe du programme source.

Un interpréteur est en général plus facile à mettre en œuvre qu'un compilateur. Mais l'interprétation d'un programme source est beaucoup plus lente que l'exécution directe d'un programme objet équivalent.

Dans les deux cas, on voit apparaître, en plus des programmes qui réalisent une application particulière, des programmes auxiliaires (les traducteurs ou les interpréteurs) dont le seul but est de faciliter la mise en œuvre des programmes d'application sur une machine. L'ensemble de ces programmes auxiliaires constitue le **logiciel de base**. Les fonctions réalisées par le logiciel de base comprennent en particulier : les fonctions liées à la production et à l'exécution des programmes (traducteur, interpréteurs, relieurs, chargeurs etc.), les fonctions liées à la conservation des informations, les fonctions d'entrée-sortie et enfin les fonctions liées au partage d'une machine entre plusieurs utilisateurs. De ces quatre familles de fonctions les trois dernières sont généralement mises en œuvre par le système d'exploitation de la machine (Unix, Windows NT ou autre). Les fonctions fournies par le système d'exploitation sont rendues accessibles aux utilisateurs au moyen d'un langage dit **langage de commande** ou **shell** qui est interprété par le système.

Le but de la suite de ce document est de présenter un exemple de langage d'assemblage et de son utilisation pour mettre en œuvre les diverses abstractions fournies par les langages Ada ou C. Le langage d'assemblage choisi est l'assembleur Gnu pour les microprocesseurs Intel 80XXX sous Linux, avec XXX au moins au moins égal à 386. Ce document ne peut donc pas se substituer aux

cours ou aux TD dont il peut tout juste être un faible support. Il ne peut pas non plus se substituer aux 3 manuels d'Intel décrivant les processeurs de la famille 80XXX.

Comme les langages d'assemblage sont très liés à l'architecture des processeurs pour lesquels ils génèrent du code, nous commençons, au chapitre 2, par présenter l'architecture de la famille des Intel 80XXX qui peut être utilisée par un utilisateur Linux. En effet, beaucoup des fonctions fournies par les processeurs Intel 80XXX sont inutilisables par un utilisateur sous Linux. Les chapitre 4 et 5 sont consacrés au sous-ensemble du langage d'assemblage Gnu que nous utilisons. Les mécanismes qui permettent aux processeurs Intel 80XXX d'accéder à la mémoire seront étudiés dans le chapitre 6. Dans le chapitre 7, nous nous intéressons aux problèmes de codage en mémoire des données, le codage des instructions étant quant à lui abordé dans le chapitre 11. Le chapitre 8 présentent la mise en œuvre des appels de fonctions et de sous programmes en assembleur. Le chapitre 9 montrent la mise en œuvre des entrées-sorties à l'aide des fonctions C *printf* et *scanf*. Les chapitres 3 et 10 s'intéressent respectivement à la traduction en assembleur des différentes structures de contrôle et de données disponibles en C et en Ada. Le chapitre 12 présente comment assembler et mettre au point un programme écrit en assembleur. Le dernier chapitre enfin donne la bibliographie des principaux ouvrages référencés.

2. L'architecture des microprocesseurs Intel après le 80386

Nous nous limitons volontairement à la description du sous ensemble des microprocesseurs les plus récents de la famille Intel 80X86. Cette famille a commencé son existence avec les 8086/8088 (machines à mots de 16 bits capables d'adresser 1Mo de mémoire) pour ensuite passer aux 80186/80188 puis au 80186 suivi des 80286 avant de s'enrichir des microprocesseurs que nous allons sommairement décrire et qui comprennent, outre le 80386, les microprocesseurs 80486, Pentium et Pentium Pro. Comme très souvent quand il s'agit de composants informatiques, la complexité de l'architecture s'explique en partie par le fait que leurs concepteurs ont voulu garder une compatibilité « complète » entre les microprocesseurs de toute la gamme et également parce qu'ils ont voulu fournir des fonctions de plus en plus complexes.

La présentation qui suit est volontairement simplifiée et passe sous silence toutes les fonctions du microprocesseur qui ne sont pas accessibles par un utilisateur développant et exécutant ses programmes sous Linux. C'est dire qu'elle ne peut pas dispenser de la consultation des documents originaux [Brey 1997] [Intel 1999] qui ont servi à sa rédaction et dont les références sont récapitulées dans la section 13 Bibliographie.

2.1. Le modèle de programmation fourni

Les instructions exécutées et les données qu'elles manipulent sont dans deux espaces mémoire séparés : mémoire (de) données et mémoire (de) programme dont chacun peut être considéré comme un tableau de 4 Giga-octets dont l'index va de la valeur 0 à la valeur $(2^{32} – 1)$[1].

Quoiqu'il n'y ait pas de contrainte d'alignement des données ou des instructions dans leur mémoire respective, il est recommandé, pour des raisons d'efficacité des accès, de les aligner sur des frontières de mot, double mot ou même quadruple mot chaque fois que cela est possible.

Les données accessibles dans la mémoire de données (donc que l'on peut adresser) sont les octets, les mots de 16 bits et les doubles mots de 32 bits. L'adresse d'un mot ou d'un double mot est

[1] L'index, appelé adresse, est codé en entier non signé sur 32 bits, voir 6.1.6.

celle de son premier octet qui est, comme nous le verrons au chapitre 7, son octet de poids faible. La figure 1 représente la valeur hexadécimale des quatre octets rangés à partir de l'adresse hexadécimale 0x804943c de la mémoire de données. Le mot à l'adresse 0x804943c vaut 0xbbaa et le double mot à la même adresse 0xddccbbaa

0x804943c	0xaa	0xbb	0xcc	0xdd

Figure 1 : Adressage de la mémoire de données

Les instructions de la mémoire programme sont de taille variable, allant de 1 à 13 octets. A la fin d'une instruction, le registre EIP (*Extended Instruction Pointer*) contient l'adresse de l'instruction suivante.

En plus de ces deux espaces mémoire, le microprocesseur qui exécute le programme accède à des mémoires appelées registres, que nous décrivons dans la sous section suivante (2.2). Les différents modes que peut utiliser le processeur pour désigner les opérandes de l'opération qu'il exécute font l'objet du chapitre 6.

2.2. Les registres

Les microprocesseurs Intel qui nous intéressent fournissent des registres de 8, 16 et 32 bits (Figure 2). Dans les instructions des microprocesseurs qui les utilisent explicitement, les registres de 8 bits sont dénotés respectivement AH, AL, BH, BL, CH, CL, DH, DL ; ceux de 16 bits : AX, BX, CX, DX, DI, SI et FLAGS et enfin ceux de 32 bits EAX, EBX, ECX, EDX, EBP, ESP, EDI, ESI, EIP et EFLAGS.

<table>
<tr><td>Notation</td><td colspan="2">31 16 15 0</td><td>Nom usuel</td></tr>
<tr><td rowspan="2">EAX</td><td rowspan="2"></td><td colspan="2">AX</td><td rowspan="2">Accumulator</td></tr>
<tr><td>AH</td><td>AL</td></tr>
<tr><td rowspan="2">EBX</td><td rowspan="2"></td><td colspan="2">BX</td><td rowspan="2">Base Index</td></tr>
<tr><td>BH</td><td>BL</td></tr>
<tr><td rowspan="2">ECX</td><td rowspan="2"></td><td colspan="2">CX</td><td rowspan="2">Count</td></tr>
<tr><td>CH</td><td>CL</td></tr>
<tr><td rowspan="2">EDX</td><td rowspan="2"></td><td colspan="2">DX</td><td rowspan="2">Data</td></tr>
<tr><td>DH</td><td>DL</td></tr>
<tr><td>ESP</td><td colspan="3"></td><td>Stack Pt.</td></tr>
<tr><td>EBP</td><td colspan="3"></td><td>Base Pt.</td></tr>
<tr><td>EDI</td><td></td><td colspan="2">DI</td><td>Dest. Index</td></tr>
<tr><td>ESI</td><td></td><td colspan="2">SI</td><td>Source Index</td></tr>
<tr><td>EIP</td><td colspan="3"></td><td>Instruction Pt.</td></tr>
<tr><td>EFLAGS</td><td></td><td colspan="2">FLAGS</td><td>Flag</td></tr>
</table>

Figure 2 : Liste et désignation des registres

Certains de ces registres : EAX, EBX, ECX, EDX, EBP, EDI et ESI ont des fonctions multiples et les autres sont spécialisés. Nous présentons rapidement les fonctions de ces différents registres :

- **EAX**. C'est l'accumulateur. Il peut être référencé comme un registre 32 bits, comme un registre de 16 bits (AX) ou comme deux registres de 8 bits (AH et AL). Si on utilise le registre de 16 bits ou l'un des deux registres de 8 bits seule la portion désignée sera éventuellement modifiée, la partie restante des 32 bits ne sera donc pas concernée par l'opération. Pour certaines instructions, les instructions arithmétiques notamment, l'accumulateur intervient comme un registre spécialisé. Mais dans beaucoup d'autres instructions il peut être utilisé pour contenir l'adresse d'un emplacement en mémoire.
- **EBX**. C'est le registre d'index (cf. 6.1.6 Adressage indirect avec base, déplacement et index). Il est utilisable comme EBX, BX, BH et BL. EBX peut contenir l'adresse d'un emplacement en mémoire programme, dans certaines instructions comme par exemple l'instruction de saut *jmp*. Il peut également servir à adresser des données en mémoire données.
- **ECX**. C'est le registre qui sert de compteur dans beaucoup d'instructions (les déplacements et les rotations utilisent CL, les boucles utilisent ECX ou CX). Il peut être également utilisé pour adresser des données en mémoire données.
- **EDX**. C'est un registre de données qui contient une partie du résultat dans les multiplications et une partie du dividende dans les divisions. Il peut également servir à adresser la mémoire données.
- **EBP**. Ce registre contient toujours une adresse en mémoire données pour y transférer des données. EBP est utilisé comme comme base d'adressage des variables locales et des paramètres d'une fonction en cours d'exécution.
- **EDI**. C'est un registre général 16 bits (DI) ou 32 bits (EDI).
- **ESI**. Comme EDI il est utilisable comme registre général 16 bits (SI) ou 32 bits (ESI). Dans les instructions de manipulation de chaînes, il contient l'adresse de la source.
- **EIP**. Ce registre contient, à la fin d'une instruction, l'adresse de l'instruction suivante à exécuter. Il peut être modifié par les instructions de saut ou d'appel et de retour de sous-programme.
- **ESP**. Ce registre est supposé contenir une adresse dans une zone gérée en pile de la mémoire données.
- **EFLAGS**. Ce registre est le registre d'état du microprocesseur. Nous ne décrivons que les indicateurs que nous utiliserons :

31 12	**11**				**7**	**6**						**0**
	O				**S**	**Z**						**C**

Figure 3 : Quelques indicateurs utiles du registre EFLAGS

- **C.** Carry (bit 0 du registre EFLAGS). Il contient la retenue après une addition ou une soustraction.
- **Z.** Zéro (bit 6 du registre EFLAGS). Il contient 1 si le résultat d'une opération arithmétique ou logique est nul et 0 sinon.

- **S.** Signe (bit 7 du registre EFLAGS). Il prend la valeur du bit de signe du résultat après l'exécution d'une opération arithmétique ou logique.
- **O.** Overflow (bit 11 du registre EFLAGS). Il indique un débordement après une opération dont les opérandes sont considérées comme de valeurs signées. Pour des opérations sur des valeurs non signées il est ignoré.

3. Traduction en assembleur des principales structures de contrôle C

Le but de cette section est de donner une méthode de traduction systématique des schémas classiques de contrôle que l'on rencontre dans tout langage de programmation, et avec lequel tout programmeur est plus ou moins familier. Les méthodes proposées, à défaut de conduire à une programmation toujours très efficace, permettent d'obtenir un programme correct. Il est conseillé dans un premier temps de suivre les schémas proposés, puis de supprimer les instructions inutiles (instructions de saut vers des instructions de saut). Avec de l'expérience, vous trouverez directement la solution améliorée.

La syntaxe choisie pour exprimer les structures de contrôle à traduire est celle du langage C [Cassagne 98]. Comme nous ne supposons pas que le lecteur est familier avec la sémantique associée à ce langage, nous rappellerons la sémantique associée à chacune des instructions que nous traduisons.

Les instructions *jmp*, *je* et *jne* représentent respectivement un saut inconditionnel, un saut à la condition opérande 1 = opérande 2 (de l'instruction précédente), et un saut à la condition opérande 1 != opérande 2 (voir section 7).

3.1. Conditionnelles

Il y a une conditionnelle classique qui peut se décliner dans sa version incomplète (***si*** *condition* ***alors*** *instruction*) ou complète (***si*** *condition* ***alors*** *instruction* ***sinon*** *instruction*)

3.1.1. Simple : if (expression) instruction

La valeur d'*expression* est calculée et si elle est différente de zéro *instructions* est exécutée. En assembleur on aura donc :

```
        <calcul de la valeur d'expression avec résultat dans %eax>
        cmpl $0,%eax
        je   finsi
        <instructions>
finsi:
```

3.1.2. Complète : if (expression) instruction1 else instruction2

La valeur d'*expression* est calculée et si elle est différente de zéro *instruction1* est exécutée si non c'est *instruction2*. En assembleur on aura donc :

```
        <calcul de la valeur d'expression avec résultat dans %eax>
        cmpl $0,%eax
        je else
        <instruction1>
        jmp finsi
```

```
else:    <instruction2>
finsi:
```

3.2. Itérations

3.2.1. Boucle : for (expression$_1$; expression$_2$; expression$_3$) instruction ;

La valeur de *expression$_1$* est calculée (en fait *expression$_1$* peut être composée d'une suite d'expressions séparées par des virgules et qui doivent alors être calculée en séquence), puis *expression$_2$* est évaluée. Si le résultat de l'évaluation d'*expression$_2$* est différent de zéro alors *instruction* est exécutée suivi du calcul d'*expression$_3$* et on recommence la suite d'opérations depuis l'évaluation d'*expression$_2$*. On peut coder cette suite d'opérations de deux façons différentes. La première en traduisant littéralement ce que nous venons de décrire et la seconde en utilisant l'instruction spécialisée *loop* du processeur qui permet d'évaluer efficacement des boucles du type :

```
for (i = 0 ; i < 100 ; i++) <instruction>
```

L'instruction *loop* ôte 1 au contenu du registre ECX et effectue un branchement à l'adresse fournie en opérande si le registre ECX est différent de zéro. Il existe des formes conditionnelles de l'instruction *loop* qui permettent de combiner le fait que ECX est différent de zéro après qu'on ait diminué de 1 sa valeur avec le fait qu'une comparaison ait donné le résultat égal (*loope*) ou non égal (*loopne*).

Traduction littérale

```
          <calcul d'expression1>
iterf:    <évaluation d'expression2 avec mise du résultat dans %eax>
          cmpl $0,%eax
          je finfor
          <instruction>
          <évaluation d'expression3>
          jmp iterf
finfor:   …
```

Traduction utilisant l'instruction *loop* de « **for** (i = 100 ; i > 0 ; i--) <instruction> »

```
          movl $100, %ecx
iterf:    <instruction>
          loop iterf
          …
```

3.2.2. Tant que faire : while (expression) instruction

La valeur courante de *expression* est calculée et si elle est non nulle alors instruction est évaluée. Cette suite d'instructions est répétée tant qu'*expression* a une valeur non nulle. En assembleur on aura donc :

```
while:    <calcul de la valeur d'expression dans %eax>
          cmpl $0,%eax
          je finwhile
whcorps:  <instruction>
          jmp while
finwhile: …
```

3.2.3. Faire tant que : do instruction while (expression)

Cette instruction est identique à la précédente sauf que les instructions de la boucle sont effectuées au moins une fois.

```
whcorps:   <instruction>
cdeval:    <calcul de la valeur d'expression dans %eax>
           cmpl $0,%eax
           jne whcorps
           …
```

4. Syntaxe de l'assembleur

La syntaxe qui est présentée ici est volontairement moins permissive que celle de l'assembleur *Gnu*. Mais nous espérons que les restrictions apportées vous éviteront de commettre des erreurs difficiles à détecter.

Un programme se présente comme une liste d'unités, une unité tenant sur une seule ligne. Il est possible (et même recommandé pour aérer le texte) de rajouter des lignes blanches. Il y a trois sortes de lignes que nous allons décrire maintenant.

4.1. Les commentaires

Un commentaire, comme dans le langage C, peut commencer par /*, se terminer par */ et comporter un nombre quelconque de lignes. Une autre forme de commentaire commence sur une ligne par le caractère # et se termine par la fin de ligne.

Exemple :

```
/* Ceci est un commentaire
pouvant tenir sur plusieurs lignes
*/

# Ceci est un commentaire se terminant a la fin de la ligne
```

4.2. Les instructions machines

Elles ont la forme suivante :

```
[Etiquette] [opération] [opérandes]     [# commentaire]
```

Les champs entre crochets peuvent être omis. Une ligne peut ne comporter qu'un champ étiquette, une opération peut ne pas avoir d'étiquette associée, ni d'opérande, ni de commentaire… Les champs doivent être séparés par des espaces ou des tabulations.

4.2.1. Le champ étiquette

C'est la désignation symbolique d'une adresse de la mémoire de données ou d'instructions qui peut servir d'opérande à une instruction ou à une directive de l'assembleur. Une étiquette est une suite de caractères alphanumériques[2] commençant par une lettre et terminée par le caractère « : ». On appelle nom de l'étiquette la chaîne de caractères alphanumériques située à gauche du caractère

[2] Aux lettres de l'alphabet majuscules et minuscules et aux chiffres décimaux sont ajoutés les trois caractères « $. _ ».

« : ». Il est déconseillé de prendre pour nom d'étiquette un nom de registre, d'instruction ou de directive.

Plusieurs étiquettes peuvent être associées à la même opération ou à la même directive.

Une étiquette ne peut être définie qu'une seule fois dans une unité de compilation. Sa valeur est relative, comme nous le verrons plus loin, à la section dans laquelle elle est définie (code données) et est égale à son adresse d'implantation dans la mémoire correspondante (données ou programme). La valeur d'une étiquette ne peut donc pas être, en général, connue avant le chargement.

Exemple :

```
Lab1 : $lab2 :
_Lab3 : $Lab2 :
       movl $0,%eax # les quatre étiquettes repèrent l'instruction
```

4.2.2. Le champ opération

Pour les opérations pouvant opérer sur plusieurs tailles d'opérandes, le champ opération est formé du nom de l'opération suivi immédiatement d'une des lettres *l*, *w* ou *b* selon que l'opération porte sur un double mot (32 bits), un mot (16 bits) ou un octet. Pour les autres opérations, le champ opération porte le nom donné par le constructeur sans suffixe.

Pour les opérations ayant deux opérandes, l'un des deux opérandes est un registre et l'autre peut être désigné par l'un des modes d'adressage fourni par le microprocesseur. Pour chaque opération le constructeur spécifie les modes d'adressage qui sont permis pour ses opérandes.

En assembleur :

- Les registres sont désignés par leur nom précédé du caractère %. Par exemple %EAX, %AX, %AH, %AL pour les quatre registres EAX, AX, AH, AL.
- Un opérande immédiat est toujours noté en assembleur par sa valeur précédée du caractère $. Exemple :

```
movl $0,%eax
.set DEUX,2                # DEUX est une constante valant 2
movb $DEUX,%ah
```

- Adresse directe de la mémoire de données : On donne soit le nom de l'étiquette qui repère la donnée soit la valeur de l'adresse. Exemple :

```
.data
xint .int 1235
.text
.set DEUX,2               # DEUX est une constante valant 2
movl xint, %eax           # eax = xint
movb DEUX, %bl            # bl = contenu de l'octet 2 de la mem de donnees
```

- Indirect registre de la mémoire de données : (%R) ou R est le nom de l'un des registres 32 bits permis.
- Indirect registre de la mémoire programme : *%R où R est le nom de l'un des registres permis.
- Indirect avec déplacement de la mémoire données : D(%R) où R est le nom de l'un des registres 32 bits permis et D la valeur du déplacement.

- Indirect mémoire programme, avec adresse directe du relai : On le note *adIci , si adIci est une étiquette définie dans la mémoire données, repérant un élément de la mémoire programme.
- Indirect avec déplacement de la mémoire d'instructions : *D(%R) où R est le nom de l'un des registres 32 bits permis et D la valeur du déplacement. Si le déplacement est nul on peut écrire *(%R).
- Indirect avec déplacement et index de la mémoire de données : D(%R1,%R2) où R1 et R2 sont les noms de registres permis et D la valeur du déplacement.
- Indirect avec index typé de la mémoire de données : D(%R1,%R2, F) où R1 et R2 sont les noms de registres permis, D le déplacement et F est le facteur multiplicatif à appliquer au contenu de R2. Les seules valeurs permises pour F sont 1, 2, 4 ou 8.

Les déplacements ou les valeurs immédiates peuvent être des expressions arithmétiques simples faisant intervenir les opérateurs arithmétiques : *, /, %,+, -, cités dans leur ordre de priorité décroissante. Le caractère / désigne la division entière et le caractère % le reste dans la division entière. L'expression ne doit pas utiliser de parenthèse. L'expression ne doit pas faire intervenir de symbole non défini dans l'unité de compilation courante.

Un déplacement peut être le résultat du calcul de la différence de deux adresses symboliques *(etiqu1 – etiqu2)* est une valeur entière signée qui ne dépend ni de l'adresse d'implantation du code, ni de celle des données et qui peut, donc, être calculée à l'assemblage. En revanche, toute expression du type : *etiq* + *n* où *n* est une constante, dépend de l'adresse d'implantation du segment de code ou de données qui contient l'étiquette *etiq* elle représente donc une adresse.

4.3. Les directives d'assemblages

Les directives d'assemblage ou plus simplement les directives sont des ordres donnés à l'assembleur et ne sont donc pas des instructions machines. Une directive est toujours précédée d'un caractère « . » (point). Nous décrivons les principale directives disponibles dans la section suivante.

5. Les Principales directives d'assemblage

Il y a trois familles de directives : les directives de sectionnement du programme, les directives de définition de données ou de constantes et les autres directives. Après avoir décrit ces trois familles nous donnons l'exemple d'un programme complet calculant le PGCD de deux nombres écrit en assembleur.

5.1. Les directives de sectionnement

Nous avons vu que les microprocesseurs auxquels nous nous intéressons adressent deux mémoires séparées : la mémoire données et la mémoire programme. Nous verrons, quand nous traiterons la définition des données, que l'on peut encore distinguer deux types de données : celles qui ont une valeur initiale définie statiquement par le programmeur et celles pour lesquelles la valeur ne sera définie qu'à l'exécution.

5.1.1. .text [sous-section]

Cette directive dit à l'assembleur d'assembler les instructions qui suivent à la suite des instructions de la sous section d'instructions numérotée *sous-section*. Si le numéro *sous-section* est omis il est par défaut égal à zéro.

5.1.2. .data [sous_section]

Cette directive indique à l'assembleur d'assembler les unités de programme suivante derrière les données de la sous-section de données numérotée *sous-section*. Si le numéro de sous-section est omis alors le nombre 0 est pris par défaut.

5.1.3. .section .rodata

Cette directive indique à l'assembleur d'assembler les données suivantes qui seront placées dans une sous section spéciale qui à l'exécution sera en lecture seule (*read only memory*) et non exécutable.

5.1.4. .bss

La section bss est une section particulière qui n'est effectivement créée que lors du chargement en mémoire du programme, et n'est donc pas incluse dans le fichier objet généré par le compilateur. Elle ne peut donc contenir aucune donnée initialisée, et sera remplie avec des 0 avant exécution du programme. On s'attend donc à trouver dans cette section uniquement des instructions de type *.skip* (voir plus bas) et des définitions de symboles.

5.2. Les directives de définition de données

On distingue deux type de données, les données initialisées des données non initialisées. La raison de ceci est que les données non initialisées, placées dans la section .bss, ne seront générées que lors du chargement du programme en mémoire, et ne sont pas stoquées dans les fichiers objets. Cela réduit donc la taille des fichiers générés par l'assembleur.

5.2.1. Déclaration des données non initialisées :

- **[étiquette] .skip taille (,valeur) :** La directive *.skip* permet de réserver un nombre d'octets égal à *taille* à l'adresse *étiquette*. La zone est réservée dans la section courante, à l'emplacement de la directive. On utilisera essentiellement la directive .skip dans une section .bss pour définir des données non initialisées, mais il est cependant possible de le faire dans n'importe quelle autre section, auquel cas la place réservée sera effectivement incluse dans le fichier objet.
- **.lcomm nom, taille (,nombre) :** La directive *lcomm* permet de réserver un nombre d'octets égal à *taille* ; l'octet 0 est à l'adresse *nom*. La syntaxe de *nom* est celle permise pour les noms d'étiquettes (cf. la section précédente). *.lcomm* peut apparaître n'importe où dans le code, mais l'emplacement réservé est automatiquement placé dans une section bss, quelque soit la section courante. Le paramètre *nombre,* facultatif, permet de réserver *nombre* zones mémoire de taille *taille.*

5.2.2. Déclaration de données initialisées

L'assembleur permet de déclarer divers types de données initialisées : des octets, des mots (16 bits), des doubles mots (32 bits), des quadruples mots (64 bits), voire des octuples mots (128 bits) et des chaînes de caractères. On rappelle qu'il toujours plus efficace d'aligner les données sur des frontières au moins multiples de leur taille.

- **[étiquette] .byte expressions** où *expressions* est une suite comprenant 0 ou plusieurs expressions séparées par des virgules. Chaque expression est assemblée dans l'octet suivant

celui où on a rangé la valeur de l'expression précédente. Exemple : la ligne suivante permet de réserver un tableau de 4 octets avec les valeurs initiales 42, 4, 8 et –1. Le premier élément de ce tableau est à l'adresse *Tabb* de la mémoire de données.

```
Tabb: .byte 6*7, 4, 8, 0xff
```

- **[étiquette] .hword expressions / [étiquette] .short expressions** où *expressions* est une suite comprenant 0 ou plusieurs expressions séparées par des virgules. Les expressions sont assemblées dans des mots de 16 bits consécutifs. Exemple : la ligne suivante permet de réserver un tableau de 4 mots de 16 bits avec les valeurs initiales 32767, -32768, 0 et –1. Le premier élément de ce tableau est à l'adresse *Tabw* de la mémoire de données.

```
Tabw: .hword 0x7fff, 0x8000, 0, 0xffff
```

- **[étiquette] .int expressions / [étiquette] .long expressions** où *expressions* est une suite comprenant 0 ou plusieurs expressions séparées par des virgules. Les expressions sont assemblées dans des doubles mots de 32 bits consécutifs[3]. Exemple : la ligne suivante permet de réserver un tableau de 4 doubles mots et de les initialiser avec les adresses des instructions f1, f2, f3 et f4. Le premier élément de ce tableau est à l'adresse *Tabf* de la mémoire de données.

```
Tabf: .int f1, f2, f3, f4
```

- **[étiquette] .quad bnums** où *bnums* est une suite comprenant 0 ou plusieurs grands nombres séparés par des virgules. Si les grands nombres (entiers relatifs non représentables sur 32 bits) ne sont pas représentables sur 8 octets ils sont tronqués et on ne conserve que les 8 octets de poids faible. Un message de mise en garde est émis par l'assembleur. Exemple : la ligne suivante définit un tableau de deux grands entiers dont le premier est à l'adresse *Gde* et le second à l'adresse *Gde+8*.

```
Gde: .quad 0x100000000, 0x200000000
```

- **[étiquette] .octa bnums** où *bnums* est une suite comprenant 0 ou plusieurs grands nombres séparés par des virgules. Ces nombres seront rangés sur 16 octets.
- **[étiquette] .string str / [étiquette] .ascii str** où *str* est une suite de caractère entre des caractères « " » (double quote). Comme en C, pour inclure des caractères spéciaux dans la chaîne (en particulier, le caractère double quote, mais aussi le retour chariot, le « new line », le « form feed » etc.), on les fait précéder du caractère « \ ». Ainsi « new line » se note \n. Dans le cas de *.string*, l'assembleur recopie ces caractères et termine la suite de caractères copiée par un octet à 0, réalisant ainsi une chaîne au sens du langage C. Pour obtenir une chaîne exempte de cet octet supplémentaire, on utilisera la directive *.ascii*. Exemple : la ligne suivante permet de réserver une chaîne de caractères désignée par *format*.

```
format: .string "Format printf de sortie d'entiers %08d\n"
```

5.3. Diverses autres directives

Nous présentons dans cette sous-section diverses autres directives d'assemblage permettant de définir des constantes, de forcer l'alignement de données sur des frontières particulières afin

[3] Sur d'autres architectures, la taille des objets réservés par les directives .int et .long peut varier. Il existe également une directive .word qui, dans le cas des processeurs étudiés est équivalente à .hword et .short (deux octets), mais dont la définition peut changer sur d'autres processeurs.

d'améliorer le temps d'accès à ces données, d'exporter des données vers des modules compilés ou assemblés séparément.

5.3.1. Définition de constantes

On peut utiliser deux directives :

- **.equ symbole, expression :** cette primitive affecte à *symbole* la valeur de l'expression qui doit pouvoir être évaluée à la première passe de l'assembleur ; la syntaxe de *symbole* est celle d'un nom d'étiquette telle que nous l'avons défini plus haut. Le nom défini par *symbole* ne doit pas avoir été déjà défini.
- **.set symbole, expression :** cette primitive est équivalente à la précédente, sauf qu'un symbole défini par la directive set peut voir sa valeur redéfinie. La nouvelle définition d'un symbole n'affecte que les unités de programme qui la suive.

Exemple :

```
.equ DEUX, 2 # Les constantes sont écrites en majuscule
.equ N,100
.set NIVEAU,1
…
.set NIVEAU, NIVEAU+1      # niveau vaut 2
```

5.3.2. Alignement de données

Comme nous le disions plus haut les directives d'alignement sont importantes car du bon alignement d'une donnée (mot sur une frontière de mot, double mot sur une frontière de double mot, etc.) dépend la vitesse avec laquelle le processeur pourra y avoir accès.

- **.p2align e1[, e2, e3] :** les opérandes, dont les deux derniers sont optionnels, sont des expressions absolues (dont la valeur ne dépend pas de l'implantation du programme). La première expression *e1* donne l'alignement fixé c'est à dire le nombre de bits de poids faible du compteur d'assemblage associé à la section courante qui doivent être à zéro après l'alignement. La seconde expression donne la valeur qui doit être affectée aux octets qui sont sautés pour atteindre la frontière demandée. Par défaut si on est dans une section de programme *e2* vaudra le code de l'opération *nop*, et si on est dans une section de données il vaudra 0. La dernière expression *e3* indique le nombre maximum d'octets que l'on accepte d'ajouter pour atteindre la frontière fixée par e1. Si la valeur fixée par e3 devait être dépassée alors l'opération d'alignement n'est pas effectuée.

5.3.3. Exportation de variables

Par défaut, tout nom symbolique utilisé dans une unité de compilation, et non défini dans cette unité (c.a.d. n'intervenant ni comme un nom de constante, ni comme un nom d'étiquette), est considéré comme externe. L'importation d'entités définies dans d'autres unités d'assemblage est donc implicite. En revanche, pour exporter vers d'autres unités de compilation des noms de constantes, de variables, ou des sous-programmes définis localement à l'unité courante, on doit le faire explicitement à l'aide de la directive suivante :

- **.global symbole :** cette directive rend le nom défini par *symbole* accessible à l'éditeur de liens et donc utilisable par d'autres modules de compilation ou d'assemblage. En fait *symbole* peut être une liste de noms de constantes ou d'étiquettes séparés par des virgules.

5.3.4. Compteur d'assemblage

Le compteur d'assemblage de la section courante peut toujours être désigné par le caractère « . ». Les opération du type : « . = . + 4 » sont tout à fait permises et reviennent à sauter 4 octets dans la section courante où cette opération est rencontrée. On peut également définir :

```
adici: .int .
```

adici est une variable dont la valeur initiale est sa propre adresse.

5.4. Un exemple simple : le pgcd

```
/* Ce programme calcule le pgcd entre les variables entières x et y
représentées sur 16 bits */

        .section .rodata # format d'impression pour printf
format: .string "Valeur de x = %d et de y = %d\n"

        .data
x:      .hword 256
y:      .hword 4096
        .text
        .p2align 2              # force l'alignement à une frontière de double mot
        .global main            # exporte le nom main point d'entrée du programme
main:   pushl %ebp              # Convention de liaison avec l'appelant
        movl %esp,%ebp
        cmpl $0,x               # if ((x==0)||(y==0)) aller imprimer x et y
        je fin
        cmpl $0,y
        je fin
        .p2align 4,,7           # force l'alignement sur une frontière de 16
iter:   movw x,%ax              # if (x != y)
        cmpw y,%ax
        jne cont                # On continue
        jmp fin                 # si non on va imprimer x et y
        .p2align 4,,7
cont:   movw x,%ax              #if (x < y)
        cmpw y,%ax
        jle .LL                 # aller calculer  y = y –x
        movw y,%ax              # sinon calculer x = x – y
        subw %ax,x
        jmp iter
        .p2align 4,,7
.LL:    movw x,%ax              #calcul de y = y-x
        subw %ax,y
        jmp iter
        .p2align 4,,7
fin:                            #impression de x et y
        xorl %eax,%eax
        pushl y
```

```
pushl x
pushl $format
call printf
addl $12,%esp
leave                   # on rend le contrôle à l'appelant
ret
```

6. Les modes d'adressages

La plupart des instructions fournies par les microprocesseurs auxquels nous nous intéressons opèrent entre deux opérandes, dont l'un est contenu dans un registre et l'autre dans un registre ou dans la mémoire (données ou programme). On appelle mode d'adressage d'un opérande, la méthode qu'utilise le processeur pour calculer l'adresse de l'opérande. S'il s'agit d'un opérande qui est un registre, il ne s'agit pas à proprement parler d'adresse en mémoire mais de nom. S'il s'agit d'un opérande en mémoire, l'adresse sera la valeur de l'index auquel se trouve cet opérande dans le tableau correspondant (cf. sous section 2.1). En fait, les modes de calcul de l'adresse d'un opérande sont différents, selon que cet opérande appartient à la mémoire données ou programme. Nous allons donc présenter séparément les modes d'adressages disponibles pour les deux sortes de mémoires.

6.1. Les modes d'adressage de la mémoire (de) données

Pour illustrer la présentation des modes d'adressage de la mémoire de données nous allons utiliser les trois instructions : *movb*, *movw*, *movl* qui permettent de copier respectivement l'octet, le mot ou le double mot de l'opérande source, le premier spécifié dans l'instruction assembleur, dans respectivement l'octet, le mot ou le double mot de l'opérande destination. Cette instruction ne permet pas aux deux opérandes d'être dans la mémoire de données, donc l'un des opérandes est soit un registre soit une valeur immédiate. Nous allons nous intéresser à toutes les façons de calculer soit la source soit la destination. On trouvera en annexe un exemple de programme qui teste tous les modes d'adressage de la mémoire données qui font l'objet de cette sous section.

6.1.1. Adressage registre direct

Dans ce mode, la valeur de l'opérande est dans un registre et l'opérande est désigné par le nom du registre concerné qui peut-être l'un des noms de registres donnés dans la section précédente. Selon que l'opération porte sur un octet, un mot de 16 bits ou un long sur 32 bits, on doit utiliser une portion de registres de 8, 16 ou 32 bits avec le nom correspondant.

Dans le cas d'une opération où un/les opérande/s est/sont adressé/s dans le mode registre direct, le/s registre/s devra/devront avoir la même taille que celle spécifiée par l'instruction (*b* pour octet, *w* pour un mot de 16 bits et *l* pour un double mot).

En assembleur Gnu, les registres sont désignés dans les instructions par leur nom précédé du caractère %. D'un point de vue syntaxique, dans l'écriture d'une instruction mettant en jeu une source et une destination la source précède toujours la destination (*OP source, destination*).

Exemple :

```
movl %esp, %ebp     /* Copie le contenu de esp dans ebp */
movw %ax, %bx       /* Copie le contenu du registre ax dans le registre bx */
movb %al, %ah       /* Copie le registre al dans le registre ah */
```

6.1.2. Adressage immédiat

Dans ce mode, c'est la valeur de l'opérande qui est directement fournie dans l'instruction[4]. Une valeur immédiate est toujours précédée en assembleur Gnu du caractère « $ ». La valeur immédiate peut être fournie sous forme symbolique, sous forme d'un nombre décimal signé, ou sous forme d'une suite de chiffres hexadécimaux précédés des deux caractères 0x.

Exemple :

```
movb $0xff, %al     /* al := -1 */
movw $0xffff, %ax   /* ax := -1 */
movl $-1, %eax      /* eax := -1 */
movl $toto, %eax    /* eax := valeur du symbole toto  */
```

Ce dernier exemple est intéressant car *toto* peut être, selon la façon dont il est défini dans le programme, une constante, une adresse dans les données ou dans le code. Dans le cas où *toto* est une constante sa valeur est connue à l'assemblage ; si non, elle n'est connue qu'au chargement.

Attention : N'oubliez pas le caractère **$** devant l'opérande immédiat sous peine de le voir mal interprété, comme un opérande adressé directement.

6.1.3. Adressage direct

Dans ce mode, c'est l'adresse de l'opérande dans la mémoire données qui est fournie dans le champ opérande de l'instruction.

Exemple : nous supposons qu'à partir de l'adresse hexadécimale 0x804943c de la mémoire données on a rangé les quatre octets : 0xaa, 0xbb, 0xcc et 0xdd. Donc, comme nous l'avons déjà vu plus haut, l'octet à l'adresse 0x804943c est 0xaa, le mot à l'adresse 0x804943c est 0xbbaa et le double mot à l'adresse 0x804943c est le double mot 0xddccbbaa. Ce qu'on vérifie en exécutant les instructions suivantes et en visualisant le résultat avec un metteur au point.

```
movb 0x804943c,%al    /* affecte 0xaa au registre octet al */
movw 0x804943c,%ax    /* affecte 0xbbaa au registre mot ax */
movl 0x804943c,%eax   /* affecte 0xddccbbaa au registre eax */
```

6.1.4. Adressage indirect registre

Dans ce mode l'adresse de l'opérande est contenue dans le registre. La syntaxe Gnu de ce mode d'adressage est : **(%REGISTER)** où REGISTER est l'un quelconque des registres EAX, EBX, ECX, EDX, EDI et ESI. Les registres ESP et EBP sont réservés en principe pour l'adressage de la pile ainsi que nous le verrons dans la section 8 (Conventions de liaison entre programmes et sous-programme). Cependant rien, pas même l'assembleur, n'interdit de les utiliser comme des registres banalisés.

Exemple : après l'instruction « `movl $0x804943c, %ebx` » qui place 0x804943c dans le registre EBX, les instructions « `movb (%ebx),%al` », « `movw (%ebx),%ax` » et « `movl`

[4] L'adresse de l'opérande est donc, dans ce cas, l'adresse de la mémoire programme qui suit immédiatement celle du code de l'instruction.

`(%ebx),%eax` » donnent les mêmes résultats que les trois instructions de l'exemple donné en 6.1.3.

6.1.5. Adressage indirect avec base et déplacement

Dans ce mode, interviennent un registre, appelé *registre de base*, et une constante signée (cf. section 7), appelée *déplacement*. Le registre peut être, comme précédemment, l'un quelconque des registres EAX, EBX, ECX, EDX, EDI et ESI. EBP est également utilisé comme registre de base, mais uniquement pour « baser » des données contenues dans la pile. La syntaxe associée par l'assembleur Gnu à ce mode est : **Dep(%REGISTER)**. Pour calculer l'adresse de l'opérande, le processeur ajoute au contenu du registre de base REGISTER la valeur sur 4 octets du déplacement signé.

Exemple : Après l'exécution de « `movl $0x8049436,%ebx` », `6(%ebx)` repérera l'opérande à l'adresse 0x804943c (0x804943c = 6 + 0x8049436) de la mémoire données et donc « `movb 6(%ebx),%al` », « `movw 6(%ebx),%ax` » et « `movl 6(%ebx),%eax` » ont un effet équivalent aux trois instructions de l'exemple de la sous section 6.1.4.

Ce mode d'adressage facilite en assembleur l'utilisation d'enregistrement (cf. « record » ADA ou « struct » C) puisque chaque champ d'un enregistrement est à un déplacement fixe de son début.

6.1.6. Adressage indirect avec base, déplacement et index

Dans ce mode, interviennent deux registres et une valeur constante signée. La syntaxe assembleur Gnu de ce mode d'adressage est : **Dep(%REGISTER1, %REGISTER2)** où Dep est une constante signée sur 4 octets, REGISTER1 et REGISTER2 sont pris parmi les registres EAX, EBX, ECX, EBP, EDX, EDI et ESI. Comme précédemment, EBP est utilisé de préférence pour adresser des données rangée dans la pile.

L'adresse de l'opérande est obtenue en ajoutant le contenu des deux registres avec le déplacement considéré comme un entier signé sur un double mot.

Ce mode d'adressage permet d'adresser facilement des tableaux qu'ils soient ou non inclus dans des enregistrements. Pour le réaliser il suffit de prendre pour *Dep* le déplacement du premier élément du tableau dans la structure, dans REG1 l'adresse de base de la structure et dans REG2 la valeur de l'index de l'élément auquel on veut accéder multiplié par la taille d'un élément de tableau.

Exemple : *str* est une structure qui, à un déplacement de TAB, contient un tableau de mots dont le premier élément est à l'index 0 et dont on veut lire dans le registre BX la valeur du 5eme élément.

```
movl $str, %eax            /* eax = adresse de base de la structure str */
movl $8,%ecx               /* ecx = valeur de l'index de l'élément à accéder */
movw TAB(%eax,%ecx),%bx    /* bx prend la valeur du 5eme element de TAB */
```

6.1.7. Adressage avec base, déplacement et index typé

Dans ce mode, interviennent deux registres REGISTER1 et REGISTER2 pris parmi EAX, EBX, ECX, EDX, EDI, EBP et ESI, un déplacement Dep qui est une constante signée sur un double mot et un facteur multiplicatif de l'index MUL qui peut valoir 1, 2, 4 ou 8. La syntaxe assembleur de ce mode d'adressage est : **Dep(%REGISTER1, %REGISTER2, MUL)**.

L'adresse de l'opérande est obtenue en ajoutant au déplacement signé le contenu du registre REGISTER1 et le produit du contenu de REGISTER2 par MUL. Pour des tailles d'élément de

tableau valant 1, 2, 4 ou 8 ce mode d'adressage permet de simplifier le calcul de l'adresse des éléments du tableau. Ainsi avec ce mode d'adressage l'exemple précédent (6.1.7) s'écrit :

```
movl $str, %eax            /* eax = adresse de base de la structure str */
movl $4,%ecx               /* ecx = valeur de l'index de l'élément à accéder */
movw TAB(%eax,%ecx,2),%bx  /* bx prend la valeur du 5eme element de TAB */
```

6.2. Les modes d'adressage de la mémoire programme

Deux types d'instructions ont besoin d'adresser la mémoire programme, les instructions de saut (inconditionnel ou conditionnel) et l'instruction d'appel de sous programme. Nous illustrons les différents mode d'adressage en utilisant l'instruction de saut inconditionnel qui s'appelle *jump* et dont la syntaxe assembleur est **jmp <dest>** où <dest> désigne la façon de calculer l'adresse de l'instruction à exécuter après le saut.

Nous avons choisi *jmp* parce qu'elle est simple et permet comme l'appel de sous programme d'utiliser tous les modes d'adressages de la mémoire d'instructions, ce qui n'est pas le cas des instructions de saut conditionnel. Les modes d'adressage de la mémoire programme sont au nombre de trois : l'adressage direct, l'adressage relatif à la valeur du registre EIP (pointeur d'instruction) et enfin l'adressage indirect qui utilise un relais qui est soit un registre soit un double mot de la mémoire données. Tous ces modes d'adressage sont illustrés par un programme dont le texte est donné en annexe.

6.2.1. Adressage direct

Le champ opérande <*dest*> de l'instruction *jmp* contient l'adresse de branchement donc l'adresse à affecter au registre EIP. Pour que l'assembleur génère un saut avec adressage direct il faut lui donner un champ <dest> numérique égal à la valeur décimale, hexadécimale ou octale de l'adresse. Si on donne une valeur symbolique l'assembleur *Gnu* génére une instruction de saut avec un mode d'adressage relatif au pointeur d'instruction.

Exemple :

```
jmp 0x080483cc        /* eip = 0x080483cc */
```

6.2.2. Adressage relatif au registre EIP

Le champ opérande contient sur 8, 16 ou 32 bits une valeur **signée** représentant le déplacement, positif ou négatif, qu'il faut ajouter au registre EIP pour atteindre l'instruction que l'on veut exécuter après l'exécution de l'instruction de saut. Le champ <dest> est le nom de l'étiquette qui repère dans le code l'instruction que l'on veut exécuter après l'instruction de saut.

Exemple :

```
Ici :  jmp Ici      /* Attention le déplacement est généré sur 8 bits et vaut –2 */
```

6.2.3. Adressage indirect

Il a deux formes :

1. Adressage indirect dont le relais est un registre. En assembleur <dest> s'écrira *%REG (ne pas oublier le caractère * avant %) où REG est l'un quelconque des registres EAX, EBX, ECX, EDX, ESP, EBP, EDI, ESI. L'instruction exécutée après le saut est l'instruction dont l'adresse est contenue dans REG.
2. Adressage indirect dont le relais est un double mot de la mémoire données. La syntaxe utilisée pour <dest> est cette fois de l'un quelconque des types permettant d'adresser la mémoire de donnée précédé d'un caractère '*' :
   - *Dep(%REG) ou *(%REG) ou *DEP(%REG1,%REG2) ou *(%REG1,%REG2) ou *(, %REG) ou *DEP(,%REG)
   - *AdRelais

   REG est pris dans EBX, ECX, EDX, ESP, EBP, EDI, ESI. Le mot de la mémoire de données, soit à l'adresse obtenue en sommant *Dep* avec la valeur courante de *REG1* ou de *REG1* et *REG2*, soit à l'adresse *AdRelais*, contient l'adresse, dans la mémoire programme, de l'instruction à exécuter après le saut. Ce mode d'adressage est évidemment très intéressant pour construire des aiguillages à l'aide de tableaux d'adresses de fonctions.

Exemple :

```
       .data
adIci: .int Ici              /* definition du relais d'indirection vers Ici */
       .text
Ici:   movl $Ici,%eax        /* eax = adresse d'Ici */
       jmp *%eax             /* remonte à Ici ; boucle infinie */
       movl $adIci, %eax     /* eax = adresse relais d'indirection */
       jmp *(%eax)           /* remonte a Ici ; boucle infinie */
       xorl %eax,%eax        /* eax = 0 */
       jmp *adIci(%eax)      /* remonte à Ici ; boucle infinie */
       jmp *adIci            /* remonte à Ici ; boucle inifinie */
```

7. Représentation des entiers sur n bits

Pour les microprocesseurs qui nous intéressent n vaut 8, 16 ou 32. A noter que les entiers du langage C sont codés, sur ce type de machine, sur 32 bits en complément à 2.

7.1. Entiers naturels

C'est la numérotation classique en base 2. Dans toute la suite on utilisera pour ces nombres le terme d'entiers non signés. Avec *n* bits on peut représenter tous les entiers de l'intervalle 0.. $2^n - 1$ (nombre constitué de n bits à 1).

n	intervalle	Valeur max en hexadécimal
8	0 .. 255	FF
16	0 .. 65535	FFFF
32	0.. 4 294 967 295	FFFFFFFF

7.2. Entiers relatifs

Pour les représenter sur n bits on utilise une représentation dite en complément à 2 que l'on peut définir comme suit :

Pour tout x appartenant à l'intervalle $[-2^{n-1}, 2^{n-1} - 1]$ on note *r* sa représentation en complément à 2 et *val(r)* la valeur de cette représentation si on l'interprète comme un entier naturel. Alors on aura :

- Si x est positif ou nul alors r est tel que x = val(r)
- Si x est négatif alors r est tel que $x = val(r) - 2^n \Rightarrow val(r) = 2^n - (-x)$

Exemple :

n	intervalle	Val min en hexa	Val max en hexa
8	-128 .. 127	80	7F
16	-32 768 .. 32767	8000	7FFF
32	-2 147 483 648 .. 2 147 483 647	80000000	7FFFFFFF

7.3. Remarques

- Avec la représentation en complément à deux, la représentation de –1 est toujours un nombre dont tous les bits sont égaux à 1.
- Pour passer d'une représentation d'un nombre sur n bits à sa représentation sur m bits (m > n), il suffit de recopier le bit de poids fort de sa représentation sur n bits dans les bits ajoutés. Voir l'instruction MOVSX (MOVe and Sign eXtend) qui permet de copier un octet dans un mot de 16 bits ou un mot de 16 bits dans un mot de 32 bits en étendant son signe. Par exemple –127 est représenté en hexadécimal sur 8 bits par 81 et sur 16 bits par FF81.
- L'opération inverse de la précédente (troncature) n'est possible que si le mot est représentable sur n bits. On doit vérifier que tous les bits éliminés sont identiques au bit de poids fort restant.

Par exemple, -129 est représenté sur 16 bits par FF7F et n'est pas représentable sur 8 bits car 7F est la représentation de 127.

7.4. Représentation des entiers en mémoire

7.4.1. Entier sur un octet

Comme l'indique la figure suivante, les bits sont numérotés, par convention, de la droite vers la gauche du poids faible vers le poids fort. Ceci est toujours vrai quelle que soit la position de l'octet dans un mot, un double mot ou un registre.

7	6	5	4	3	2	1	0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Figure 4 : Poids des bits d'un octet

7.4.2. Entier sur un mot

L'adresse du mot est celle de l'octet de poids faible qui précède l'octet de poids fort ainsi que l'illustre la figure ci-dessous :

Octet 0 : Poids faible								0ctet 1 : Poids fort							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8

Figure 5 : Représentation d'un entier sur un mot de 16 bits

7.4.3. Entier sur un double mot

L'adresse du double mot est celle du mot de poids faible qui précède le mot de poids fort (cf 7.4.3). Les octets sont donc rangés dans l'ordre inverse de leur poids dans la valeur de l'entier qu'ils codent.

Mot 0 : Poids faible																Mot1 : Poids fort															
Octet 0								Octet 1								Octet 2								Octet 3							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	23	22	21	20	19	18	17	16	31	30	29	28	27	26	25	24

Figure 6 : Représentation d'un entier sur un mot de 32 bits

7.5. Représentation des entiers dans les registres

Un registre (de 16 ou 32 bits) a le poids de ses bits qui croit de la droite vers la gauche. Le bit le plus à droite ayant le poids 0 pour EAX, AX , AL et AH. Le bit le plus à gauche ayant le poids 31 pour EAX, 15 pour AX et 7 pour AH et AL.

Registre 32 bits																															
Mot poids fort																Registre 16 bits															
																Octet poids fort : RH								Octet poids faible : RL							
																7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 7 : Entier dans un registre

7.6. Comparaison d'entiers

On est conduit dans les programmes à utiliser des entiers signés ou non signés. Comme l'ordre entre ces nombre est différent (l'entier non signé 0xff est plus grand que 0x00, mais l'entier signé 0xff est plus petit que 0x00) il existe deux ensembles de branchements conditionnels distincts selon que la comparaison ou, plus généralement, l'opération positionnant les codes conditions qu'ils suivent concerne une opération sur des entiers signés ou non signés.

La comparaison des entiers signés S et D correspond à effectuer l'opération D – S et à positionner les codes conditions en fonction du résultat. Après cette opération on utilise les instructions de branchement conditionnel suivantes :

- **JG** (Jump Greater) // S < D.
- **JL** (Jump Lower) // S > D.
- **JGE** (Jump Greater or Equal) // S <= D.
- **JLE** (Jump Lower or Equal) // S >= D.
- **JE** (Jump Equal) // S = D.
- **JNE** (Jump Not Equal) // S != D.

Après la comparaison des entiers non signés S et D, on utilise les instructions de branchement conditionnel suivantes :

- **JA** (Jump Above) // S < D.
- **JB** (Jump Below) // S > D.
- **JAE** (Jump Above or Equal) // S <= D.
- **JBE** (Jump Below or Equal) // S >= D.
- **JE** (Jump Equal) // S = D.
- **JNE** (Jump Not Equal) // S != D.

8. Conventions de liaison entre programmes et sous-programme

On dispose d'une instruction d'appel de sous-programme qui est appelée *call* et qui admet les mêmes mode d'adressage que l'instruction de saut *jmp* que nous avons décrite sous-section 6.2. Pour le retour au programme appelant on dispose de l'instruction *ret*. Ces instructions utilisent implicitement une zone de mémoire adressée par le registre ESP (*Extend Stack Pointer*) : la première, *call*, pour y ranger l'adresse de retour et la seconde, *ret*, pour y retrouver l'adresse de retour. Nous allons donc nous intéresser d'abord à la gestion de la pile adressée par ESP puis aux instructions d'appel et de retour de procédure.

8.1. Gestion de la pile, les instruction push, pop, pusha, popa

En dehors des effets de bord effectués par les instruction *call* et *ret* la pile peut être manipulée directement par les instructions *push{l | w}* et *pop{l | w}* qui permettent respectivement d'empiler ou de dépiler un mot ou un double mot selon que leur suffixe est *w* ou *l*. *Il n'est donc pas possible d'empiler ou de dépiler un octet*.

Quand la pile se remplit la valeur du pointeur de pile contenue dans ESP diminue de 2 ou de 4 selon qu'on empile un mot de 16 bits ou un double mot. Quand la pile se vide la valeur du pointeur de pile augmente de 2 ou 4 selon qu'on dépile un mot ou un double mot.

Le pointeur de pile pointe toujours sur l'octet de plus faible poids de l'entité au sommet de pile (mot ou double mot). L'opération *push* commence donc par diminuer le pointeur de pile de la taille de l'entité que l'on empile puis recopie la valeur empilée. L'opération *pop* recopie la valeur dans l'opérande puis augmente la valeur du pointeur de pile de la taille de l'opérande.

Exemple : on suppose que le registre EBP contient 0xbffffb18 et que le registre ESP contient 0xbffffb14, la figure 8.1 illustre l'effet de l'instruction « `pushl %ebp` » sur la pile (zone mémoire adressée par ESP).

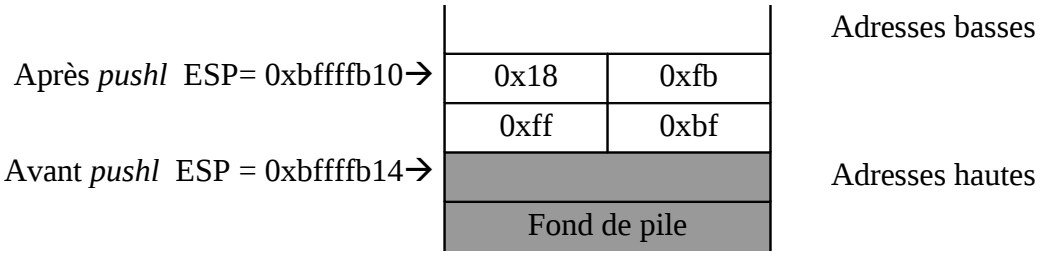


Figure 8 : Effet de pushl sur la pile

La figure suivante (Effet du popl %eax sur EAX et sur la pile) illustre l'effet de l'instruction « `popl %eax` » si cette instruction suit immédiatement l'instruction « `pushl %ebp` » précédente.

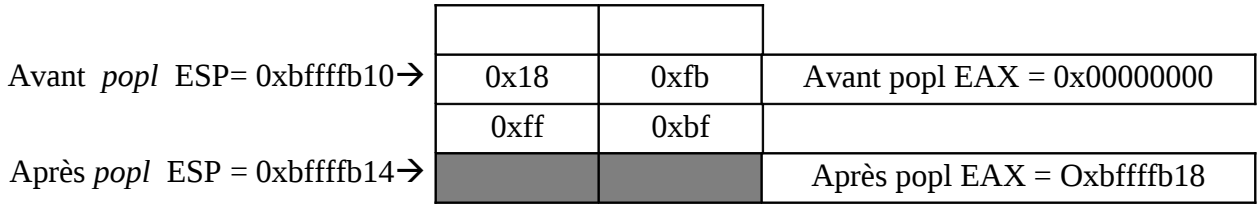


Figure 9 : Effet du *popl %eax* sur EAX et sur la pile

La figure (Effet du pushw sur la pile) illustre l'effet de l'instruction « `pushw %bp` »

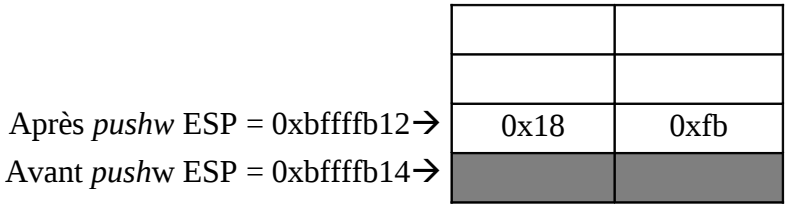


Figure 10 : Effet du *pushw* sur la pile

Les opération *pushal* et *popal* permettent de sauver sur la pile (respectivement de restaurer à partir de la pile) l'ensemble des registres de 32 bits à l'exception des registres EFLAG et EIP.

8.2. Les instructions d'appel et de retour de sous-programme

8.2.1. L'instruction *call* d'appel de sous-programme

Les machines Intel fournissent une instruction appelée *call*, qui permet à la fois d'effectuer un branchement à une adresse fournie en opérande et de sauver à l'adresse stockée dans le registre ESP l'adresse de l'instruction qui suit l'instruction *call*. Après l'instruction *call*, la valeur stockée dans le registre ESP a été diminuée de 4, et donc pointe sur l'adresse de retour. L'instruction *call* est donc une combinaison indivisible d'un empilement de l'adresse qui suit l'instruction *call* et d'une instruction *jmp* à l'adresse fournie en opérande. L'instruction *call* admet les mêmes mode d'adressage que l'instruction *jmp*.

Pour utiliser une instruction *call*, il faut donc être sûr que le programme qui s'exécute possède une pile et que ESP a été initialisé pour pointer sur le sommet de cette pile. C'est effectivement le cas pour tout programme s'exécutant sous Unix et dont l'exécutable a été fabriqué à l'aide de la commande *gcc*.

8.2.2. L'instruction *ret* de retour de sous-programme

Cette instruction copie dans le pointeur d'instruction EIP le contenu du mot de 32 bits dont l'adresse est contenue dans ESP et augmente le contenu de ESP de 4. L'instruction *ret* est donc la combinaison indivisible du retrait de la pile d'un double mot et d'un branchement à l'adresse dépilée. Pour qu'il n'y ait pas d'erreur à l'exécution il faut être sûr que le double mot au sommet de pile pointe bien sur une instruction dans la mémoire d'instructions.

8.3. Passage de paramètres, gestion des variables locales et utilisation des registres

8.3.1. Passage des paramètres

L'association de *call* et *ret* permet donc la mise en œuvre simple de sous programmes éventuellement récursifs qui n'utilisent pas de paramètre. Si on veut pouvoir utiliser des paramètres, il faut les allouer dans la pile, pour permettre la récursivité, et fixer des conventions pour que l'appelant sache dans quel ordre il doit copier les valeurs des paramètres sur la pile s'il veut que l'appelé puisse les exploiter. Avant l'appel d'une fonction $f(p_1, p_2, \ldots, p_n)$ les paramètres $p_n, p_{n-1}, \ldots, p_1$ seront rangés dans la pile et dans cette ordre.

On empile toujours des multiples de 4 octets si on veut suivre les conventions des langages C et Ada.

- **Paramètre passé « par adresse »** : on empile l'adresse de la variable soit quatre octets. En C, c'est le cas des paramètres déclarés sous la forme : *p(Untype *truc)* et qui à l'appel de *p* prennent la forme *p(&muche)*. En Ada, c'est le cas des paramètres structurés en mode « out » ou « in out ».
- **Paramètre passé par valeur de type simple** (entier, booléen, caractère, énumération etc.) on empile la valeur effective sur 4 octets **cadrée à gauche**. C'est à dire que les valeurs comme les caractères ou les entiers courts occupent les octets de poids faible du double mot empilé. En Ada on utilise ce type de passage pour les mode « in », « in out » ou « out » sur n'importe quel type simple.

Exemple : On veut appeler la procédure p(char c, int i, boolean b) avec pour c la valeur associée au caractère ASCII X (x majuscule) qui se note 'X', pour i la valeur 5 et pour b la valeur TRUE.

```
.equ TRUE,1
pushl $TRUE
pushl $5
pushl $'X'
call p
addl $12, %esp # libère la zone de paramètres sur la pile
```

Paramètres structurés passés par valeur. En général, en Ansi C on préfère passer ces paramètres par variable (passer l'adresse de base de la structure). En Ada, la façon de passer le paramètre structuré dépend du compilateur quel que soit le mode du paramètre. Si on veut quand même passer les paramètres structurés par valeur, on empile la valeur sur le même nombre d'octets.

Un exemple commenté est fourni dans l'annexe 4. On constatera qu'on ne change pas l'ordre des données dans la structure, que l'on s'arrange pour respecter les contraintes de frontières des éléments de la structure. Ainsi l'entier *i* qui suit le caractère *c* devant être à une frontière de double mot, implique que le caractère est alloué seul dans l'octet de **poids faible** d'un double mot.

8.3.2. Allocation des variables locales

Les variables locales n'ayant un sens qu'au cours de l'exécution d'une procédure elles sont allouées dans la pile ce qui en facilite l'allocation à l'entrée et la libération à la sortie de la procédure. Pour allouer *n* octets dans la pile il suffit d'enlever *n* au contenu du pointeur de pile. Compte tenu des contraintes déjà signalées sur la pile, on n'allouera que des nombres pairs d'octets.

8.3.3. Adressage par la procédure appelée des variables locales et des paramètres

On va les adresser au moyen du mode d'adressage base déplacement que nous avons déjà présenté. On utilise comme registre de base le registre spécialisé EBP. Comme chaque procédure base ses variables locales et ses paramètres avec le registre EBP, la procédure appelée doit, avant de modifier sa valeur la sauver dans la pile. L'annexe 1 page 39 illustre ce qu'est le contexte d'un sous programme dans la pile lorsqu'il s'exécute.

On notera que les paramètres ont toujours un déplacement positif et les variables locales ont un déplacement négatif relativement à la base de pile courante.

8.3.4. Paramètres, variables locales et sauvegarde pour une procédure P

Pour obtenir le résultat illustré dans l'annexe 0, chaque fonction ou procédure doit exécuter un prologue et un épilogue qui respecte le schéma :

```
/* Prologue de la procédure P qui a besoin de TVL_P octets de variable locales */
P :    pushl %ebp           # sauve la base de l'appelant
       movl  %esp, %ebp     # initialise la base de pile de P
       subl TVL_P,%esp      # alloue TVL_P octets sur la pile pour les variables locales
       pushal               # sauve les registres généraux (EAX, ECX, EDX, EBX,
                            # ESP, EBP, ESI, EDI) sur la pile

/* Epilogue de la procédure P qui restaure la valeur des registres de l'appelé sa base de
variable locales et de paramètre et lui rend le contrôle à l'adresse suivant l'appel */
       popal                # restaure la valeur des registres généraux
       leave                # libère les variables locales et restaure EBP
                            # ESP ← EBP, EBP ← POP()
       ret                  # rend le contrôle à l'appelant
```

8.3.5. Appel d'une fonction ou d'un sous-programme

La séquence d'appel de la procédure P(p1, p2, … pn) doit respecter le schéma suivant :

```
<empilement de pn>
<empilement de pn-1>
…
<empilement de p1>
call P
addl $4*n, %esp
```

8.3.6. Utilisation des registres

Dans la mesure du possible il vaut mieux, pour des raisons d'efficacité d'accès, utiliser les registres pour représenter les variables locales de type simple. Dans ce cas il est évidemment nécessaire d'effectuer la sauvegarde des registres de l'appelant que l'on utilise afin de lui permettre de retrouver son contexte d'exécution au retour de l'appel.

On ne peut évidemment pas toujours se dispenser d'allouer les variables locales dans la pile. C'est en particulier le cas d'une variable locale dont la taille dépasse 4 octets où d'une variable locale dont on doit passer l'adresse en paramètre d'appel à une autre procédure.

On peut également utiliser les registres pour y implanter des paramètres d'appel ou de retour de type simple. Evidemment dans ce cas et au moins pour les paramètres de retour on ne doit effectuer ni sauvegarde, ni restauration.

Attention : si vous appelez des fonctions Unix leur résultat est rendu dans EAX (qui sera donc détruit) et elles peuvent de plus détruire les registres EDX et ECX**.**

9. Les entrées sorties à l'aide des fonctions C *printf* et *scanf*

Les entrées sorties sur le terminal seront faites en utilisant les primitives d'entrées sorties *printf* et *scanf* que fournit la bibliothèque C. Nous conseillons donc au lecteur de se reporter aux ouvrages sur le langage C, notamment à celui de Bernard Cassagne [Cassagne 98], pour ce qui est de la définition exhaustive des formats de données associés aux différents types de données à imprimer ou à lire. Nous nous contentons dans cette section de préciser comment doivent être fournis à l'appel la chaîne spécifiant le message éventuel et le format des différents paramètres à lire ou à écrire.

9.1. Exemples simples en langage C

- Impression des valeurs de deux entiers :

```
int i = 12 ;
int j = 32 ;
printf("La valeur de i est %d et celle de j %d \n", i, j) ; /* \n : retour chariot */
```

imprimera : *La valeur de i est 12 et celle de j 32*

- Impression d'une chaîne de caractères :

```
char message[] = "Hello world!";
printf("Message : %s\n", message);     /* message = &message[0] */
```

imprimera : *`Message : Hello world!`*

- Lecture d'un entier :

```
int i;
scanf("%d", &i);       /* le caractère & indique que l'on passe une adresse */
```

permettra de lire un entier tapé à la console et de le stocker dans la variable i.

- Lecture d'une chaîne de caractères :

```
char message[13] ;
scanf("%s", message);     /* message = &message[0] */
```

permettra de lire un message d'au plus 12 caractères tel que « Hello world ! », tapé à la console, et de le stocker dans la variable message. Attention, il faut prévoir un caractère supplémentaire pour stocker le caractère de fin de chaine `\0`.

9.2. Appel de printf

On veut traduire en assembleur l'appel à la fonction C à un nombre variable de paramètres :

printf(format, $param_1$, $param_2$, …, $param_n$) ;

où *format* est une chaîne de caractères qui contient du texte et la spécification des paramètres $param_1$, $param_2$, …, $param_n$. La chaîne format est identique à ce qu'elle serait dans un programme C voulant imprimer le même ensemble de paramètres. On aura donc en assembleur.

```
       .section .rodata
Format: .string format
       .equ nb_param,…
       …
       .section text
       …
       #Appel de printf(format, param1 , param2 , …, paramn )
                               # sauvegarde éventuelle de EAX, EDX, ECX
       pushl paramn            # On empile les paramètres à partir de paramn
       pushl paramn-1
       …
       pushl param1
       pushl $Format           # On empile l'adresse de la chaîne Format
       call printf             # appel de la fonction
       addl $nb_param*4,%esp   # on dépile les paramètres
                               # restauration éventuelle de EAX, EDX, ECX
       …
```

9.3. Appel de scanf

On veut traduire en assembleur l'appel à la fonction C à un nombre variable de paramètres :

scanf(format, &$param_1$, &$param_2$, …, &$param_n$) ;

où *format* est une chaîne de caractères qui contient du texte et la spécification du type des paramètres $param_1$, $param_2$, …, $param_n$ qu'on lit sur le clavier. La chaîne format est identique à ce qu'elle serait dans un programme C voulant lire et formater le même ensemble de paramètres.

```
       .section .rodata
Format: .string format
       .equ nb_param,…
       …
       .section text
       …
       #Appel de scanf(format, &param1 , &param2 , …, &paramn )
                               # sauvegarde éventuelle de EAX, EDX, ECX
       pushl $paramn           # On empile l'adresse des paramètres à partir de paramn
       pushl $paramn-1
       …
       pushl $param1
```

```
pushl $Format            # On empile l'adresse de la chaîne Format
call scanf               #appel de la fonction
addl $nb_param*4,%esp    #on dépile les paramètres
                         # restauration éventuelle de EAX, EDX, ECX
…
```

10. Traduction des principaux type de données [5]

On a déjà présenté dans la section 7 le principe du codage des entiers sur n bits. Les entiers (les types *int* ou *long int* du langage C) sont représentés en complément à 2 sur 32 bits. Les entiers courts (les *short int* du langage C) sont représentés en complément à 2 sur 16 bits. Les caractères (type *char* du langage C) sont représentés par leur code ASCII sur un octet. Une chaîne de caractères est représentée par la suite des codes ASCII des caractères qui la composent suivie d'un caractère nul (dont tous les bits sont à zéro). Pour les autres types de données (tableaux ou enregistrements) on a a priori le choix de la représentation sauf si on veut utiliser des modules écrits dans un langage autre que l'assembleur. Dans ce dernier cas, il faut respecter les choix du compilateur du langage. Les choix d'implantation que nous présentons sont ceux du compilateur Gnu du langage C sur Unix.

10.1. Type énuméré, booléen

Le type booléen n'existe pas à proprement parler dans le langage C. Implicitement le type booléen a la forme d'un type énuméré :

```
typedef enum {FALSE = 0, TRUE = 1} boolean_t
```

qui est implanté dans un entier non signé (donc occupe 4 octets). Quand les expressions intervenant dans une conditionnelle (cf. section 3.1) ne sont pas booléennes mais entières le résultat de leur évaluation est projeté sur l'ensemble {false, true} avec la convention que toute expression non nulle est associée à *true* et toute expression nulle est associée à *false*.

Attention ! Contrairement au langage C et au langage Ada, l'assembleur ne fournit pas d'opérations logiques *OR* (||) , *AND* (*&&*), *NOT* (*!*). Les opérations fournies par l'assembleur et qui ont pour nom **AND**, **NOT** et **OR** sont des opérations qui opèrent bit à bit entre leurs opérandes. La conséquence est qu'il existe des couples d'entiers *a* et *b* différents et non nuls tels que AND a, b donne un résultat nul, or *a* et *b* étant non nuls ce sont des représentations possibles de la valeur booléenne *true*.

10.2. Article d'enregistrement

Considérons l'exemple suivant :

```
typedef enum {TRUE, FALSE} t_boolean ;       // définition du type _booléen
typedef struct {                             // définition d'un type enregistrement
      boolean_t a ;                          // un booléen
      short int f ;                          // un entier signé codé sur 16 bits
      int b;                                 // un entier signé codé sur 32 bits
```

[5] Cette section reprend presque mot pour mot des parties de la section 7 des notes du cours de logiciel de base Xavier Nicollin [Nicollin 1997] en faisant les adaptations nécessaires pour tenir compte du changement d'assembleur. La lecture de l'original de Xavier Nicollin sera fort utile à tous ceux qui veulent en savoir plus, notamment sur l'implantation des tableaux multidimensionnels et des listes.

```
    char c, d, e ;                    // trois caractères
} t_enreg ;                           // le type créé s'appelle t_enreg
```

Toute variable du type *t_enreg* aura ses différents champs rangés dans l'ordre, à des adresses consécutives. Chaque champ doit occuper un nombre d'octets multiple de 4 et donc on peut être amené à utiliser jusqu'à trois octets de remplissage. Soit *t* une variable du type *t_enreg*. Cette variable occupe 16 octets qui seront organisés en mémoire comme suit :

	Octet 0	Octet 1	Octet 2	Octet 3
Adresse de t →	a			
	f		alignement	alignement
	b			
	c	d	e	alignement

L'accès aux différents champs de la variable *t* s'effectue par adressage indirect avec base et déplacement relativement à son adresse. Afin d'augmenter la lisibilité du code on utilise des constantes symboliques pour les déplacements.

```
.equ D_a, 0
.equ D_f, 4
.equ D_b, 8
.equ D_c, 12
.equ D_d, 13
.equ D_e, 14
.equ SOF_t_enreg, 16
```

Si on veut traduire en assembleur l'affectation au champ *d* de la variable *t* la valeur 'X', on aura différentes traductions possibles selon le statut de la variable *t* (*t.d = 'X'*).

- t est une variable globale au programme : t sera donc déclarée :

```
.lcomm t, SOF_t_enreg
```

et l'affectation pourra être réalisée de deux façons différentes :

```
movl $t, %eax              # eax contient l'adresse de l'enregistrement t
movb $'X', D_d(%eax)       # t.d = 'X'
```

ou alors en utilisant le mode d'adressage direct :

```
movb $'X', t+D_d           # t.d = 'X'
```

- t est une variable locale du programme et est donc allouée dans la pile et adressée par un déplacement négatif relativement à la base de pile courante déclaré sous la forme :

```
.set D_t, …
```

Sous cette hypothèse on a encore deux façons d'accéder au champ *d* :

```
lea D_t(%ebp), %eax        # eax adresse de la variable t
movb $'X', D_d(%eax)       # t.d = 'X'
```

ou alors directement :

```
movb $'X', D_t+D_d(%ebp)   # t.d = 'X'
```

- t est un paramètre passé par référence qui est à un déplacement *D_ad_t* positif sur la pile par rapport à la base courante de pile :

```
.set D_ad_t, …              # déplacement relativement à ebp de l'adresse de t
```

l'accès au champ d de la variable t se fera alors :

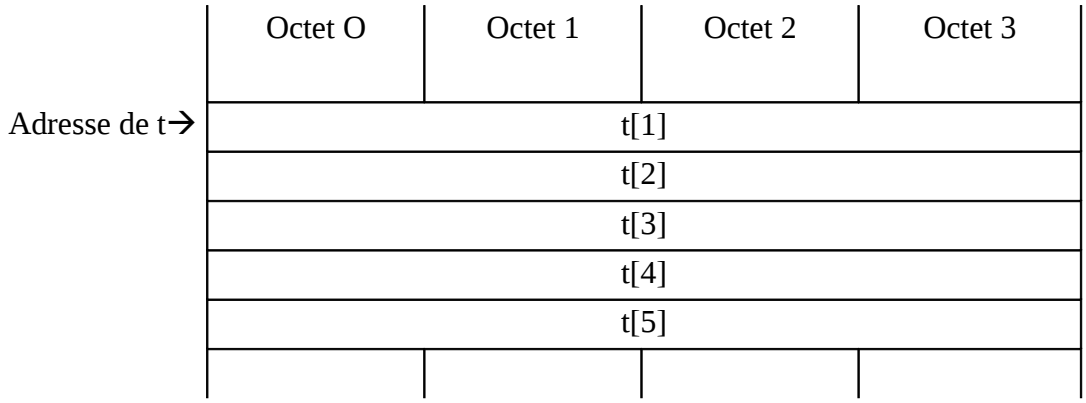
```
movl   D_ad_t(%ebp), %eax  # eax = adresse de t
movb   $'X', D-d(%eax)     # t.d = 'P'
```

10.3. Tableaux

Soit la définition de type suivante :

```
typedef int tab1[1..5];
```

Et soit *t* une variable de type *tab1*. Les élément de t seront rangé en mémoire dans l'ordre des adresses croissantes :



Pour accéder à l'élément t[i] on effectue le calcul :

Adresse(t[i]) = Adresse(t) – (a * (taille d'un élément)) + i * (taille d'un élément)

où *a* désigne l'indice du premier élément du tableau donc dans cet exemple *a* vaut *1*. On remarque que Adresse(t) – (a * (taille d'un élément)) est l'adresse où serait rangé l'élément 0 de ce tableau si il existait. On l'appelle **l'adresse virtuelle du tableau** t. L'adresse virtuelle d'un tableau est donc l'adresse de son élément d'index 0 qu'il soit fictif ou non.

On définit le déplacement pour trouver l'adresse virtuelle relativement à l'adresse du tableau :

```
.equ D_t_0, -1*4           # -(indice du premier élément)*(taille d'un élément)
```

supposons que l'on veuille traduire en assembleur l'affectation :

```
t[i] = 42 ;
```

où i n'est connu qu'au cours de l'exécution. Comme précédemment nous allons distinguer plusieurs cas selon le statut de la variable *t*.

- **t est une variable globale :**

```
.equ NB_ELEM,5
.equ TAILLE,4
.lcomm t, NB_ELEM*TAILLE
```

  - Première solution :

```
# la valeur calculée pour i est rangée dans %ecx
movl $t+D_t_0,%ebx  # ebx = adresse virtuelle de t
movl $42, 0(%ebx,%ecx,4)   # t[i] = 42
```

On utilise ici l'adressage indirect avec base déplacement et index typé. On profite ainsi du fait que la taille des élément du tableau est compatible avec les facteurs multiplicatifs de l'indice fournis par ce mode d'adressage. Ce mode d'adressage est utile lorsqu'on dispose d'une adresse de base (ici l'adresse virtuelle de t) dans un registre, d'un déplacement connu statiquement (ici 0) et d'un déplacement calculé dynamiquement (ici : i*taille d'un élément). Si la taille de l'élément ne correspond pas à l'un des 4 facteurs disponibles, on est obligé d'effectuer le calcul de %ecx « à la main », c'est à dire que l'on doit effectuer le produit de *i* par la *TAILLE* et affecter le résultat au registre choisi comme index .

```
movl %ecx,%eax              # eax = i
mull $TAILLE                # eax = i*taille
movl $t+D_t_0,%ebx          # ebx = adresse virtuelle de t
movl $42,0(%ebx, %eax)      # t[i] = 42
```

- Seconde solution : (le déplacement pour obtenir l'adresse virtuelle de t est utilisé à la fin)

```
# calcul de i dans %ecx
movl   $t,%ebx
movl   $42,D_t_0(%ebx,%ecx,4)
```

- **t est une variable locale ou un paramètre passé par valeur**. Dans ces deux cas le tableau est alloué sur la pile et l'adresse de t est un déplacement négatif ou positif relativement à la base de pile courante. On aura :

```
.set D_t,…          #déplacement de t dans la pile relativement au registre ebp.
```

On écrit alors :

```
# le calcul de i est  fait dans ecx
lea    D_t(%ebp),%ebx            # ebx = adresse de t
movl   $42,D_t_0(%ebx,%ecx,4)
```

Ou bien

```
lea D_t+D_t_0(%ebp),%ebx    # ebx = adresse virtuelle de t
movl $42,0(%ebx,%ecx,4)
```

Ou encore plus simplement, sans utiliser *ebx*

```
movl $42, D_t+D_t_0(%ebp,%ecx,4)
```

- **t est un paramètre passé par référence**. On note *D_ad_t* le déplacement de l'adresse relatif au registre *ebp* sur la pile. On aura :

```
.set D_ad_t,…
# Calcul de i dans ecx
movl   D_ad_t(%ebp),%ebx          # ebx = adresse de t
movl   $42,D_t_0(%ebx,%ecx,4)
```

10.4. Pointeurs

Un pointeur est représenté par l'adresse de l'objet pointé. Il est donc codé comme un entier non signé sur 32 bits. La valeur *null* des langages Ada et C est représentée par 0 :

```
.equ null,0
```

Considérons la définition d'un type liste d'entier en C :

```
typedef struct elem *liste ;  // Une liste est un pointeur sur le premier élément
typedef struct elem {
         int  valeur ;        // valeur de l'élement
         liste suivant ;      // adresse du suivant dans la liste
} t_element ;
```

Les constantes symboliques associées au type *t_element* sont :

```
.equ SOF_t_element,8          // taille d'un élément du type
.equ D_valeur,0               // déplacement du champ valeur
.equ D_suivant,4              // déplacement du champ suivant
```

Soit p un élément de type *liste* et qui pointe sur un objet de type *t_element*, et traduisons

```
p = p->suivant,
```

En considérant les trois cas d'allocation en mémoire possibles de la variable p on aura :

- p est une variable globale :

```
.lcomm p,4                    // p est créée mais pas initialisé
…                             // p à ce point pointe sur un objet de type t_elem
movl $p,%ebx                  // ebx pointe sur l'objet pointé par p
movl D_suivant(%ebx),%ebx     // ebx pointe sur l'objet pointé par le champ
                              // suivant p
movl %ebx,p                   // p = p->suivant
```

- p est un paramètre ou une variable locale

```
.set D_p,…                    // déplacement de p relativement à la base de pile
…
movl D_p(%ebp),%ebx           // ebx = p
movl D_suivant(%ebx),%ebx     // ebx = p->suivant
movl %ebx, D_p(%ebp)          // p = ebx
```

- p est un paramètre passé par référence

```
.set D_ad_p,…                 // déplacement de l'adresse de p sur la pile
…                             movl D_ad_p(%ebp),%eax // eax = adresse de p
movl (%eax),%ebx              // ebx = adresse de p
movl D_suivant(%ebx),%ebx     // ebx = p->suivant
movl %ebx,(%eax)              // p = p->suivant
```

L'allocation dymamique s'effectue en Ada au moyen de *new* et en C au moyen de la fonction *(void*) malloc(int taille_en_octets)*. En assembleur, pour allouer dynamiquement des enregistrements ou tout autre type de variable on peut utiliser, comme en C, la fonction *malloc*. Cette fonction effectue l'allocation de la taille demandée dans une zone mémoire, appelée **tas**,

associée au programme. Elle retourne l'adresse de la zone allouée dans le registre *eax*. Si l'allocation n'est pas possible *malloc* retourne dans *eax* la valeur *null*.

11. Codage des instructions

Nous décrivons dans cette section le format général du codage binaire des instructions de l'Intel. Cette section reprend en la simplifiant la documentation du constructeur qui est disponible sur le site web d'Intel [Intel 99]. La liste des instructions peut être trouvée en annexe.

11.1. Format général des instructions

Les instructions de la famille de processeurs Intel sont toujours un sous ensemble du format général donné dans la figure 8. Une instruction peut comporter : un octet de préfixe optionnel, un ou deux octets associés au code opération de l'instruction, suivi si c'est nécessaire de la spécification du mode d'adressage des opérandes consistant en un octet appelé *ModR/M* pouvant être suivi d'un octet d'index ou *Scalable Index Byte* (SIB). Ces champs peuvent être suivis d'une valeur de déplacement codée sur 1 ou 4 octets et/ou d'une valeur immédiate codée sur 1, 2 ou 4 octets.

Préfixe	Code OP	ModR/M	Index SIB	Déplacements	Valeur Immédiate
Optionnel 1 octet	1 ou 2 octets	1 octet si nécessaire	1 octet si nécessaire	déplacement sur 1 ou 4 octets, ou rien	donnée immédiate sur 1, 2, 4 octets ou rien

Figure 11 : Format général d'une instruction

7 6	5 3	2 0
Mod	Reg/Op	R/M

7 6	5 3	2 0
Echelle	Index	Base

Figure 12 : Format des octets ModR/M et SIB

11.2. Préfixe des instructions

Ce champ n'est utilisé que lorsqu'on veut utiliser l'un des registres de 16 bits (AX, BX, CX, etc.) dans une opération ou plus généralement opérer sur un mot. Dans ce cas le préfixe est présent et a la valeur hexadécimale 0x66.

11.3. Code opération

Le champ code opération peut être codé sur 8, 16 ou 19 bits. Dans le dernier cas, les deux premiers octets constituent la **partie primaire** du code opération et les trois bits restant du code opération, codés dans les bits 3, 4 et 5 de l'octet *ModR/M*, constituent son **extension**. Pour spécifier que l'opération codée porte sur un mot (dans ce cas l'instruction est précédée du préfixe 0x66) ou un double mot, le codage du code opération comprend un bit appelé bit *W* qui vaut 1. Si ce bit vaut 0 l'opération porte sur un octet. Le bit *W* est le plus souvent le bit 0 du champ code opération quand celui-ci est codé sur un seul octet.

Si l'instruction utilise une donnée immédiate sur 8 bits qui doit être étendue à 16 ou 32 bits avant que l'opération soit effectuée, l'extension peut être effectuée avec ou sans recopie du bit de

signe. Le mode choisi est sélectionné par le bit appelé *S* du code opération qui vaut 0 si on opère sans extension du bit de signe et 1 sinon.

Pour les instructions conditionnelles (branchement conditionnel ou positionnement si condition) le champ condition (*tttn*) est codé pour spécifier la condition que l'on teste. Les bits *ttt* spécifient la condition et le bit n spécifie si on s'intéresse à la condition (n = 0) ou à sa négation (n = 1). Pour un code opération dont la partie primaire est codée sur un seul octet , le champ *tttn* est situé dans les bits 3, 2, 1 et 0 du code opération. Pour un code opération dont la partie primaire du code opération est codée sur deux octets, le champ *tttn* est dans les bits 3, 2, 1 et 0 du second octet du code opération. L'annexe 11 page 66 contient les valeurs de *tttn* et leur correspondance en terme de la condition testée.

11.4. Octet ModR/M et octet Index (Scalable Index Byte)

Les instructions qui utilisent un opérande en mémoire ont un octet *ModR/M* qui suit la partie primaire du code opération. La partie *ModR/M* est constituée comme le montre la figure 9 de trois champs :

- Le champ *Mod* qui, combiné avec le champ *R/M*, permet de coder les modes d'adressage permis.
- Le champ *Reg/Op* peut spécifier soit un nom de registre, soit la partie secondaire du code opération.
- Le champ *R/M* peut spécifier un nom de registre comme opérande ou être combiné avec le champ *Mod* pour coder un mode d'adressage.

Pour certains modes d'adressage on a besoin d'un octet supplémentaire pour spécifier complètement le mode d'adressage de l'opérande en mémoire. Cet octet est appelé octet d'index typé ou *Scalable Index Byte* (SIB). L'octet SIB comprend également trois champs :

- L'*Echelle (SS)* qui spécifie le facteur à appliquer à la valeur de l'index..
- L'*Index* proprement dit qui spécifie le nom du registre d'index.
- La *Base*, qui spécifie le nom du registre de base.

La sous-section 11.6 précise comment sont codés les octets *ModR/M* et *SIB*.

11.5. Déplacements et données immédiates

Certains modes d'adressage utilisent un déplacement qui est codé immédiatement à la suite soit du champ *ModR/M*, soit du champ *SIB* s'il est présent. Le déplacement s'il est requis peut être codé sur 1 ou 4 octets.

Si l'instruction spécifie un opérande immédiat, son codage suit le champ déplacement éventuel. Le codage d'une donnée immédiate peut comporter 1, 2 ou 4 octets.

11.6. Codage du mode d'adressage

Les valeurs des octets *ModR/M* et *SIB* ainsi que les modes d'adressage correspondants sont donnés respectivement dans les tables en annexe pages 62 et 64.

12. Assemblage, édition de liens et mise au point d'un programme assembleur

Nous présentons dans cette section les principales commandes qui vous permettront d'assembler, de construire un exécutable et de mettre au point un programme écrit en assembleur Gnu.

12.1. Fichier source

Les fichiers sources doivent avoir l'extension ***.s***. Vous pouvez les produire avec n'importe quel éditeur, si vous utilisez *(x)emacs* pensez qu'il possède un mode assembleur et donc pensez à choisir et à redéfinir les paramètres par défaut de ce mode qui ne vous conviendraient pas.

Un programme assembleurs peut comporter plusieurs modules sources en assembleurs que vous pourrez assembler séparément, puis relier entre eux au moyen de l'éditeur de lien *ld* d'Unix [S. Chamberlain 94], soit assembler et relier entre eux les modules en un seul appel à la commande **gcc**.

12.2. Assemblage et construction d'un exécutable à l'aide de la commande gcc

Pour assembler un fichier *file.s* il suffit d'appeler la commande :

as –a file.s [–o file.o]

L'option ***–a*** vous vous permet d'obtenir un listing contenant la table des symboles. Si vous voulez avoir la liste dans le fichier *file.l* plutôt que sur votre écran, redirigez la sortie sur un fichier

(> file.l)

L'option ***–o*** vous permet de donner un nom au binaire objet produit, si non le binaire est produit dans le fichier ***a.out***.

Pour faire un exécutable à partir de fichiers *file1.o, file2.o, …fileN*, générés par l'assembleur il suffit d'appeler la commande **gcc file1.o file2.o …fileN.o –o file**

Attention : pour que la commande précédente s'exécute sans erreur, l'un des fichiers *file.o* doit obligatoirement exporter un point d'entrée de nom ***main***. Si vous ne voulez pas définir un point d'entrée *main* dans votre programme, il vous faudra utiliser les possibilités fournies par l'éditeur de lien d'Unix et pour cela vous reporter à la documentation [S. Chamberlain 94].

La commande :

gcc –Wa,-a file.s –o file

est équivalente aux deux premières commandes : l'exécutable est dans ***file*** et la liste dans ***file.l***

12.3. Mettre au point votre code avec gdb

gdb *file*	appel de gdb sur l'exécutable *file*.
quit	Pour quitter le metteur au point.
help *cde*	Obtention d'aide sur la commande *cde.*
br *[etiq]*	Pose de point d'arrêt à l'entrée de *etiq* de votre programme.
info br	Liste des points d'arrêt courants.
delete *i.*	Suppression du point d'arrêt numéro *i*.
run	Lancement de l'exécution du programme chargé depuis le début

	jusqu'au premier point d'arrêt s'il y en a de posé.
si	Exécution du programme pas à pas (instruction par instruction).
ni	Exécution du programme pas à pas mais en sautant les appels de fonctions.
cont	Poursuite de l'exécution jusqu'au prochain point d'arrêt.
RET	Retour chariot : exécute à nouveau la commande précédente.
CTRL-p	Prompte la commande précédente.
i reg	Visualisation de tous les registres.
i reg *eax ebx*	Visualisation de registres particuliers, par exemple *eax* et *ebx.*
frame	Visualisation du bloc de pile associé à la fonction courante.
frame *i*	Visualisation du bloc de pile n° i (ième niveau d'appel) ou à l'adresse i.
bt	Liste de blocs couramment empilés.
x/nfu *addr*	Visualisation de la mémoire : • *n* nombre d'unités à visualiser • *f* format d'affichage prend ses valeurs dans : **x** hexadécimal, **d** décimal signé, **u** décimal non signé, **t** binaire, **i** instruction, **s** chaîne terminée par 0 • *u* taille d'une unité qui peut être : **b** octet, **h** demi-mot, **w** mot, **g** mot de 64 bits • *addr* est l'adresse, donnée sous forme hexadécimale ou symbolique, du début de la zone à visualiser. *Exemple :* *x/15i main # affiche 15 instructions à partir de l'adresse main.*
display	Surveillance de la valeur d'une variable, d'un registre ou du contenu d'une zone-mémoire à chaque pas d'exécution. Sans paramètre : liste des points de surveillance courants. Paramètre(s) : • *[/nfu] x* # valeur de la variable x. *# Exemple : display n* • *$reg* # contenu du registre x. *# Exemple : display $eax* • / nfu *$reg* /* contenu de la zone mémoire pointée par le registre $reg. *Exemples :* o *display /4wx $ebx # valeur hexadécimale de 4 doubles mots à partir de l'adresse contenue dans EBX.*

	○ *display /s $ebx# chaîne de caractères pointée par EBX. */*
set {type}addr = value	Modification d'un emplacement mémoire quelconque : • *type* le type de la variable. Le champ type prend ses valeurs dans *{short, int, char}* (en fait tout type de base du langage C). • *addr* son adresse. • *value* la valeur à écrire.
set $reg = value	Modification du contenu d'un registre : • *reg* est le nom du registre que l'on veut modifier • *value* est la valeur décimale, octale ou hexadécimale que l'on veut donner au registre. *Exemples :* *set $eax = 0xf3* *affecte la valeur hexadécimale 0xf3 au registre eax* *set $eax = ($eax &0xffff0000) | 0xfedc* *affecte la valeur hexadécimale 0xfedc au mot de poids faible de eax sans modifier la valeur du mot de poids fort.*

Attention :

Certains étudiants n'utilisent pas *gdb* pour la mise au point de programmes complexes. Ne comprenant pas ce qui se passe réellement lors de l'exécution de leurs programmes, ils les corrigent plus ou moins au hasard. La raison mise en avant est le temps nécessaire à taper les commandes *gdb* avant de lancer l'exécution du programme. Pour éviter ce problème utilisez un **fichier de commande** dans lequel vous inscrivez une fois pour toutes les commandes que vous souhaitez voir exécutées au lancement de gdb. Supposons par exemple que vous souhaitiez poser deux points d'arrêt aux étiquettes *iter* et *fin*. Par ailleurs, vous voulez visualiser , chaque fois que le programme s'interrompt, l'instruction courante et celle qui la suit ainsi que les contenus des registres EAX et ECX.

Dans ce cas, il vous faudra inscrire dans un fichier de commande :

break iter
break fin
display $eax
display $ecx
display /2i $eip

Si le fichier de commande s'appelle *cde* et le programme à mettre au point *pgcd*, on lancera la mise au point par : *gdb –x cde pgcd*

Le programme s’arrêtera à chaque passage sur l’étiquette *iter* et lors du passage sur l’étiquette *fin*. Entre les points d’arrêts, il sera possible de travailler pas à pas *(si)* ou de façon continue *(cont)* et donc de comprendre finement le comportement du programme.

13. Bibliographie

[Kip R. Irvine 1999]

Assembly Language for Intel-Based Computer, 3rd Edition, Prentice-Hall Upper Saddle River NewJersey 07458, ISBN 0-13-660390-4, 667 pages.

[Barry B. Brey 1997]

The Intel Microprocessors 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, and Pentium Pro Processor – Architecture, Programming, and Interfacing ; Fourth Edition, Prentice-Hall Hall Upper Saddle River NewJersey ; ISBN 0 – 13 – 260670-4, 907 pages.

[Intel 1999]

Intel Architecture Software Developer's manual, Volume 2 : Instruction set reference, order number 243192, 1999.

http://developer.intel.com/software/idap/resources/technical_collateral/pentiumiiixeon/index.htm

[Bernard Cassagne 1998]

Introduction au langage ANSI C, disponible en pdf à l'URL :

http://www-clips.imag.fr/commun/bernard.cassagne/

[Dean Elsner, Jay Fenlason & friends 1994]

Using as, The GNU Assembler, 1994. Disponible au format PostScript à l'URL :

http://www.gnu.org/manual/gas-2.9.1/as.html

[Steve Chamberlain]

Using ld, The GNU linker, ld version 2, 1994, Disponible au format PostScript à l'URL :

http://www.gnu.org/manual/

[Richard M. Stallman and Roand H. Pesh 1998]

Debugging with GDB, 1998, Disponible au format PostScript à l'URL :

http://www.gnu.org/manual/gdb-4.17/gdb.html

[Yann Rouzaud 1991]

Programmation en assembleur 68000 sur DPX2000, 1991, polycopié 26 pages, épuisé.

[Sacha Krakowiak 1982]

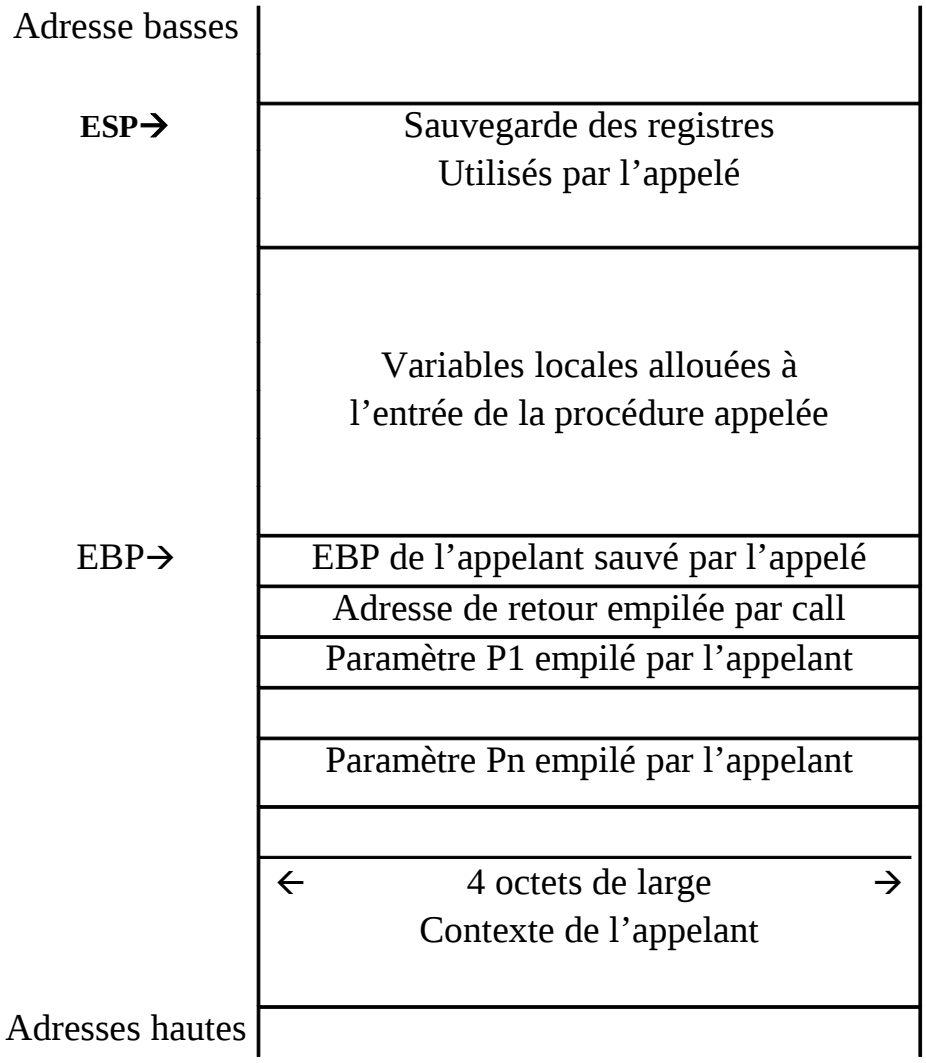
Logiciel de base , polycopié université J. Fourier 80 pages, dernier tirage 1982, épuisé.

[Xavier Nicollin 1997]

Ensimag première année – Logiciel de base : note de cours, 1997, polycopié 40 pages.

/perms/nicollin/LoBa/notes_de_cours.tex.ps

Annexe 1 : Schéma d'un contexte de sous programme dans la pile



Annexe 2 : Test des modes d’adressage de la mémoire de données

```
.file "adresse.s"
        .section .rodata
        .p2align 5                # force l’alignement sur une frontière de 32

.LC1:   /* Spécification d'une sortie hexadecimale sur 8 bits pour printf */
        .string "valeur de l'octet lu:%02x\n"

.LC2:   /* Spécification d'une sortie hexadécimale sur 16 bits pour printf */
        .string "valeur du mot lu : %04x\n"

.LC3:   /* Spécification d'une sortie hexadécimale sur 32 bits  pour printf */
        .string "valeur du double mot lu: %08x\n"

        .section .data
        .globl xl, xw, xb         /* exporte les variables xl, xw, xb */

xl:     .int 0x7fffffff           /* reservation d'un mot initialisé de 32 bits */
xw:     .hword 0x7fff             /* reservation d'un  mot initialisé de 16 bits */
xb:     .byte 0x7f                /* reservation d'un octet initialisé */

        .equ TTAB,200             /* taille du tableau tab */
        .equ TT_CHAR, 48          /* taille du tableau t_char */
        .equ D_T_CHAR,48          /* deplacement du tableau t_char dans str */
        .equ TSTR,D_T_CHAR+TT_CHAR/* taille de la structure str */

        .lcomm tab,TTAB,1         /* réservation du tableau tab */
        .lcomm str,TSTR,32        /* réservation de la structure str */

        .text

        .globl main, Rdam, Iam, Mdam, Iram, Bpiam, Briam, Siam

        /* Test des modes d'adressage */

        /* Register direct addressing mode */

Rdam:   xorl %eax,%eax            /* eax = 0 */
        incb %al                  /* al = al + 1*/
        movb %al,%ah              /* ah = al et eax = 257 !*/
        xorw %ax,%ax              /* ax = 0  */
        incb %ah                  /* ah = 1  et eax 256 !*/

        /* Immediate addressing mode *

Iam:    xorl %eax,%eax
        movb $0xff,%al            /* al = -1 et eax = 255 */
        movb $0xff,%ah            /* ah = -1 et ax = -1 et eax = 65535 */
        movw $0x1234,%ax          /* ax et eax = 0x1234, ah = 0x12 al = 0x34 */
        movl $-1,%eax             /* eax = -1 */

        /* Memory direct addressing mode */
```

ANNEXES

```
Mdam:   movb $0,xb              /* data_mem[xb].b= 0 */
        movw $0,xw              /* data_mem[xw].w= 0 */
        movl $0,xl              /* data_mem[xl].l= 0 */

        /* Indirect register addressing mode */

Iram:   movl $xb,%eax           /* eax = adresse de xb */
        movb $255,(%eax)        /* xb = -1 */
        movl $xw,%eax           /* eax = adresse de xw */
        movw $257,(%eax)        /* xw = 257 */
        movl $xl,%eax           /* eax = adresse de xl */
        movl $0x123456,(%eax)   /* xl = $12345678 */

        /* Base plus Index addressing mode *
        /* for (i = 0; i < TTAB; i++) tab[i]=i) */
        /* Dans le cas ou tab est un tableau d'octets, de mots de 16 et 32 bits */

Bpiam:  movl $0,%ebx            /*cas des octets  i = 0 */
iter1:  movl $tab,%eax          /* eax contient l'adresse de tab */
        cmpl $TTAB,%ebx         /* i < TTAB */
        jge suite1              /* non alors c'est fini */
        movb %bl,(%eax,%ebx)    /* oui : tab[i] = i (byte)*/
        movl $0,%edx            /* on relit dans edx ce qu'on a ecrit */
        movb (%eax,%ebx),%dl
        pushl %edx              /* on va l'imprimer */
        pushl $.LC1
        call printf
        addl $8,%esp
        incl %ebx               /* i++  au suivant */
        jmp iter1

suite1: movl $0,%ebx            /* cas mots de 16 bits i = 0 */
iter2:  movl $tab,%eax
        cmpl $TTAB-1,%ebx
        jge suite2
        movw %bx,(%eax,%ebx)
        movl $0,%edx
        movw (%eax,%ebx),%dx
        pushl %edx
        pushl $.LC2
        call printf
        addl $8,%esp
        addl $2,%ebx
        jmp iter2

suite2: movl $0,%ebx            /* cas double mots i = 0*/
iter3:  movl $tab,%eax
        cmpl $TTAB-3,%ebx
        jge suite3
        movl %ebx,(%eax,%ebx)
        movl (%eax,%ebx),%edx
        pushl %edx
        pushl $.LC3
        call printf
        addl $8,%esp
```

```
        addl $4,%ebx
        jmp iter3

suite3: /* Base relative plus index addressing mode */
        /* for (i=0; i < TT_CHAR; i++) str.t_char[i]=i*/

Briam:  movl $str,%eax            /* eax = adresse de base de str */
        movl $0,%ebx              /* ebx joue le role de i */
iter4:  cmpl $TT_CHAR,%ebx        /* i < TT_CHAR */
        jge suite4                /* non c'est fini */
        movb %bl,D_T_CHAR(%eax,%ebx)
        incl %ebx                 /* i++ */
        jmp iter4

suite4: movl $0,%ebx              /* On traite le cas des mots de 16 bits*/
iter5:  cmpl $TT_CHAR-1,%ebx
        jge suite5
        movw %bx,D_T_CHAR(%eax,%ebx)
        addl $2,%ebx              /* i=++ */
        jmp iter5

suite5: movl $0,%ebx              /* On traite le cas des doubles mots */
iter6:  cmpl $TT_CHAR-3,%ebx
        jge suite6
        movl %ebx,D_T_CHAR(%eax,%ebx)
        addl $4,%ebx              /* i++ */
        jmp iter6

suite6: /* Scaled index addressing mode */
        /* for (i=0; i < TT_CHAR; i++) str.t_char[i];*/

        .equ BFAC,1               /* facteur multiplicatif de l'index */
        .equ HFAC,2
        .equ WFAC,4

Siam:   movl $0,%ebx              /* ebx joue le role de i */
iter7:  movl $str,%eax            /*eax = adresse de base de str */
        cmpl $TT_CHAR/BFAC,%ebx   /* i < TT_CHAR */
        jge suite7                /* non c'est fini */
        movb %bl,D_T_CHAR(%eax,%ebx,BFAC)
        movl $0,%edx
        movb D_T_CHAR(%eax,%ebx,BFAC),%dl
        pushl %edx
        pushl $.LC1
        call printf
        addl $8,%esp
        incl %ebx
        jmp iter7

suite7: movl $0,%ebx
iter8:  movl $str,%eax            /*eax = adresse de base de str */
        cmpl $TT_CHAR/HFAC,%ebx
        jge suite8
        movw %bx,D_T_CHAR(%eax,%ebx,HFAC)
        movw D_T_CHAR(%eax,%ebx,HFAC),%dx
```

```
        pushl %edx
        pushl $.LC2
        call printf
        addl $8,%esp
        incl %ebx
        jmp iter8

suite8: movl $0,%ebx
iter9:  movl $str,%eax            /*eax = adresse de base de str */
        cmpl $TT_CHAR/WFAC,%ebx
        jge suite9
        movl %ebx,D_T_CHAR(%eax,%ebx,WFAC)
        movl D_T_CHAR(%eax,%ebx,WFAC),%edx
        pushl %edx
        pushl $.LC3
        call printf
        addl $8,%esp
        incl %ebx
        jmp iter9

suite9: /* fin du test des modes d'adressage de la mémoire */

        popal                     /* restauration des registres de l'appelant */
        leave                     /* restaure la base de pile */
        ret                       /* rend le contrôle à l'appelant */
```

Annexe 3 : Test des modes d’adressage de la mémoire programme

```
        .section .rodata
        .p2align 5

        /* Chaine de définition pour effectuer des printf de doubles mots */

.LC3:   .string "valeur du double mot rangé : %08x\n"
        .data
adIram: .int Iram                /* adresse dans la section text */
        .text
        .globl main
main:
Iram :
        pushl %ebp               /* respect des conventions de liaison */
        movl %esp,%ebp
        pushal

        /* Test des modes d'adressage de la mémoire d'instruction */

        jmp 0x080483cc           /* jmp à l'adresse absolue 0x080483cc = Iram !*/
        jmp Iram                 /* jmp relatif au pointeur d'instruction */
        movl $Iram,%ecx
        jmp *%ecx                /* jmp à l'adresse contenue dans ecx */
        movl $adIram,%eax
        jmp *0(%eax)             /* jmp à l'adresse contenu 0(%eax)*/
        xorl %ecx,%ecx           /* ecx = 0 */
        jmp *adIram(%ecx)        /* jmp à l'adresse contenue à adIram[0] */
        jmp *adIram              /* jmp à l'adresse contenue à ladresse adIram */

        /* fin du test des modes d'adressage */

        popal                    /* restaure les registres*/
        leave
        ret
```

Annexe 4 : Résultat de l’assemblage du programme calculant le pgcd

```
 1                      .file  "pgcd.s"
 2
 3                /* Ce programme calcule le pgcd entre les variables
 4                 * entière x et y codée sur 16 bits
 5                */
 6                      .section .rodata
 7 0000 56616C65  format:      .string "Valeur de x = %d et de y = %d\n"
 7      75722064
 7      65207820
 7      3D202564
 7      20657420
 8                      .data
 9 0000 0001      x:     .hword 256
10 0002 0010      y:     .hword 4096
11                      .text
12                      .p2align 2   # force l'alignement à une frontière de mot
13                      .global main # exporte le nom du point d'entrée du programme
14
15                main:
16 0000 55              pushl %ebp   # Convention de liaison avec l'appelant
17 0001 89E5            movl %esp,%ebp
18
19 0003 66833D00        cmpw $0,x    # if ((x==0)||(y==0)) aller imprimer x et y
19      00000000
20 000b 7448            je fin
21 000d 66833D02        cmpw $0,y
21      00000000
22 0015 743E            je fin
23                      .p2align 4,,7# force l'alignenment sur une frontière de 16
24                iter:
25 0017 66A10000        movw x,%ax   # if (x != y)
25      0000
26 001d 663B0502        cmpw y,%ax
26      000000
27 0024 7502            jne cont     # On continue
28 0026 EB2D            jmp fin      # si non on va imprimer x et y
29                      .p2align 4,,7
30                cont:
31 0028 66A10000        movw x,%ax   #if (x < y)
31      0000
32 002e 663B0502        cmpw y,%ax
32      000000
33 0035 7E0F            jle .LL      # aller calculer  y = y -x
34 0037 66A10200        movw y,%ax   # sinon calculer x = x - y
34      0000
35 003d 66290500        subw %ax,x
35      000000
36 0044 EBD1            jmp iter
37                      .p2align 4,,7
```

```
  38                .LL:
  39 0046 66A10000          movw x,%ax
  39      0000
  40 004c 66290502          subw %ax,y
  40      000000
  41 0053 EBC2              jmp iter
  42
  43                        .p2align 4,,7
  44
  45                fin:    /* impression de x et y */
  46 0055 31C0              xorl %eax,%eax
  47 0057 66A10200          movw y,%ax
  47      0000
  48 005d 50                pushl %eax
  49 005e 66A10000          movw x,%ax
  49      0000
  50 0064 50                pushl %eax
  51 0065 68000000          pushl $format
  51      00
  52 006a E8FCFFFF          call printf
  52      FF
  53 006f 83C40C            addl $12,%esp
  54
  55 0072 C9                leave
  56 0073 C3                ret
  57

DEFINED SYMBOLS
                           *ABS*:00000000 pgcd.s
              pgcd.s:7     .rodata:00000000 format
              pgcd.s:9     .data:00000000 x
              pgcd.s:10    .data:00000002 y
              pgcd.s:15    .text:00000000 main
              pgcd.s:45    .text:00000055 fin
              pgcd.s:24    .text:00000017 iter
              pgcd.s:30    .text:00000028 cont

UNDEFINED SYMBOLS
printf
```

Annexe 5 : Passage de paramètres de type structuré

```
typedef struct{
      char c;
      int i;
      char d, b;
} enreg;

void p1(enreg X1);
void p2(enreg *X2);

void p1(enreg X1){
  enreg Y;
  Y.c = X1.c;
  Y.i = X1.i;
  Y.b = X1.b;
  Y.d = X1.d;
}
void p2(enreg *X2){
  enreg Y;
  Y.c = X2->c;
  Y.i = X2->i;
  Y.b = X2->b;
  Y.d = X2->d;
}
int main(){
  enreg u;
  u.c = 'd';
  u.i = -1;
  u.b = 'F';
  u.d = 'G';
  p1(u);
  p2(&u);
}
```

=-=- **TRADUCTION EN ASSEMBLEUR DU PROGRAMME C PRECEDENT -=-=**

```
        /* Adressage de la structure C enreg */

        .set Dc,0                 #deplacement du champ c
        .set Di,4                 #deplacement du champ i
        .set Db,8                 #deplacement du champ b
        .set Dd,9                 #deplacement du champ d
        .equ ENREG_SIZE,12        #taille en octet d'un element de type enreg

        /* Definition de la fonction p1(enreg X1) */

        # adressage des variables locales et des parametres

        .set X1,8                 #deplacement du parametre X1 dans la pile
        .set Y, -ENREG_SIZE       #deplacement de la variable locale Y dans la pile

        .text
        .p2align 5
        .global p1

p1:     pushl %ebp
        movl %esp,%ebp
        subl $ENREG_SIZE,%esp     #alloue Y sur la pile
        pushal                    # sauve tous les registres

        movb X1+Dc(%ebp),%al
        movb %al,Y+Dc(%ebp)       # Y.c = X1.c
        movl X1+Di(%ebp),%eax
        movl %eax,Y+Di(%ebp)      #Y.i = X1.i
        movb X1+Db(%ebp),%al
        movb %al,Y+Db(%ebp)       #Y.b = X1.b
        movb X1+Dd(%ebp),%al
        movb %al,Y+Dd(%ebp)       #Y.d = X1.d

        popal                     #restaure les registres
        leave
        ret

        /* Definition de la fonction p2(enreg *X2) */

        # adressage des parametres et des variables locales

        .set X2,8                 #deplacement du parametre X2 dans la pile
        .set Y, -ENREG_SIZE       #deplacement de Y dans la pile

        .text
        .p2align 5

        .global p2

p2:     pushl %ebp
        movl %esp,%ebp
        subl $ENREG_SIZE,%esp     #alloue Y sur la pile
        pushal                    # sauve tous les registres
```

```
movl X2(%ebp),%eax          # EAX pointeur sur la structure
movb Dc(%eax),%dl
movb %dl,Y+Dc(%ebp)         # Y.c = X2->c
movl Di(%eax),%edx
movl %edx,Y+Di(%ebp)        #Y.i = X2->i
movb Db(%eax),%dl
movb %dl,Y+Db(%ebp)         #Y.b = X2->b
movb Dd(%eax),%dl
movb %dl,Y+Dd(%ebp)         #Y.d = X2->d

popal                       #restaure les registres
leave
ret
```

```
        .text
        .p2align 5

        /* definition du programme principal */

        # adressage des variable locale

        .set U, -ENREG_SIZE       #deplacement de u dans la pile

        .global main

main:   pushl %ebp
        movl %esp,%ebp
        subl $ENREG_SIZE,%esp     #alloue u sur la pile
        pushal                    # sauve tous les registres

        # initialisation de la structure u

        movb $'d',U+Dc(%ebp)      # u.c = 'd'
        movl $-1, U+Di(%ebp)      # u.i = -1
        movb $'F',U+Db(%ebp)      # u.b = 'F'
        movb $'G',U+Dd(%ebp)      # u.d = 'G'

        # appel de la procesure p1

        movl U+Db(%ebp),%eax
        pushl %eax                # empile le double mot b,d
        movl U+Di(%ebp),%eax
        pushl %eax                # empile le double mot i
        movl U+Dc(%ebp),%eax
        pushl %eax                # empile le double mot  c
        call p1
        addl $ENREG_SIZE,%esp     # libere l'espace des param

        # appel de la procedure p2

        leal U(%ebp),%eax               # calcule l'adresse de u dans EAX
        pushl %eax                      # empile l'adresse de u
        call p2
        addl $4,%esp                    # libere l'espace du parametre

        popal                           #restaure les registres
        leave
        ret
```

Annexe 6 : Exemple de diverses formes de boucle d’itération

<rédaction différée>

Annexe 7 : codage des instructions

Le codage des instruction est abordé en détail dans la section 11 page 32. On rappelle ici par commodité la syntaxe générale d'une istruction assembleur :

Préfixe	Code OP	ModR/M	Index SIB	Déplacements	Valeur Immédiate
Optionnel 1 octet	1 ou 2 octets	1 octet si nécessaire	1 octet si nécessaire	déplacement sur 1 ou 4 octets, ou rien	donnée immédiate sur 1, 2, 4 octets, ou rien

- Préfixe : présent si l'opération porte sur un mot. Sa valeur hexadécimale vaut alors 0x66.

- Code OP : cf annexe 8. Certains codes OP peuvent comporter un certain nombre de bits qu'il faut préciser :

  s (1 bit) : placé à 1 s'il faut étendre le bit de signe lorsque l'on a affaire à une conversion d'une représentation à une représentation de plus grande taille, 0 sinon.

  w (1 bit) : placé à 1 si l'on travaille sur des mots ou des doubles mot, 0 si l'on travaille sur des octets.

  tttn (4 bits) : type de test à effectuer, cf. annexe 11

- ModR/M : l'octet ModR/M, s'il est présent (cf annexe 8), comprend trois éléments, dont les valeurs sont disponibles dans l'annexe 9.

7 6	5 3	2 0
Mod	Reg/Op	R/M

- SIB : l'octet SIB, s'il est présent, comprend également trois éléments. Voir annexe 10.

7 6	5 3	2 0
Echelle	Index	Base

Annexe 8 : Codes OP

Formats des instructions	Codage
ADC - Add with Carry	
registre1 vers registre 2	0001 000w : 11 reg1 reg2
registre2 vers registre 1	0001 001w : 11 reg1 reg2
mémoire vers registre	0001 001w : mod reg r/m
registre vers mémoire	0001 000w : mod reg r/m
immédiat vers registre	1000 00sw : 11 010 reg : donnée immédiate
immédiate vers AL,AX ou EAX	0001 010w : donnée immédiate
immédiate vers mémoire	1000 00sw : mod 010 r/m : donnée immédiate
ADD – Add	
registre1 vers registre 2	0000 000w : 11 reg1 reg2
registre2 vers registre 1	0000 001w : 11 reg1 reg2
mémoire vers registre	0000 001w : mod reg r/m
registre vers mémoire	0000 000w : mod reg r/m
immédiat vers registre	1000 00sw : 11 000 reg : donnée immédiate
immédiate vers AL,AX ou EAX	0000 010w : donnée immédiate
immédiate vers mémoire	1000 00sw : mod 000 r/m : donnée immédiate
AND – Logical AND	
registre1 vers registre 2	0010 000w : 11 reg1 reg2
registre2 vers registre 1	0010 001w : 11 reg1 reg2
mémoire vers registre	0010 001w : mod reg r/m
registre vers mémoire	0010 000w : mod reg r/m
immédiat vers registre	1000 00sw : 11 100 reg : donnée immédiate
immédiate vers AL,AX ou EAX	0010 010w : donnée immédiate
immédiate vers mémoire	1000 00sw : mod 000 r/m : donnée immédiate
BOUND – Check Array Against Bounds	0110 0010 : mod reg r/m
BSF – Bit Scan Forward	
registre1 vers registre 2	0000 1111 : 1011 1100 : 11 reg1 reg2
mémoire vers registre	0000 1111 : 1011 1100 : mod reg r/m
BSR – Bit Scan Reverse	
registre1 vers registre 2	0000 1111 : 1011 1101 : 11 reg1 reg2
mémoire vers registre	0000 1111 : 1011 1101 : mod reg r/m
BSWAP – Byte Swap	0000 1111 : 1100 1 reg
BT – Bit Test	

registre, immédiate	0000 1111 : 1011 1010 : 11 100 reg : donnée im8
mémoire, immédiate	0000 1111 : 1011 1010 : mod 100 r/m : donnée im8
registre1, registre2	0000 1111 : 1010 0011 : 11 reg2 reg1
mémoire, registre	0000 1111 : 1010 0011 : mod reg r/m
BTC – Bit Test and Complement	
registre, immédiate	0000 1111 : 1011 1010 : 11 111 reg : donnée im8
mémoire, immédiate	0000 1111 : 1011 1010 : mod 111 r/m : donnée im8
registre1, registre2	0000 1111 : 1011 1011 : 11 reg2 reg1
mémoire, registre	0000 1111 : 1011 1011 : mod reg r/m
BTR – Bit Test and Reset	
registre, immédiate	0000 1111 : 1011 1010 : 11 110 reg : donnée im8
mémoire, immédiate	0000 1111 : 1011 1010 : mod 110 r/m : donnée im8
registre1, registre2	0000 1111 : 1011 0011 : 11 reg2 reg1
mémoire, registre	0000 1111 : 1011 0011 : mod reg r/m
BTS – Bit Test and Set	
registre, immédiate	0000 1111 : 1011 1010 : 11 101 reg : donnée im8
mémoire, immédiate	0000 1111 : 1011 1010 : mod 101 r/m : donnée im8
registre1, registre2	0000 1111 : 1010 1011 : 11 reg2 reg1
mémoire, registre	0000 1111 : 1010 1011 : mod reg r/m
CALL – Call Procedure	
directe	1110 1000 : déplacement sur 32 bits
indirecte registre	1111 1111 : 11 010 reg
mémoire indirecte	1111 1111 : mod 010 r/m
CBW – Convert Byte to Word	1001 1000
CDQ – Convert Doubleword to Qword	1001 1001
CLC – Clear Carry Flag	1111 1000
CMC – Complement Carry Flag	1111 0101
CMOVcc – Conditional Move	
Registre2 vers registre1	0000 1111 : 0100 tttn : 11 reg1 reg2

Mémoire vers registre	0000 1111 : 0100 tttn : mod reg r/m
CMP – Compare Two Operands	
registre1 avec registre2	0011 100w : 11 reg1 reg2
registre2 avec registre1	0011 101w : 11 reg1 reg2
mémoire avec registre	0011 101w : mod reg r/m
registre avec mémoire	0011 100w : mod reg r/m
immédiat avec registre	1000 O0sw : 11 111 reg : donnée immédiate
immédiate avec AL,AX ou EAX	0011 110w : donnée immédiate
immédiate avec mémoire	1000 O0sw : mod 111 r/m : donnée immédiate
CWD – Convert Word to Doubleword	1001 1001
CWDE - Convert Word to Doubleword	1001 1000
DEC – Decrement by 1	
registre	1111 111w : 11 001 reg
registre (autre forme de codage)	0100 1 reg
mémoire	1111 111w : mod 001 r/m
DIV – Unsigned Divide	
AL, AX ou EAX par un registre	1111 011w : 11 110 reg
AL, AX ou EAX par une donnée en mémoire	1111 011w : mod 110 r/m
ENTER – Make stack frame for function	1100 1000 : deplacement sur 16 bits : niveau 8 bits
IDIV – Signed Divide	
AL, AX ou EAX par un registre	1111 011w : 11 111 reg
AL, AX ou EAX par une donnée en mémoire	1111 011w : mod 111 r/m
IMUL – Signed Multiply	
AL, AX ou EAX par un registre	1111 011w : 11 101 reg
AL, AX ou EAX par une donnée en mémoire	1111 011w : mod 101 r/m
registre1 par registre2	0000 1111 : 1010 1111 : 11 reg1 reg2
registre par mémoire	0000 1111 : 1010 1111 : mod reg r/m
registre1 par donnée immédiate vers registre2	0110 10s1 : 11 reg1 reg2 : donnée immédiate
mémoire par donnée immédiate vers registre	0110 10s1 : mod reg r/m : donnée immédiate
INC – Increment by 1	
registre	1111 111w : 11 000 reg

registre (autre codage) mémoire	0100 0 reg 1111 111w : mod 000 r/m
Jcc – Jump if Condition is Met déplacement sur 8 bit déplacement sur 32 bits	 0111 tttn : déplacement sur 8 bits 0000 1111 : 1000 tttn : déplacement sur 32 bits
JMP – Unconditional Jump court direct indirect via un registre indirect via un relais en mémoire	 1110 1011 : déplacement sur 8 bits 1110 1001 : déplacement sur 32 bits 1111 1111 : 11 100 reg 1111 1111 : mod 100 r/m
LAHF – Load Flags into AH Register	1001 1111
LEA – Load Effective Address	1000 1101 : mod reg r/m
LEAVE – High Level Function Exit	1100 1001
LOCK – Assert LOCK #Signal Prefix	1111 0000
LOOP – Loop Count	1110 0010 : déplacement sur 8 bits
LOOPZ / LOOPE – Loop Count while Zero / Egal	1110 0001 : déplacement sur 8 bits
LOOPNZ / LOOPNE – Loop Count while not Zero / Egal	1110 0000 : déplacement sur 8 bits
MOV – Move Data registre1 vers registre2 registre2 vers registre1 mémoire vers registre registre vers mémoire donnée immédiate vers registre donnée immédiate vers registre (variante) donnée immédiate vers mémoire mémoire vers AL,AX ou EAX AL, AX, EAX vers mémoire	 1000 100w : 11 reg1 reg2 1000 101w : 11 reg1 reg2 1000 101w : mod reg r/m 1000 100w : mod reg r/m 1100 O11w : 11 000 reg : donnée immédiate 1011 wreg : donnée immédiate 1100 O11w : mod 000 r/m : donnée immédiate 1010 000w : déplacement sur 32 bits 1010 001w : déplacement sur 32 bits
MOVSX- Move with Sign-Extend registre2 vers registre1 mémoire vers registre	 0000 1111 : 1011 111w : 11 reg1 reg2 0000 1111 : 1011 111w : mod reg r/m
MOVZX- Move with Zero-Extend registre2 vers registre1 mémoire vers registre	 0000 1111 : 1011 011w : 11 reg1 reg2 0000 1111 : 1011 011w : mod reg r/m

MUL – Unsigned Multiply	
AL, AX, ou EAX avec un registre	1111 011w : 11 100 reg
AL, AX, ou EAX avec une mémoire	1111 011w : mod 100 reg
NEG – Two's Complement Negation	
Registre	1111 011w : 11 011 reg
mémoire	1111 011w : mod 011 r/m
NOP – No Operation	1001 0000
N0T – One's Complement Negation	
Registre	1111 011w : 11 010 reg
mémoire	1111 011w : mod 010 r/m
OR – Logical Inclusive OR	
registre1 vers registre2	0000 100w : 11 reg1 reg2
registre2 vers registre1	0000 101w : 11 reg1 reg2
mémoire vers registre	0000 101w : mod reg r/m
registre vers mémoire	0000 100w : mod reg r/m
donnée immédiate vers registre	1000 O0sw : 11 001 reg : donnée immédiate
donnée immédiate vers AL, AX, EAX	0000 110w : donnée immédiate
donnée immédiate vers mémoire	1000 O0sw : mod 001 r/m : donnée immédiate
POP – Pop a Word from the Stack	
registre	1000 1111 : 11 000 reg
registre (autre codage)	0101 1reg
mémoire	1000 1111 : mod 000 r/m
POPA/POPAD – Pop All general Registers	0110 0001
PUSH – Push Word to the Stack	
registre	1111 1111 : 11 110 reg
registre (autre codage)	0101 0reg
mémoire	1111 1111 : mod 110 r/m
donné immédiate	0110 10s0 : donnée immédiate
PUSHA / PUSHAD – Push All General registers	0110 0000
PUSHF / PUSHFD – Push Flags registers	1001 1100
RCL – Rotate through Carry Left	
registre de 1 position	1101 000w : 11 010 reg
mémoire de 1 position	1101 000w : mod 010 r/m
registre du nombre de positions	1101 001w : 11 010 reg

contenu dans CL	
mémoire du nombre de positions contenu dans CL	1101 001w : mod 010 r/m
registre du nombre de positions donné	1100 000w : 11 010 reg : donnée immédiate sur 8 bits
mémoire du nombre de positions donné	1100 000w : mod 010 r/m : donnée immédiate sur 8 bits
RCR– Rotate through Carry Right	
registre de 1 position	1101 000w : 11 011 reg
mémoire de 1 position	1101 000w : mod 011 r/m
registre du nombre de positions contenu dans CL	1101 001w : 11 011 reg
mémoire du nombre de positions contenu dans CL	1101 001w : mod 011 r/m
registre du nombre de positions donné	1100 000w : 11 011 reg : donnée immédiate sur 8 bits
mémoire du nombre de positions donné	1100 000w : mod 011 r/m : donnée immédiate sur 8 bits
RET – return from function	
Pas d'argument	1100 0011
Ajout d'une valeur immédiate à SP	1100 0010 : valeur immédiate sur 16 bits
ROL – Rotate Left	
registre de 1 position	1101 000w : 11 000 reg
mémoire de 1 position	1101 000w : mod 000 r/m
registre du nombre de positions contenu dans CL	1101 001w : 11 000 reg
mémoire du nombre de positions contenu dans CL	1101 001w : mod 000 r/m
registre du nombre de positions donné	1100 000w : 11 000 reg : donnée immédiate sur 8 bits
mémoire du nombre de positions donné	1100 000w : mod 000 r/m : donnée immédiate sur 8 bits
ROR – Rotate Right	
registre de 1 position	1101 000w : 11 001 reg
mémoire de 1 position	1101 000w : mod 001 r/m
registre du nombre de positions contenu dans CL	1101 001w : 11 001 reg
mémoire du nombre de positions contenu dans CL	1101 001w : mod 001 r/m

registre du nombre de positions donné	1100 000w : 11 001 reg : donnée immédiate sur 8 bits
mémoire du nombre de positions donné	1100 000w : mod 001 r/m : donnée immédiate sur 8 bits
SAHF – Store AH into Flags	1001 1110
SAL – Shift Arithmetic Left	même instruction que SHL
SAR – Shift Arithmetic Right	
registre de 1 position	1101 000w : 11 111 reg
mémoire de 1 position	1101 000w : mod 111 r/m
registre du nombre de positions contenu dans CL	1101 001w : 11 111 reg
mémoire du nombre de positions contenu dans CL	1101 001w : mod 111 r/m
registre du nombre de positions donné	1100 000w : 11 111 reg : donnée immédiate sur 8 bits
mémoire du nombre de positions donné	1100 000w : mod 111 r/m : donnée immédiate sur 8 bits
SBB – Integer Substraction with Borrow	
registre1 avec registre2	0001 100w : 11 reg1 reg2
registre2 avec registre1	0001 101w : 11 reg1 reg2
mémoire avec registre	0001 101w : mod reg r/m
registre avec mémoire	0001 100w : mod reg r/m
donnée immédiate avec registre	1000 00sw : 11 011 reg : donnée immédiate
donnée immédiate avec AL, AX, EAX	0001 110w : donnée immédiate
donnée immédiate avec mémoire	1000 00sw : mod 011 r/m : donnée immédiate
SETcc – Byte Set on condition	
Registre	0000 1111 : 1001 tttn : 11 000 reg
mémoire	0000 1111 : 1001 tttn : mod 000 r/m
SHL – Shift Left	
registre de 1 position	1101 000w : 11 100 reg
mémoire de 1 position	1101 000w : mod 100 r/m
registre du nombre de positions contenu dans CL	1101 001w : 11 100 reg
mémoire du nombre de positions contenu dans CL	1101 001w : mod 100 r/m
registre du nombre de positions donné	1100 000w : 11 100 reg : donnée immédiate sur 8 bits
mémoire du nombre de positions	1100 000w : mod 100 r/m : donnée immédiate

donné	sur 8 bits
SHLD – Double precision Shift Left	
registre du nombre de positions donné	0000 1111 : 1010 0100 : 11 reg2 reg1 : donnée immédiate sur 8 bits
mémoire du nombre de positions donné	0000 1111 : 1010 0100 : mod reg r/m : donnée immédiate sur 8 bits
registre du nombre de positions contenu dans CL	0000 1111 : 1010 0101 : 11 reg2 reg1
mémoire du nombre de positions contenu dans CL	0000 1111 : 1010 0101 : mod reg r/m
SHR – Shift Right	
registre de 1 position	1101 000w : 11 101 reg
mémoire de 1 position	1101 000w : mod 101 r/m
registre du nombre de positions contenu dans CL	1101 001w : 11 101 reg
mémoire du nombre de positions contenu dans CL	1101 001w : mod 101 r/m
registre du nombre de positions donné	1100 000w : 11 101 reg : donnée immédiate sur 8 bits
mémoire du nombre de positions donné	1100 000w : mod 101 r/m : donnée immédiate sur 8 bits
SHRD – Double precision Shift Right	
registre du nombre de positions donné	0000 1111 : 1010 1100 : 11 reg2 reg1 : donnée immédiate sur 8 bits
mémoire du nombre de positions donné	0000 1111 : 1010 1100 : mod reg r/m : donnée immédiate sur 8 bits
registre du nombre de positions contenu dans CL	0000 1111 : 1010 1101 : 11 reg2 reg1
mémoire du nombre de positions contenu dans CL	0000 1111 : 1010 1101 : mod reg r/m
SUB – Integer Substraction	
registre1 avec registre2	0010 100w : 11 reg1 reg2
registre2 avec registre1	0010 101w : 11 reg1 reg2
mémoire avec registre	0010 101w : mod reg r/m
registre avec mémoire	0010 100w : mod reg r/m
donnée immédiate avec registre	1000 00sw : 11 101 reg : donnée immédiate
donnée immédiate avec AL, AX, EAX	0010 110w : donnée immédiate
donnée immédiate avec mémoire	1000 00sw : mod 101 r/m : donnée immédiate
TEST – Logical Compare	

registre1 avec registre2	1000 010w : 11 reg1 reg2
mémoire avec registre	1000 010w : mod reg r/m
donnée immédiate avec registre	1111 011w : 11 000 reg : donnée immédiate
donnée immédiate avec AL, AX, EAX	1010 100w : donnée immédiate
donnée immédiate avec mémoire	1111 011w : mod 000 r/m : donnée immédiate
XADD – Exchange and Add	
registre1 avec registre2	0000 1111 : 1100 000w : 11 reg1 reg2
mémoire avec registre	0000 1111 : 1100 000w : mod reg r/m
XCHG – Exchange R/M with Register	
registre1 avec registre2	1000 011w : 11 reg1 reg2
AL, AX ou EAX avec reg	1001 0reg
mémoire avec registre	1000 011w : mod reg r/m
XLAT / XLATB – Table Look up Translation	1101 0111
XOR – Logical Exclusive OR	
registre1 vers registre2	0011 000w : 11 reg1 reg2
registre2 vers registre1	0011 001w : 11 reg1 reg2
mémoire vers registre	0011 001w : mod reg r/m
registre vers mémoire	0011 000w : mod reg r/m
donnée immédiate vers registre	1000 O0sw : 11 110 reg : donnée immédiate
donnée immédiate vers AL, AX, EAX	0011 010w : donnée immédiate
donnée immédiate vers mémoire	1000 O0sw : mod 110 r/m : donnée immédiate

Annexe 9 : Format de l’octet ModR/M

La table donne dans sa première colonne intitulée *Adresse Effective* la liste des 24 modes d’adressage de la mémoire. Chacune des valeurs du champ *R/M* permet de coder trois registre. Le registre effectivement utilisé dépend du préfixe pour choisir entre EAX et AX et du code opération, ainsi que de l’attribut de taille pour choisir entre (EAX, AX) et AL. Les seconde et troisième colonnes donnent la valeur des champs *Mod* et *R/M*.

Les premières lignes de la table donnent la signification du champ *Reg/Opcode*. La ligne « *Reg* = » donne la valeur des trois bits du champ servant à désigner le second opérande qui doit être un registre. Si l’instruction n’a pas de second opérande ces trois bits servent à l’extension du code opération de l’instruction.Le corps de la table donne en hexadécimal les différentes valeurs permises pour l’octet *ModR/M*. Les bits 3, 4 et 5 de l’octet donnent le numéro de colonne dans la table et les bits restants : 0, 1, 2, 6, 7 le numéro de la ligne de la table. Ainsi l’octet 26 correspond à l’élément en septième ligne cinquième colonne (6,4) de la table.

ANNEXES

R8			AL	CL	DL	BL	AH	CH	DH	BH
R16			AX	CX	DX	BX	SP	BP	SI	DI
R32			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
Code opération			0	1	2	3	4	5	6	7
REG =			000	001	010	011	100	101	110	111
Adresse effective	**Mod**	**R/M**	**Valeur de l'octet ModR/M en hexadécimal**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--]		100	04	0C	14	1C	24	2C	34	3C
Depl32		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
Disp8[EAX]	01	000	40	48	50	58	60	68	70	78
Disp8[ECX]		001	41	49	51	59	61	69	71	79
Disp8[EDX]		010	42	4A	52	5A	62	6A	72	7A
Disp8[EBX]		011	43	4B	53	5B	63	6B	73	7B
Disp8[--][--]		100	44	4C	54	5C	64	6C	74	7C
Disp8[EBP]		101	45	4D	55	5D	65	6D	75	7D
Disp8[ESI]		110	46	4E	56	5E	66	6E	76	7E
Disp8[EDI]		111	47	4F	57	5F	67	6F	77	7F
Disp32[EAX]	10	000	80	88	90	98	A0	A8	B0	B8
Disp32[ECX]		001	81	89	91	99	A1	A9	B1	B9
Disp32[EDX]		010	82	8A	92	9A	A2	AA	B2	BA
Disp32[EBX]		011	83	8B	93	9B	A3	AB	B3	BB
Disp32[--][--]		100	84	8C	94	9C	A4	AC	B4	BC
Disp32[EBP]		101	85	8D	95	9D	A5	AD	B5	BD
Disp32[ESI]		110	86	8E	96	9E	A6	AE	B6	BE
Disp32[EDI]		111	87	8F	97	9F	A7	AF	B7	BF

Annexe 10 : Format de l'octet SIB

Cette table donne l'interprétation des 256 valeurs possibles de l'octet *SIB*. Elle spécifie donc lequel des registres est utilisé comme registre de base et lequel est utilisé comme index. On a quatre groupes de codage selon le facteur multiplicatif qui doit s'appliquer à l'index avant le calcul de l'adresse effective.

R32			EAX	ECX	EDX	EBX	ESP	[*][6]	ESI	EDI
Base =			0	1	2	3	4	5	6	7
Base =			000	001	010	011	100	101	110	111
Index typé	**SS**	**Index**	**Valeur de l'octet SIB en hexadécimal**							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
AUCUN		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
AUCUN		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	99	9A	9B	9C	9D	9E	9F
AUCUN		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7

[6] [*] signifie un déplacement codé sur 32 bits sans base si le champ Mod vaut 00 et [EBP] sinon. Cela fournit donc les modes d'adressage disp32[,index], disp8[EBP][index] (Mod = 01) et disp32[EBP][index] (Mod = 10)

[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
AUCUN		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

Annexe 11 : Codage du champ tttn spécifiant le test à effectuer

tttn	Mnémonique	Condition
0000	O	Overflow
0001	NO	No overflow
0010	B, NAE	Below, Not above or equal
0011	NB, AE	Not below, Above or equal
0100	E, Z	Equal, Zero
0101	NE, NZ	Not equal, Not zero
0110	BE, NA	Below or equal, Not above
0111	NBE, A	Not below or equal, Above
1000	S	Sign
1001	NS	Not sign
1010	P, PE	Parity, Parity Even
1011	NP PO	Not parity, Parity Odd
1100	L, NGE	Less than, Not greater than or equal to
1101	NL, GE	Not less than, Greeater than or equal to
1110	LE, NG	Less than or equal, Not greater than
1111	NLE,G	Not less than or equal, Greater than