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

Ensimag et
Section Télécom commune à l'Ensimag et l'Erg

Première année.

Brouillon : vendredi 13 octobre 2000

X. Rousset de Pina

Cette documentation présente un sous ensemble des possibilités offertes par l'assembleur Gnu[1] disponible sous Linux et dont on trouvera la description complète dans le document *Using as* http://www.gnu.org/manual/gas-2.9.1/as.html.

Sommaire

[1] Le projet GNU a été lancé en 1984 pour développer un système, le système GNU, dont le logiciel soit distribué librement et gratuitement et qui soit semblable au système Unix complet.

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 pour ensuite passer aux 80186/80188 (machine à mots de 16 bits capable d'adresser 1Mo de mémoire) 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 saurait dispenser de la consultation des documents originaux qui ont servi à sa rédaction et dont les références sont récapitulées en annexe.

Le modèle de programmation fourni

Les instructions exécutées et les données qu'elles manipulent sont dans deux espaces mémoire séparés dont chacun peut être considéré comme un tableau de 4 Gigaoctet dont l'index va de la valeur 0 à la valeur $(2^{32} – 1)$.
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 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 double mot de 32 bits. L'adresse d'un mot ou d'un double mot est celle de son premier octet. 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 d'instruction 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 allons décrire dans la fin de cette section avant de nous intéresser dans la section suivante aux différents modes que peut utiliser le processeur pour désigner les opérandes de l'opération qu'il exécute.

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.

Notation	31 16	15 0	Nom usuel
EAX		AX	Accu.
		AH | AL	
EBX		BX	Base Index
		BH | BL	
ECX		CX	Count
		CH | CL	
EDX		DX	Data
		DH | DL	
ESP			Stack Pt.
EBP			Base Pt.
EDI		DI	Dest. Index
ESI		SI	Source Index
EIP			Instruction Pt.
EFLAGS		FLAGS	Flag

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 des 32 bits sera éventuellement modifiée, la partie restante ne sera donc pas concernée par l'opération. Pour les instructions

arithmétiques et certaines autres instructions 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. Il est utilisable comme EBX, BX, BH et BL. EBX peut, dans certaines instructions contenir l'adresse d'un emplacement en mémoire programme. Il peut également servir à adresser les données.
- **ECX.** C'est le registre qui sert de compteur dans beaucoup d'instructions (les instructions de répétition d'une opération sur les chaînes utilisent CX, les déplacements et les rotations utilisent CL, les boucles utilisent ECX ou CX). Il peut être également utilisé pour adresser des données en mémoire.
- **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 les données.
- **EBP.** Ce registre contient toujours une adresse en mémoire pour y transférer des données. EBP est utilisé comme nous le verrons plus loin comme base d'adressage des variables locales et des paramètres d'une fonction en cours d'exécution.
- **EDI.** Utilisable comme registre général 16 bits (DI) ou 32 bits (EDI) . EDI contient l'adresse destination dans les instructions manipulant des chaînes.
- **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 l'adresse d'une zone de mémoire gérée en pile.
- **EFLAGS.** Ce registre est le registre d'état du microprocesseur. Nous ne décrivons que les indicateurs que nous utiliserons :

31…… ..12	**11**	**10**			**7**	**6**		**4**				**0**
	O	**D**			**S**	**Z**		**A**				**C**

Figure 3 Quelques indicateurs utiles du registre EFLAGS

- **C.** C'est le bit 0 du registre EFLAGS. Il contient la retenue après une addition ou une soustraction.
- **A.** C'est le bit 4 du registre EFLAGS. Il contient le report observé entre les bits 3 et 4 du résultat pour une addition ou une soustraction. Il n'est utilisé que les opération DAA et DAS qui opèrent sur des nombre décimaux codés binaires.
- **Z.** C'est le bit 6 du registre EFLAGS. Il contient 1 si le résultat d'une opération arithmétique ou logique est nul et 0 sinon.

- **S.** C'est le bit 7 du registre EFLAGS. Il prend la valeur du bit de signe du résultat après l'exécution d'une opération arithmétique ou logique.
- **D.** C'est le bit 10 du registre EFLAGS. Il sélectionne le mode ajout ou diminution pour DI et SI pendant les instructions sur des chaînes. Il est positionné par l'instruction STD et effacé avec l'instruction CLD. Si D vaut 1 les registres sont diminués automatiquement et augmentés automatiquement si non.
- **O.** C'est le bit 11 du registre EFLAGS. Il indique un débordement après une opération sur des valeurs signées. Pour des opérations sur des valeurs non signées il est ignoré.

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 la mémoire des données ou dans celle des instructions. On appelle mode d'adressage d'un opérande, la façon dont le processeur calcule 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 mais de nom. S'il s'agit d'un opérande en mémoire de données ou de programme l'adresse sera la valeur de l'index auquel se trouve cet opérande dans le tableau correspondant. En fait les modes de calcul de l'adresse d'un opérande sont différents selon que cet opérande appartient à la mémoire des données ou à celle des instructions. Nous allons donc présenter séparément les modes d'adressages disponibles pour la mémoire de données et pour celle d'instructions.

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 respectivement de copier 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 toute les façons de calculer soit la source soit la destination.

Adressage registre direct

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 ou les deux opérandes sont adressés dans le mode registre direct les deux registres 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 long sur 32 bits).
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 */
```

Adressage immédiat

Dans ce mode c'est la valeur effective de l'opérande qui est fournie dans l'instruction. Une valeur immédiate est toujours précédée en assembleur 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 /* affecte à eax la 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 ou une adresse dans les données ou dans le code. Dans le premier cas la valeur de *toto* est connue à l'assemblage dans le second elle n'est connue qu'au chargement.
Attention : N'oubliez pas le caractère **$** devant l'opérande immédiat sous peine de le voir mal interprété et notamment comme un opérande adressé directment.

Adressage direct

Dans ce mode c'est l'adresse de l'opérande dans la mémoire de 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 de 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 des instructions suivantes

```
movb 0x80483c3,%al      /* affecte 0xaa au registre octet al */
movw 0x80483c3,%ax      /* affecte 0xbbaa au registre mot ax */
movl 0x80483c3,%eax     /* affecte 0xddccbbaa au registre eax */
```

Adressage indirect registre

Dans ce mode l'adresse de l'opérande est contenue dans le registre. La syntaxe assembleur 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 plus avant dans ce document. 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 EAX, les instructions `movb (%ebx),%al / movl (%ebx),%ax / movl (%ebx),%eax` donnent les mêmes résultats que les trois instructions de l'exemple de la sous section précédente.

Adressage avec base et déplacement

Comme précédemment ce mode d'adressage permet d'adresser indirectement la mémoire. Il fait intervenir un registre, appelé registre de base, et une constante signée, 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 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 de 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 précédente.
Ce mode d'adressage facilite en assembleur l'utilisation de structures de données à la C puisque chaque champ de la structure est à un déplacement fixe de son début.

Adressage avec base, déplacement et index

Ce mode d'adressage fait intervenir deux registres et une valeur constante signée. La syntaxe assembleur de ce mode d'adressage est :

Dep(%REGISTER1, %REGISTER2)

où Dep est une constante signée sur 4 octets, REGISTER1 et REGISTER2 sont l'un des registres EAX, EBX, ECX, EBP, EDX, EDI et ESI. Comme précédemment EBP est utilisé de préférence pour adresser les données 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 4 octets.

Ce mode d'adressage permet d'adresser facilement des tableaux qu'ils soient ou non inclus dans des structures. Pour le réaliser il suffit de prendre pour Dep le déplacement 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.

Exemple :

str est une structure qui, à un déplacement de TAB, contient un tableau de mots de 16 bits dont on veut lire la valeur du 5eme élément dans le registre BX.

```
movl $str, %eax  /* eax = adresse de base de la structure str */
movl $10,%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 */
```

Adressage avec base, déplacement et index typé

Ce mode d'adressage fait intervenir deux registres REGISTER1 et REGISTER2 pris parmi EAX, EBX, ECX, EDX, EDI, EBP et ESI, un déplacement Dep qui une constante signée sur 32 bits 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.

Exemple :

Avec ce mode d'adressage l'exemple précédent s'écrira :

```
movl $str, %eax  /* eax = adresse de base de la structure str */
movl $5,%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 */
```

Les modes d'adressage de la mémoire d'instructions

Deux types d'instructions ont besoin d'adresser la mémoire d'instructions, 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 d'instructions 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 long mot de la mémoire de données.

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 générera une instruction de saut avec un mode d'adressage relatif au pointeur d'instruction.

Exemple :

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

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 */

Adressage indirect

Il y a en comme nous l'avons déjà mentionné deux formes :

1. Adressage indirect dont le relais d'indirection 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 d'indirection est un mot de la mémoire de données. La syntaxe utilisée pour <dest> est cette fois du type :

   *Dep(%REG) -- REG est pris dans EBX, ECX, EDX, ESP, EBP, EDI, ESI. Ou du type

   *(%REG) si le champ Dep est nul et que REG contient l'adresse du relais d'indirection en mémoire.

   *AdRelai

   Le mot de la mémoire de données, soit à l'adresse *AdRelai*, soit à l'adresse obtenue en sommant *Dep* avec la valeur courante de *REG*, contient l'adresse, dans la mémoire d'instructions, 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 */
     movl $adIci, %eax    /* eax = adresse relais d'indirection */
     jmp (%eax)           /* remonte a Ici */
     xorl %eax,%eax
     jmp  *adIci(%eax)    /* remonte à Ici */
     jmp  *adIci          / remonte à Ici
```

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 en complément à 2 sur 32 bits.

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 hexadécimale
8	0 .. 255	FF

16	0 .. 65535	FFFF
32	0.. 4 294 967 295	FFFFFFFF

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 $x = val(r) - 2^n$

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

Remarques

1. Avec la représentation en complément à deux la représentation de –1 est toujours un nombre dont tous les bits sont égaux à 1.
2. 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 sur 8 bits par 81 et sur 16 bits par FF81.
3. L'opération inverse de la précédente 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.

Représentation des entiers en mémoires

Entier sur un octet

Comme l'indique la figure (Figure 4 Poids des bits d'un octet) 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

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 (Figure 5 Représentation d'un entier sur un mot de 16 bits).

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

Entier sur un double mot

L'adresse du double mot est celle du mot de poids faible qui précède le mot de poids fort (Figure 6 Représentation d'un entier sur un mot de 32 bits). 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

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

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.

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
```

Les instructions machines.

Elles ont la forme suivante :

[Etiquette] [operation] [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.

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 « : ». 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.

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

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 (code données) dans laquelle elle est définie et est égale à son adresse d'implantation dans la mémoire (données ou instructions) dans laquelle elle est définie. 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
```

Le champ opération

Pour les opérations pouvant opérer sur plusieurs tailles d'opérande, 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 mot de 32 bits, un mot de 16 bits ou un caractère. Pour les autres opérations le champ opération porte le nom donné par le constructeur sans suffixe.

Pour les opérations ayant deux opérandes, sauf dans le cas d'opérations sur les chaînes, l'un des deux opérande 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 d'instructions : *%R où R est le nom de l'un des registres permis.
- Indirect avec déplacement de la mémoire de données : D(%R) où R est le nom de l'un des registres 32 bits permis et D la valeur du déplacement.
- Indirect mémoire, avec adresse directe du relais. On le note *adIci , si adIci est une étiquette définie dans la mémoire de données, ou *(%R) si le déplacement est nul.
- 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.
- 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.

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 instruction machines. Une directive est toujours précédée d'un caractère « . » (point). Nous décrivons les principale directives disponibles dans la section suivante.

Les Principales directives d'assemblage

Nous allons distinguer trois familles de directives : les directives de sectionnement du programme et les directives de définition de données et constantes et les autres. Après avoir décrit ces trois familles nous donnons un exemple de programme complet.

Les directives de sectionnement

Nous avons vu que les microprocesseurs auxquels nous nous intéressions adressent deux mémoires séparées une où sont implantées les instructions, l'autre où sont implantées les données. Nous

verrons quand nous traiterons la définition des données que l'on peut encore distinguer deux types de données celle qui ont une valeur initiale défini statiquement par le programmeur et celles pour lesquelles la valeur ne sera définie qu'à l'exécution. Pour l'instant nous ne nous intéressons qu'au directives permettant de distinguer données et code.

.text [sous-section]

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

.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.

.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 et non exécutable.

Les directives de définition de données

Comme nous l'avons signalé plus haut on distingue les données initialisées des données non initialisées. La raison de ceci est que les données non initialisée ne sont pas réellement générés par l'assembleur mais par le chargeur, cela réduit donc la taille des fichier objets générés par l'assembleur.

Déclaration des données non initialisées : .lcomm nom, taille

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). Les données non initialisées sont allouées à la suite de toutes les données initialisées dans une zone qui est mise à zéro.

Déclaration de données initialisées

L'assembleur permet de déclarer 5 types de données initialisée : des octets, des mots de 16 bits, des doubles mots de 32 bits, des quadruples mots de 64 bits et des chaînes de caractères.

- **[étiquette] .byte expressions**

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**

  expressions est une suite comprenant 0 ou plusieurs expressions séparées par des virgules. Chaque expression est assemblée dans le mot de 16 bits suivant.

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, 0xfffff
```

- **[étiquette] .int expressions**.

  expressions est une suite comprenant 0 ou plusieurs expressions séparées par des virgules. Chaque expression est assemblée dans le mot de 32 bits suivant.

Exemple :

La ligne suivante permet de réserver un tableau de 4 doubles mots et de les initialiser avec les adresses des sous programmes 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**

  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 faibles. Un message de mise en garde est imprimé.

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] .string str**

  str est une suite de caractère entre des caractères « " » « double quote ». Comme en C, pour inclure dans la chaîne des caractères spéciaux (en particulier, le caractère double quote, mais

aussi le retour chariot, le « new line », le « form feed » etc.), on les fait précéder du caractère « \ ». 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.

Exemple :

La ligne suivante permet de réserver une chaîne de caractères désignée par *format*.

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

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

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
```

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.

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. 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 dispose 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.

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 octet 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.

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
      movw y,%ax
      pushl %eax
      movw x,%ax
      pushl %eax
      pushl $format
      call printf
      addl $12,%esp

      leave       # on rend le contrôle à l'appelant
      ret
```

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 vu précédemment. Pour le retour au programme appelant on dispose de l'instruction *ret*. Ces instructions utilisent implicitement une zone de mémoire adressé 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.

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 suivante (Figure 8 Effet de pushl sur la pile) illustre l'effet de l'instruction :

```
pusl %ebp
```

sur la pile (zone mémoire adressée par ESP).

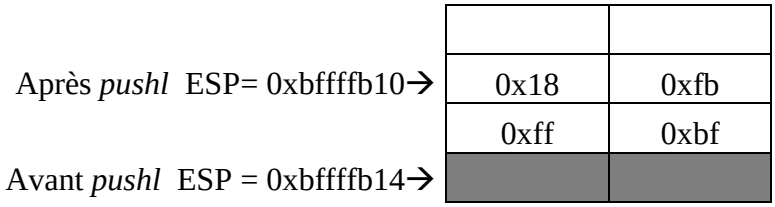


Figure 8 Effet de pushl sur la pile

La figure suivante (Figure 9 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 :

pusl %ebp

précédente.

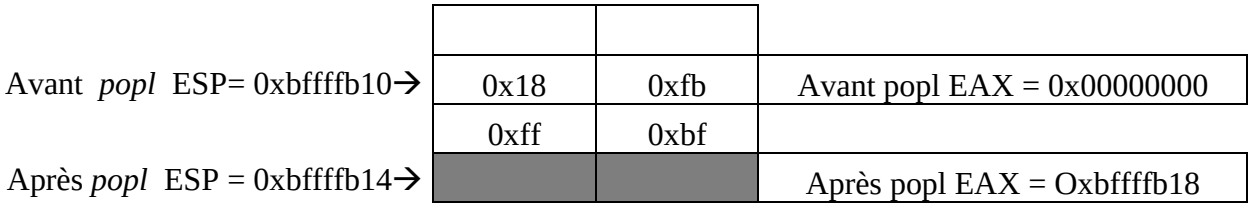


Figure 9 Effet du popl %eax sur EAX et sur la pile

La figure (Figure 10 Effet pushw sur la pile) illustre elle l'effet de l'instruction :

Pushw %bp

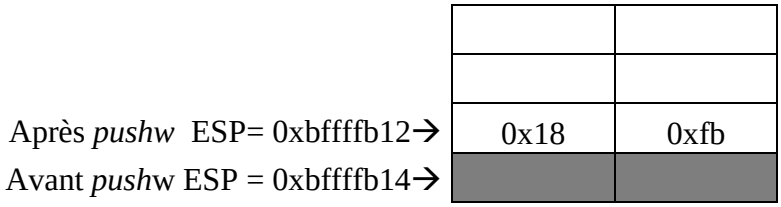


Figure 10 Effet pushw sur la pile

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.

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

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

Les machines Intel auxquelles nous nous intéressons 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 a une pile qui lui est associée et que ESP a été initialisé pour pointer sur son sommet. C'est effectivement le cas

pour tout programme s'exécutant sous Unix et dont l'exécutable à été fabriqué à l'aide de la commande *gcc*.

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

Cette instruction copie dans le pointeur d'instruction EIP le contenue 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 d'un dépilement d'un double mot et d'un branchement à l'adresse dépilée. Pour qu'il n'y ait pas d'erreur il faut être sûr que le double mot au sommet de pile pointe bien sur une instruction dans la mémoire d'instructions.

Passage de paramètres et gestion des variables locales

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 valeur des paramètres sur la pile si il veut que l'appelant puisse les exploiter. Avant l'appel d'une fonction $f(p_1,p_2, \ldots,p_n)$ les paramètres p_n, p_{n-1},…, p_1 seront rangés sur la pile et dans cette ordre.

On empile toujours multiple de 4 octets si on veut suivre les conventions du langage C.

- **Paramètre passé par adresse** : on empile l'adresse de la variable soit quatre octets
- **Paramètre passé par valeur de type simple** (entier, booléen, caractère, énumération etc.) on empile la valeur effective sur 4 octets cadrée à gauche

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, on préfère passer ces paramètres par variables (passer l'adresse de base de la structure). Cependant si on veut quand même les passer 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.

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.

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 appelé doit avant de modifier la valeur du EBP elle doit sauver cette valeur dans la pile. L'annexe 0 illustre ce qu'est le conteste d'un sous programme dans la pile lorsqu'il s'exécute.

Paramètres, variables locales et sauvegarde de la procédure P

Pour obtenir ce résultat 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
      addl  $-TVL_P,%esp      # alloue TVL_P octets sur la pile pour les variables locales
      pushad                  # sauve tous les registres 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 */
      popad                   # restaure la valeur des registres sauf celle de EBP
      leave                   # cette instruction libère les variables locales et restaure EBP
      ret                     # rend le contrôle à l'appelant
```

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

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 détruire EDX et ECX

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], pour ce qui est de la définition exhaustive des formats de données associés aux différents types de données à imprimer ou à lire.

Traduction en assembleur des principales structures de contrôle

<se reporter à Y. Rouzeau pour l'instant >

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

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.

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 définir les paramètres de ce mode.

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.l] 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, ajoutez l'indicateur '= ' immédiatement après ***a*** sans espace et suivi sans espace du nom du fichier où le listing doit être généré.

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 du fichier généré par l'assembleur il suffit d'appeler la commande

gcc file.o –o file

–o permet de donner un nom à l'exécutable.

La commande :

gcc –Wa, -a=file.l, file.s –o file

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

Mettre au point votre code avec gdb

Pour appeler gdb sur l'exécutable *file* :

gdb *file*

Pour mettre un point d'arrêt à l'entrée du programme :

br main

Pour exécuter son programme instruction par instruction

si

Pour exécuter son programme au pas à pas en sautant les appels de fonctions

ni

Pour visualiser tous les registres

i reg

Pour visualiser des registres particuliers , par exemple les registres *eax* et *ebx*

i reg eax ebx

Pour visualiser la mémoire

x/nfu addr

- *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*

Pour modifier un emplacement mémoire quelconque

set {type}addr = value

- *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

Pour modifier le contenu d'un registre

set $REG = value

- *REG* est le nom du registre que l'on veut modifier non précédé du caractère %.
- *value* est la valeur décimale, octale ou hexadécimale que l'on veut donner au registre.

Exemple

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.

Pour quitter le metteur au point :

quit

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.

[Bernard Cassagne]

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

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

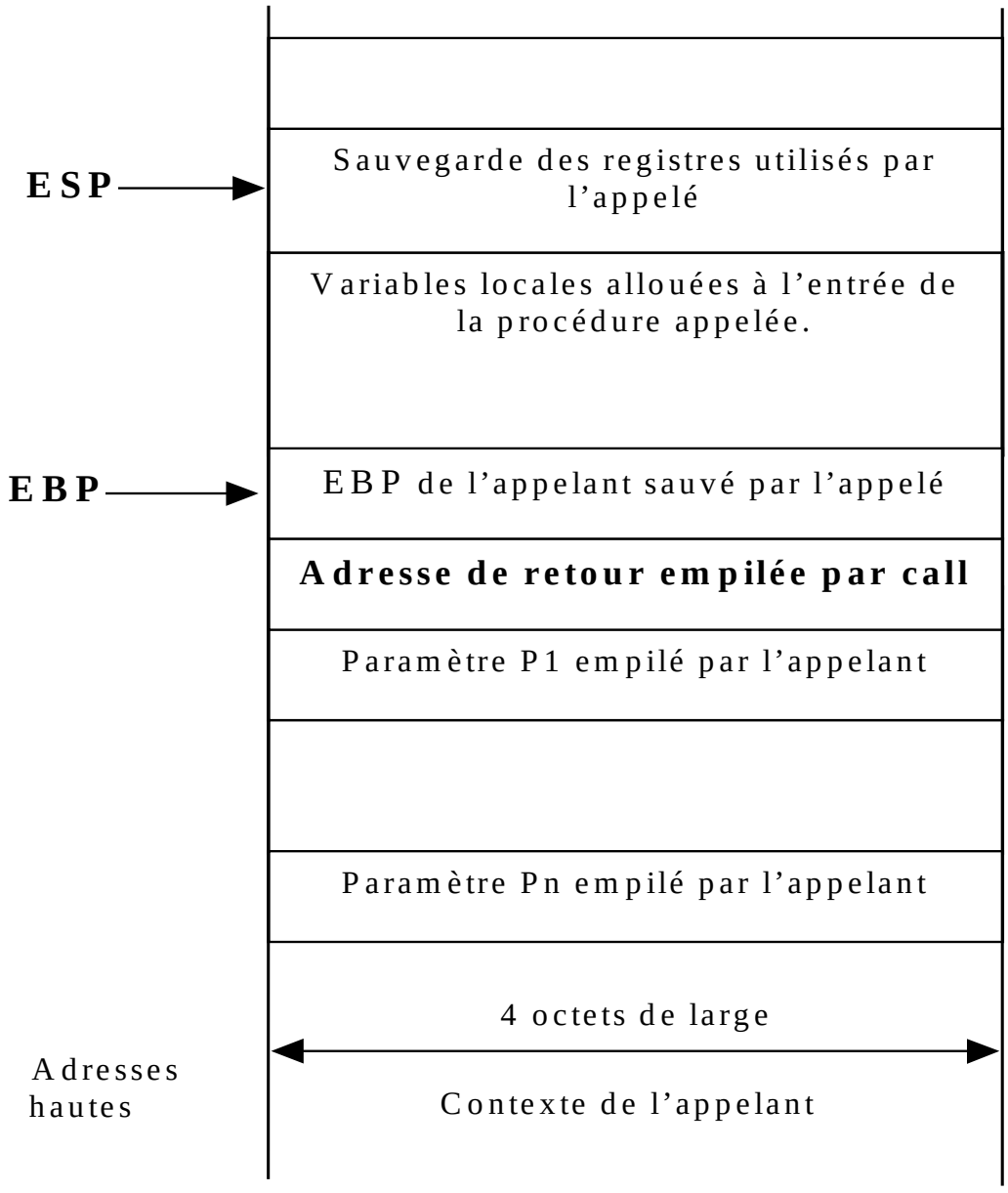
Debugging with GDB, 1998, Disponible au format PostScript à l'URL :
http://www.gnu.org/manual/gdb-4.17/gdb.html

[Yann Rouzaud 1991]

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

Annexe 0 :

Schémas d'un contexte de sous programme dans la pile



Annexe 1 :

Un programme de 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
main:
        pushl   %ebp           /* sauvegarde base frame appelant */
        movl    %esp,%ebp      /* base les var. locales et paramètres de l'appelé */
        pushal                 /* sauvegarde des registres */

        /* 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 */

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:
```

```
        le/* 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 2

Un programme de test des modes d'adressage de la mémoire d'instructions

```
   .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 3

Résultat de l'assemblage du programme calculant le pgcd

```
GAS LISTING pgcd.s

   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  # on remarque l'inversion des octets
  10 0002 0010          y:      .hword 4096 # poids faible suivi de poids fort
  11                            .text
  12                            .p2align 2  # alignement à une frontière de mot
  13                            .globl main # exporte le point d'entrée
  14
  15                    main:
  16 0000 55                    pushl %ebp  # Convention de liaison
  17 0001 89E5                  movl %esp,%ebp
  18
  19 0003 833D0000              cmpl $0,x   # if ((x==0)||(y==0)) aller imprimer
  19      000000
  20 000a 7447                  je fin
  21 000c 833D0200              cmpl $0,y
  21      000000
  22 0013 743E                  je fin
  23                            .p2align 4,,7 # aligne sur une frontière de 16
  24                    iter:
  25 0015 66A10000              movw x,%ax  # if (x != y)
  25      0000
  26 001b 663B0502              cmpw y,%ax
  26      000000
  27 0022 7502                  jne cont          # On continue
  28 0024 EB2D                  jmp fin           # si non on va imprimer x et y
  29                            .p2align 4,,7
  30                    cont:
  31 0026 66A10000              movw x,%ax  #if (x < y)
  31      0000
  32 002c 663B0502              cmpw y,%ax
  32      000000
  33 0033 7E0F                  jle .LL     # aller calculer  y = y -x
  34 0035 66A10200              movw y,%ax  # sinon calculer x = x - y
  34      0000
  35 003b 66290500              subw %ax,x
  35      000000
  36 0042 EBD1                  jmp iter
  37                            .p2align 4,,7
  38                    .LL:
  39 0044 66A10000              movw x,%ax
```

```
  39      0000
  40 004a 66290502            subw %ax,y
  40      000000
  41 0051 EBC2                jmp iter
  42
  43                          .p2align 4,,7
  44
  45                    fin:  /* impression de x et y */
  46 0053 31C0                xorl %eax,%eax
  47 0055 66A10200            movw y,%ax
  47      0000
  48 005b 50                  pushl %eax
  49 005c 66A10000            movw x,%ax
  49      0000
  50 0062 50                  pushl %eax
  51 0063 68000000            pushl $format
  51      00
  52 0068 E8FCFFFF            call printf
  52      FF
  53 006d 83C40C              addl $12,%esp
  54
  55 0070 C9                  leave
  56 0071 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:00000053 fin
              pgcd.s:24    .text:00000015 iter
              pgcd.s:30    .text:00000026 cont

UNDEFINED SYMBOLS
printf
```

Annexe 4

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
```