Cours 3 - CPU 80386

Halim Djerroud

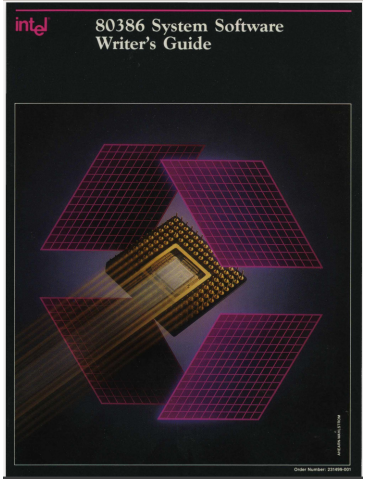
Université Gustave Eiffel
ESIEE
PARIS

révision : 1.0

Bibliographie

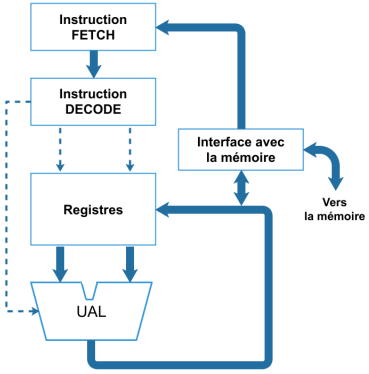
- 80386 programmer's reference manual
- GNU AS : `https://sourceware.org/binutils/docs/as/`

La plus part des illustration présentées dans ce document sont extraites du *80386 programmer's reference manual*



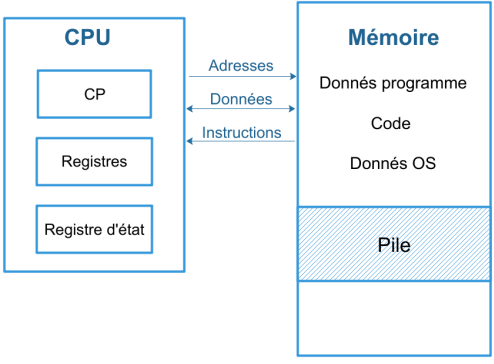
Architecture de 80386

- 3 phases, sous le pilotage de l'unité de contrôle (UC)
  1. **FETCH** : lecture en mémoire l'instruction
     - le CPU va lire l'instruction se trouvant à l'adresse PC
     - il la rapatrie dans le registre IR (Instruction Register)
     - il incrémente PC pour préparer l'instruction suivante
  2. **DECODE** : décodage de l'instruction
     - En fonction du code opération, l'unité de contrôle calcule :
       1. les chemins que vont devoir emprunter les données à la phase suivante
       2. les opérations qui leur seront (éventuellement) appliquées par l'UAL
  3. **EXECUTE** : exécution proprement dite de l'instruction
     - les données suivent les chemins (bus interne) calculés précédemment
     - l'UAL effectue l'opération demandée par l'unité de contrôle
- Puis le processeur reprend à la phase 1 pour l'instruction suivante



Architecture de 80386

- CP : Compteur de programme
  - Adresse de la prochaine instruction
  - e.g. EPI (32 bits), RIP (64 bits)
- Registres :
  - Très utilisés par les programmes
  - e.g. EAX, EBX, ESP...
- Registre d'état :
  - Statu des instruction récentes
  - Utilisé pour réaliser les branchement (SI... SINON ...)
- Mémoire :
  - Vue comme un tableau de case mémoires
  - Contient le code objet du programme, les données du programme, la pile, etc.



Architecture 80386

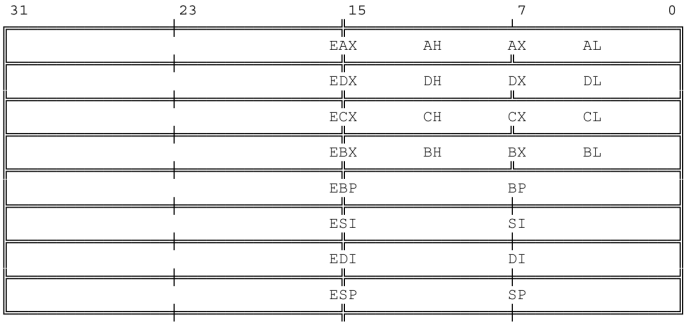
La machine répète tout le temps le étapes suivantes :

1. Récupérer l'instruction suivante de la mémoire (le PC indique la prochaine instruction)
2. Décoder l'instruction (dans l'unité de contrôle)
3. Exécuter l'instruction
4. Mettre à jour l'état de manière appropriée
5. Aller à l'étape 1

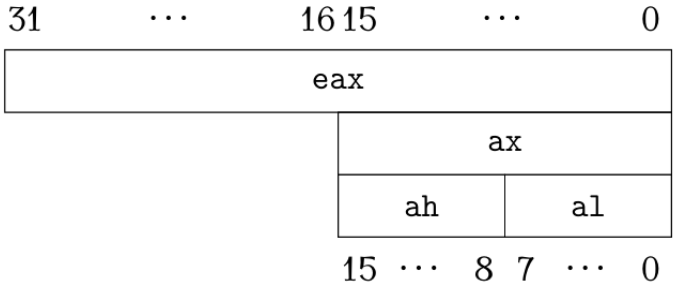
Les registres généraux

Chapitre 2 : Basic Programming Model

Les 80386 dispose de de plusieurs registres de travail. L'utilisateur peut utiliser soit un registre en entier ou seulement une partie du registre.



Subdivision du registre eax



Les registres généraux

- **EAX,AX,AH,AL :** Appelé aussi le registre Accumulateur. Il est utilisé pour l'accès au port d'E/S, l'arithmétique, les appels d'interruption, etc...
- **EBX,BX,BH,BL :** Appelé le registre de base ou registre d'index. Il est utilisé comme pointeur de base pour l'accès à la mémoire et aussi pour obtenir des valeurs de retour d'interruption.
- **ECX,CX,CH,CL :** Appelé le registre compteur. Il est utilisé comme compteur de boucle et pour les décalages et aussi pour obtenir des valeurs d'interruption.
- **EDX,DX,DH,DL :** Appelé le registre de données. Il est utilisé pour l'accès au port d'E/S, l'arithmétique, certaines interruptions. Ce registre est aussi utilisé pour certaines divisions.

```
32 bits : EAX EBX ECX EDX
16 bits : AX BX CX DX
 8 bits : AH AL BH BL CH CL DH DL
```

Les registres de la pile

- **EBP BP :** Registre de pointeur de base de pile. Contient l'adresse de base de la pile.
- **ESP SP :** registre de pointeur de pile. Contient l'adresse supérieure de la pile.
- **EDI SI :** Registre d'index source. Utilisé pour la copie de chaînes de caractères et de matrices.
- **EDI DI :** Registre d'index de destination. Utilisé pour les chaînes de caractères, la copie de matrices de mémoire et pour l'adressage de pointeurs lointains avec le registre ES.

Les registres segments

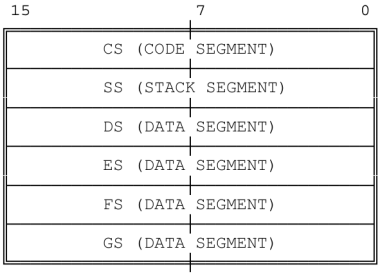


FIGURE – Les registres segments

Les registres segments

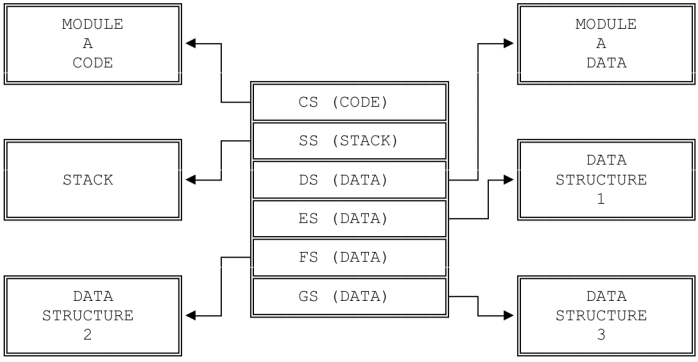


FIGURE – Les registres segments

Pourquoi utiliser des segments

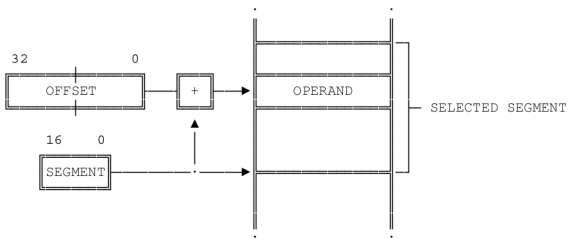


FIGURE – Pointeur à deux composants

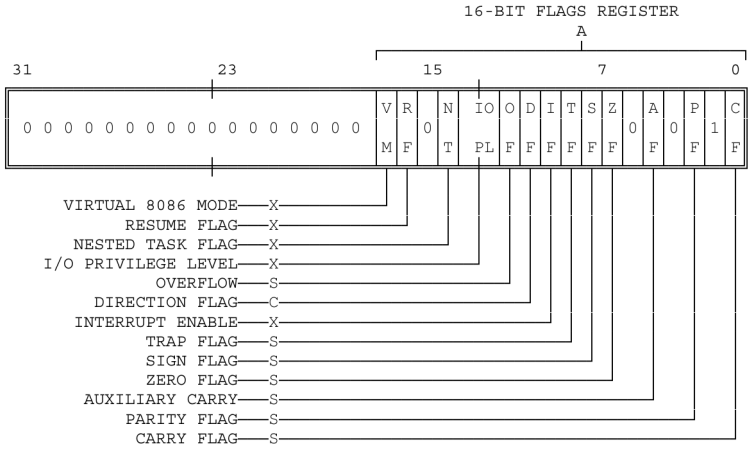
Les registres segments

- **CS (Code Segment) :** Contient le segment de code dans lequel le programme s'exécute
- **DS (Data Segment) :** Contient le segment de données auquel le programme accède
- **ES,FS,GS :** Ce sont des registres de segments supplémentaires disponibles pour l'adressage de pointeur lointain comme la mémoire vidéo par exemple
- **SS (Stack Segment)** Contient le segment de pile (Stack) utilisé par le programme. Il a parfois la même valeur que DS

Pas d'inquiétudes

Nous allons pas utiliser ces registres dans ce cours

Registre d'état



S = STATUS FLAG, C = CONTROL FLAG, X = SYSTEM FLAG

NOTE: 0 OR 1 INDICATES INTEL RESERVED. DO NOT DEFINE

Registre d'état

Le registre EFLAGS contient l'état du processeur. Il est modifié par de nombreuses instructions et est utilisé pour comparer certains paramètres, boucles conditionnelles et sauts conditionnels. Chaque bit contient l'état d'un paramètre spécifique de la dernière instruction.

Bit	Flag	Description
0	CF	Carry flag
2	PF	Parity flag
4	AF	Auxiliary carry flag
6	ZF	Zero flag
7	SF	Sign flag
8	TF	Trap flag
9	IF	Interrupt enable flag
10	DF	Direction flag
11	OF	Overflow flag
12-13	IOPL	I/O Priviledge level
14	NT	Nested task flag
16	RF	Resume flag
17	VM	Virtual 8086 mode flag
18	AC	Alignment check flag (486+)
19	VIF	Virutal interrupt flag
20	VIP	Virtual interrupt pending flag
21	ID	ID flag

Brève introduction à l'assembleur GNU

- L'assembleur utilisé dans ce cours : **GNU**, les instruction doit être écrite comme suit :

```
mov source, destination
```

- Les registres : une opérande qui indique un registre doivent être préfixé par un %.
  e.g. `%eax, %ax, %al, %ah, %ebx, ...`
- Les opérandes immédiats (constantes) : sont préfixées par un $.
  e.g. `$12, $0xFF0C5, $0b110011, $01247, $msg, ...`
- La taille des opérandes est codée dans le dernier caractère de l'instruction :
  b 8 bits (**b**`yte`), **w** 16 bits (**w**`ord`) et **l** 32 bits (**l**`ong`)

Type	Abréviation	Taille en octets	Taille en bits	Exemple
byte	b	1	8	al
word	w	2	16	ax
dword	l	4	32	eax

Introduction

Syntaxe assembleur :

- **AT&T**
  OPERATION SOURCE, DESTINATION
  Example :
  mov $123, %eax
- Intel
  OPERATION DESTINATION, SOURCE
  Example :
  mov eax, 123

Préfixe de variable

La taille des paramètres est intégrée au nom du registre :
`RAX, EAX, AX, AL` impliquent respectivement qword (64 bits), long (double word, 32 bits), word (16 bits) et byte (octet 8 bits).

- **AT&T**
  b = *byte* (8 bit).
  s = *single* (32-bit floating point).
  w = *word* (16 bit).
  l = *long* (32 bit integer or 64-bit floating point).
  q = *quad* (64 bit).
  t = *ten bytes* (80-bit floating point).

Exemple de manipulation de registres

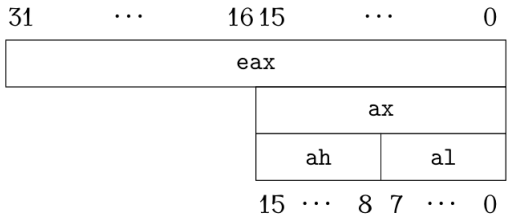


FIGURE – Subdivision du registre eax

```
movb $5, %al  # sauvgarde la veur 5 dans le registre al
movb $6, %ah  # sauvgarde la veur 5 dans le registre ah
movw $7, %ax  # sauvgarde la veur 7 dans le registre ax
```

Les instructions binaires

Chapitre 17 : Instruction Set

- Les instructions arithmétiques
- Les opérations sur les bits
- Les instructions de comparaison et de branchement

Les instructions arithmétiques

Chapitre 17 : Instruction Set

Instruction	**Description**
add op1, op2	op2 ← op1 + op2
sub op1, op2	op2 ← op2 - op1
neg reg	reg ← -reg
inc reg	reg ← reg+1
dec reg	reg ← reg - 1
imul op (signé ou **mul** non signé)	**edx :eax** ← **eax** ×*op*
imul dest, op	dest ← dest ×*op*
imul dest, op, immédiat	dest ← op ×*immdiat*
idiv op (**div** non signé)	**eax** ← **edx :eax** / op **edx** ← **edx :eax** mod op Ne pas oublier de mettre 0 dans **edx** avant d'appeler **idiv**

Les instructions binaires

Instruction	Description
and op1,op2	op2 ← op1 & op2
or op1,op2	op2 ← op1 | op2
xor op1,op2	op2 ← op1 ^ op2
not reg	reg ← ~reg
shl reg, immédiat	reg ← reg << immédiat
shr reg, immédiat	reg ← reg >> immédiat
sal reg, immédiat	reg ← reg << immédiat signé
sar reg, immédiat	reg ← reg >> immédiat signé
rol reg, immédiat	reg ← reg décalage circulaire gauche de imm
ror reg, immédiat	reg ← reg décalage circulaire droit imm
rcl reg, immédiat	reg : CF ← reg : CF reg décalage circulaire gauche imm
rcr reg, immédiat	reg : CF ← reg : CF decalageCircDroitede imm

Les instructions de comparaison et de branchement

Instruction	Description
cmp op1,op2	calcul de op2 - op1 sans stocker le résultat. Mise à jour ZF,CF et OF
jmp op	branchement inconditionnel à l'adresse op
jz op	branchement à op si ZF=1
jnz op	branchement à op si ZF=0
jo op	branchement à op si OF=1
jno op	branchement à op si OF=0
js op	branchement à op si SF=1
jns op	branchement à op si SF=0
jc op	branchement à op si CF=1
jnc op	branchement à op si CF=0
jp op	branchement à op si PF=1
jnp op	branchement à op si PF=0

Les instructions de branchement après `cmp`

Signées :

Instruction	Description
je op	branchement à op si op1 = op2
jne op	branchement à op si op1 != op2
jl op (**jnge**)	branchement à op si op1 < op2
jle op (**jng**)	branchement à op si op1 <= op2
jg op (**jnle**)	branchement à op si op1 > op2
jge op (**jnl**)	branchement à op si op1 >= op2

Les instructions de branchement après `cmp`

Non signées :

Instruction	Description
je op	branchement à op si op1 = op2
jne op	branchement à op si op1 != op2
jb op (**jnae**)	branchement à op si op1 < op2
jbe op (**jna**)	branchement à op si op1 <= op2
ja op (**jnbe**)	branchement à op si op1 > op2
jae op (**jnb**)	branchement à op si op1 >= op2

Exemple de manipulation de registres

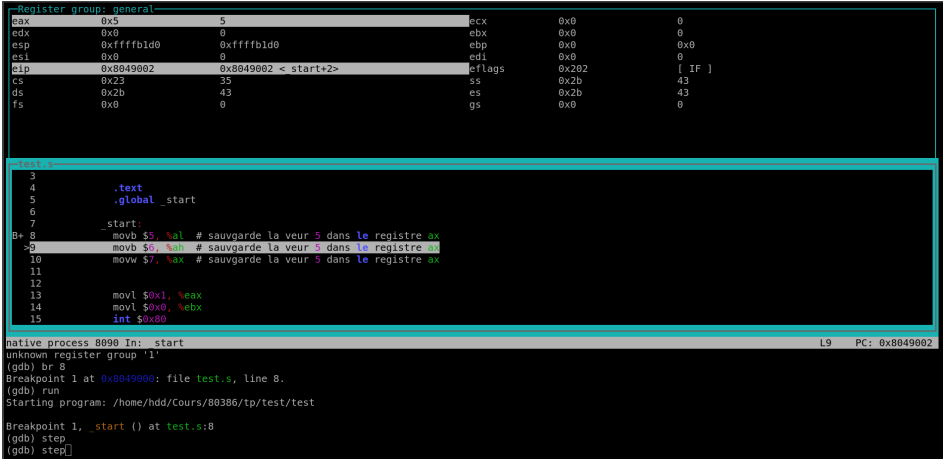


FIGURE – Exemple après exécution de la première instruction

Structure d'un if

```
if (x == 5) {
    x = x + 2;
}
else {
    x = x - 4;
}
```

```
.data
x:  .int 10
...
.text
...
if:
    cmpl $5, x
    jne else
    addl $2, x
    jmp end_if
else:
    subl $4, x
end_if:
```

Structure d'une boucle while

```
char x = 10
...
while (x > 5) {
    x = x - 1;
}
```

```
.data
x:  .byte  10
.text
.global main
main:
    movl    %esp, %ebp
    xorl    %eax, %eax
    movl    x,    %ecx
while:
    cmpl    $5,   %ecx
    jle     end_while
    dec     %ecx
    movl    %ecx, x
    jmp     while
end_while:
    ret
```

Structure d'une boucle for

```
char i, x = 10
for (i = 0; i < 5; i ++) {
    x = x + 4;
}
```

```
.data
    x:  .byte 10
.bss
        .lcomm i, 1
.text
    .global main
main:
        movl    %esp, %ebp
        pushl   %ebp
        movl    $0, i
for:    cmpl    $5, i
        jae     end_for
        addl    $4, x
        addl    $1, i
        jmp     for
end_for:
        leave
        ret
```

Appler la fonction `printf`

```
printf(" ******************\n");
printf("* GNU AS IS THE BEST *\n");
printf(" ******************\n");
```

```
.data
str1: .asciz " ******************\n"
str2: .asciz "* GNU AS IS THE BEST *\n"
str3: .asciz " ******************\n"

.text
.global main

main:
    movl %esp, %ebp
    pushl $str1
    call printf
    pushl $str2
    call printf
    pushl $str3
    call printf
    add $12,%esp
    ret
```

Structure d'une boucle for

```c
int i;
for (i = 1; i <= 10; i ++) {
    printf("%d\n",i );
}
```

```
str : .asciz "%d\n"
.text
.global main
main:

    movl $1, %eax
    movl $10, %ebx
loop:
    pushl %eax
    pushl %ebx
    pushl %eax
    pushl $str
    call printf
    addl $8, %esp
    popl %ebx
    popl %eax
    cmpl %eax, %ebx
    je end
    incl %eax
    jmp loop
end:
    ret
```

Les modes d'adressages

Lorsqu'un opérande réside dans le segment de données du programme, le processeur doit calculer son adresse effective (EA : Effective Address) en se basant sur l'expression suivante :

$$EA = base + scale * index + disp$$

L'assembleur GNU traduit l'expression précédente en celle-ci :

$$EA = disp(base, index, scale)$$

- **disp** : un déplacement facultatif. **disp** peut être un symbole ou un entier signé
- **scale** : un scalaire qui multiplie index. Il peut avoir la valeur 1, 2, 4 ou 6. Si le scale n'est pas spécifié, il est substitué par 1
- **base** : un registre 32 bits optionnel. Souvent, on utilise EBX et EBP (si l'opérande est dans la pile) comme base
- **index** : un registre 32 bits optionnel. Souvent, on utilise ESI et EDI comme index

Les adresses mémoire

- **AT&T**
  - base : Registre de 32 bits (contenant le plus souvent une adresse)
  - index (Optionnel) : Registre de 32 bits (contenant le plus souvent une adresse)
  - scale (Optionnel) : Facteur valant 1, 2, 4 ou 8 multipliant index
  - disp (Optionnel) : Déplacement (displacement), ajouté ou déduit à la fin du calcul
  - segreg (Optionnel) : Segment mémoire (Segment Register) indiquant le segment dans lequel se trouve la donnée

Les modes d'adressage : mémoire de données

- Types de l'opérande source
  - Registre
  - Mémoire
  - immédiat, valeur codée dans l'instruction
  - implicite, valeur qui n'apparait pas dans l'instruction
- Types de l'opérande destination
  - Registre
  - Mémoire

ATTENTION

Une instruction ne peut pas avoir deux opérandes de type mémoire.

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 0*x*

```
movb $0xff,    %al    # al  = -1
movw $0xffff, %ax    # ax  = -1
movl $-1,      %eax   # eax = -1
movl $lim,     %eax   # eax = la valeur du symbole lim
```

Adressage direct

En little endian à l'adresse `0x80C0439` se trouve le :

`double mot : 0xDDCCBBAA`
`mot : 0xBBAA`
`octet : 0xAA`

```
#affecte 0xaa au registre octet al
movb 0x80C0439, %al

# affecte 0xbbaa au registre mot ax
movw 0x80C0439, %ax

# affecte 0xddccbbaa au registre eax
movl 0x80C0439, %eax
```

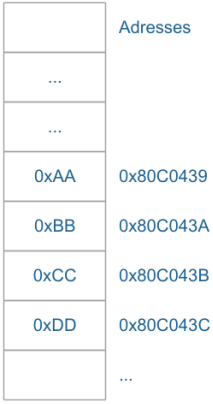


FIGURE – Mémoire

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 : `(%REGISTRE)`
- REGISTER est l'un quelconque des registres EAX, EBX, ECX, EDX, EDI et ESI.
- Les registres ESP et EBP sont réservés pour l'adressage de la pile

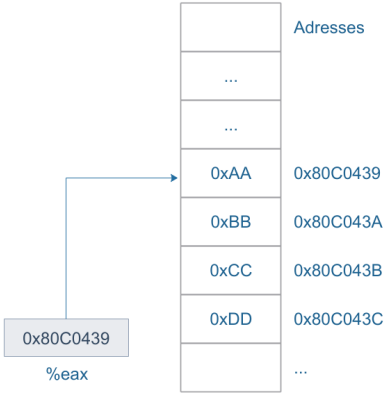
Exemple : Adressage indirect registre

```
#Mettre l'@ dans le registre %ebx
movb 0x80C0439, %ebx

#affecte 0xaa au registre octet al
movb (%ebx), %al

# affecte 0xbbaa au registre mot ax
movw (%ebx), %ax

# affecte 0xddccbbaa au registre eax
movl (%ebx), %eax
```



Adressage avec base et déplacement

- Dans ce mode l'adresse de l'opérande est contenue dans le registre + un déplacement
- La syntaxe assembleur de ce mode d'adressage est : `Dep(%REGISTRE)`
- REGISTER est l'un quelconque des registres EAX, EBX, ECX, EDX, EDI, ESI, ESP et EBP
- Pour calculer l'adresse de l'opérande on ajoute au contenu du registre de base `REGISTER` la valeur sur 4 octets du déplacement signé.

Exemple : Adressage avec base et déplacement

Quelle est la valeur de `%ax` après l'exécution de ces instructions :

```
movl $0x80C0439, %ebx
movw 2(%ebx), %ax
```

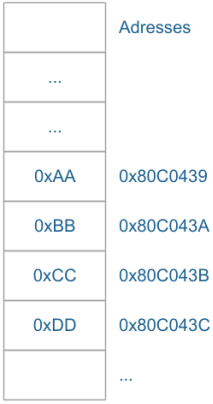


FIGURE – Mémoire

Exemple : Adressage avec base, déplacement et index

```
    .data
tab_int:    .int 10, 20, 30
    .text
    .global main
main:
    movl %esp, %ebp #for correct debugging
    # write your code here
    xorl    %eax,   %eax

    leal    tab_int,    %ebx
    addl    $2,         (%ebx)
    addl    $2,         4(%ebx)
    addl    $2,         8(%ebx)
    ret
```

Adressage avec base, déplacement et index

- Ce mode d'adressage fait intervenir deux registres (`REGISTRE1` et `REGISTRE2`) et une valeur constante signée `Dep`
- La syntaxe assembleur de ce mode d'adressage est : `Dep(%REGISTRE1, %REGISTRE2)`
- REGISTER est l'un quelconque des registres EAX, EBX, ECX, EDX, EDI, ESI, ESP et EBP
- 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.

Adressage avec base, déplacement et index typé

- Ce mode d'adressage fait intervenir deux registres (`REGISTRE1` et `REGISTRE2`), un déplacement `Dep` qui consiste en une valeur constante signée et finalement 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(%REGISTRE1, %REGISTRE2, MUL)`
- REGISTER est l'un quelconque des registres EAX, EBX, ECX, EDX, EDI, ESI, ESP et EBP
- 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 : Adressage avec base, déplacement, index et type

```
.data
tableau:
    .int 1, 5, 2, 4,
somme:
    .int 0
.text
.global main
main:
    movl %esp, %ebp
    xor %eax, %eax
    xor %ebx, %ebx
    mov $tableau, %ecx
tant_que:
    cmp $4, %eax
    je fin
    add (%ecx, %eax,4), %ebx
    inc %eax
    jmp tant_que
fin:
    mov %ebx, somme
    ret
```

Les modes d'adressage : mémoire d'instructions

Deux types d'instructions ont besoin d'adresser la mémoire d'instructions :

- les instructions de saut (inconditionnel ou conditionnel)
- les instructions d'appel de sous programmes.

L'instruction de saut inconditionnel qui s'appelle **jump**, la syntaxe assembleur est :

```
jmp <dest>
```

- `<dest>` désigne la façon de calculer l'adresse de l'instruction à exécuter après le saut

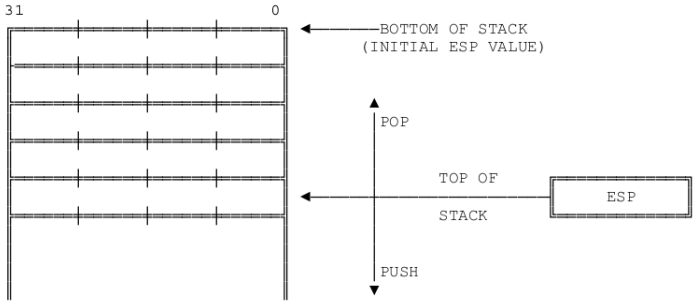
Les illustrations des modes d'adressages se ferons avec l'instruction inconditionnel `jmp`

Les modes d'adressage de la mémoire d'instructions :

- adressage direct
- adressage relatif à la valeur du registre EIP
- adressage indirect qui utilise un relais qui est soit un registre soit un long mot de la mémoire de données

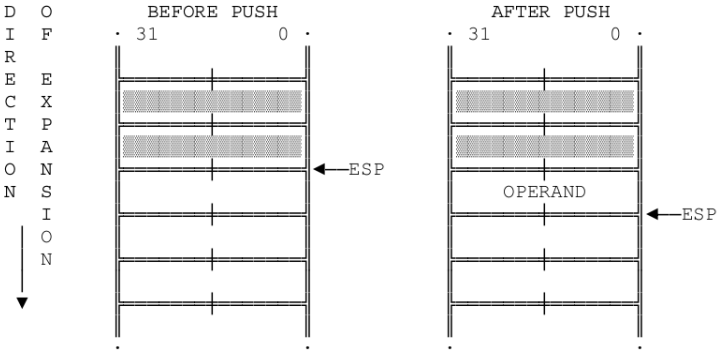
La pile : principe de fonctionnement

- `EBP` : registre de pointeur de base de pile
- `ESP` : registre de pointeur de pile. Contient l'adresse supérieure de la pile.



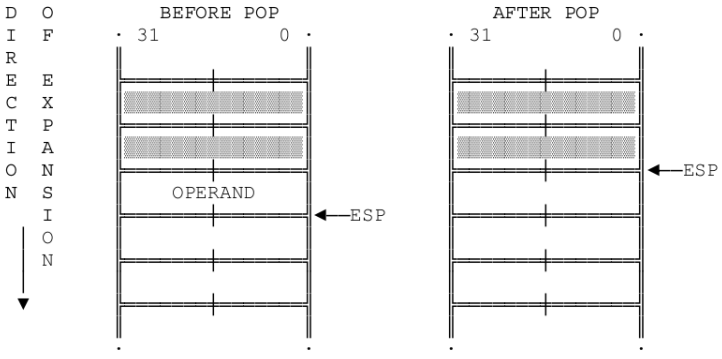
La pile : principe de fonctionnement

- PUSH :



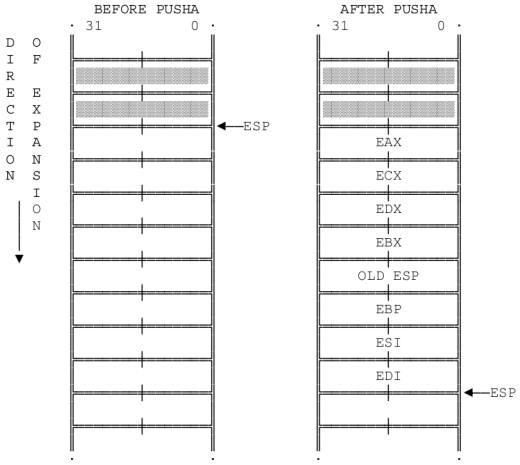
La pile : principe de fonctionnement

- POP :



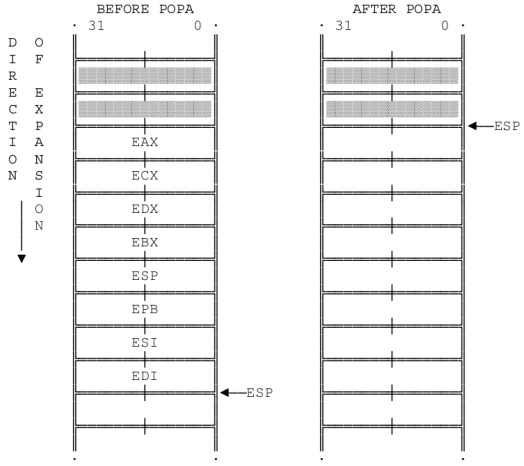
La pile : principe de fonctionnement

- PUSHA :



La pile : principe de fonctionnement

- POPA :



Les variables locales

- On alloue l'espace requis par les variables locales en diminuant `esp`
- On peut alors mettre les valeurs des variables locales dans `(%esp)`, `8(%esp)` ... , en utilisant l'adressage indirete.
- A la fin du sous programme, on libère l'espace mémoire correspondant avec l'instruction `mov %epb, %ese`

La pile : Les sous programmes

Définition d'un sous programme :

- Code écrite une seule fois que l'on peut exécuter à partir de différents endroits du programme
- Implémentation des procédures et des fonctions des langages de haut niveau

- A la fin de l'exécution d'un sous programme, l'exécution doit se poursuivre avec l'instruction qui suit l'appel
- Il faut mémoriser l'adresse de retour et faire un saut à cette adresse en utilisant l'adressage indirect.

Les instruction d'appel et retour de sous programmes :

- **`call`** `op` : saut inconditionnel à **op** et empile l'adresse de l'instruction suivante
- **`ret`** : dépile une adresse et saute à cette adresse

Appel de fonction : Exemple

```
.text
.global main

f:
    pushl   %ebp
    movl    %esp,   %ebp
    pushl   $30
    # .....
    addl    $4, %esp
    popl    %ebp
    ret
main:
    movl %esp, %ebp #for correct debugging
    pushl   %ebp
    movl    %esp,   %ebp
    pushl   $5
    pushl   $10

    call f
    addl    $8, %esp
    movl    %esp, %ebp
    ret
```

La convention d'appel C

1. Avant d'appeler un sous-programme, l'appelant doit sauvegarder le contenu de certains registres enregistré par l'appelant. Les registres enregistrés par l'appelant sont EAX, ECX, EDX. Si vous voulez le contenu de ces registres à conserver à travers l'appel de sous-programme, sauvegardez-les sur la pile.
2. Pour passer des paramètres au sous-programme, sauvegardez-les sur la pile avant l'appel. Les paramètres doivent être sauvegardés dans l'ordre inverse (c'est-à-dire le dernier paramètre en premier) - puisque la pile grandit, le premier paramètre sera stocké à l'adresse la plus basse (cette inversion des paramètres était historiquement utilisée pour permettent aux fonctions de passer un nombre variable de paramètres).
3. Pour appeler le sous-programme, utilisez l'instruction d'appel. Cette instruction place l'adresse de retour au-dessus de les paramètres sur la pile et se branche sur le code du sous-programme.
4. Après le retour du sous-programme (c'est-à-dire immédiatement après l'instruction d'appel), l'appelant doit supprimer les paramètres de la pile. Cela restaure la pile à son état avant l'exécution de l'appel.
5. L'appelant peut s'attendre à trouver la valeur de retour du sous-programme dans le registre EAX.
6. L'appelant restaure le contenu des registres enregistrés par l'appelant (EAX, ECX, EDX) en les retirant de la pile. L'appelant peut supposer qu'aucun autre registre n'a été modifié par le sous-programme.

Les règles de l'appelé (1)

1. Au début de la sous-routine, la fonction doit pousser la valeur de EBP sur la pile, puis copier la valeur de ESP dans EBP en utilisant les instructions suivantes :

```
pushl %ebp
movl %ebp, %esp
```

2. Ensuite, allouez les variables locales en faisant de la place sur la pile. Rappelez-vous, la pile grandit vers le bas, donc pour faire de la place sur le dessus de la pile, le pointeur de pile doit être décrémenté. La quantité de décrémentation du pointeur de pile dépend du nombre de variables locales nécessaires. Par exemple, si 3 entiers locaux (4 octets chacun) étaient requis, le pointeur de pile devrait être décrémenté de 12 pour faire de la place pour ces variables locales. C'est à dire :

```
subl %esp , 12
```

Comme pour les paramètres, les variables locales seront situées à des décalages connus par rapport au pointeur de base (%ebp).

Les règles de l'appelé (2)

3. Ensuite, les valeurs de tous les registres désignés comme enregistrés par l'appelé qui seront utilisés par la fonction doivent être enregistrées. Pour sauvegarder les registres, poussez-les sur la pile. Les registres enregistrés par l'appelé sont EBX, EDI et ESI (ESP et EBP seront également préservés par la convention d'appel, mais n'ont pas besoin d'être poussés sur la pile au cours de cette étape). Une fois ces trois actions effectuées, le fonctionnement réel du sous-programme peut se poursuivre. Lorsque le sous-programme est prêt à revenir, les règles de convention d'appel continuent :
4. Lorsque la fonction est terminée, la valeur de retour de la fonction doit être placée dans EAX si elle n'y est pas déjà.
5. La fonction doit restaurer les anciennes valeurs de tous les registres enregistrés par l'appelé (EBX, EDI et ESI) qui ont été modifiés. Le contenu du registre est restauré en le détachant de la pile. Notez que les registres doivent être sautés dans l'ordre inverse dans lequel ils ont été poussés.

Les règles de l'appelé (3)

6. Ensuite, nous libérons les variables locales. La façon évidente de le faire pourrait être d'ajouter la valeur appropriée au pointeur de pile (puisque l'espace a été alloué en soustrayant la quantité nécessaire du pointeur de pile). En pratique, une façon moins sujette aux erreurs de désallouer les variables consiste à déplacer la valeur du pointeur de base dans le pointeur de pile, c'est-à-dire :

   ```
   movl %esp , %ebp
   ```

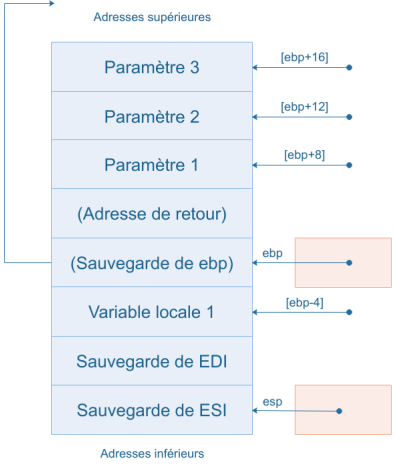
   Cette astuce fonctionne car le pointeur de base contient toujours la valeur que le pointeur de pile contenait juste avant l'allocation des variables locales.

7. Immédiatement avant de revenir, nous devons restaurer la valeur du pointeur de base de l'appelant en supprimant EBP de la pile. Rappelez-vous, la première chose que nous avons faite à l'entrée du sous-programme a été de pousser le pointeur de base pour sauvegarder son ancienne valeur.

8. Enfin, nous revenons à l'appelant en exécutant une instruction `ret`. Cette instruction trouvera et supprimera l'adresse de retour appropriée de la pile.

La pile : principe de fonctionnement

- Une image de la pile en mémoire lors de l'exécution du corps de myFunc :



Structure d'une boucle for

```
.text
.global main
.global myFunc
myFunc:
/***Prologue de sous-programme standard ***/
    pushl   %ebp                    # Sauvegarder l'ancienne valeur du pointeur de base
    movl    %esp, %ebp              # Mettre la nouvelle vlaeur du pointeur de base
    subl    $4, %esp                # Réserver de l'espace pour 1 variables locale de 4 octets
    pushl   %edi                    # Savegarde des registres que la fonction modifie
    pushl   %esi                    # ici on modifie %edi et %esi
                                    # pas besoinde sauvegarder EAX, EBP ou ESP

    /*** Corps du sous-programme ***/
    movl     8(%ebp),    %eax       # Mettre la valeur du paramètre1 dans EAX
    movl    12(%ebp),    %esi       # Mettre la valeur du paramètre1 dans ESI
    movl    16(%ebp),    %edi       # Mettre la valeur du paramètre1 dans EDI

    movl    %edi,       -4(%ebp)    # Metre %edi dans la variable locale
    addl    %esi,       -4(%ebp)    # Ajouter %esi dans la variable locale
    addl    -4(%ebp),   %eax        # Ajouter le contenu de la variable locale
                                    # dans %eax (résultat final)

    /***Epilogue de sous-programme standard ***/
    popl    %esi
    popl    %edi
    movl    %ebp, %esp
    popl    %ebp
    ret
```

Structure d'une boucle for

```
main:
    movl    %esp, %ebp #for correct debugging
    pushl   $10
    pushl   $20
    pushl   $30
    call    myFunc
    addl    $12, %esp
    ret
```

Structure d'une boucle for

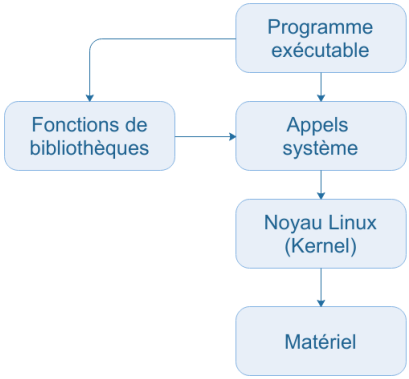
```c
#include <stdio.h>
extern int myFunc(int, int, int);
int main ( ) {
    int x = 3;
    printf("myFun = %d", myFunc(10,20,30));
    return 0 ;
}
```

```
$ gcc -m32 main.c myFunc.s
```

Les appels système : Définition

- Un appel système (*system call* ou *syscall*) provoque une interruption pour demander au système d'exploitation d'accomplir une tâche
- Un appel système peut être vu comme une fonction primitive fournie par le noyau accessible par interruption
- Se référer à la section 2 des pages de manuel. e.g. appel système **`read`** :

```
$ man -S 2 read
```



Quelques appels système importants

Manipulation de système de fichiers :

- **`open`** : Ouvrir et éventuellement créer un fichier
- **`close`** : Fermer un descripteur de fichier
- **`read`** : Lire à partir d'un descripteur de fichier
- **`write`** : Écrire dans un descripteur de fichier

Contrôle des processus :

- **`exit`** : Mettre fin au processus d'appel
- **`fork`** : Créer un processus enfant
- **`wait`**, **`waitpid`** : Attendez que le processus change d'état

Les appels système

Les appels systèmes sont listé dans les headers du noyau linux ici :

`/usr/include/x86_64-linux-gnu/asm/unistd_32.h`

```
#ifndef _ASM_X86_UNISTD_32_H
#define _ASM_X86_UNISTD_32_H 1

#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_waitpid 7
#define __NR_creat 8
#define __NR_link 9
#define __NR_unlink 10
....
```

Appels systèmes en assembleur

1. Choix de l'appel système (identification du numéro de l'appel)
2. Préparation de l'appel système (passage des paramètres)
   - Affectation des paramètres aux registres :
     - `eax` : Numéro de l'appel système
     - `ebx` : Argument 0
     - `ecx` : Argument 1
     - `edx` : Argument 2
     - `esi` : Argument 3
     - `edi` : Argument 4
     - `ebp` : Argument 5
3. Appel du system call (appel sous forme d'intéruption)
   - Utilisation de l'instruction **`int`** pour provoquer une interruption avec le paramètre **`0x80`** pour indiquer au noyau qu'on souhaite appeler une fonction système
     - `int $0x80`

Quelques numéros d'appels système importants

N°	Appel système	%eax	arg0 (%ebx)	arg1 (%ecx)	arg2 (%edx)	arg3 (%esi)	arg4 (%edi)	arg5 (%ebp)
0	restart_syscall	0x00	-	-	-	-	-	-
1	exit	0x01	int error_code	-	-	-	-	-
2	fork	0x02	-	-	-	-	-	-
3	read	0x03	unsigned int fd	char *buf	size_t count	-	-	-
4	write	0x04	unsigned int fd	const char *buf	size_t count	-	-	-
5	open	0x05	const char *filename	int flags	umode_t mode	-	-	-
6	close	0x06	unsigned int fd	-	-	-	-	-
7	waitpid	0x07	pid_t pid	int *stat_addr	int options	-	-	-
8	creat	0x08	const char *pathname	umode_t mode	-	-	-	-
9	link	0x09	const char *oldname	const char *newname	-	-	-	-

La liste complète est fourni dans les documents de cours sans la section *ressources*
ou dans les headers du noyau linux ici :
`/usr/include/x86_64-linux-gnu/asm/unistd_32.h`

Exemple avec l'appel système **write** et **exit**

```
.data
str:    .asciz "Hello World\n"
        len = . - str - 1
.text   .global _start
_start:
        /** Appel System write **/
        movl $0x4, %eax
        movl $0x1, %ebx
        movl $str, %ecx
        movl $len, %edx
        int $0x80

        /** Appel System exit **/
        movl $0x1, %eax
        movl $0x0, %ebx
        int $0x80
```

Compilation :

```
as --32 -o helloWorld.o helloWorld.s
ld -m elf_i386 -o hello helloWorld.o
```