

Programmation assembleur : aperçu

*Programmation en assembleur : NASM
Module M14 Semestre 4*

Printemps 2010

Equipe pédagogique : Younès El Amrani, Abdelhakim El Imrani, Faissal Ouardi

Tout l'assembleur dans un transparent

- L'assembleur Nasm est un langage de mnémoniques
- Nasm permet de programmer la machine au niveau le plus bas.
- Nasm utilise des codes d'opérations qui activent les circuits combinatoires du processeur.
- Nasm accède aux registres de la machine. Les registres sont des circuits séquentiels.
- Nasm accède et manipule directement la mémoire en utilisant des adresses vues comme des adresses d'octets
- Nasm possède 5 types fondamentaux :
 1. byte (1 octet = 8 bits) ,
 2. word (2 octets = 16 bits)
 3. double word (4 octets = 32 bits)
 4. Quadruple word (8 octets = 64 bits)
 5. Ten Bytes (10 octets = 80 bits)

Format d'un Programme NASM

```
SEGMENT .data                                     ;; données initialisées
```

```
textHello: DB "Hello, World !" , 0 ;; DB signifie Define Byte
```

```
SEGMENT .bss                                     ;; les données non initialisées
```

```
textReponseHello: RESB 15                       ;; RESB signifie Réserve Byte
```

```
SEGMENT .text                                    ;; Le code exécutable
```

```
_main:                                           ;; main est par convention l'entrée du code / le début
```

```
    ENTER 0, 0                                   ;; Le prologue : on commence une fenêtre de pile
```

```
    OpCode_1 OPERANDE(S)_1                       ;; CODE OPERATION 1 suivi des opérandes ;
```

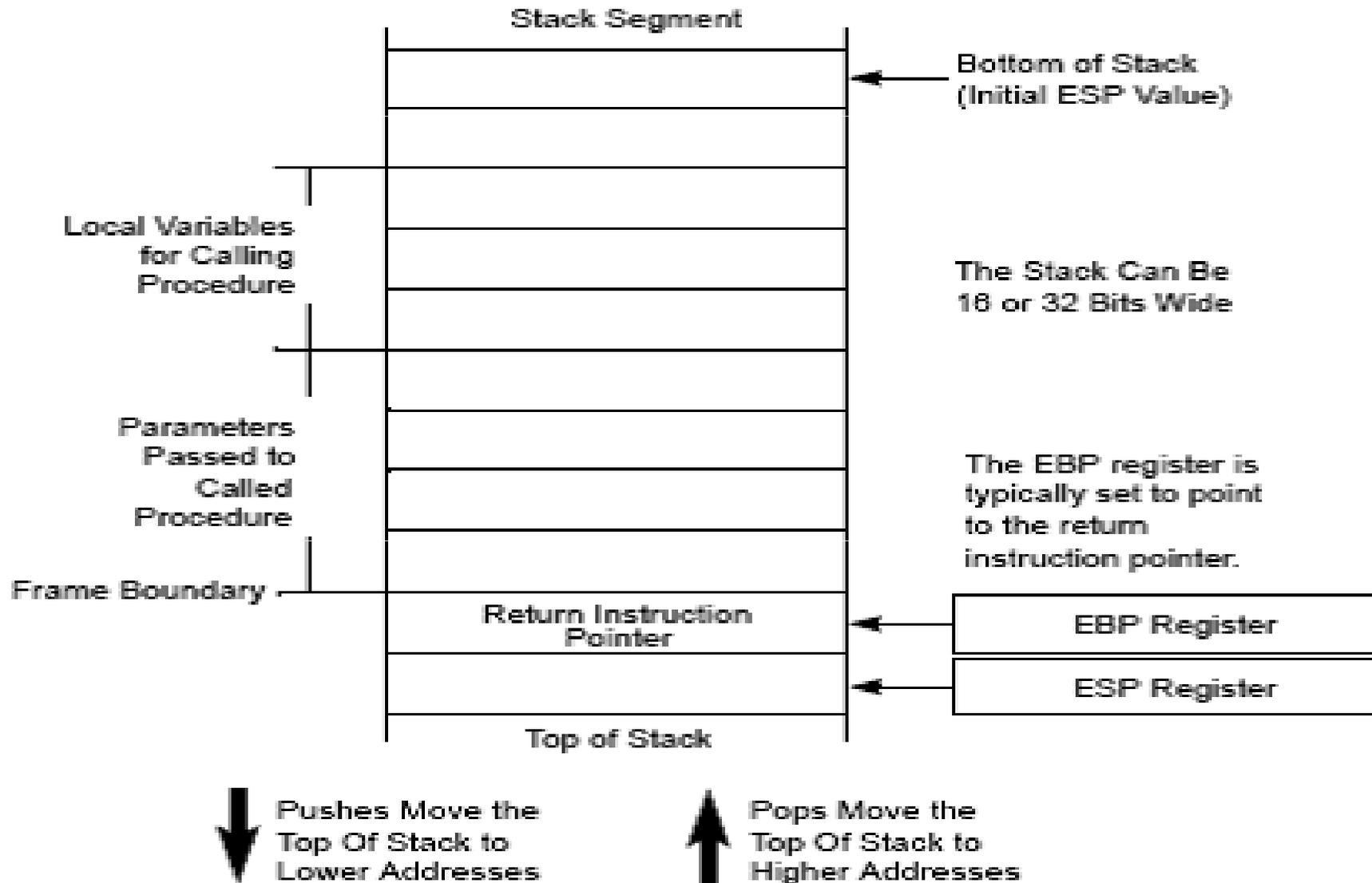
```
    ...
```

```
    OpCode_n OPERANDE(S)_n                       ;; CODE OPERATION n suivi des opérandes ;
```

```
    LEAVE                                        ;; On referme la fenêtre de pile ;
```

```
    RET                                          ;; On restaure le compteur de programme
```

Le segment de pile



Données initialisées en NASM

```
SEGMENT .data    ;;; Le segment .data contient les données initialisées
textHello:      DB "Hello, World!", 0    ;;; texte qui se termine par zéro
integerFormat:  DB "%d", 0              ;;; texte format %d de printf du C
stringFormat:   DB "%s", 0              ;;; texte format %s de printf du C
characterFormat: DB "%c", 0             ;;; texte format %c de printf du C
charVal1:       DB 'C'                  ;;; Le caractère 'C'
intVal1:        DD 15                    ;;; L'entier 15 codé sur 32 bits
intVal2:        DD 20                    ;;; L'entier 20 codé sur 32 bits
```

Allocation de données initialisées

Nombre d'octets	Mnémonique	Description	Attribut
1	D B	Définit un byte (un octet)	Byte
2	D W	Définit un mot (un " Word " en anglais)	Word
4	DD	définit un double mot (Double Word)	Double word
8	DQ	définit un mot quadruple (quadruple word)	Quad word
10	DT	Définit dix octets (dix bytes)	Ten bytes

5.1. FUNDAMENTAL DATA TYPES

The fundamental data types of the IA are bytes, words, doublewords, and quadwords (refer to Figure 5-1). A byte is eight bits, a word is 2 bytes (16 bits), a doubleword is 4 bytes (32 bits), and a quadword is 8 bytes (64 bits).

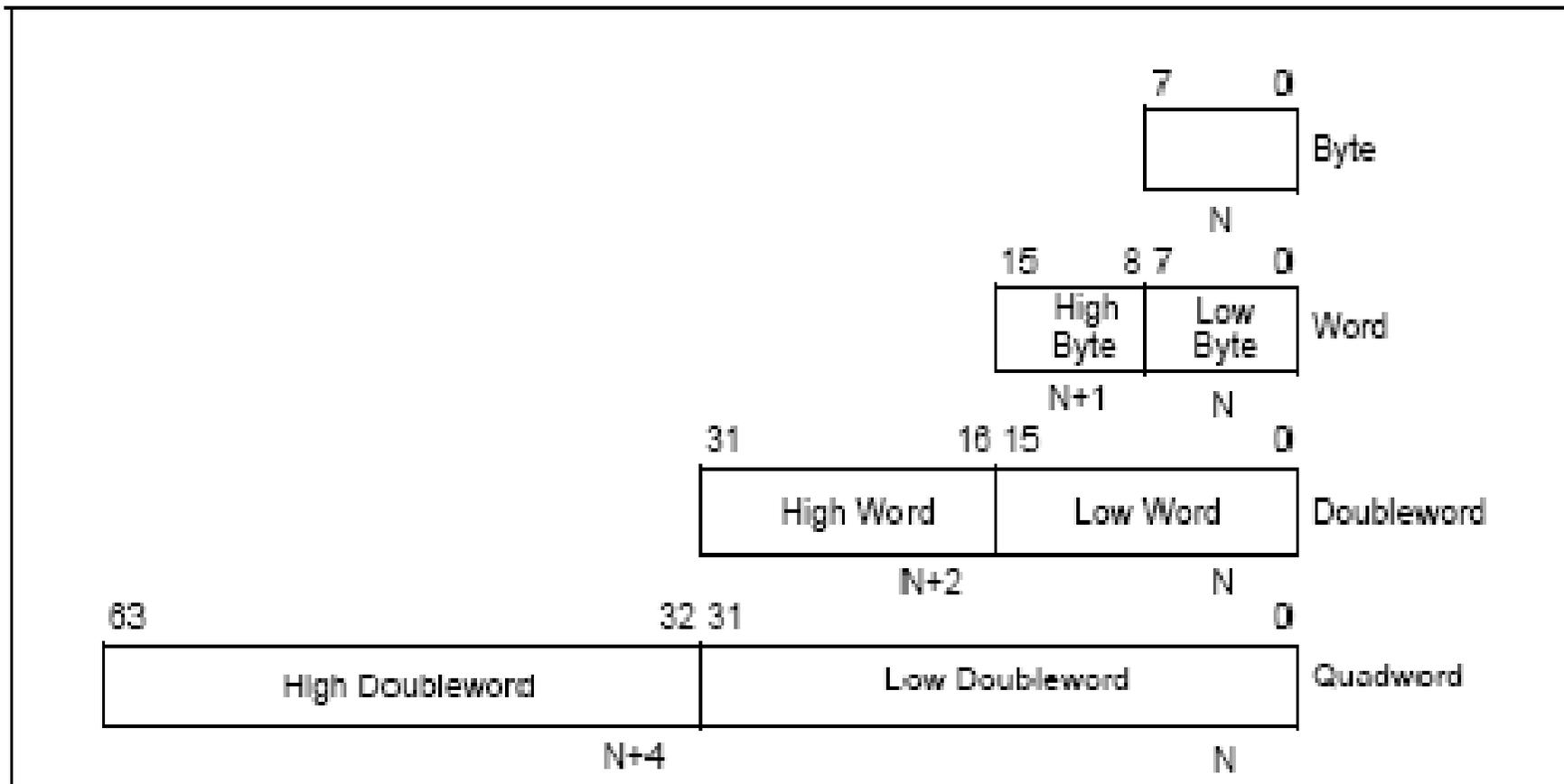
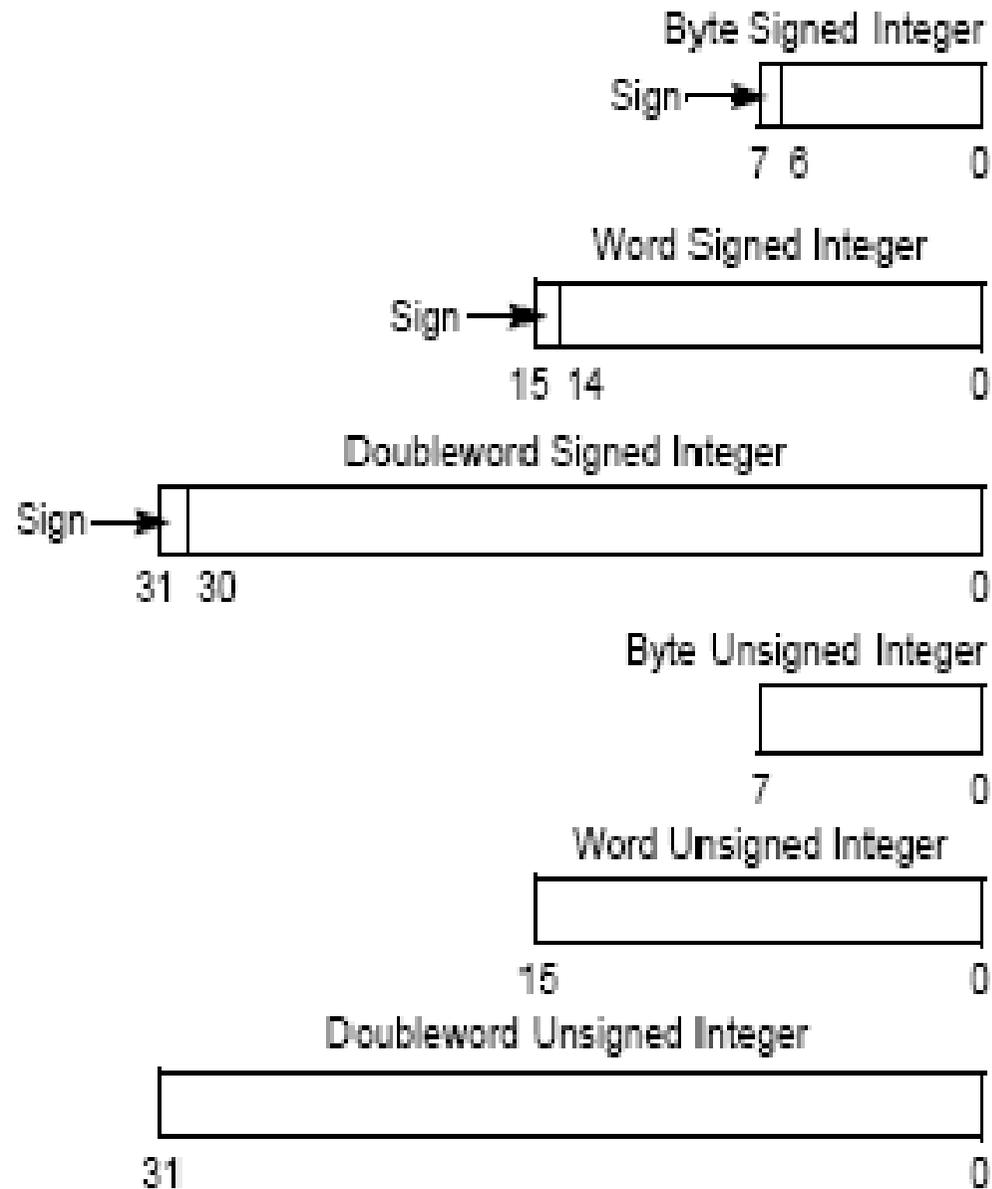


Figure 5-1. Fundamental Data Types



Données non-initialisées en NASM

```
SEGMENT .bss                ;;segment de données non initialisées
    textHello:             RESB 15    ;; Réserve de 15 Bytes (15 caractères)
    textSalam:             RESB 5     ;; Réserve de 5 Bytes (5 caractères)
    tableauInt1:           RESD 10    ;; Réserve de 10 entiers : c'est un tableau
    tableauInt2:           RESD 15    ;; Réserve de 15 entiers : c'est un tableau
    charVal1:              RESB 1     ;; Réserve de 1 seul Byte (1 seul caractère)
    charVal2:              RESB 1     ;; Réserve de 1 seul Byte (1 seul caractère)
```

;;; DANS CE SEGMENT, LES DONNEES NE RECOIVENT PAS DE VALEUR

;;; INITIALE.

;;; UN ESPACE LEUR EST RESERVE POUR CONTENIR LA / LES VALEUR(S)

Allocation de données non initialisées

Nombre d'octets	Mnémonique	Description	Nom
1 * n	RESB n	Réserve n octet(s)	Byte
2 * n	RESW n	Réserve n mots (1 mot = 2 octets)	Word
4 * n	RESD n	Réserve n double mot(s) (1 Double mot = 4 octets)	Double word
8 * n	RESQ n	Réserve un quadruple mot (quadruple word)	Quad word
10 * n	REST n	Réserve dix octets (dix bytes)	Ten bytes

Données initialisées en NASM

```
SEGMENT .data    ;;; Le segment .data contient les données initialisées

textHello:      DB "Hello, World!", 0    ;;; texte qui se termine par zéro
integerFormat:  DB "%d", 0              ;;; texte format %d de printf du C
stringFormat:   DB "%s", 0             ;;; texte format %s de printf du C
characterFormat: DB "%c", 0            ;;; texte format %c de printf du C
```

Programme proprement dit

```
SEGMENT .data    ;;; Le segment .data contient les données initialisées  
    textHello:   DB "Hello, World !" , 0    ;;; texte : terminé par zéro  
    integerFormat: DB "%d" , 0             ;;; format %d de printf du C  
    stringFormat: DB "%s" , 0             ;;; format %s de printf du C  
    characterFormat: DB "%c" , 0          ;;; format %c de printf du C
```

```
SEGMENT .text    ;;; marque le début du segment de code  
    global _main ;;; symboles exportés vers l'extérieur  
    extern _printf , _scanf ;;; fonctions importées
```

Programme qui utilise la librairie C

_main:

enter 0, 0

;;; Ouvre une fenêtre de pile

push textHello

;;; ADRESSE DU TEXTE HELLO

push stringFormat

;;; EMPILER FORMAT

call _printf

;;; APPEL DE FONCTION

add ESP, 8

;;; DEPILER 2 FOIS (ADD ESP,4)

mov EAX, 0

;;; Signifie fin du programme

leave

;;; libère la zone utilisée sur la pile

ret

;;; restaure EIP.

;;; EIP signifie extended Instruction

;;; Pointer

Quelques abbréviations

- **reg8** dénote un registre général 8 bits
- **reg16** dénote un registre général 16 bits
- **reg32** dénote un registre général de 32 bits.
- **imm** dénotes une valeur immédiate de 8, 16 ou 32 bits.
- **imm8** dénotes une valeur immédiate de 8 bits.
- **imm16** dénotes une valeur immédiate de 16 bits.
- **imm32** dénotes une valeur immédiate de 32 bits.
- **mem** dénotes une référence mémoire générique de 8, 16 ou 32 bits
- **mem8**, **mem16**, **mem32**, **mem64** dénote une référence mémoire respectivement de 8, 16, 32 et 64 bits.
- **r/m8** dénote un registre ou une référence mémoire de 8 bits.
- **r/m16**, **r/m32** dénote un registre ou une référence mémoire respectivement de 16 bits et 32 bits.

PUSH : documentation NASM

- PUSH : Push Data on Stack
- PUSH reg16 ; o16 50+r [8086]
- PUSH reg32 ; o32 50+r [386]
- PUSH imm8 ; 6A ib [186]
- PUSH imm16 ; o16 68 iw [186]
- PUSH imm32 ; o32 68 id [386]

En mode 32 bits : PUSH décrémente le sommet de pile (ESP) de 4 bytes, et empile la valeur donnée à [SS:ESP]

En mode 16 bits : PUSH décrémente le sommet de pile (SP) de 2 bytes, et empile la valeur donnée à [SS:SP] .

LE SEGMENT .data : vue binaire !!

00000000 48	H:	DB 'H'
00000001 65	e:	DB 'e'
00000002 6C	l:	DB 'l'
00000003 64	d:	DB 'd'
00000004 73	s:	DB 's'
00000005 25	pourcentage:	DB '%'
00000006 00	zero:	DB 0
00000007 48656C6C6F20576F72-	textHello:	DB "Hello World" , 0
00000010 6C6400		
00000013 257300	stringFormat:	DB "%s" , 0

LE SEGMENT .text : vue binaire !!

```
00000004  68[07000000]    push    textHello
00000009  68[13000000]    push    stringFormat
0000000E  E8(00000000)    call   _printf
00000013  81C404000000    add    ESP , 4

00000019  B800000000      mov    eax , 0
0000001E  C9              leave
0000001F  C3              ret
```

Principaux registres du processeur

1. Le compteur de programme EIP, Ce compteur indique (contient) l'adresse de la prochaine instruction à exécuter. Il ne peut être manipulé directement par programme.
2. Le fichier des registres d'entiers: il contient huit registres. Chaque registre peut contenir une valeur de 32 bits dans le cas d'une architecture 32 bits comme IA32. Ces registres sont:
 - i. EAX , EBX , ECX , EDX (généraux)
 - ii. ESI , EDI (Utilisés par les chaînes)
 - iii. ESP , EBP (Utilisés par la pile)

Ajouter ici les registres de IA32 à partir de la documentation d'intel

Typage niveau assembleur

En assembleur on ne fait pas de distinction entre

- (1) Les entiers signés et les entiers non signés
- (2) Les adresses (pointeurs) entres elles
- (3) Entre les adresses et les entiers
- (4) Entre les entiers et les caractères
- (5) Entre structures, unions, classes, tableaux, etc

**TOUT TYPE EST UN ENSEMBLES DE
UN OU PLUSIEURS OCTETS...**

Typage niveau assembleur

QUESTION: Mais alors, qui effectue le typage des programmes en assembleur ??

RÉPONSE: C'est le programmeur qui assure le typage... Il n'a pas intérêt à se tromper...

Typage: dans la tête du programmeur

Structures de données coté assembleur

Les tableaux, les structures et les unions sont perçus pareillement au niveau de l'assembleur.

Les structures de données composées sont perçues comme

« des *collections d'octets contigus* (côte à côte) »

Tableaux \equiv structures \equiv unions \equiv "collections d'octets contigus"

instruction = opcode + opérande(s)

- Une instruction à un code d'opération noté opcode + une à plusieurs opérandes.
- Les opérandes (si nécessaires) contiennent
 - les valeurs qui constituent les données de l'opération à effectuer et / ou
 - l'adresse de la destination du résultat.
- Syntaxe:
 - Les « () » signifient optionnel.
 - Le « * » signifie 0 ou plusieurs.
 - Le « | » signifie ou.

(label:) OPCODE (operande)* (; commentaire)

instruction = opcode + opérande(s)

- Syntaxe:

- Les « () » signifient optionnel.
- Le « * » signifie 0 ou plusieurs.
- Le « | » signifie ou.

(label:) OP CODE (operande)* (; commentaire)

Les instructions = opération-code +
opérandes: **OPCODE (*op*)***

(label:) OPCODE (operande*) (; commentaire)

• **Exemples:**

;;; **Pour écrire la valeur décimale 1 dans le registre
EAX:**

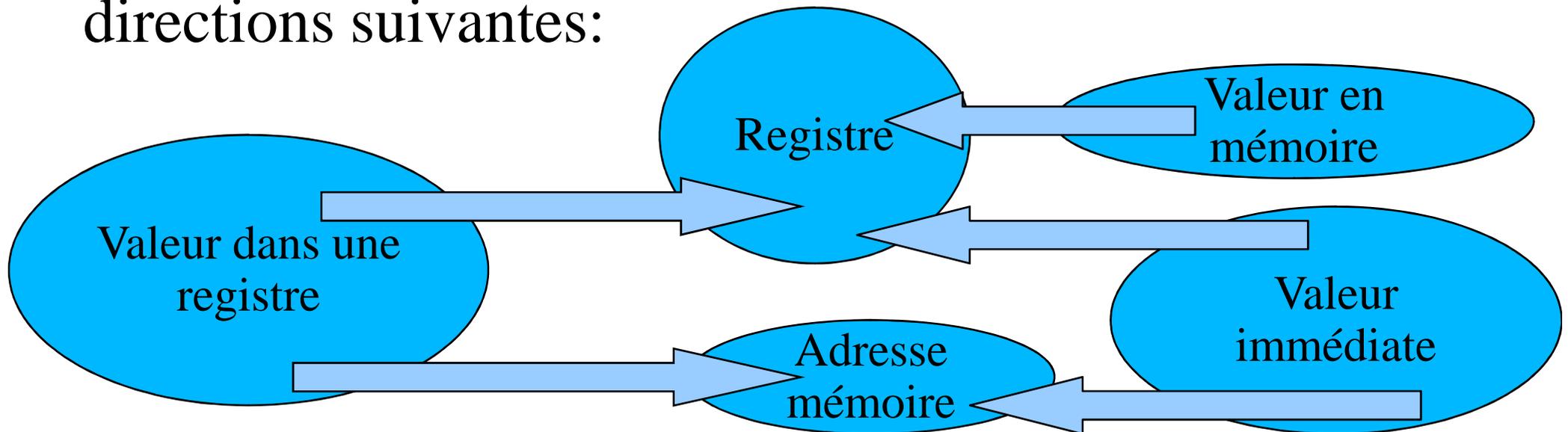
mov EAX , 1 ; écrit 1 dans eax

;;; **Pour lire Mémoire(EBP+8) et l'écrire dans ESP**

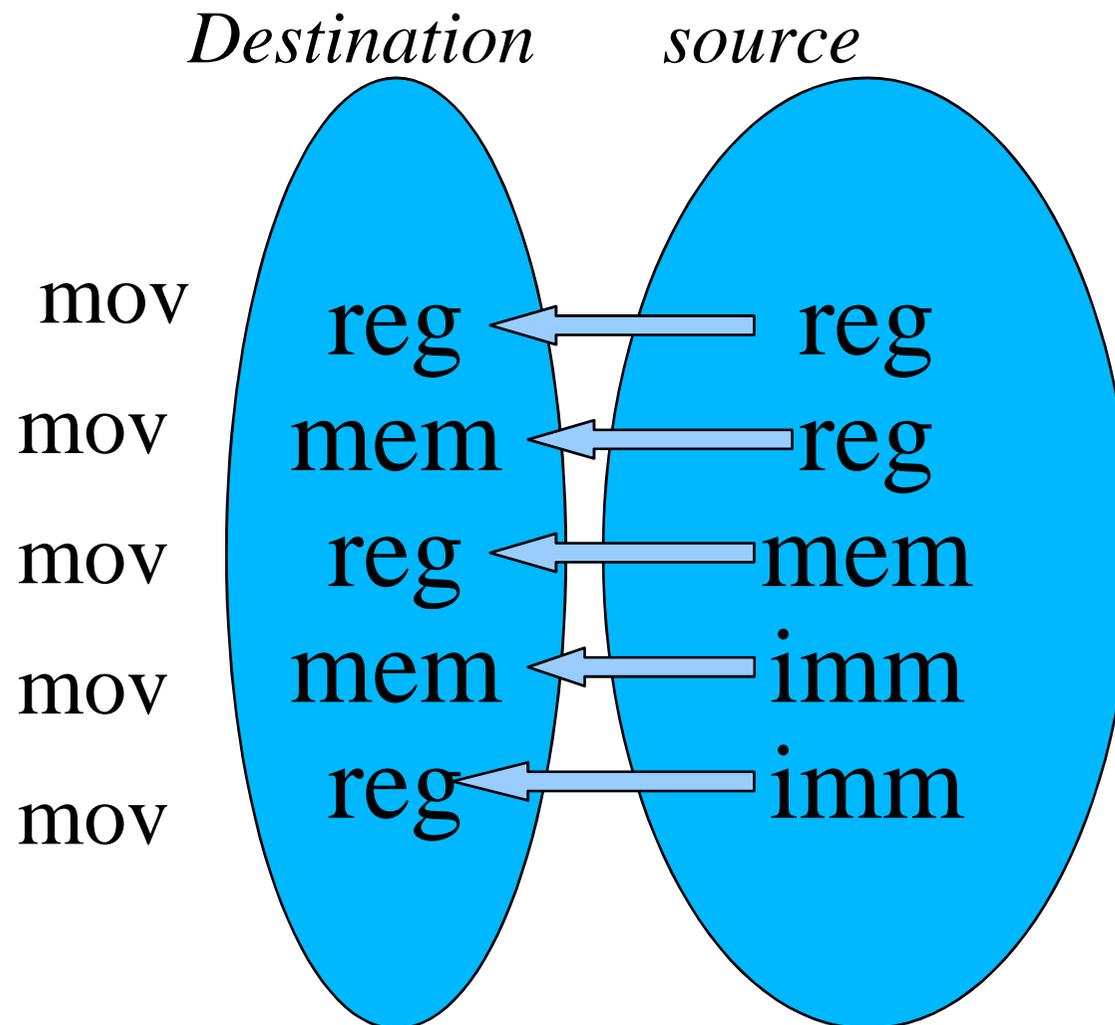
mov ESP , [EBP + 8]

Instructions de transfert de données

- Parmi les instructions les plus utilisées figure celle qui déplace/transfère les données. C'est l'instruction dont le code d'opération (**opcode**) est **MOV**. Les transferts peuvent se faire dans les directions suivantes:



L'instruction mov, jamais de mémoire à mémoire : il faut passer par un registre



Instructions de transfert de données

- Une opération *MOV* ne peut avoir deux opérandes qui sont toutes les deux des adresses mémoires. Autrement dit, il n'existe pas en IA32 de transfert direct de la mémoire vers la mémoire: il faut passer par un registre. Cette limitation est liée au matériel et non pas à la théorie !

Format des instructions dans un assembleur

(label:) opcode (opérandes) (; commentaire)*

→ Généralement on distingue deux types d'instructions:

- 1- les instructions directives donnent des directives qui ne sont pas des instructions exécutables. Ex: *extern _printf*
- 2- les instructions exécutables. Ex: *PI: DD 3.14 ; le réel pi*

→ Exemples d'instructions *directive*:

extern _printf ; printf sera importée

global _main ; le _main sera exporté

Formats d'opérandes en assembleur IA32

Modes d'adressage

Type	Forme	Valeur d'opérandes	Nom
Immédiate	Imm= 12q,10,0xA,Ah,\$0A,1010b	Valeur immédiate de Imm)	Immédiate
Registre	REG	Valeur contenue dans REG	Registre
Mémoire	[Imm]	Mémoire (Imm)	Absolu
Mémoire	[REG]	Mémoire (REG)	Indirect
Mémoire	[REGb + Imm])	Mémoire (REGb+Imm)	Base + déplacement
Mémoire	[REGb + REGi]	Mémoire (REGb + REGi)	Indexé
Mémoire	[Imm + REGb + REGi]	Mémoire (Imm + REGb + REGi)	Indexé
Mémoire	[REGi * s]	Mémoire (REGi* s)	Indexé, pondéré
Mémoire	[Imm+ REGi * s]	Mémoire (Imm + (REGi* s))	Indexé, pondéré
Mémoire	[REGb+ REGi * s]	Mémoire (REGb + (REGi* s))	Indexé, pondéré
Mémoire	[Imm + REGb + REGi * s]	Mémoire (Imm+REGb+(REGi* s))	Indexé, pondéré

Note 1: REG peut être un des registres EAX , EBX, ECX, EBP, ESP, EDI, ESI etc ...

Note 2: Mémoire(@) signifie contenue de la mémoire à l'adresse @

Mémoire		Registres	
Adresse	Valeur	Registre	Valeur
0x100	0xFF	EAX	0x100
0x104	0xAB	ECX	0x1
0x108	0x13	EDX	0x3
0x10C	0x11		

Que valent les opérandes suivantes ?

EAX	
[0x104]	
0x108	
[EAX]	
[EAX+4]	
[9+EAX+EDX]	
[260+ECX+EDX]	
[0xFC+ECX*4]	
[EAX+EDX*4]	

Exemples:

Soit la mémoire

Et les registres

Suivants:

Adresse	Valeur
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Registre	Valeur
EAX	0x100
ECX	0x1
EDX	0x3

Que valent les opérandes suivantes ?

% eax	0x100	
[0x104]	0xAB	
0x108	0x108	
[EAX]	0xFF	
[EAX+4]	0xAB	
[9+EAX+EDX]	0x11	$9+0x100+0x3=0x10C$
[260+ECX+EDX]	0x13	$260=16*16+4=0x104$
[0xFC+ECX*4]	0xFF	$0xFC+0x1*4 = 0x100$
[EAX+EDX*4]	0x11	$0x100+0x3*4=0x10C$

Code niveau machine des instructions IA32

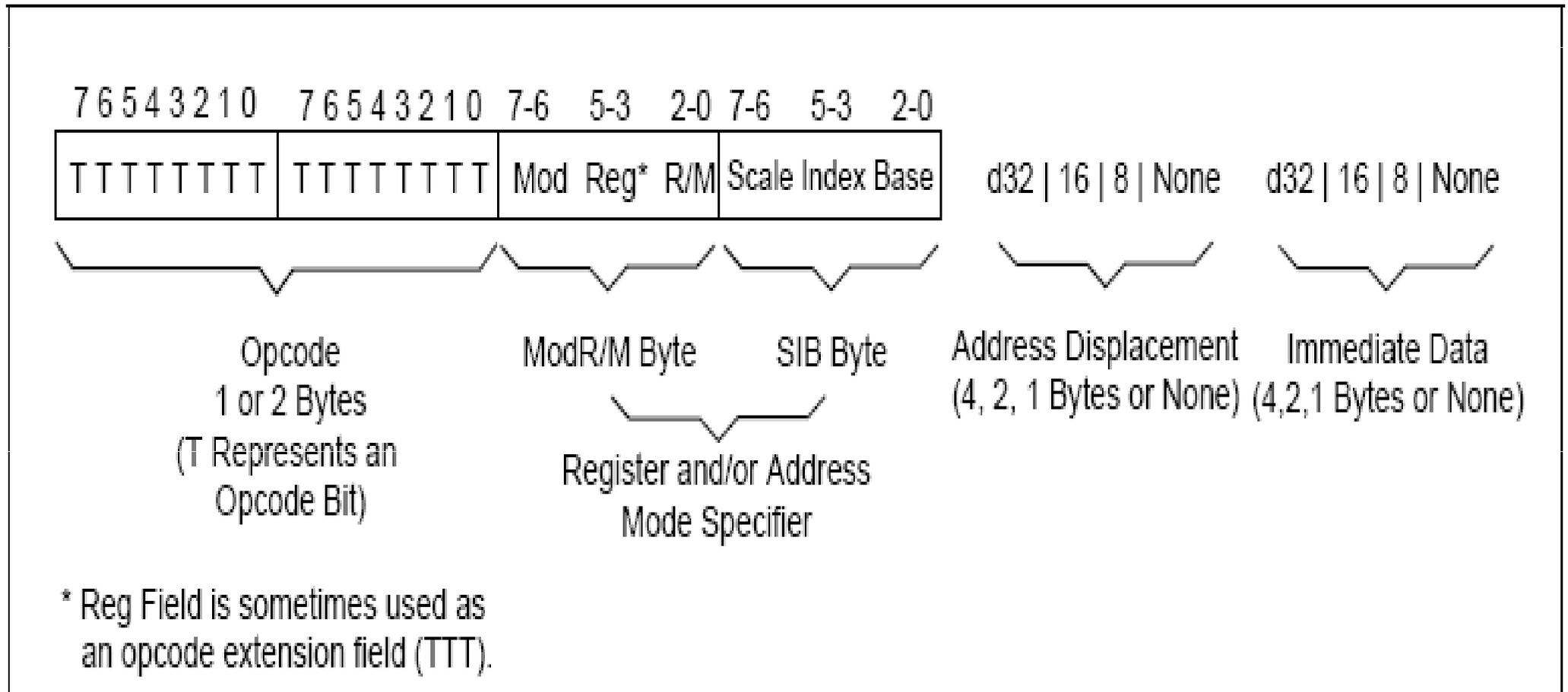


Figure B-1. General Machine Instruction Format

Mettre les différentes OPCCODE de
jmp, jne, je, etc au lieu de ADC

Exemple 1 : ADC addition avec retenue

ADC—Add with Carry

Opcode	Instruction	Description
14 <i>ib</i>	ADC AL, <i>imm8</i>	Add with carry <i>imm8</i> to AL
15 <i>iw</i>	ADC AX, <i>imm16</i>	Add with carry <i>imm16</i> to AX
15 <i>id</i>	ADC EAX, <i>imm32</i>	Add with carry <i>imm32</i> to EAX
80 <i>2 ib</i>	ADC <i>r/m8</i> , <i>imm8</i>	Add with carry <i>imm8</i> to <i>r/m8</i>
81 <i>2 iw</i>	ADC <i>r/m16</i> , <i>imm16</i>	Add with carry <i>imm16</i> to <i>r/m16</i>
81 <i>2 id</i>	ADC <i>r/m32</i> , <i>imm32</i>	Add with CF <i>imm32</i> to <i>r/m32</i>
83 <i>2 ib</i>	ADC <i>r/m16</i> , <i>imm8</i>	Add with CF sign-extended <i>imm8</i> to <i>r/m16</i>
83 <i>2 id</i>	ADC <i>r/m32</i> , <i>imm8</i>	Add with CF sign-extended <i>imm8</i> into <i>r/m32</i>
10 <i>lr</i>	ADC <i>r/m8</i> , <i>r8</i>	Add with carry byte register to <i>r/m8</i>
11 <i>lr</i>	ADC <i>r/m16</i> , <i>r16</i>	Add with carry <i>r16</i> to <i>r/m16</i>
11 <i>lr</i>	ADC <i>r/m32</i> , <i>r32</i>	Add with CF <i>r32</i> to <i>r/m32</i>
12 <i>lr</i>	ADC <i>r8</i> , <i>r/m8</i>	Add with carry <i>r/m8</i> to byte register
13 <i>lr</i>	ADC <i>r16</i> , <i>r/m16</i>	Add with carry <i>r/m16</i> to <i>r16</i>
13 <i>lr</i>	ADC <i>r32</i> , <i>r/m32</i>	Add with CF <i>r/m32</i> to <i>r32</i>

Exemple 1 : ADC addition avec retenue

- L'instruction ADC additionne le premier opérande (nommé destination) avec le second opérande (nommé source) et le drapeau de retenue. ADC met son résultat dans l'opérande destination (le premier en lisant de gauche à droite)
- Les deux opérandes ne peuvent être simultanément des adresses en mémoire.
- L'opérande source peut-être une valeur immédiate.
- Le drapeau CF (carry flag) représente une retenue d'un addition précédente.

Exemple 1 : ADC

Operation

$DEST \leftarrow DEST + SRC + CF;$

Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

Exemple 2 : CMP compare 2 opérandes

CMP—Compare Two Operands

Opcode	Instruction	Description
3C <i>ib</i>	CMP AL, <i>imm8</i>	Compare <i>imm8</i> with AL
3D <i>iw</i>	CMP AX, <i>imm16</i>	Compare <i>imm16</i> with AX
3D <i>id</i>	CMP EAX, <i>imm32</i>	Compare <i>imm32</i> with EAX
80 <i>7 ib</i>	CMP <i>r/m8</i> , <i>imm8</i>	Compare <i>imm8</i> with <i>r/m8</i>
81 <i>7 iw</i>	CMP <i>r/m16</i> , <i>imm16</i>	Compare <i>imm16</i> with <i>r/m16</i>
81 <i>7 id</i>	CMP <i>r/m32</i> , <i>imm32</i>	Compare <i>imm32</i> with <i>r/m32</i>
83 <i>7 ib</i>	CMP <i>r/m16</i> , <i>imm8</i>	Compare <i>imm8</i> with <i>r/m16</i>
83 <i>7 id</i>	CMP <i>r/m32</i> , <i>imm8</i>	Compare <i>imm8</i> with <i>r/m32</i>
38 <i>lr</i>	CMP <i>r/m8</i> , <i>r8</i>	Compare <i>r8</i> with <i>r/m8</i>
39 <i>lr</i>	CMP <i>r/m16</i> , <i>r16</i>	Compare <i>r16</i> with <i>r/m16</i>
39 <i>lr</i>	CMP <i>r/m32</i> , <i>r32</i>	Compare <i>r32</i> with <i>r/m32</i>
3A <i>lr</i>	CMP <i>r8</i> , <i>r/m8</i>	Compare <i>r/m8</i> with <i>r8</i>
3B <i>lr</i>	CMP <i>r16</i> , <i>r/m16</i>	Compare <i>r/m16</i> with <i>r16</i>
3B <i>lr</i>	CMP <i>r32</i> , <i>r/m32</i>	Compare <i>r/m32</i> with <i>r32</i>

Détailler CMP r/m32, r32 et CMP
r/m32, i32

Exemple 2 : CMP compare 2 opérandes

Description

This instruction compares the first source operand with the second source operand and sets the status flags in the EFLAGS register according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction. When an immediate value is used as an operand, it is sign-extended to the length of the first operand.

Exemple 2 : CMP compare 2 opérandes

- L'instruction CMP compare deux opérandes : le premier avec le second, puis positionne les drapeaux du registre d'état conformément au résultat.
- La comparaison est effectuée en soustrayant le second opérande du premier. CMP positionne les drapeaux du registre d'état de la même façon que l'instruction SUB (soustraction)
- Lorsqu'une valeur immédiate est utilisée comme seconde opérande, alors elle est étendue en conservant son signe à la taille du premier opérande.

Les Macros en NASM

```
%macro nomMacro n      ;;; n est le nombre de paramètres
    OPCODE1 <operandes>
    ...
    OPCODEp <operandes>
%endmacro
```

;;; Dans le corps de la macro,

;;; %1 dénote le premier argument en ligne arg1

;;; %n référence argn

APPEL DE LA MACRO :

```
nomMacro arg1 , ... , argn
```

Donnez la définition des macros pour les appels de fonctions avec arguments