

MI01 – Automne 2011

Programmation des processeurs IA-32

Stéphane Bonnet

Poste : 52 56

Courriel : stephane.bonnet@utc.fr

Sommaire

1. Modèles de programmation
2. L'assembleur
3. Types de données
4. Modèle mémoire
5. Registres
6. Modes d'adressage
7. Jeu d'instruction

Quelques références

- Processeur et Coprocesseur, R. Hummel.
Dunod Tech, 1992.
- The Art of Assembly Language, R. Hyde.
No Starch Press, 2003. Disponible en ligne:
<http://webster.cs.ucr.edu/AoA/>
- Intel 64 and IA-32 Architectures Software
Developer's Manuals. Disponibles en ligne:
<http://www.intel.com/products/processor/manuals>

Sommaire

1. Modèle de programmation
2. Les outils
3. Types de données
4. Modèle mémoire
5. Registres
6. Modes d'adressage
7. Jeu d'instruction

1. Modèle de programmation

- Représente le processeur tel qu'il est vu par le programmeur d'applications.
- Différent du modèle physique du processeur (électronique) :

Abstraction du fonctionnement du processeur

- Dans ce cours : modèle IA-32 (*Intel Architecture 32-bit*), défini par Intel pour les 80386 et ses processeurs ultérieurs.

1. Modèle de programmation

Le modèle de programmation comporte les aspects suivants :

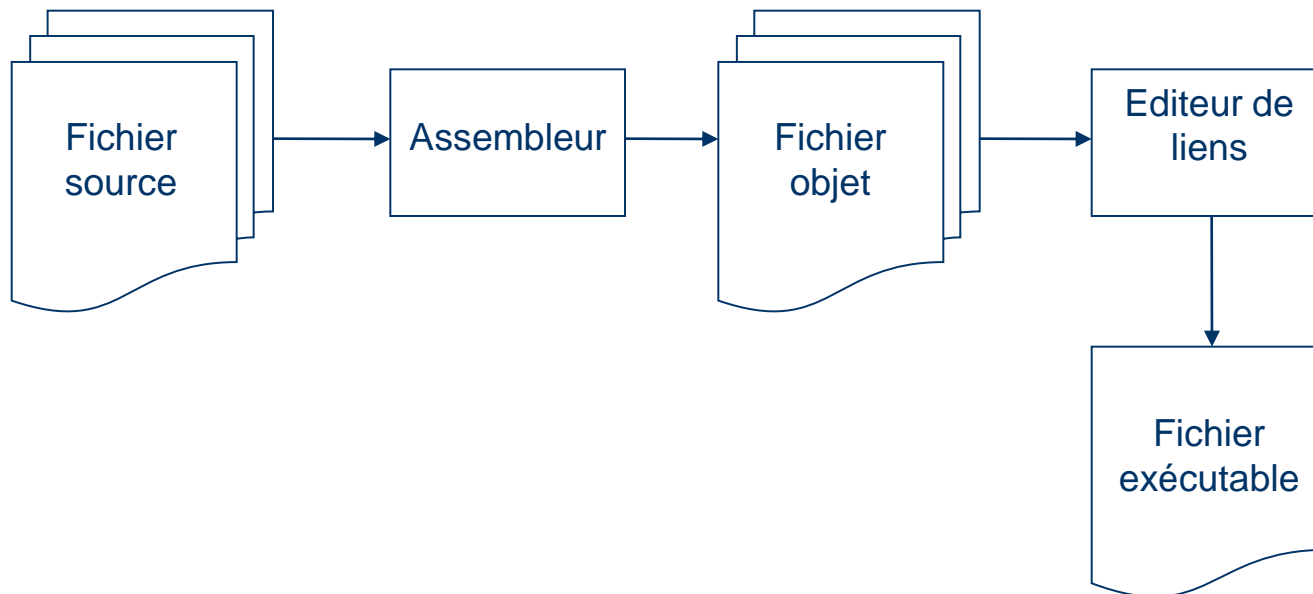
- Types de données manipulables
- Organisation de la mémoire
- Registres internes du processeur
- Modes d'adressage
- Format des instructions
- Jeu d'instructions

Sommaire

1. Définition
2. Les outils
 1. L'assembleur
 2. L'éditeur de liens
3. Types de données
4. Modèle mémoire
5. Registres
6. Modes d'adressage
7. Jeu d'instruction

2.1. L'assembleur

- L'assembleur est un logiciel de traduction de programmes écrits en langage d'assemblage en fichiers exécutables (langage machine).



2.1. L'assembleur

L'assembleur transforme ceci...
(langage d'assemblage) :

MOV AL, 0D0h

En ceci (code machine) :

10110000 11010000

2.1. L'assembleur

Le format d'une ligne de programme est :

étiquette: MOV op1, op2 ; commentaire

- « étiquette » (optionnelle) représente l'adresse virtuelle de l'instruction. Elle permet d'y faire référence ailleurs dans le programme. C'est une constante symbolique.
- MOV représente l'action à effectuer (instruction). On appelle **mnémonique** le nom des instructions.
- op1 et op2 sont les opérandes (données) sur lesquelles on veut agir.

Exemple

```
depart:            MOV  eax, ebx            ; Copier ebx dans eax
```

2.1. L'assembleur

- Syntaxe (Convention « INTEL »)

- Registres : le nom du registre

`MOV EAX, 12`

- Adresses mémoire : crochets

`MOV [6000h], 12`

`MOV [adresse], 12` ; Adresse est une étiquette

- Constantes (débutent par un **chiffre** ou un **signe**)

- Décimales par défaut `MOV EAX, 12`
- Binaires, suffixe « b » `MOV EAX, 1100b`
- Octales, suffixe « o » `MOV EAX, 14o`
- Hexadécimales, suffixe « h » `MOV EAX, 0Ch`
- Négatives (quelque soit la base) `MOV EAX, -0Ch`
 - Produit le complément à 2 de la valeur absolue

- Etiquettes

- Chaîne alphanumérique ne débutant **pas par un chiffre**.
- La « valeur » d'une étiquette est l'adresse en mémoire de l'objet qui la suit (instruction, donnée ...)

2.1. L'assembleur

- En général, dans une instruction (ex : MOV op1, op2)
 - op1 représente la **destination**: endroit où le résultat sera stocké.
 - op2 représente la **source**: donnée qui sera combinée avec op1 pour calculer le résultat de l'instruction.
- Sauf (rares!) exceptions, la source et la destination **doivent être de même taille.**

2.1. L'assembleur

- L'assembleur détermine automatiquement la taille des opérandes quand il le peut :
 - `MOV EAX, FFh` Opérandes 32 bits car EAX est un registre 32 bits.
 - `MOV [6000h], FFh` On ne sait pas...
 - Il faut spécifier explicitement la taille de l'opérande :
 - `MOV byte ptr [6000h], FFh` 8 bits
 - `MOV word ptr [6000h], FFh` 16 bits
 - `MOV dword ptr [6000h], FFh` 32 bits

2.2. Rôle de l'éditeur de liens

- L'assemblage (ou la compilation) d'un fichier source produit un fichier objet. Il peut y en avoir plusieurs dans un programme
- L'éditeur de liens prend en entrée les fichiers objets
- L'éditeur de liens rassemble ces fichiers et décide des adresses mémoire définitives des données et instructions
- L'éditeur de liens produit en sortie un fichier exécutable.

2.2. Rôle de l'éditeur de liens

1		
2	00000000	
3		
4	00000000	
5	00000000	00000000
6	00000004	0000000C 0000002A
7	00000012	00000012 00000017
8		
9		
10	00000000	
11	00000000	
12	00000000	B9 00000000
13	00000005	A1 00000000r
14	0000000A	
15	0000000A	03 048D 00000004r
16	00000011	41
17	00000012	83 F904
18	00000015	75 F3
19	00000017	A3 00000000r
20		

Adresse virtuelle **Contenu de la mémoire**

```
.386
.model flat

.data
sum:      dd 0
tab:      dd 12,42,18,23

.code
_start:
main:
    mov ecx, 0
    mov eax, [sum]
@loop:
    add eax, [tab + ecx * 4]
    inc ecx
    cmp ecx, 4
    jne @loop
    mov [sum], eax
end
```

2.2. Rôle de l'éditeur de liens

- Dans ce listing toutes les références à la mémoire sont notées avec le suffixe 'r'. Ceci indique que la référence n'est pas résolue.
- C'est l'éditeur de liens qui définira quelle est l'adresse virtuelle exacte correspondante, et non l'assembleur, pour fournir une image que le système d'exploitation peut charger en mémoire.
- Ainsi, si l'éditeur de liens décide que les données débutent à l'adresse virtuelle 80040000, la ligne 15 sera remplacée par :

15 0000000A 03 04 8D 80040004 add eax, [tab + ecx * 4]

dans le programme exécutable final.

Sommaire

1. Définition
2. Les outils
- 3. Types de données**
4. Modèle mémoire
5. Registres
6. Modes d'adressage
7. Jeu d'instruction

3. Types de données

- Le processeur peut manipuler directement ses **types fondamentaux**
 - L'octet (*byte*)
 - Le mot de 16 bits (*word*)
 - Le double mot de 32 bits (*double word, dword*)
 - Représentés en complément à 2, signe = bit de poids fort.
- Types secondaires traités par des unités spécifiques
 - Flottants (standard IEEE 754)
 - Quadruples mots de 64 bits (*quad word, qword*)

3. Types de données

- En C par exemple, on peut écrire `int x;`
=> À toute donnée est associée un **type** qui en identifie la nature (ici, sur IA32, x est un entier signé exprimé sur 32 bits).
- Cependant, les microprocesseurs ne s'occupent que de suites d'octets. Ils ne se soucient pas de savoir si ce sont des entiers, des caractères, s'ils sont signés ou non signés... c'est le programmeur qui identifie la signification des données, pas le processeur !

Sommaire

1. Définition
2. Les outils
3. Types de données
- 4. Modèle mémoire**
5. Registres
6. Modes d'adressage
7. Jeu d'instruction

4. Modèle de mémoire

- Dépend du mode de fonctionnement du processeur. Dans ce cours on considère le modèle dit « flat » :
 - les adresses virtuelles varient de 0 à $2^{32} - 1$
 - Il existe 3 espaces d'adressage, appelés segments, superposés :
 - Un segment de code de 4 Go, qui contient les instructions du programme,
 - Un segment de données de 4 Go,
 - Un segment de pile de 4 Go.

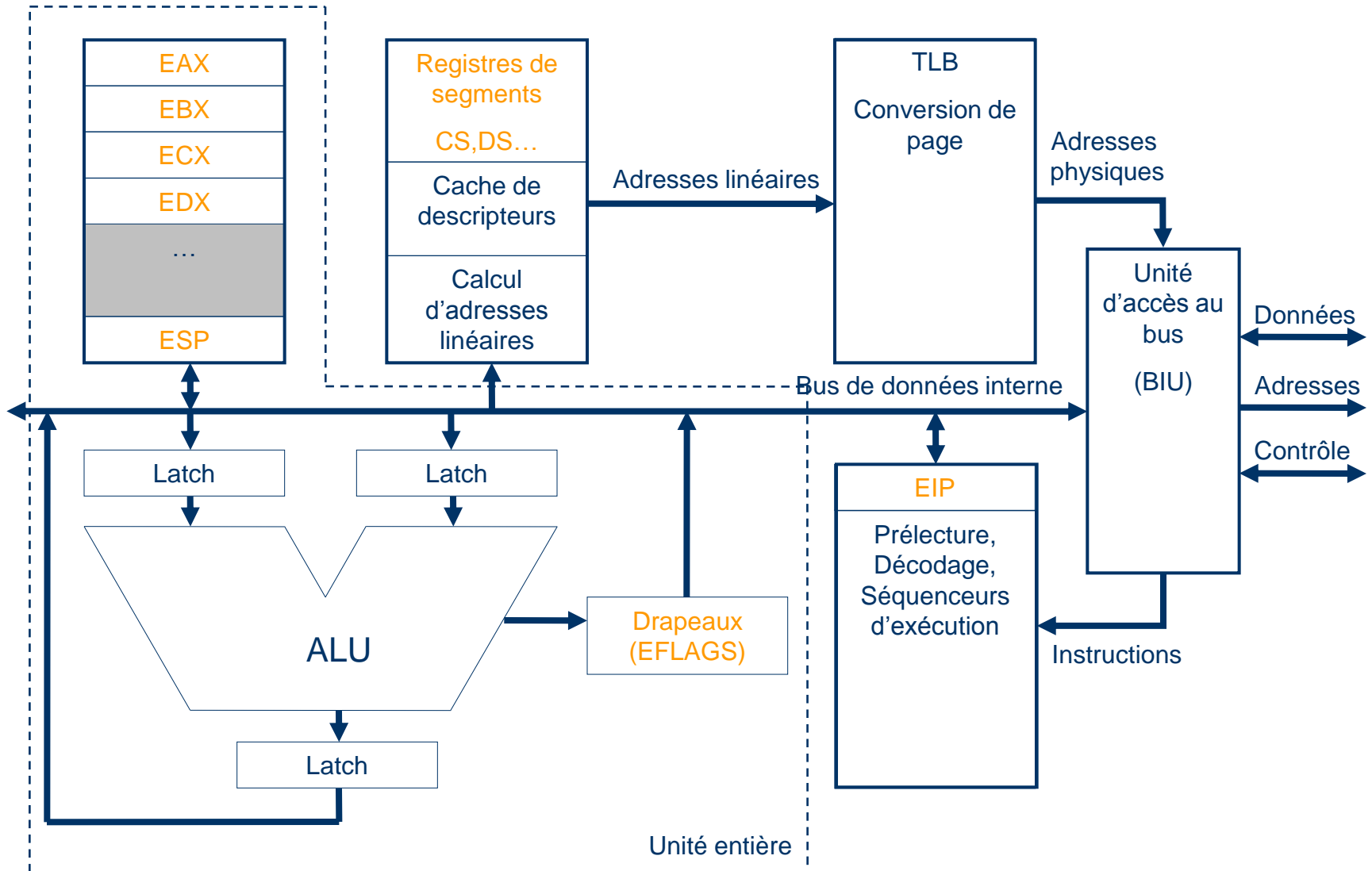
Sommaire

1. Définition
2. Les outils
3. Types de données
4. Modèle mémoire
- 5. Registres**
 1. Registres généraux
 2. Index et pointeurs
 3. Drapeaux
 4. Registres de segmentation
6. Modes d'adressage
7. Jeu d'instruction

5. Registres

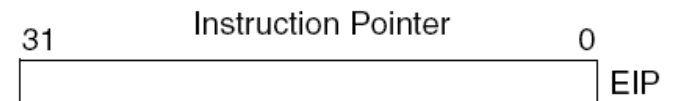
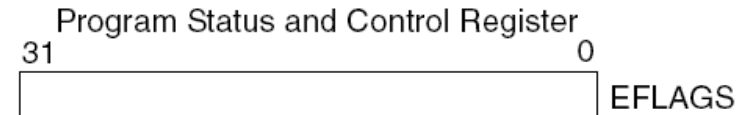
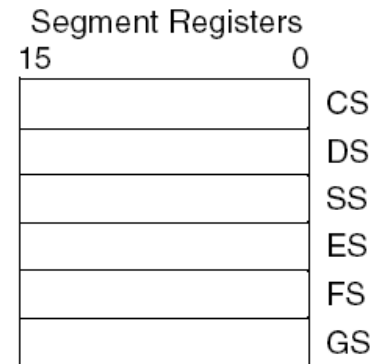
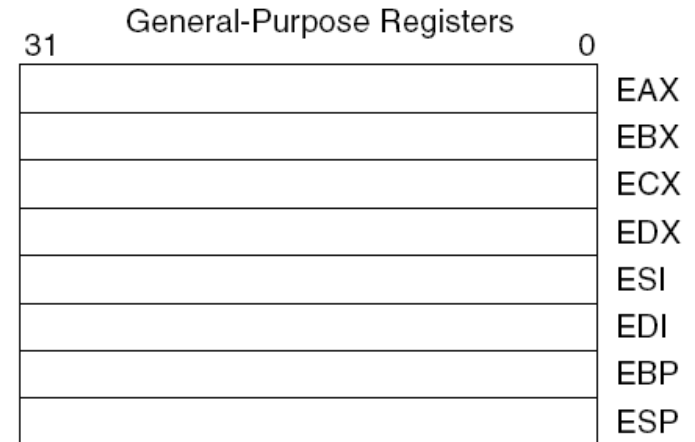
- Zones de mémoire internes au processeur, qui permettent de :
 - Stocker des valeurs intermédiaires dans les calculs
 - Représenter l'état du processeur
 - Savoir où trouver en mémoire l'instruction à exécuter
 - Savoir où trouver en mémoire les données du programme...

5. Registres



5. Registres

- 4 types de registres
 - Registres généraux
 - Registres d'index
 - Registres pointeurs
 - Registres de segment
- Registre des drapeaux (EFLAGS)
- Pointeur d'instruction (EIP)



5.1. Registres d'usage général

- Opérandes pour les instructions arithmétiques et logiques
- Valeurs intermédiaires dans les calculs
- Peuvent être adressés comme :
 - 1 registre 32 bits
 - 1 registre 16 bits
 - 2 registres 8 bits

General-Purpose Registers								
31	16	15	8	7	0		16-bit	32-bit
			AH		AL		AX	EAX
			BH		BL		BX	EBX
			CH		CL		CX	ECX
			DH		DL		DX	EDX
			BP					EBP
			SI					ESI
			DI					EDI
			SP					ESP

5.2. Registres d'index

- Utilisés pour l'adressage indexé en mémoire
- Opérandes pour les instructions arithmétiques et logiques
- Peuvent être adressés comme :
 - 1 registre 32 bits
 - 1 registre 16 bits.

General-Purpose Registers							
31	16	15	8	7	0	16-bit	32-bit
			AH		AL	AX	EAX
			BH		BL	BX	EBX
			CH		CL	CX	ECX
			DH		DL	DX	EDX
			BP				EBP
			SI				ESI
			DI				EDI
			SP				ESP

5.2. Registres pointeurs

- ESP est le pointeur de pile
 - Le processeur s'attend toujours à disposer d'une pile valide
- EBP est utilisé pour manipuler des données sur la pile
- Peuvent être adressés comme :
 - 1 registre 32 bits
 - 1 registre 16 bits

General-Purpose Registers							
31	16	15	8	7	0	16-bit	32-bit
			AH		AL	AX	EAX
			BH		BL	BX	EBX
			CH		CL	CX	ECX
			DH		DL	DX	EDX
			BP				EBP
			SI				ESI
			DI				EDI
			SP				ESP

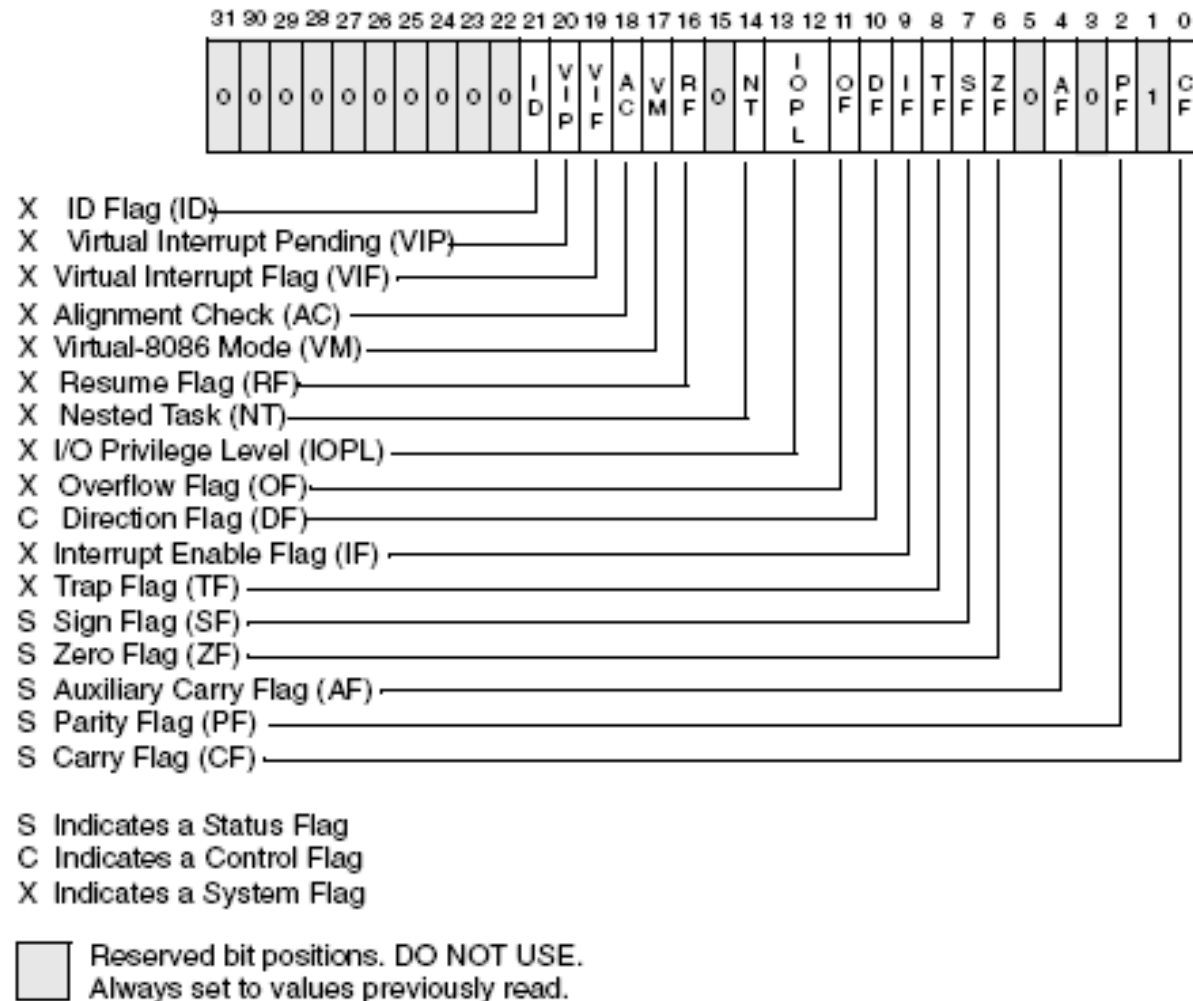
5.2. Registre pointeur d'instructions

- **EIP** est le pointeur d'instructions : contient l'adresse du premier octet de l'instruction qui **suit immédiatement** l'instruction en cours
- Il n'est pas adressable : on ne peut pas le modifier directement
- Seules les instructions de saut et d'appel/retour de sous-programme peuvent le modifier

5.3. Drapeaux (EFLAGS)

- Représentent l'état du processeur et contrôlent son comportement
- Il est divisé en bits, chaque bit est appelé *drapeau* (flag)
- Un drapeau est soit :
 - **Positionné** (bit = 1)
 - **Effacé** (bit = 0)
- Mis à jour **automatiquement** lors de l'exécution de la plupart des instructions : permet d'agir en fonction du résultat de l'instruction

5.3. Drapeaux




5.3. Drapeaux d'état

- **CF : drapeau de retenue**
- PF : drapeau de parité
- AF : drapeau de retenue auxiliaire
- **ZF : drapeau zéro**
- **SF : drapeau de signe**
- **OF : drapeau de débordement**

5.3. Drapeau de retenue (CF)

- Indique si une opération arithmétique a provoqué un débordement

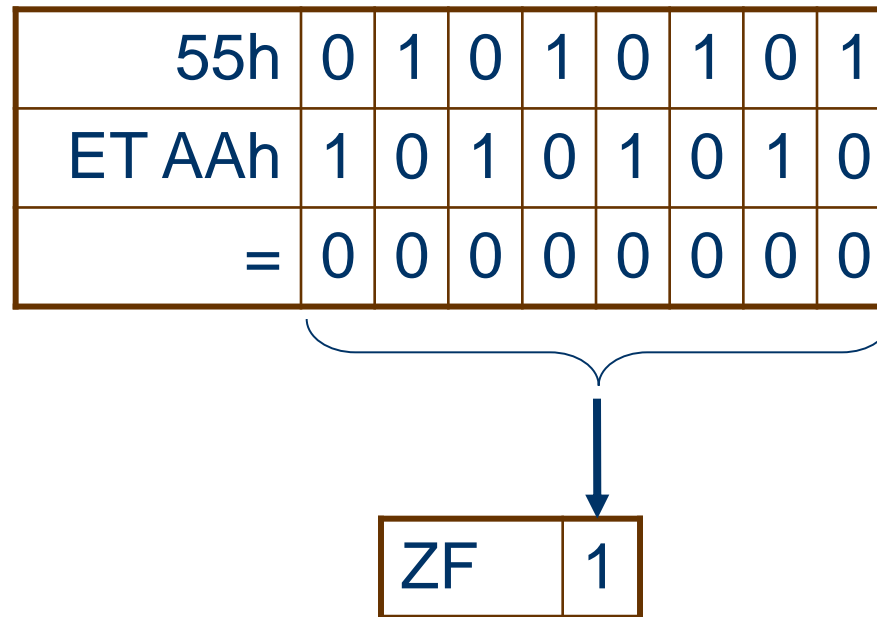
55h		0	1	0	1	0	1	0	1
+B1h		1	0	1	1	0	0	0	1
=	1	0	0	0	0	0	1	1	0



CF	1
----	---

5.3. Drapeau Zéro (ZF)


- Indique un résultat nul lors d'une opération arithmétique ou logique



5.3. Drapeau de signe SF

- Reçoit le contenu du bit le plus significatif après une opération arithmétique et logique
- N'a de sens que sur les nombres signés !

-2h	1	1	1	1	1	1	1	0
+1h	0	0	0	0	0	0	0	1
= -1h	1	1	1	1	1	1	1	1



SF	1
----	---

5.3. Drapeau de débordement OF

- Indique un dépassement de capacité dans les opérations sur des nombres signés

7Fh	0	1	1	1	1	1	1	1
+7Fh	0	1	1	1	1	1	1	1
=	1	1	1	1	1	1	1	0

↓ XOR

OF	1
----	---

- Indique si le calcul a débordé sur le bit de signe, ce qui le rend invalide : ne peut arriver que si on fait l'addition de deux nombres de même signe

5.4. Registres de segment

- Registres de 16 bits, utilisés pour la gestion mémoire.
- Permettent de configurer l'espace d'adressage dédié aux éléments du programme
 - **CS** (Code Segment) pour les instructions
 - **DS** (Data Segment) pour les données
 - **SS** (Stack Segment) pour la pile
 - **ES, FS, GS** pour définir des espaces supplémentaires
- Plus de détails dans la partie « Gestion de la mémoire » du cours !

Sommaire

1. Définition
2. Les outils
3. Types de données
4. Modèle mémoire
5. Registres
- 6. Modes d'adressage**
7. Jeu d'instruction

6. Modes d'adressage

- L'adresse d'une donnée est ce qui permet de la retrouver.
- Une donnée peut être soit
 - Dans un registre interne, dans ce cas l'adresse est le nom du registre ;
 - Dans la mémoire externe, dans ce cas son adresse est la valeur qu'il faut placer sur le bus d'adresse pour pointer sur la bonne case mémoire.
 - **Le seul moyen d'accéder à une donnée est de connaître son adresse**
- La manière dont l'adresse d'une donnée est spécifiée dans une instruction est le **mode d'adressage**

6. Modes d'adressage

- Opérande immédiate

- Permet la manipulation de constantes

MOV EAX, 0FFAA4CD0h

EAX \leftarrow 0FFAA4CD0h

- Registre direct

- Permet la manipulation des registres

MOV EAX, EBX

EAX \leftarrow EBX

- Mémoire direct

- Permet la manipulation de données en mémoire dont l'adresse est constante

MOV EAX, [6000h]

EAX \leftarrow mot de 32 bits à l'adresse 6000h.

6. Modes d'adressage

- Manipulation d'une donnée d'adresse variable
 - Donnée dont on ne connaît pas l'adresse à priori, mais on sait où trouver cette adresse (ou comment la calculer)
 - => On connaît un **pointeur** vers la donnée
 - => Utilisation de modes d'adressage **indirects**.
 - Les modes d'adressage indirects diffèrent en fonction de la taille d'adresse :
 - Modes 16 bits (l'adresse est contenue dans un registre 16 bits)
 - Modes 32 bits (l'adresse est contenue dans un registre 32 bits)

6. Modes d'adressage indirects 32 bits

Forme générale

```
MOV EAX, [ rb + ri * echelle + depl ]
```

Où :

rb : registre de **base**

ri : registre **d'index**

échelle : constante multiplicative 1, 2, 4 ou 8

depl : constante signée additive (déplacement)

Chaque partie peut être omise.

6. Modes d'adressage indirects 32 bits

- Indirect simple
 - `MOV EAX, [EBX]`
 $EAX \leftarrow \text{double mot à l'adresse mémoire } EBX.$
- Indirect avec déplacement
 - `MOV EAX, [EBX + 2]`
 $EAX \leftarrow \text{double mot à l'adresse mémoire } EBX + 2$
- Indirect avec facteur d'échelle et déplacement
 - `MOV EAX, [EBX * 2 + 2]`
 $EAX \leftarrow \text{double mot à l'adresse mémoire } EBX \times 2 + 2$

6. Modes d'adressage indirects 32 bits

- Indirect indexé simple
 - `MOV EAX, [EBX + ESI]`
EAX \leftarrow double mot à l'adresse mémoire EBX + ESI.
- Indirect indexé avec déplacement
 - `MOV EAX, [EBX + ESI + 2]`
EAX \leftarrow double mot à l'adresse mémoire EBX + ESI + 2
- Indirect indexé avec facteur d'échelle et déplacement
 - `MOV EAX, [EBX + ESI * 2 + 2]`
EAX \leftarrow double mot à l'adresse mémoire EBX + ESI x 2 + 2

Modes d'adressage

- Sauf exception :

Jamais deux accès à la mémoire dans une même instruction !

`MOV [ESI], word ptr [7000h] => Interdit`



```
MOV AX, [7000h]  
MOV [ESI], AX
```

Sommaire

1. Définition
2. Les outils
3. Types de données
4. Modèle mémoire
5. Registres
6. Modes d'adressage
- 7. Instructions**
 1. Encodage
 2. Jeu d'instructions

7.1. Encodage des instructions

- Comment une instruction est représentée en mémoire de programme ?
- Une instruction se compose de
 - Une opération et, *éventuellement* :
 - Une opérande de destination
 - Une opérande source

ADD EAX , 2

$EAX \leftarrow EAX + 2$

7.1. Encodage des instructions

- L'encodage est la représentation binaire en mémoire d'une instruction. Il est directement interprétable par le processeur
 - langage machine.
- IA32 : processeurs CISC (*Complex Instruction Set Computers*)
 - encodage de taille variable, de 1 à 16 octets
 - nombre important d'instructions, modes d'adressages complets.
- processeurs RISC (*Reduced Instruction Set Computers*), tels que Sparc(SUN), ARM (Acorn), Alpha (Digital), PowerPC (IBM), MIPS...
 - encodage de taille fixe
 - peu d'instructions, peu de modes d'adressage (en général deux instructions spécialisées type *LOAD* et *STORE*)

7.1. Encodage des instructions

- Code opération (*opcode*) : permet d'identifier l'instruction

Opcode ADD (AL, EAX), donnée

0000010w

w = 0 → opérandes 8 bits

w = 1 → opérandes 32 bits

On encode ADD EAX, 2 → opérandes 32 bits, w = 1.

Opcode ADD EAX, donnée

00000101 = 05h

					05										
P1	P2	P3	P4	O2	O1			D1	D2	D3	D4	I1	I2	I3	I4
Préfixes				Code op.		mod reg r/m	sib	Déplacement				Donnée immédiate			

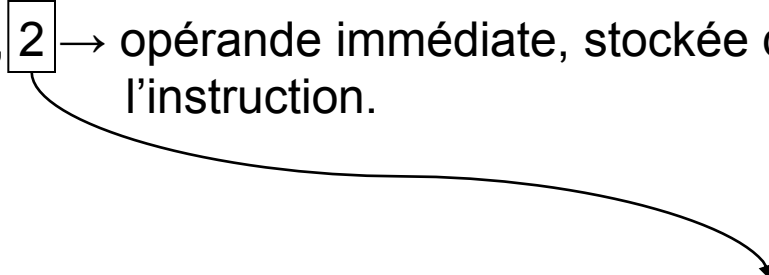
7.1. Encodage des instructions

- Opérande destination : identifie l'adresse ou sera rangé le résultat

On encode ADD EAX, 2 → l'opérande destination est implicite

- Opérande source : identifie ou trouver la première opérande de l'opération

On encode ADD EAX, 2 → opérande immédiate, stockée dans l'instruction.



					05							02	00	00	00
P1	P2	P3	P4	O2	O1			D1	D2	D3	D4	I1	I2	I3	I4
Préfixes				Code op.		mod reg r/m	sib	Déplacement				Donnée immédiate			

7.1. Encodage des instructions

- On obtient pour l'encodage de ADD EAX,2 :
05 02 00 00 00
- Seuls les octets significatifs (utilisés dans l'instruction) sont effectivement stockés en mémoire.

					05							02	00	00	00
P1	P2	P3	P4	O2	O1			D1	D2	D3	D4	I1	I2	I3	I4
Préfixes				Code op.		mod reg r/m	sib	Déplacement				Donnée immédiate			

7.1. Encodage des instructions

- **ADD ECX, [EBX + ESI * 4 + 12]**

Opcode ADD reg/mém, reg/mém

000000dw

d = 0 → destination = reg/mém

source = reg

d = 1 → destination = reg

source = reg/mém

w = 0 → opérandes 8 bits

w = 1 → opérandes 32 bits

Dans ce cas, opérandes 32 bits,

$ECX \leftarrow [EBX + ESI * 2 + 12] + ECX$

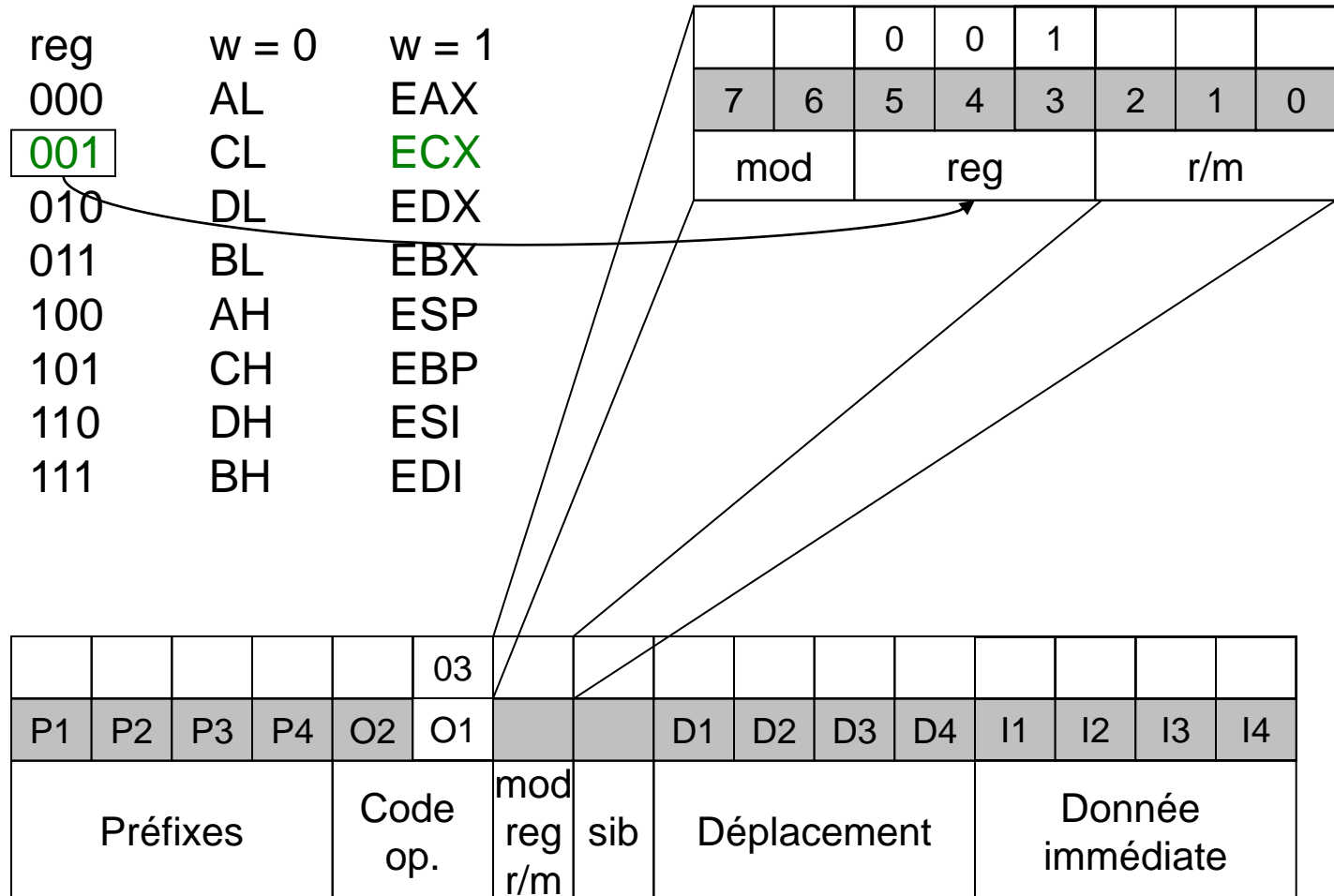
donc d ← 1, w ← 1.

00000011 = 03h

					03										
P1	P2	P3	P4	O2	O1			D1	D2	D3	D4	I1	I2	I3	I4
Préfixes				Code op.		mod reg r/m	sib	Déplacement				Donnée immédiate			

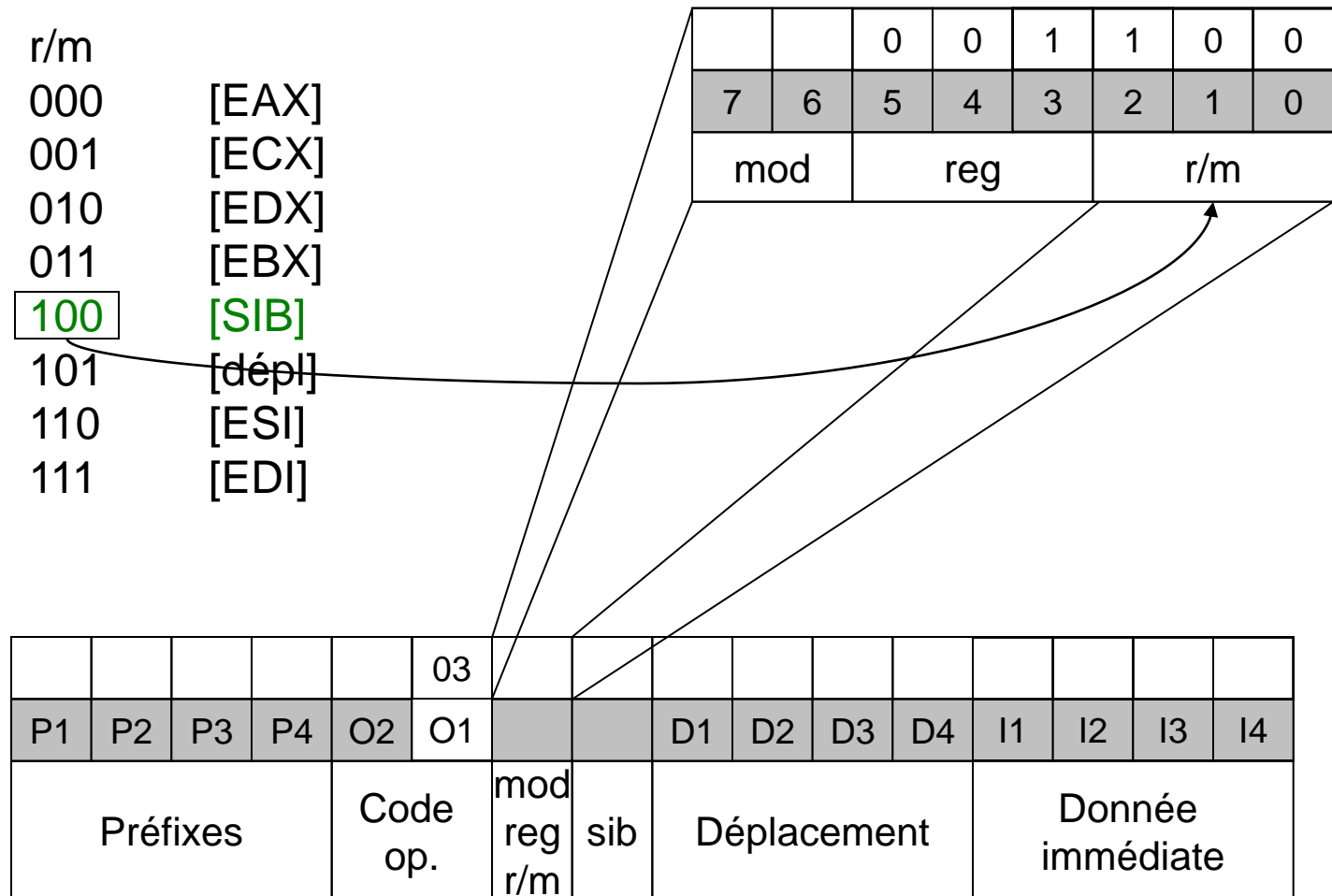
7.1. Encodage des instructions

- Opérande destination : registre ECX



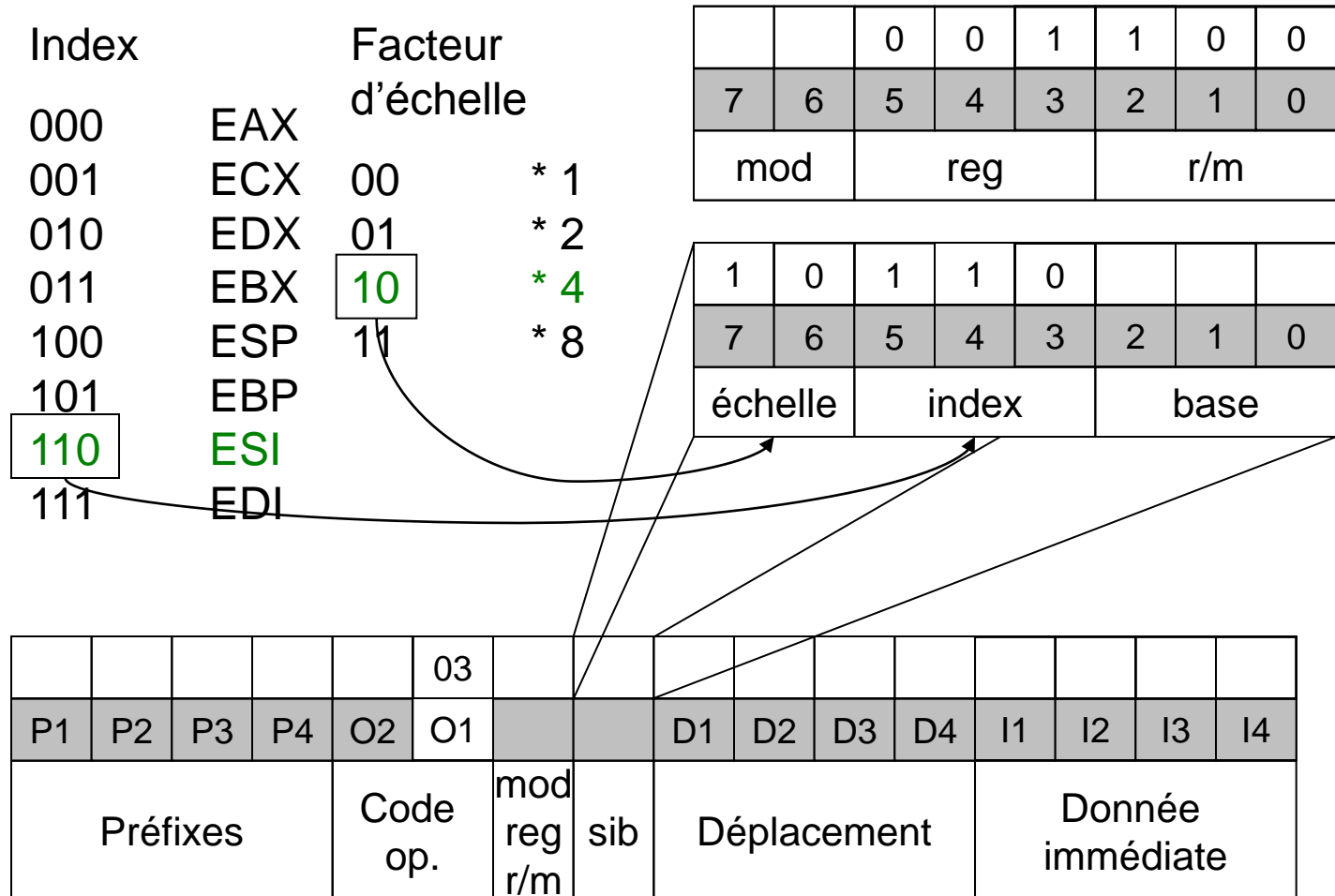
7.1. Encodage des instructions

- Opérande source : $[EBX + ESI * 4 + 12]$



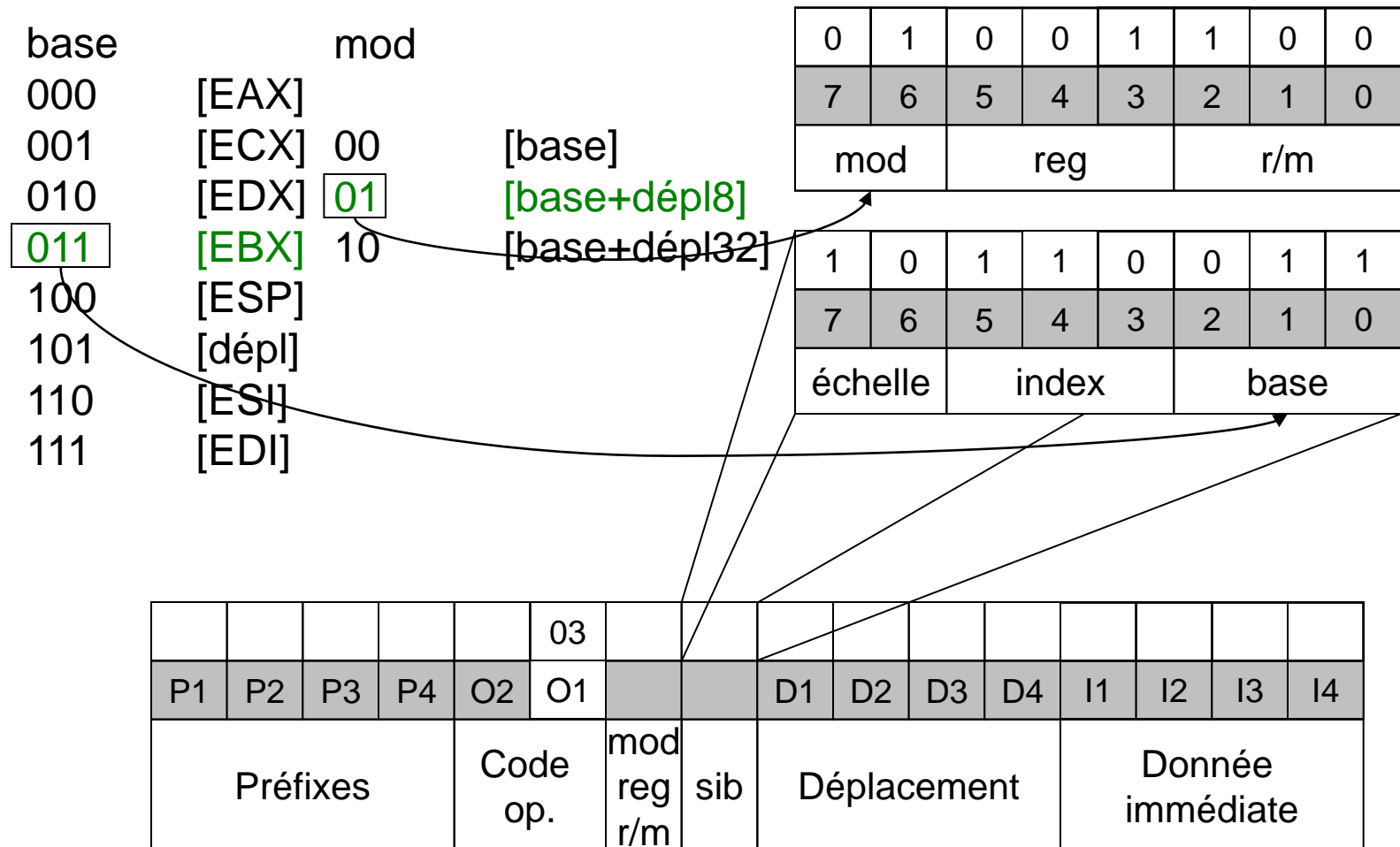
7.1. Encodage des instructions

- Opérande source : $[EBX + ESI * 4 + 12]$



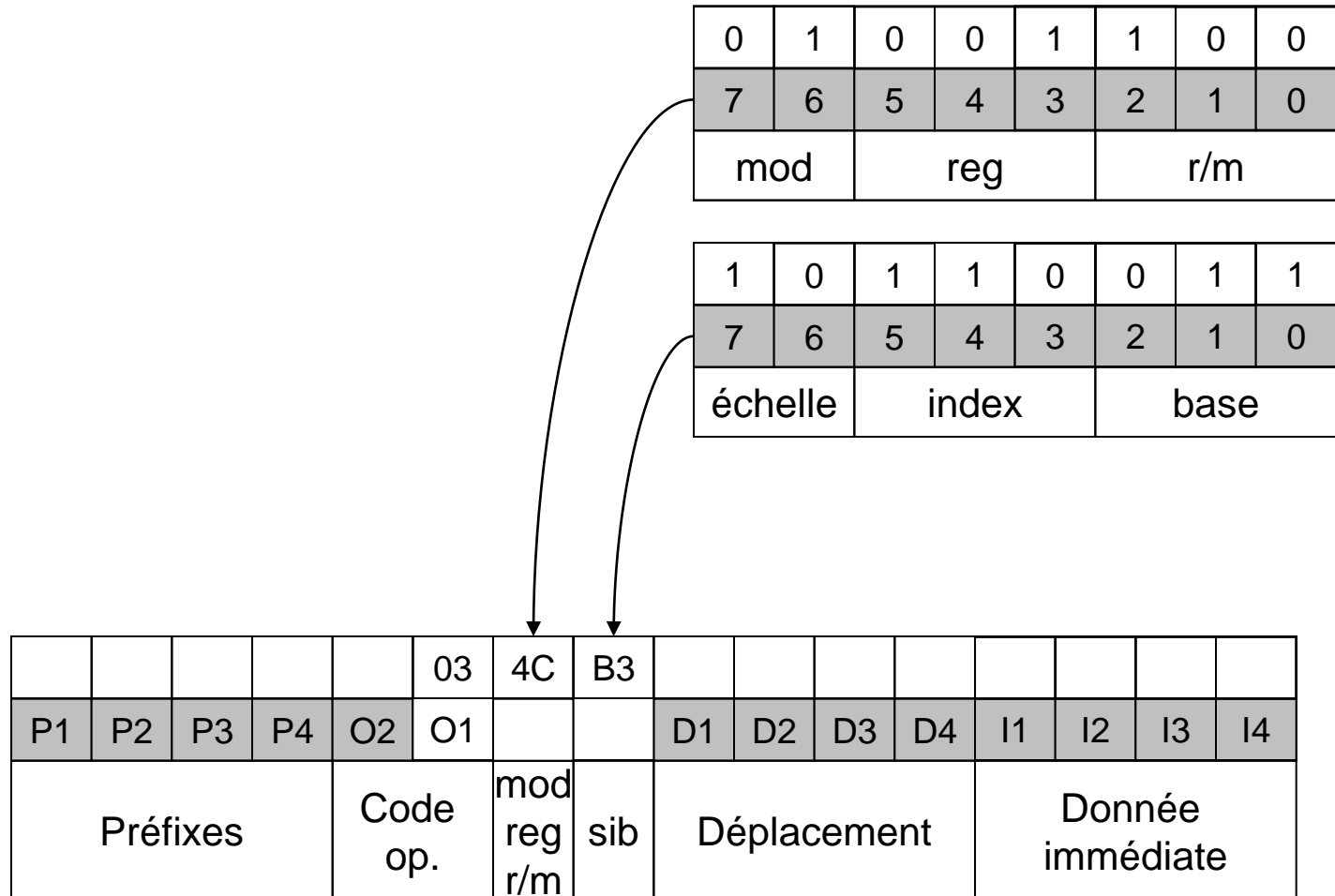
7.1. Encodage des instructions

- Opérande source : $[EBX + ESI * 4 + 12]$



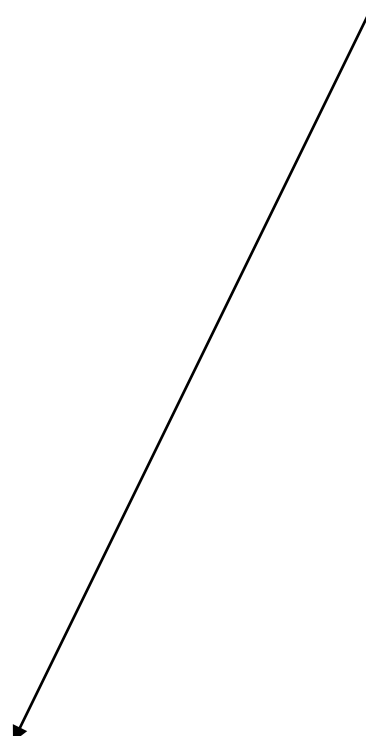
7.1. Encodage des instructions

- Opérande source : $[EBX + ESI * 4 + 12]$



7.1. Encodage des instructions


- Opérande source : $[EBX + ESI * 4 + 12]$
- Encodage final :
03 4C B3 0C



					03	4C	B3	0C							
P1	P2	P3	P4	O2	O1			D1	D2	D3	D4	I1	I2	I3	I4
Préfixes				Code op.		mod reg r/m	sib	Déplacement				Donnée immédiate			

7.1. Encodage des instructions

- Les préfixes permettent d'encoder dans l'instruction des informations supplémentaires :
 - Taille d'opérande
`ADD CX, [EBX + ESI * 4 + 12]`
 - Substitution de segment
`ADD CX, ES:[EBX + ESI * 4 + 12]`
- ...qui est donc encodée 66 26 03 4C B3 0C



		66	26		03	4C	B3	0C							
P1	P2	P3	P4	O2	O1			D1	D2	D3	D4	I1	I2	I3	I4
Préfixes				Code op.		mod reg r/m	sib	Déplacement				Donnée immédiate			

7.2. Jeu d'instruction

- Ensemble des instructions exécutables par le processeur
- Plusieurs classes d'instructions
 - Transferts de données
 - Opérations arithmétiques
 - Opérations logiques
 - Décalages et rotations
 - Tests et comparaisons
 - Transfert de contrôle
 - Gestion de la pile
 - Opérations sur les chaînes

7.2. Transferts de données

- Ne modifient pas les drapeaux en général

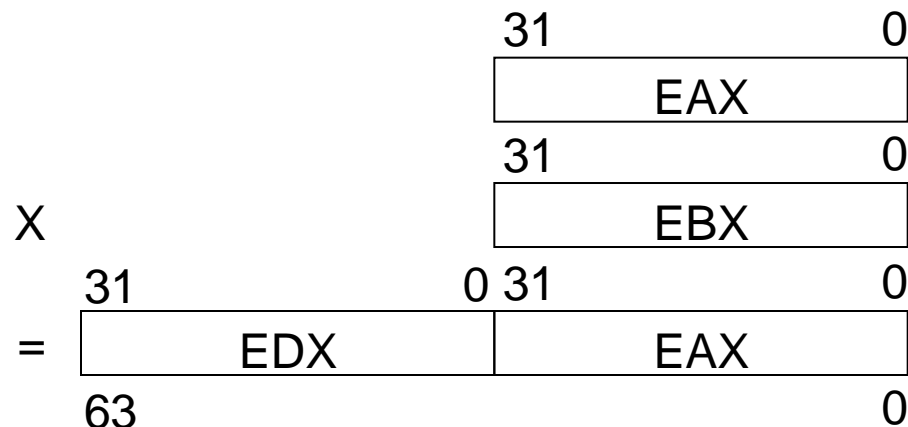
Instruction	Opération
BSWAP reg	Inverse l'ordre des octets
CMPXCHG op1, op2	Si (E)AX = op1 $op2 \leftarrow op1$ sinon (E)AX $\leftarrow op1$
LEA op1, op2	$op1 \leftarrow$ Adresse effective de op2
MOV op1, op2	$op1 \leftarrow op2$
MOVZX op1, op2	$op1 \leftarrow op2$ avec extension par 0
MOVSX op1, op2	$op1 \leftarrow op2$ avec extension de signe
XCHG op1, op2	$op1 \leftrightarrow op2$

7.2. Opérations arithmétiques

Instruction	Opération
ADC op1, op2	$op1 \leftarrow op1 + op2 + CF$
ADD op1, op2	$op1 \leftarrow op1 + op2$
DEC op1	$op1 \leftarrow op1 - 1$
DIV op	$E(AX) \leftarrow E(AX) / op$ (division non signée)
IDIV op	$E(AX) \leftarrow E(AX) / op$ (division signée)
MUL op1	$E(AX) \leftarrow E(AX) \times op$ (multiplication non signée)
IMUL op1	$E(AX) \leftarrow E(AX) \times op$ (multiplication signée)
IMUL op1, op2	$op1 \leftarrow op1 \times op2$ (multiplication signée)
INC op	$op \leftarrow op + 1$
MUL op	$E(AX) \leftarrow E(AX) \times op$ (multiplication non signée)
NEG op	$op \leftarrow -op$ (inversion arithmétique)
SBB op1, op2	$op1 \leftarrow op1 - op2 - CF$
SUB op1, op2	$op1 \leftarrow op1 - op2$

7.2. Opérations arithmétiques

- Multiplication `MUL op`, `IMUL op`
 - Le résultat de la multiplication de deux mots de x bits nécessite au plus $2x$ bits.
 - Le processeur utilise donc un registre de taille double de op pour stocker le résultat :
 - (I)`MUL BL` $\rightarrow AX \leftarrow AL * BL$
 - (I)`MUL BX` $\rightarrow DX:AX \leftarrow AX * BX$
 - (I)`MUL EBX` $\rightarrow EDX:EAX \leftarrow EAX * EBX$



7.2. Opérations arithmétiques

- Multiplication IMUL *op1, op2*
 - Dans ce cas, $op1 \leftarrow op1 \times op2$.
 - La taille du résultat n'est pas étendue : si le résultat ne tient pas dans *op1*, alors $CF \leftarrow 1$ et $OF \leftarrow 1$.
 - *op1* et *op2* doivent être de même taille.
 - *op1* et *op2* doivent être des mots ou double mots
 - destination *op1* est nécessairement un **registre**

7.2. Opérations arithmétiques

- Division *DIV op*, *IDIV op*
 - Effectue la division entière de l'accumulateur par *op*.
 - La taille du dividende dépend de la taille de *op*
 - *DIV BL* $AL \leftarrow AX / BL$
 - *DIV BX* $AX \leftarrow DX:AX / BL$
 - *DIV EBX* $EAX \leftarrow EDX:EAX / EBX$
 - Le reste de la division est calculé en même temps
 - *DIV BL* $AH \leftarrow \text{Reste}$
 - *DIV BX* $DX \leftarrow \text{Reste}$
 - *DIV EBX* $EDX \leftarrow \text{Reste}$

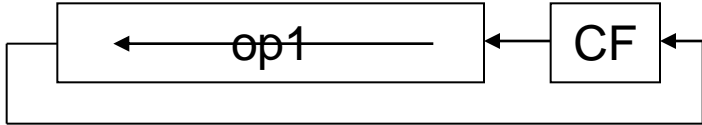
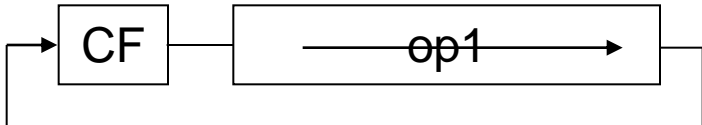
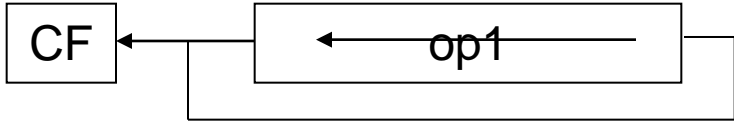
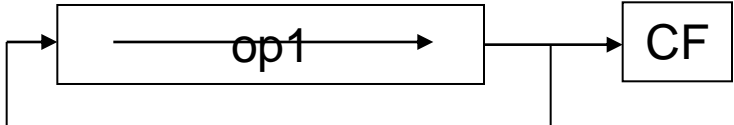
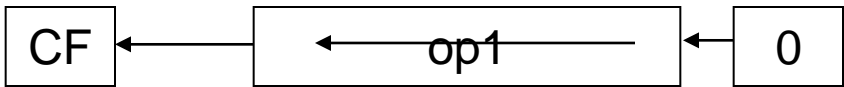
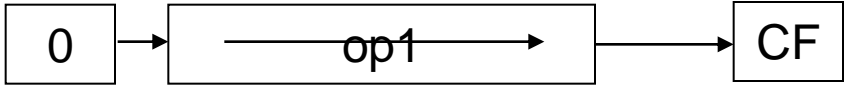
Opérations logiques

- Opérations logiques (agissent bit à bit)
 - Les drapeaux sont modifiés en fonction du résultat
 - L'instruction XOR peut être utilisée pour mettre un registre à 0 : XOR EAX, EAX est plus efficace que MOV EAX, 0.
 - L'instruction OR peut être utilisée pour tester si une donnée est nulle. Si EAX = 0, après OR EAX,EAX on a CF = 1 et EAX inchangé.

Instruction	Opération
AND op1, op2	op1 ← op1 ET op2
OR op1, op2	op1 ← op1 OU op2
NOT op	op ← NON op
XOR op1, op2	op1 ← op1 XOU op2

Décalages et rotations

- op2 peut être soit une opérande immédiate, soit CL.
- op2 représente le nombre de bits à déplacer.

Instruction	Opération
RCL op1, op2	
RCR op1, op2	
ROL op1, op2	
ROR op1, op2	
SHL op1, op2	
SHR op1, op2	

Décalages et rotations

- Les décalages peuvent être utilisés pour diviser et multiplier par 2
 - SHL AX, 1 $AX \leftarrow AX \times 2$
 - SHR AX, 1 $AX \leftarrow AX / 2$
- Problème : si AX est signé, que devient le bit de signe ?
 - SAR AX, 1 $AX \leftarrow AX / 2$, Bit 15 \leftarrow Signe de AX
 - ➔ Utiliser un décalage *arithmétique*

Tests et comparaisons

- Tests et comparaisons
 - Ces instructions modifient les drapeaux sans modifier leurs opérandes.
 - On peut opérer des test sur des bits (BT, BTS...) ou sur des opérandes classiques (CMP, TEST)

Instruction	Opération
BT op1, op2	CF \leftarrow Bit op2 de op1
BTS op1, op2	CF \leftarrow Bit op2 de op1 et mise à un de ce bit
BTR op1, op2	CF \leftarrow Bit op2 de op1 et mise à zéro de ce bit
BTC op1, op2	CF \leftarrow Bit op2 de op1 et inversion de ce bit
CMP op1, op2	Calcule op1 – op2 et positionne les drapeaux
TEST op1, op2	Calcule op1 ET op2 et positionne les drapeaux

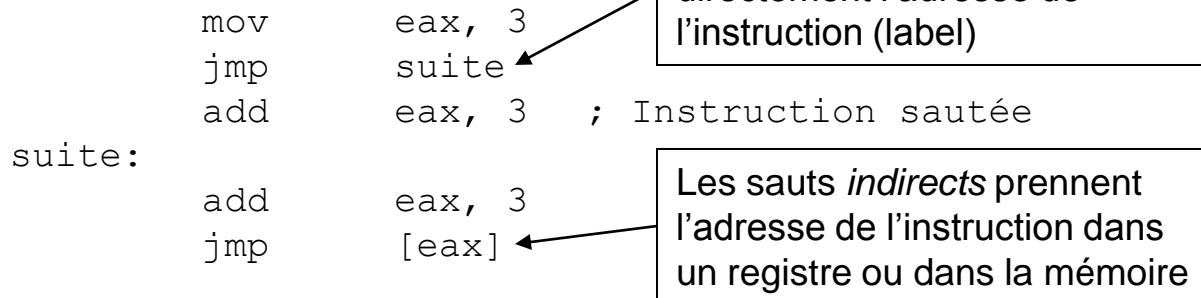
Transferts de contrôle

- Le pointeur d'instruction (EIP) est affecté.
- L'instruction suivante se trouvera donc ailleurs.
- En général, op est une adresse immédiate (déplacement)
- op peut être une référence mémoire ou un registre. Dans ce cas, le branchement est fait à l'adresse contenue dans la mémoire ou le registre.

Instruction	Opération
CALL op	Saut sous programme
INT imméd8	Saut sous programme d'interruption n° imméd8
IRET	Retour de sous programme d'interruption
Jcond dépl	Saut conditionnel – Teste les drapeaux
JCXZ	Saut si CX = 0
JCXNZ	Saut si CX ≠ 0
JMP op	Saut inconditionnel
RET	Retour de sous programme

Sauts inconditionnels

- Normalement, les instructions sont exécutées séquentiellement.
- Les sauts inconditionnels modifient le flot d'exécution, en changeant l'adresse de la prochaine instruction à exécuter.



Sauts conditionnels

- Permettent de modifier le flot d'exécution en fonction d'une condition, donc de contrôler l'exécution
- Forme :

Jcond adresse

Avec ***cond*** la condition à vérifier

adresse l'adresse à charger dans EIP si *cond* est vérifiée.

- La condition est une fonction combinatoire des drapeaux d'état du processeur f(FLAGS).
- L'algorithme réalisé est donc le suivant :
 SI f(FLAGS) = VRAI ALORS
 EIP ← *adresse*
 FINSI
- Seuls les sauts directs sont autorisés !

Sauts conditionnels

- Le saut est effectué en fonction des drapeaux
- Il faut donc utiliser une instruction qui modifie les drapeaux juste avant. Ex : `CMP EAX, EBX`

EAX ? EBX	Comparaison signée	Comparaison non signée
>	JG, JNLE	JA, JNBE
≥	JGE, JNL	JAE, JNB
=	JE, JZ	JE, JZ
≠	JNE, JNZ	JNE, JNZ
≤	JLE, JNG	JBE, JNA
<	JL, JNGE	JB, JNAE

Sauts conditionnels

- Ils permettent de traduire les structures de contrôle utilisés en programmation structurée
 - SI condition ALORS ... SINON ... FIN SI
 - FAIRE ... TANT QUE condition
 - TANT QUE condition FAIRE ... FIN TANT QUE
 - POUR intervalle FAIRE ... FIN POUR

Sauts conditionnels

- Traduction de la structure alternative SI..ALORS..SINON

```
int diff_absolue(int x, int y)
{
    if (x < y)
        return(y - x);
    else
        return(x - y);
}
```

Le code « goto » utilise l'instruction goto du C pour simuler le flot d'exécution d'un programme assembleur. A ne pas faire dans un vrai programme C !

```
int diff_absolue_goto(int x, int y)
{
    int resultat;
    if (x < y) goto inferieur;
    resultat = x - y;
    goto fini;
inferieur:
    resultat = y - x;
fini:
    return(resultat);
}
```

Sauts conditionnels

- Code assembleur de diff_absolue_goto

```
int diff_absolue_goto(int x, int y)
{
    int resultat;
    if (x < y) goto inferieur;
    resultat = x - y;
    goto fini;
inferieur:
    resultat = y - x;
fini:
    return(resultat);
}
```

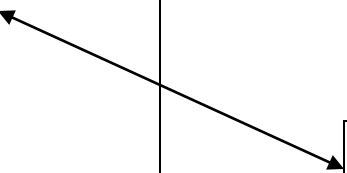
```
diff_absolue_goto:
    push    ebp
    mov     ebp, esp
    mov     edx, [ebp + 8]
    mov     eax, [ebp + 12]

    cmp     eax, edx
    jl      inferieur
    sub     eax, edx
    jmp     fini

inferieur:
    sub     edx, eax

fini:      mov     eax, edx

    pop     ebp
    ret
```



Sauts conditionnels

- Traduction de la structure itérative FAIRE...TANQUE

Forme de la structure (en C) :

```
do
    bloc_instructions
while (expression_test);
```

Forme de la structure (en code goto) :

```
boucle:
    bloc_instructions
    t = expression_test;
    if (t) goto boucle;
```

Sauts conditionnels

- Exemple FAIRE..TANTQUE

```
int somme(int n)
{
    int result = 0;
    int i = 0;
    do
    {
        result = result + i;
        i = i + 1;
    }
    while (i <= n);
    return(result);
}
```

```
somme:
        push    ebp
        mov     ebp, esp

        mov     edx, [ebp+8]
        xor     eax, eax
        xor     ecx, ecx

boucle:
        add     eax, ecx
        inc     ecx
        cmp     ecx, edx
        jle     boucle

        pop     ebp
        ret
```

Sauts conditionnels

- Traduction de la structure itérative TANTQUE ..FAIRE

Forme de la structure (en C) :

```
while (expression_test)
    bloc_instructions
```

Forme de la structure (en code goto),
traduction directe :

```
boucle:
    t = expression_test;
    if (!t)
        goto fini;
    bloc_instructions
    goto boucle;
fini:
```

Cette traduction directe à une instruction de saut dans la boucle de plus que le do..while ! Mais on peut transformer un while en do..while...

```
if (! expression_test)
    goto fini;
do
    bloc_instructions
while(expression_test);
fini:
```

Forme de la structure (en code goto) :

```
t = expression_test;
if (! t)
    goto fini;
boucle:
    bloc_instructions
    t = expression_test;
    if (t) goto boucle;
fini:
```

Sauts conditionnels

- Exemple TANT QUE..FAIRE

```
int somme(int n)
{
    int result = 0;
    int i = 0;
    while(i <= n)
    {
        result = result + i;
        i = i + 1;
    }
    return(result);
}
```

```
somme:
    push    ebp
    mov     ebp, esp

    mov     edx, [ebp+8]
    xor     eax, eax
    xor     ecx, ecx
    cmp     ecx, edx
    ja      fini

boucle:
    add     eax, ecx
    inc     ecx
    cmp     ecx, edx
    jle     boucle

fini:
    pop     ebp
    ret
```


Sauts conditionnels

- Traduction de la structure itérative POUR..FAIRE

Forme de la structure (en C) :

```
for(expression_init; expression_test; expression_maj)
    bloc_instructions
```

Ce qui est équivalent à la structure while :

```
expression_init;
while(expression_test)
{
    bloc_instructions;
    expression_maj;
}
```

...Et qui revient au do..while :

```
expression_init;
if (!expression_test) goto fini;
do
{
    bloc_instructions;
    expression_maj;
}
while(expression_test)
fini:
```

Sauts conditionnels

- Exemple POUR..FAIRE

```
int somme(int n)
{
    int result = 0;
    int i;
    for(i = 0; i <= n; i++)
    {
        result = result + i;
    }
    return(result);
}
```

```
somme:
    push    ebp
    mov     ebp, esp

    mov     edx, [ebp+8]
    xor     eax, eax
    xor     ecx, ecx
    cmp     ecx, edx
    ja      fini

boucle:
    add     eax, ecx
    inc     ecx
    cmp     ecx, edx
    jle     boucle

fini:
    pop     ebp
    ret
```

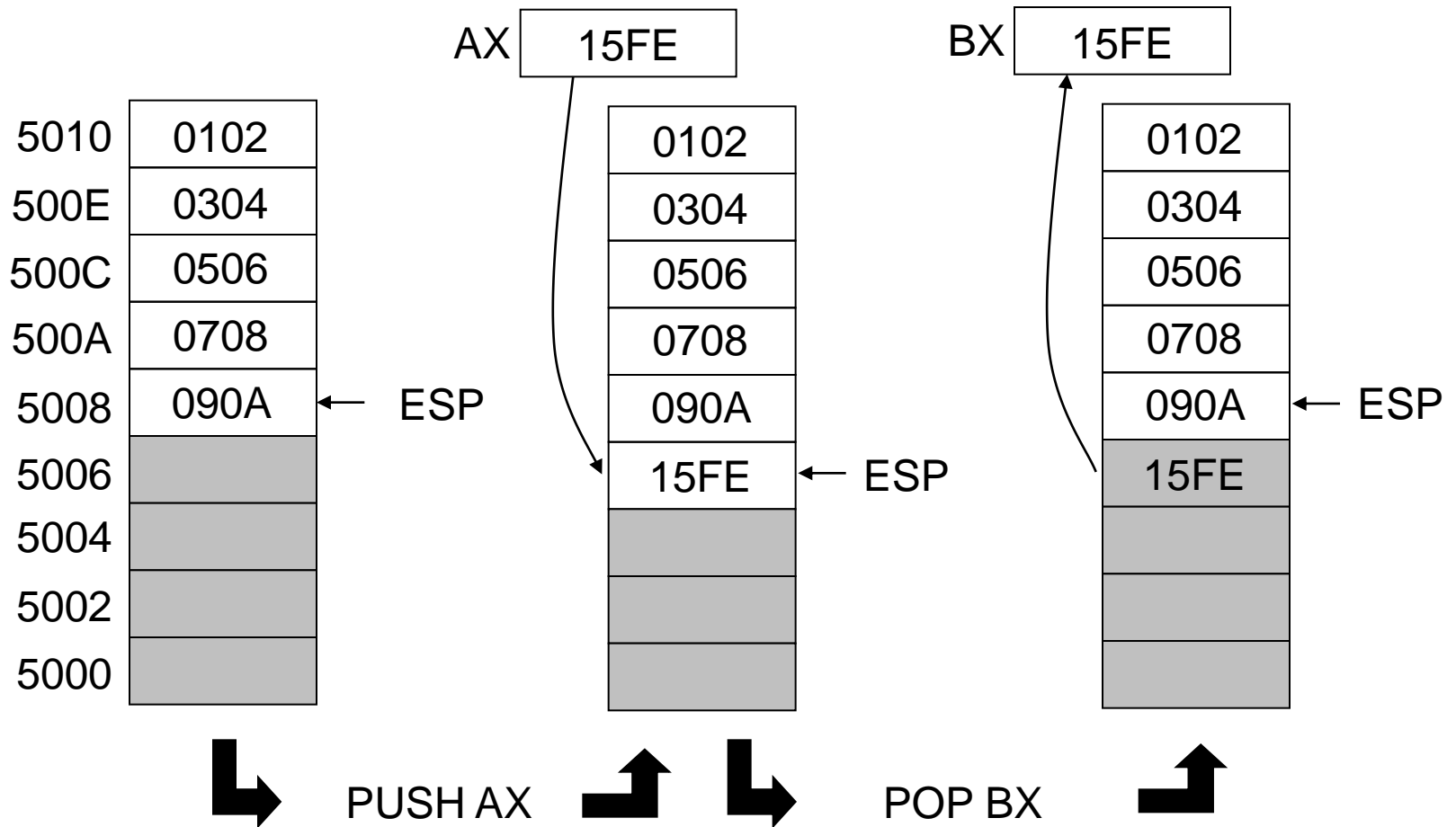
Le code est le même que celui du while !

Opérations sur la pile

- Ces instructions permettent de sauvegarder et de restaurer des valeurs sur la pile
- L'incrémentation/la décrémentation du pointeur de pile dépend de la taille de l'opérande *op* (16 ou 32 bits).

Instruction	Opération
PUSH op	$ESP \leftarrow ESP - \text{taille}(op) ; [ESP] \leftarrow op$
POP op	$op \leftarrow [ESP] ; ESP \leftarrow ESP + \text{taille}(op)$
PUSHF	$ESP \leftarrow ESP - 2 ; [ESP] \leftarrow \text{FLAGS}$
POPF	$\text{FLAGS} \leftarrow [ESP] ; ESP \leftarrow ESP + 2$
PUSHFD	$ESP \leftarrow ESP - 4 ; [ESP] \leftarrow \text{EFLAGS}$
POPFD	$\text{EFLAGS} \leftarrow [ESP] ; ESP \leftarrow ESP + 4$

Opérations sur la pile

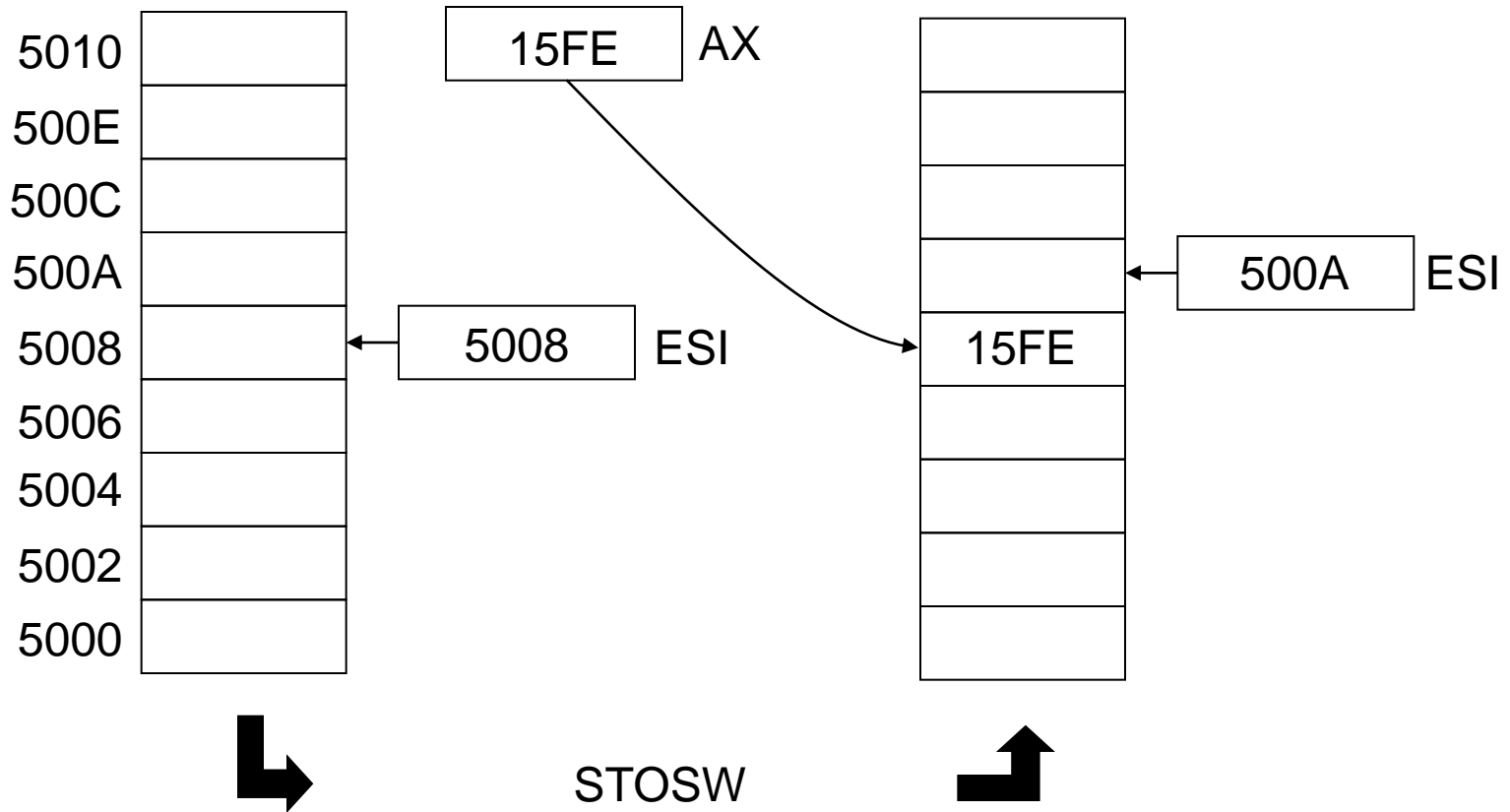


Opérations sur les chaînes

- Agissent sur des chaînes de mots mémoire
- Chaîne : suite contiguë de mots de même taille.
- Un élément de la chaîne est pointé par ESI ou EDI ;
- L'instruction agit sur cet élément, puis incrémente/décrémente automatiquement ESI/EDI

Opérations sur les chaînes

- Exemple : STOSW



Opérations sur les chaînes

- x peut prendre les valeurs :
 - B : agir sur des octets (Byte)
 - W : agir sur des mots (Word)
 - D : agir sur des double mots (Double word)

Instruction	Opération
CMPSx	Compare DS:[ESI] et ES:[EDI] et ajuste les drapeaux en conséquence (cf. CMP). ESI ← ESI +/- Taille; EDI ← EDI +/- Taille
LODSx	Accumulateur ← DS:[ESI]; ESI ← ESI +/- Taille
MOVSx	ES:[EDI] ← DS:[ESI] ESI ← ESI +/- Taille; EDI ← EDI +/- Taille
SCASx	Compare l'accumulateur avec DS:[ESI] et ajuste les drapeaux en conséquence. ESI ← ESI +/- Taille; EDI ← EDI +/- Taille
STOSx	DS:[ESI] ← Accumulateur; ESI ← ESI +/- Taille

Opérations sur les chaînes

- La valeur du drapeau DF (*Direction Flag*) détermine le sens du parcours de la chaîne
 - Si DF = 0, les pointeurs sont incrémentés
 - Si DF = 1, les pointeurs sont décrémentés.
- Deux instructions permettent de manipuler le drapeau DF
 - **CLD** : $DF \leftarrow 0$
 - **STD** : $DF \leftarrow 1$

Opérations sur les chaînes

- Préfixes de répétition
 - Ils permettent de répéter l'instruction chaîne plusieurs fois.
 - Le nombre de répétitions est contenu dans ECX.
 - Elle peuvent être interrompues par l'apparition de certaines conditions

Instruction	Opération
REP	Répéter tant que $ECX \neq 0$
REPZ/REPE	Répéter tant que $ECX \neq 0$ et $ZF = 1$
REPNZ/REPNE	Répéter tant que $ECX \neq 0$ et $ZF = 0$