Architecture et Langage Assembleur

Eric Ramat
ramat@lisic.univ-littoral.fr

Université du Littoral - Côte d'Opale

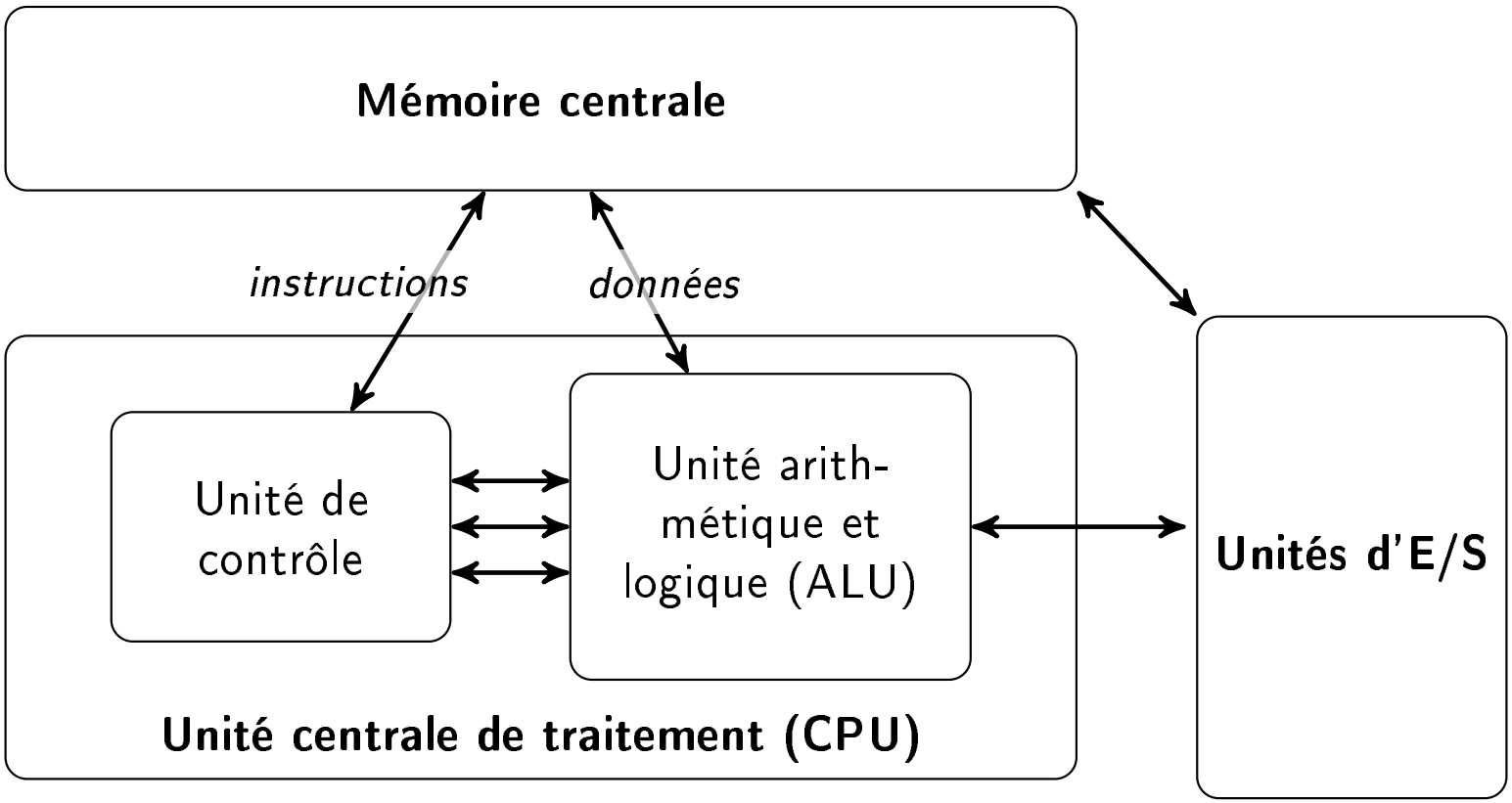
Déroulement du module

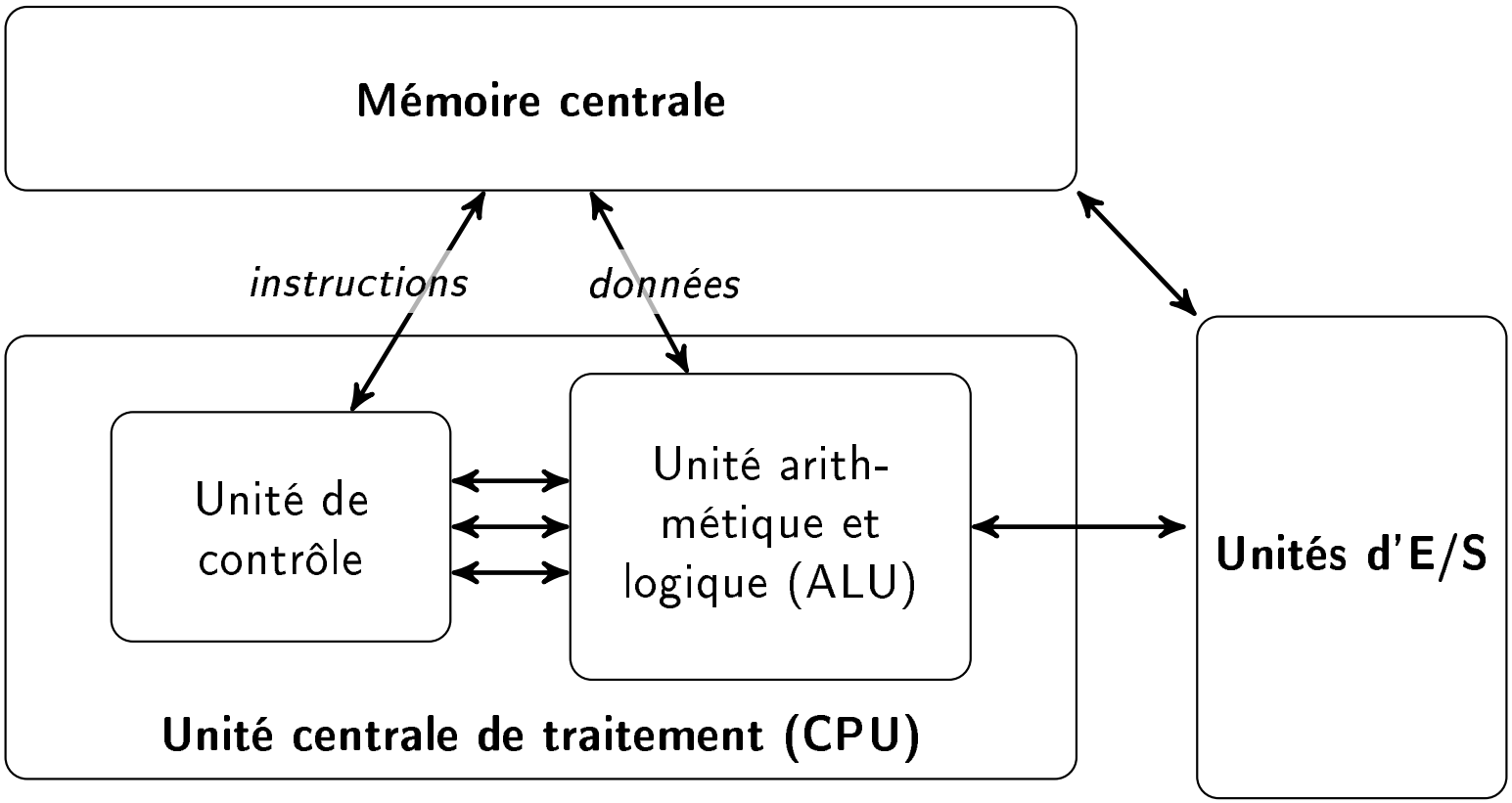
- 9h de cours
- 6 séances de 3h de travaux pratiques
- **chaque séance** de tp :
  - un compte rendu en fin de séance
  - une note

- note du module = sup(examen, $\frac{1}{2}$ (examen + moyenne des notes TP)

Sommaire

Plan


Mémoire centrale
instructions
données
Unité de contrôle
Unité arith-métique et logique (ALU)
Unité centrale de traitement (CPU)
Unités d'E/S



Conséquence de l'architecture de von Neumann: la mémoire contient données ET programmes mélangés !

- Composants électroniques: présence ou absence de courant
- États symbolisés par 0 et 1

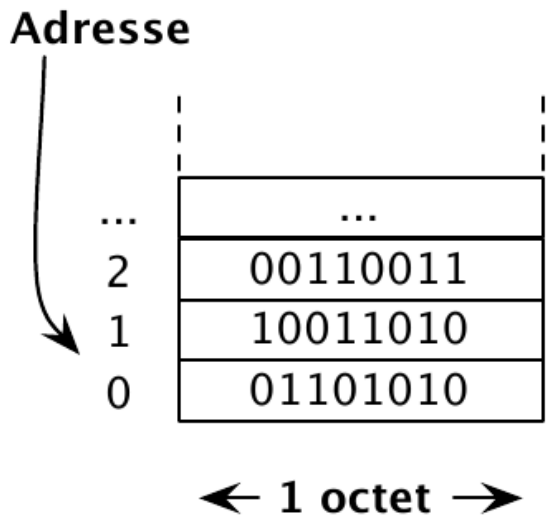
Nom	**Valeur**
Bit	0 ou 1
Octet	de 00000000 à 11111111

- Conséquence: données et programmes mélangés en mémoire et codés en binaire !!

$\rightarrow$ **Comment ça marche ??**

La Mémoire :

- tableau de cases
- chaque case contient 8 bits (1 octet)
- chaque case possède une adresse



On veut stocker

- des entiers positifs (12, 7850, ...)
- des entiers négatifs (-3, ...)
- des caractères ('A', 'r', ';', ...)
- des chaînes de caractères ("Hello", ...)
- des réels (7.873, ...)
- des instructions
- ...

Mais la mémoire
ne contient que des bits (0 ou 1)!

On veut stocker

- des entiers positifs (12, 7850, ...)
- des entiers négatifs (-3, ...)
- des caractères ('A', 'r', ';', ...)
- des chaînes de caractères ("Hello", ...)
- des réels (7.873, ...)
- des instructions
- ...

Tout coder sous forme d'entiers positifs en binaire

Mais la mémoire
ne contient que des bits (0 ou 1)!

Représentation usuelle :

$$14 = 1 \times 10^1 + 4 \times 10^0$$

Représentation positionnelle:

- choix d'une base b (10, 2, ...)
- choix de b symboles

Exemples:

- base 2 (0, 1) : $1110_2 = 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 14_{10}$
- base 3 (0, 1, 2): $112_3 = 1 \times 3^2 + 1 \times 3^1 + 2 \times 3^0 = 14_{10}$

Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée

Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée

$$\begin{array}{r|l} 23 & 10 \\ \hline & \\ & \end{array}$$

Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée

$$\begin{array}{r|l} 23 & 10 \\ \hline & \\ & 2 \end{array}$$

Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée

$$\begin{array}{r|l} 23 & 10 \\ \hline 3 & 2 \end{array}$$

Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée

$$\begin{array}{r|l} 23 & 10 \\ \hline 3 & \end{array} \qquad \begin{array}{r|l} 2 & 10 \\ \hline & \end{array}$$

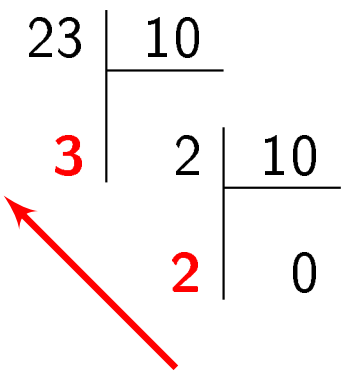
Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée

$$\begin{array}{r|l} 23 & 10 \\ \hline 3 & 2 \end{array} \qquad \begin{array}{r|l} 2 & 10 \\ \hline 0 & \end{array}$$

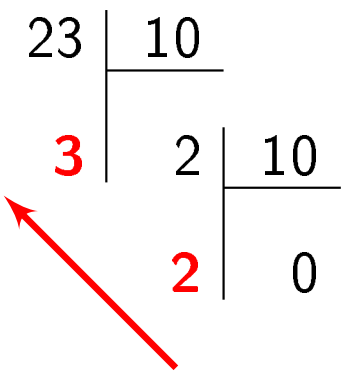
Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée

23	10	
3	2	10
	2	0

Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée



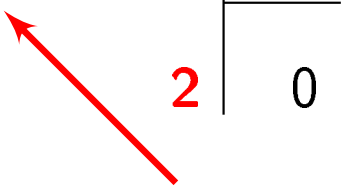
Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée



Résultat: $23 = 23_{10}$

Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée

23	10
3	2 | 10
2	0

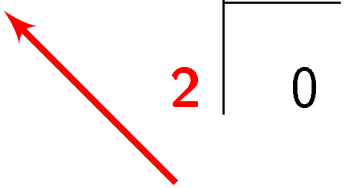


23	2

Résultat: $23 = 23_{10}$

Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée

$$\begin{array}{r|l} 23 & 10 \\ \hline \color{red}{\mathbf{3}} & 2 \end{array} \qquad \begin{array}{r|l} 2 & 10 \\ \hline \color{red}{\mathbf{2}} & 0 \end{array}$$



$$\begin{array}{r|l} 23 & 2 \\ \hline & 11 \end{array}$$

Résultat: $23 = 23_{10}$

Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée

23	10
3	2
2	0

23	2
1	11

Résultat: $23 = 23_{10}$

Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée

23	10
3	2
2	0

23	2
1	11
	2
	1
	0

Résultat: $23 = 23_{10}$

Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée

23	10
3	2
2	0

23	2
1	11
	5

Résultat: $23 = 23_{10}$

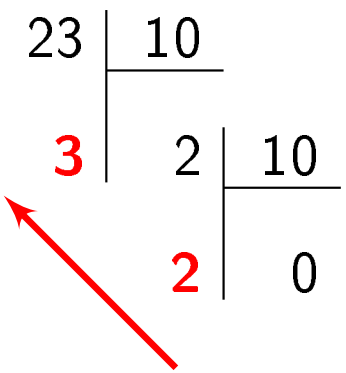
Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée

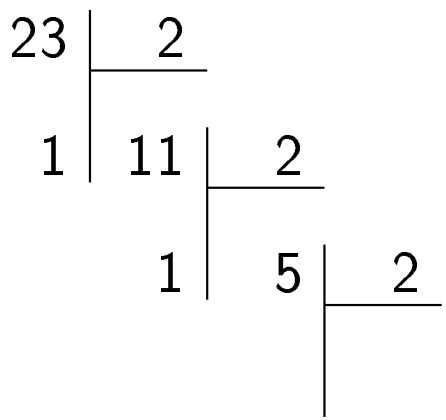
23	10
3	2
2	0

23	2
1	11
1	5

Résultat: $23 = 23_{10}$

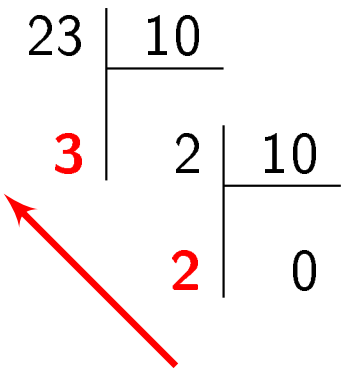
Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée

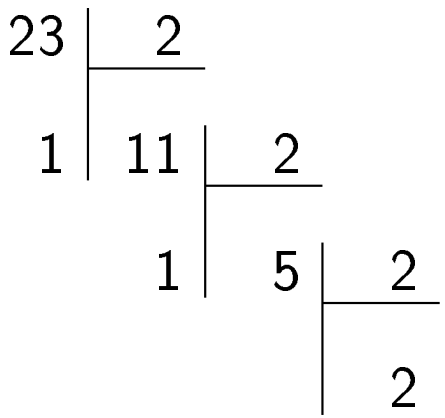




Résultat: $23 = 23_{10}$

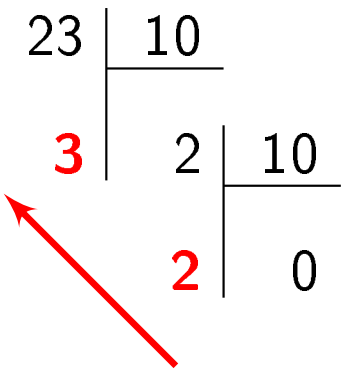
Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée

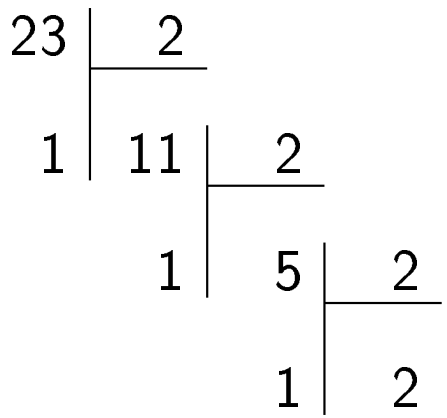




Résultat: $23 = 23_{10}$

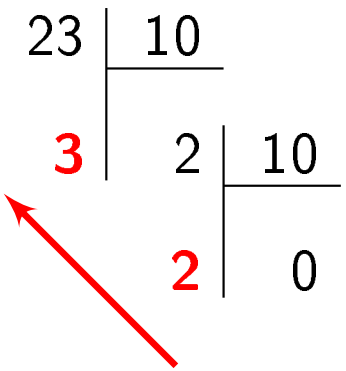
Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée

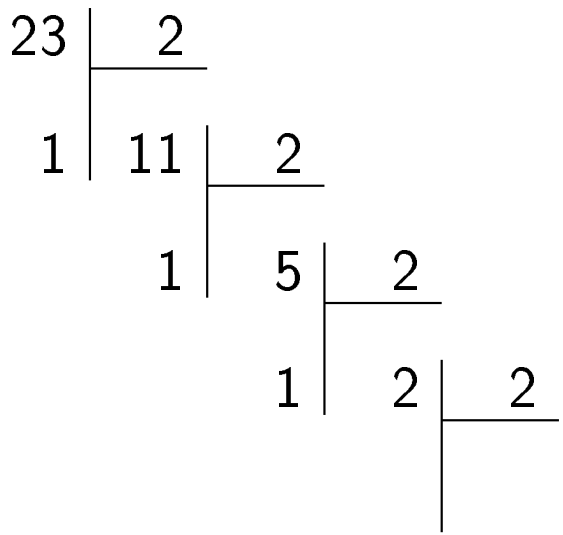




Résultat: $23 = 23_{10}$

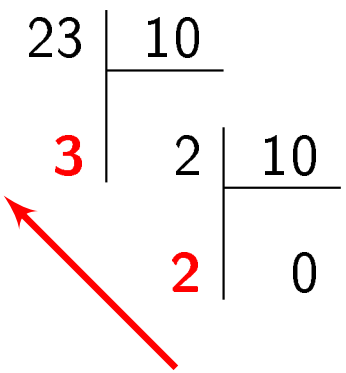
Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée

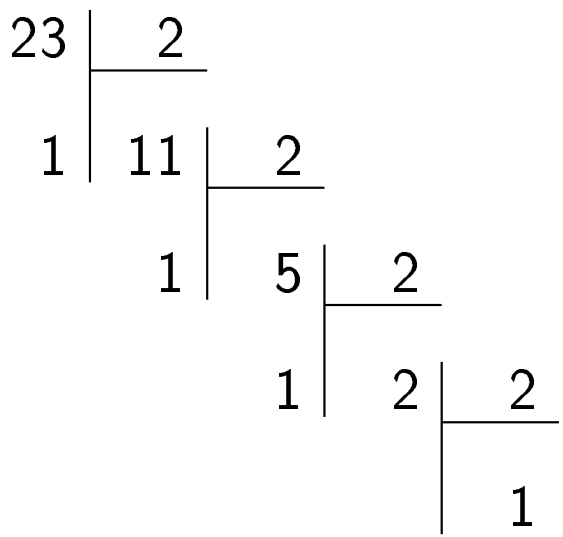




Résultat: $23 = 23_{10}$

Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée

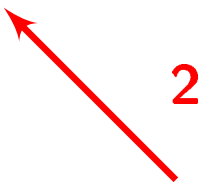




Résultat: $23 = 23_{10}$

Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée

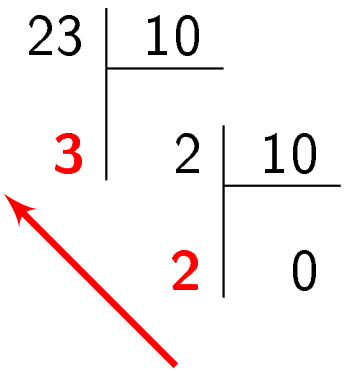
$$\begin{array}{r|l} 23 & 10 \\ \hline 3 & 2 \end{array} \qquad \begin{array}{r|l} 2 & 10 \\ \hline 2 & 0 \end{array}$$

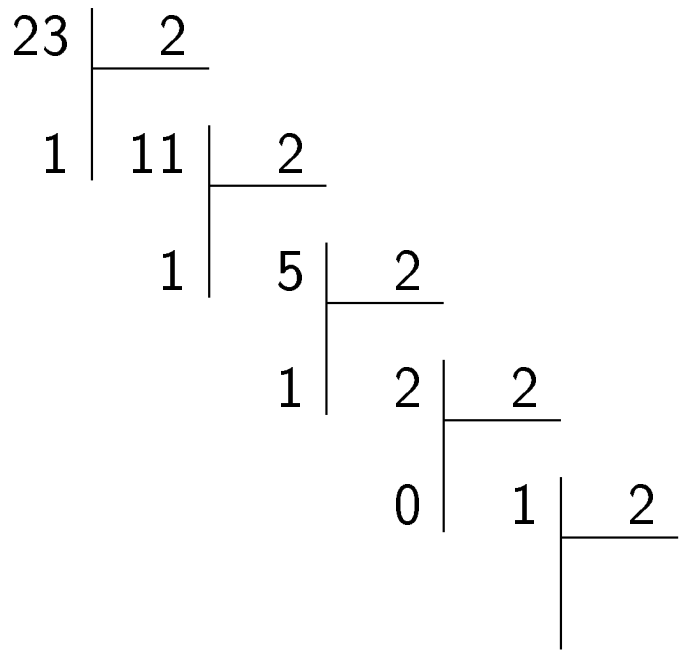


$$\begin{array}{r|l} 23 & 2 \\ \hline 1 & 11 \end{array} \qquad \begin{array}{r|l} 2 & 2 \\ \hline 1 & 5 \end{array} \qquad \begin{array}{r|l} 2 & 2 \\ \hline 1 & 2 \end{array} \qquad \begin{array}{r|l} 2 & 1 \\ \hline 0 & \end{array}$$

Résultat: $23 = 23_{10}$

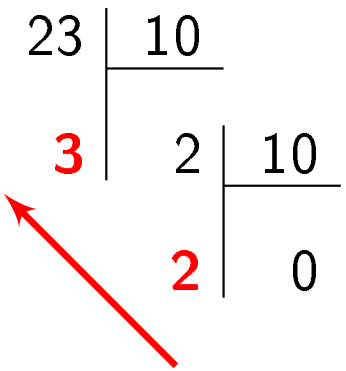
Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée

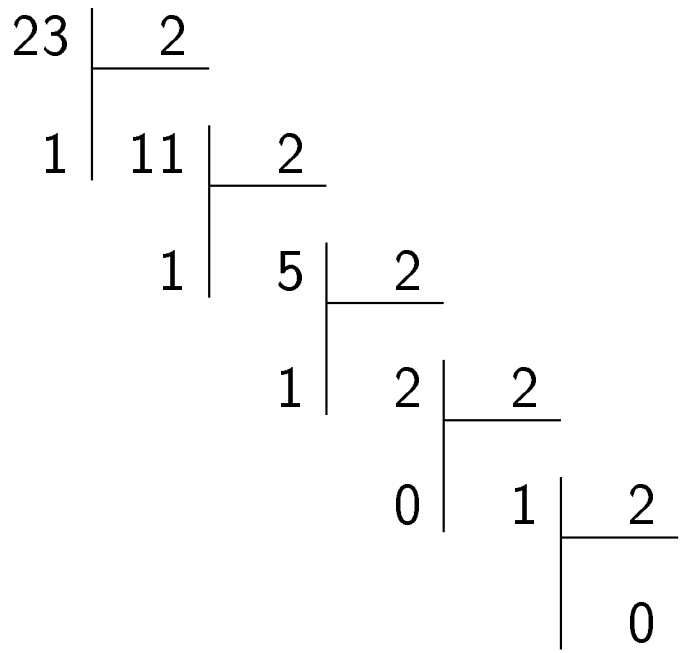




Résultat: $23 = 23_{10}$

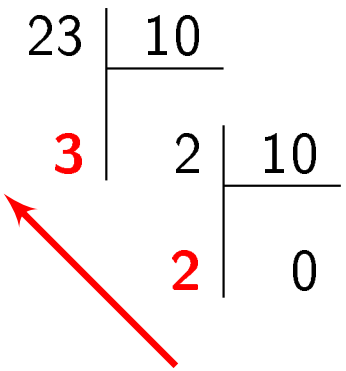
Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée

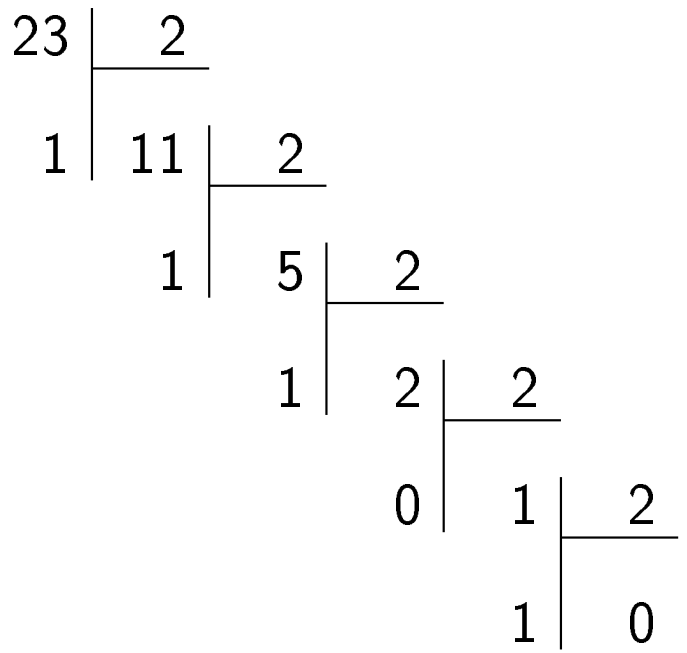




Résultat: $23 = 23_{10}$

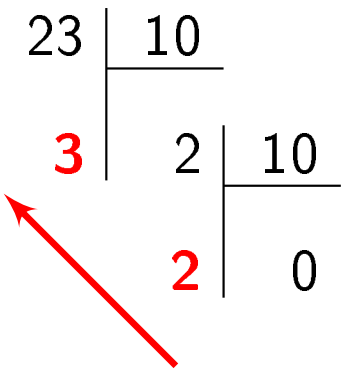
Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée

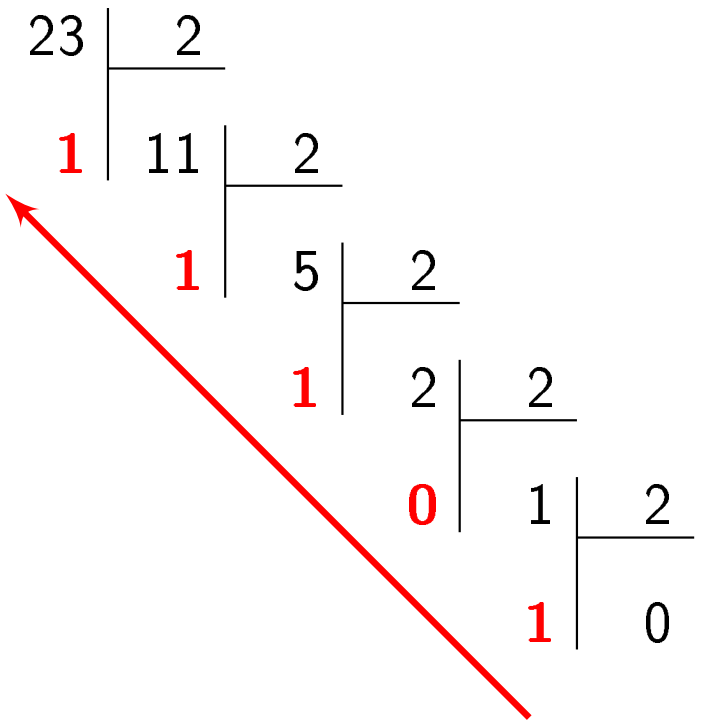




Résultat: $23 = 23_{10}$

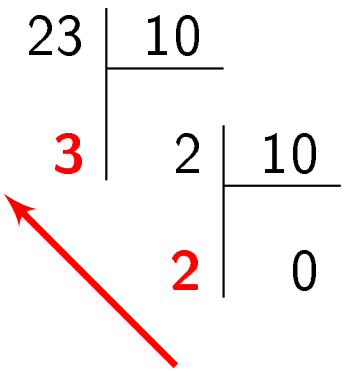
Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée

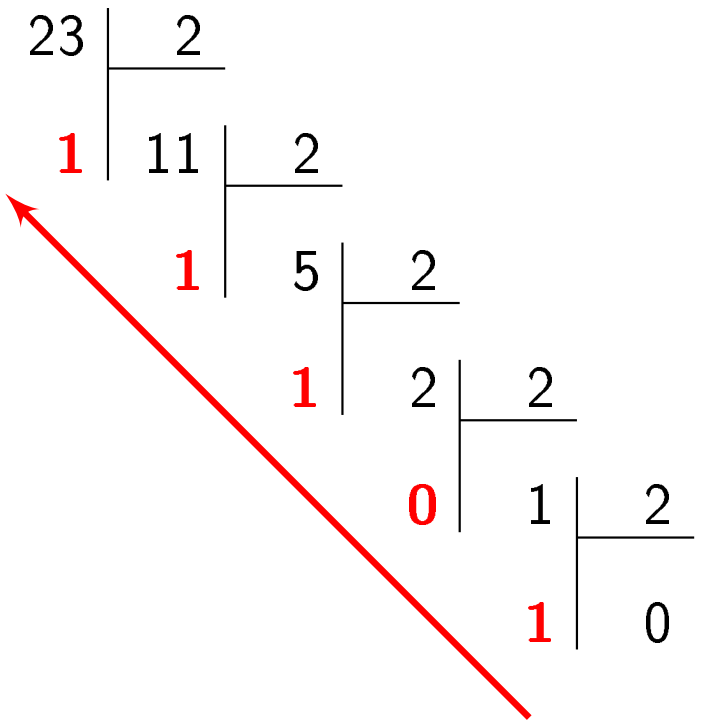




Résultat: $23 = 23_{10}$

Comment passer 23_{10} en base 10, 2, 3, etc ?
$\rightarrow$ on utilise la division itérée





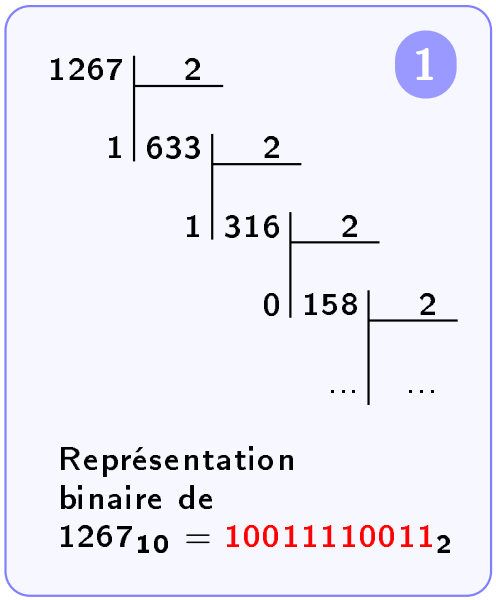
Résultat: $23 = 23_{10} = 10111_2$

Méthode pour stocker un **entier positif (non-signé)** en mémoire:

1. Représentation du nombre en binaire
2. Découpage de cette représentation en octets
3. Stockage de chaque octet consécutivement

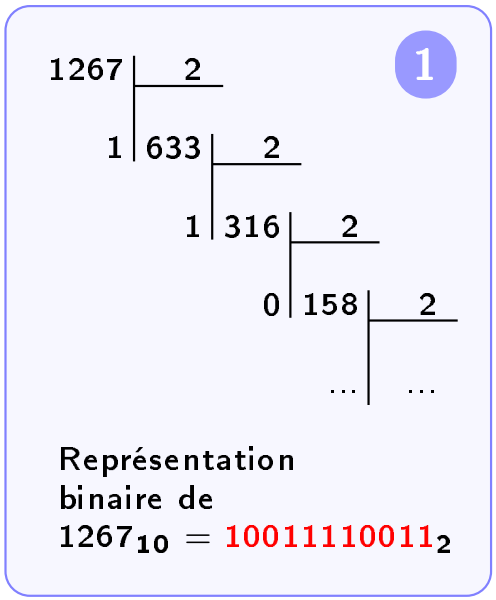
Méthode pour stocker un **entier positif (non-signé)** en mémoire:

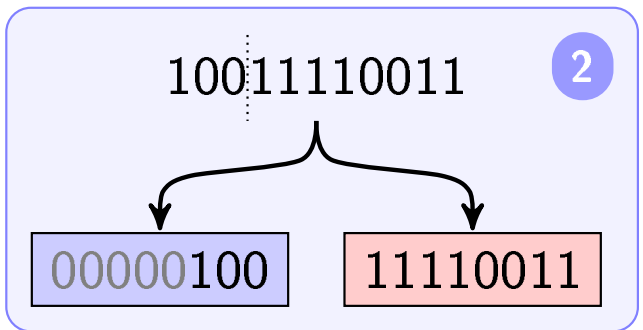
1. Représentation du nombre en binaire
2. Découpage de cette représentation en octets
3. Stockage de chaque octet consécutivement



Méthode pour stocker un **entier positif (non-signé)** en mémoire:

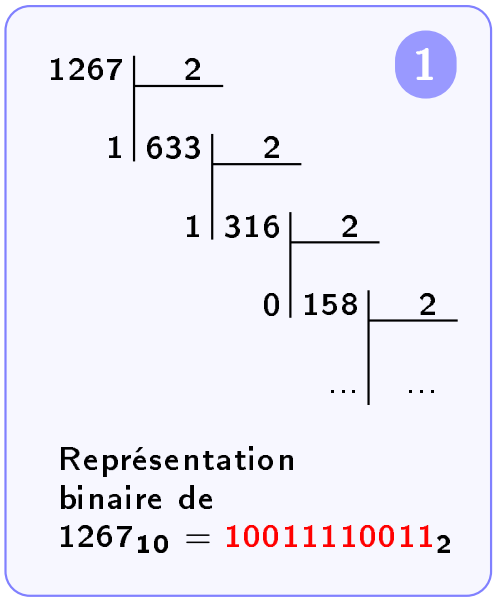
1. Représentation du nombre en binaire
2. Découpage de cette représentation en octets
3. Stockage de chaque octet consécutivement

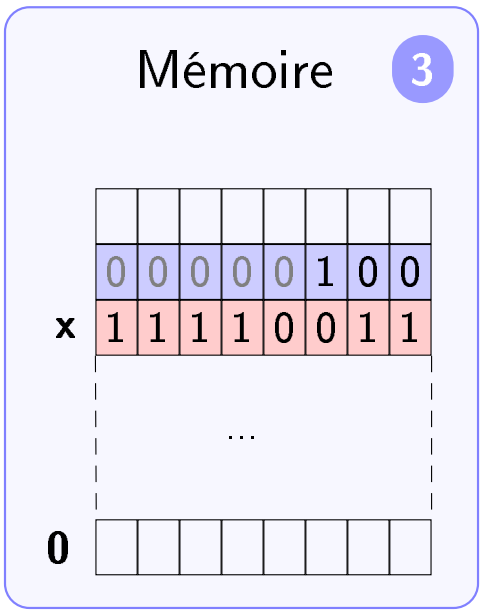


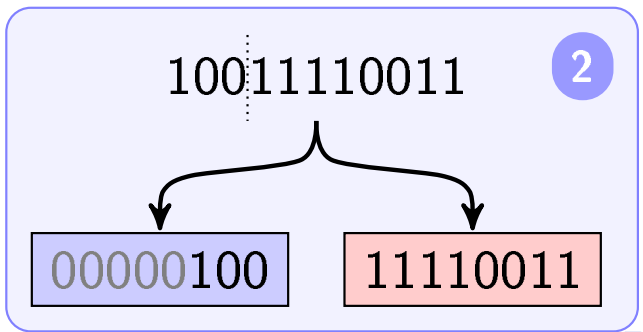


Méthode pour stocker un **entier positif (non-signé)** en mémoire:

1. Représentation du nombre en binaire
2. Découpage de cette représentation en octets
3. Stockage de chaque octet consécutivement







Entiers non-signés: entiers positifs
Entiers signés: entiers positifs *et* négatifs

Comment représenter des entiers négatifs ??

- Magnitude signée
- Complément à 1
- Complément à 2
- Biaisée

Entiers non-signés: entiers positifs
Entiers signés: entiers positifs *et* négatifs

Comment représenter des entiers négatifs ??

- Magnitude signée
- Complément à 1
- Complément à 2
- Biaisée

Complément à 1:

Le bit de poids fort correspond au signe:

- 0 = positif
- 1 = négatif

Complément à 1:

Le bit de poids fort correspond au signe:

- 0 = positif
- 1 = négatif

Le complément à 1 est obtenu en complémentant bit à bit sa valeur absolue avec 1

Complément à 1:

Le bit de poids fort correspond au signe:

- 0 = positif
- 1 = négatif

Le complément à 1 est obtenu en complémentant bit à bit sa valeur absolue avec 1

Exemple: représentation de -23_{10} ?

Complément à 1:

Le bit de poids fort correspond au signe:

- 0 = positif
- 1 = négatif

Le complément à 1 est obtenu en complémentant bit à bit sa valeur absolue avec 1

Exemple: représentation de -23_{10} ?

- $|-23_{10}| = 23_{10} = 00010111_2$

Complément à 1:

Le bit de poids fort correspond au signe:

- 0 = positif
- 1 = négatif

Le complément à 1 est obtenu en complémentant bit à bit sa valeur absolue avec 1

Exemple: représentation de -23_{10} ?

- $|-23_{10}| = 23_{10} = 00010111_2$
- Complément à 1 de $|-23_{10}|$: (inversion des bits)

$$\begin{array}{rr} & 11111111 \\ - & 00010111 \\ \hline = & 11101000 \end{array}$$

Complément à 1:

Le bit de poids fort correspond au signe:

- 0 = positif
- 1 = négatif

Le complément à 1 est obtenu en complémentant bit à bit sa valeur absolue avec 1

Exemple: représentation de -23_{10} ?

- $|-23_{10}| = 23_{10} = 00010111_2$
- Complément à 1 de $|-23_{10}|$: (inversion des bits)

$$\begin{array}{rr} & 11111111 \\ - & 00010111 \\ \hline = & 11101000 \end{array}$$

Complément à 1:

Le bit de poids fort correspond au signe:

- 0 = positif
- 1 = négatif

Le complément à 1 est obtenu en complémentant bit à bit sa valeur absolue avec 1

Exemple: représentation de -23_{10} ?

- $|-23_{10}| = 23_{10} = 00010111_2$
- Complément à 1 de $|-23_{10}|$: (inversion des bits)

$$\begin{array}{rr} & 11111111 \\ - & 00010111 \\ \hline = & 11101000 \end{array}$$

Complément à 1:
Le bit de poids fort correspond au signe:

- 0 = positif
- 1 = négatif

Le complément à 1 est obtenu en complémentant bit à bit sa valeur absolue avec 1

Exemple: représentation de -23_{10} ?

- $|-23_{10}| = 23_{10} = 00010111_2$
- Complément à 1 de $|-23_{10}|$: (inversion des bits)

$$\begin{array}{rr} & 11111111 \\ - & 00010111 \\ \hline = & 11101000 \end{array}$$

– Deux représentations pour 0: 00000000_2 et 11111111_2
– Plage de nombres représentés sur 8 bits: -127 à $+127$

Complément à 2:

Le bit de poids fort correspond au signe:

- 0 = positif
- 1 = négatif

Complément à 2:

Le bit de poids fort correspond au signe:

- 0 = positif
- 1 = négatif

Le complément à 2 est obtenu en ajoutant 1 au complément à 1 de sa valeur absolue (et inversement)

Complément à 2:

Le bit de poids fort correspond au signe:

- 0 = positif
- 1 = négatif

Le complément à 2 est obtenu en ajoutant 1 au complément à 1 de sa valeur absolue (et inversement)

Exemple: représentation de -23_{10} ?

Complément à 2:

Le bit de poids fort correspond au signe:

- 0 = positif
- 1 = négatif

Le complément à 2 est obtenu en ajoutant 1 au complément à 1 de sa valeur absolue (et inversement)

Exemple: représentation de -23_{10} ?

- $|-23_{10}| = 23_{10} = 00010111_2$

Complément à 2:
Le bit de poids fort correspond au signe:

- 0 = positif
- 1 = négatif

Le complément à 2 est obtenu en ajoutant 1 au complément à 1 de sa valeur absolue (et inversement)

Exemple: représentation de -23_{10} ?

- $|-23_{10}| = 23_{10} = 00010111_2$
- Complément à 1 de $|-23_{10}| = 11101000_2$

Complément à 2:
Le bit de poids fort correspond au signe:

- 0 = positif
- 1 = négatif

Le complément à 2 est obtenu en ajoutant 1 au complément à 1 de sa valeur absolue (et inversement)

Exemple: représentation de -23_{10} ?

- $|-23_{10}| = 23_{10} = 00010111_2$
- Complément à 1 de $|-23_{10}| = 11101000_2$
- Ajout de 1: $11101000_2 + 1 = 11101001_2$

Complément à 2:
Le bit de poids fort correspond au signe:

- 0 = positif
- 1 = négatif

Le complément à 2 est obtenu en ajoutant 1 au complément à 1 de sa valeur absolue (et inversement)

Exemple: représentation de -23_{10} ?

- $|-23_{10}| = 23_{10} = 00010111_2$
- Complément à 1 de $|-23_{10}| = 11101000_2$
- Ajout de 1: $11101000_2 + 1 = 11101001_2$

Complément à 2:
Le bit de poids fort correspond au signe:

- 0 = positif
- 1 = négatif

Le complément à 2 est obtenu en ajoutant 1 au complément à 1 de sa valeur absolue (et inversement)

Exemple: représentation de -23_{10} ?

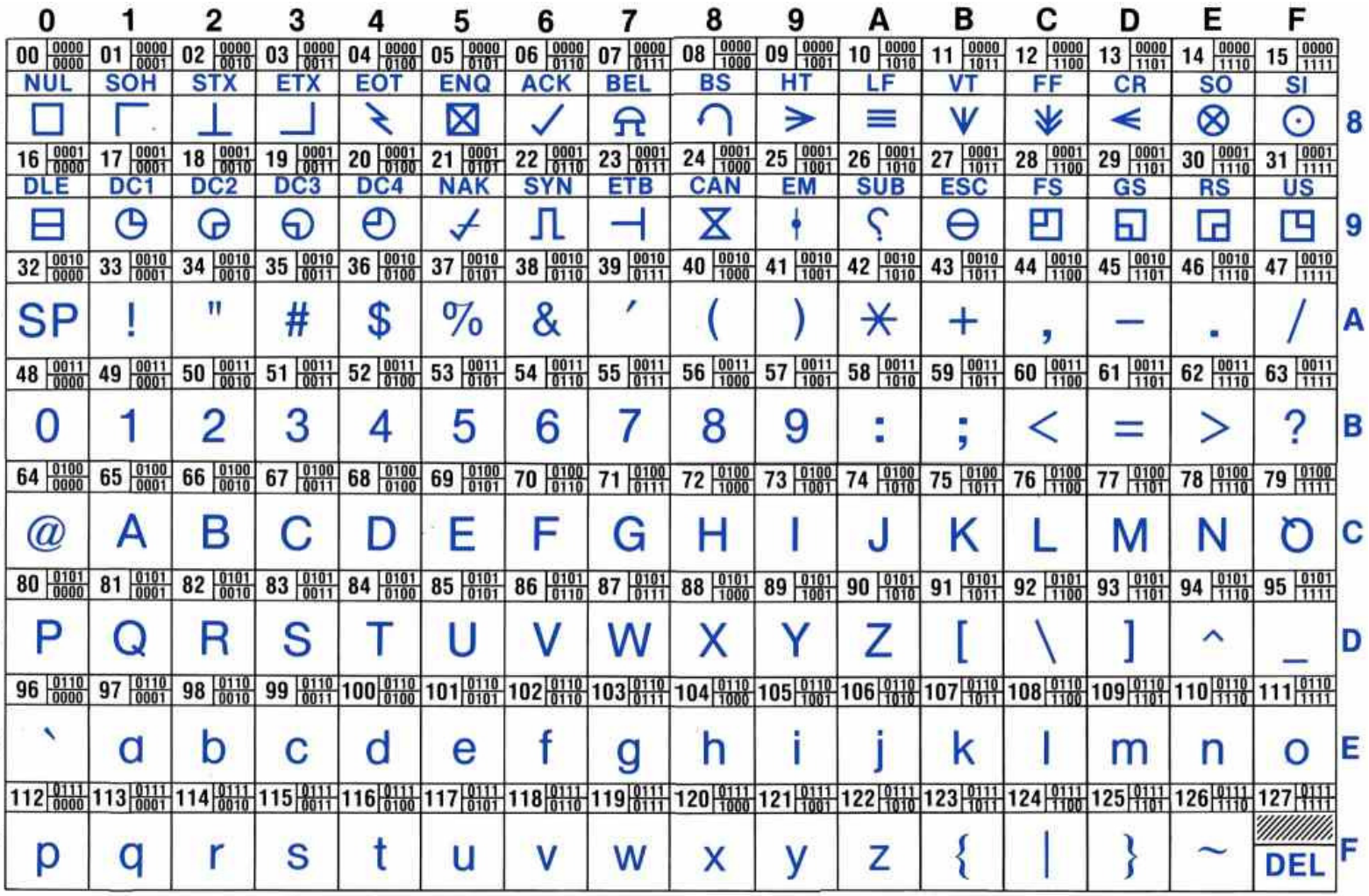
- $|-23_{10}| = 23_{10} = 00010111_2$
- Complément à 1 de $|-23_{10}| = 11101000_2$
- Ajout de 1: $11101000_2 + 1 = 11101001_2$

– Une seule représentation pour 0: 00000000_2
– Plage de nombres représentés sur 8 bits: -128 à $+127$

Plusieurs formats de représentation binaire:

- EBCDIC (*Extended Binary-Coded Decimal Interchange Code*)
  - Représentation sur 8 bits (256 caractères possibles)
  - Utilisé autrefois sur les mainframes IBM
- ASCII (*American Standard Code for Information Interchange*)
  - représentation sur 7 bits (pas d'accents)
  - ASCII étendu : sur 8 bits (mais pas normalisé)
    (ex: OEM, ANSI)
- Unicode : encodage sur 16 bits (65536 possibilités) pour représenter tous les caractères de toutes les langues

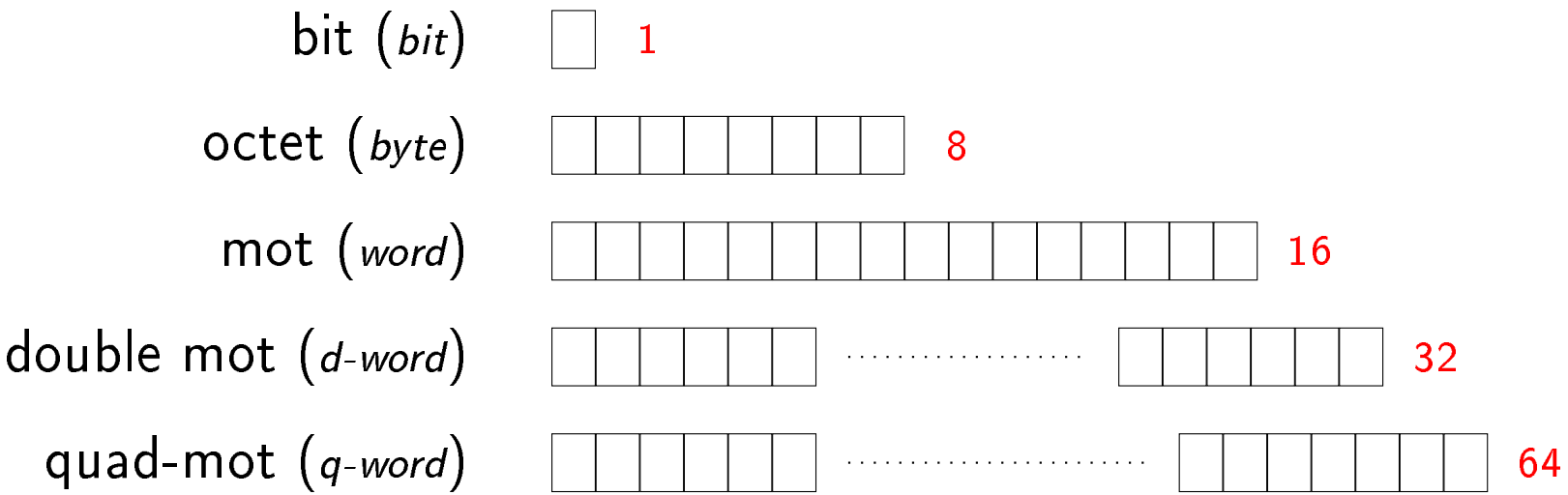
Exemple: Table ASCII restreinte



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI	8
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US	9
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	−	.	/	A
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	B
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	C
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_	D
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	E
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL	F

Non traité dans ce cours.
Voir la norme IEEE 754 pour la représentation en binaire des flottants.

Les types de données fondamentaux:

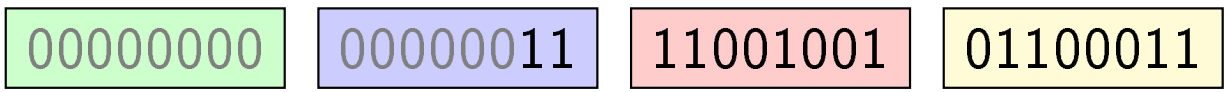


Ordre d' "empilement" lorsque stockage sur plusieurs octets ?
Exemple:

$$248163_{10} = 111100100101100011_2$$

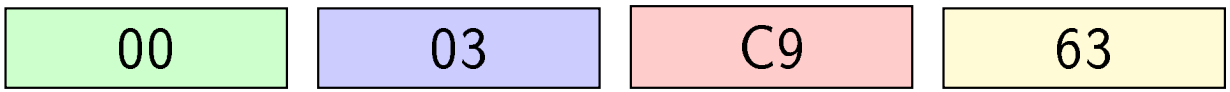
Ordre d' "empilement" lorsque stockage sur plusieurs octets ?
Exemple:

$$248163_{10} = 11110010010110001 1_2$$



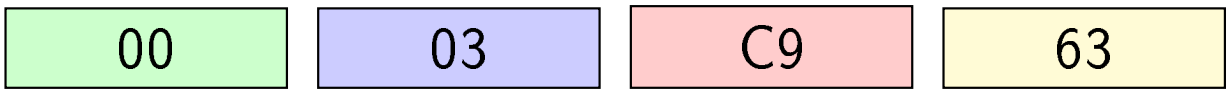
Ordre d' "empilement" lorsque stockage sur plusieurs octets ?
Exemple:

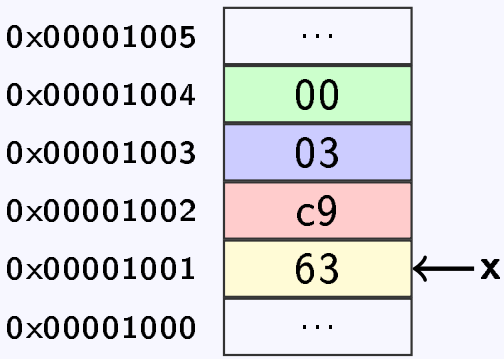
$$248163_{10} = 11110010010110001 1_2$$



Ordre d' "empilement" lorsque stockage sur plusieurs octets ?
Exemple:

$$248163_{10} = 11110010010110001 1_2$$



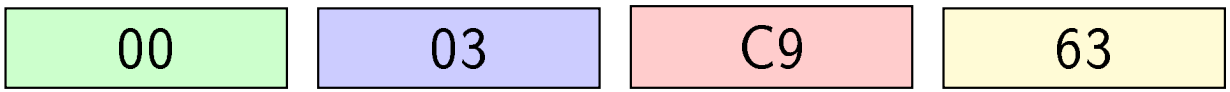


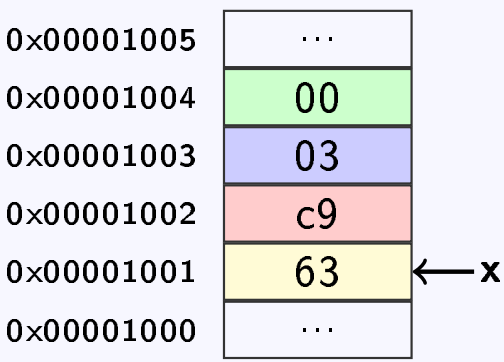
little-endian

lsb (*Less Significant Bit*) à l'adresse la plus petite

Ordre d' "empilement" lorsque stockage sur plusieurs octets ?
Exemple:

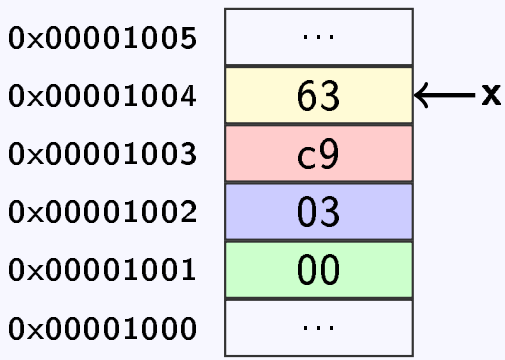
$$248163_{10} = 11110010010110001 1_2$$





little-endian

lsb (*Less Significant Bit*) à l'adresse la plus petite

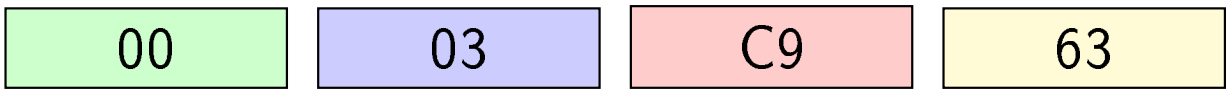


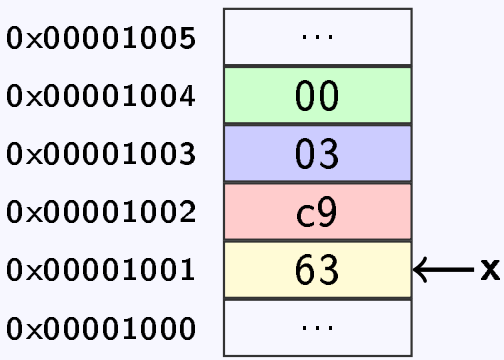
big-endian

msb (*Most Significant Bit*) à l'adresse la plus petite

Ordre d' "empilement" lorsque stockage sur plusieurs octets ?
Exemple:

$$248163_{10} = 11110010010110001 1_2$$





little-endian

lsb (*Less Significant Bit*) à l'adresse la plus petite

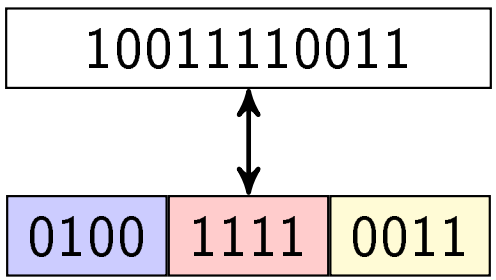
Architecture ix86: adressage par octet *little endian*

Note sur le passage de base 2 en base 16 et inversement:

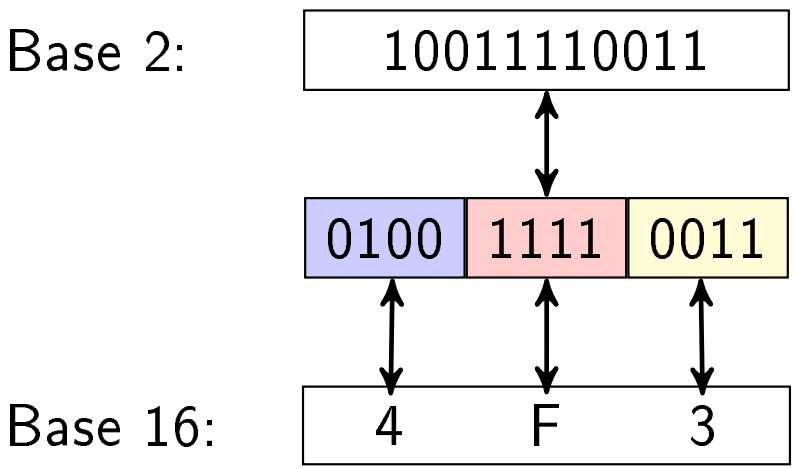
Base 2: 10011110011

Note sur le passage de base 2 en base 16 et inversement:

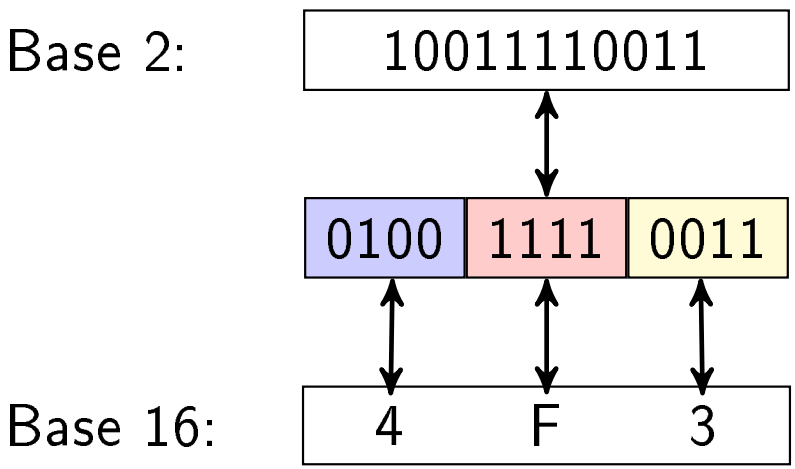
Base 2:



Note sur le passage de base 2 en base 16 et inversement:

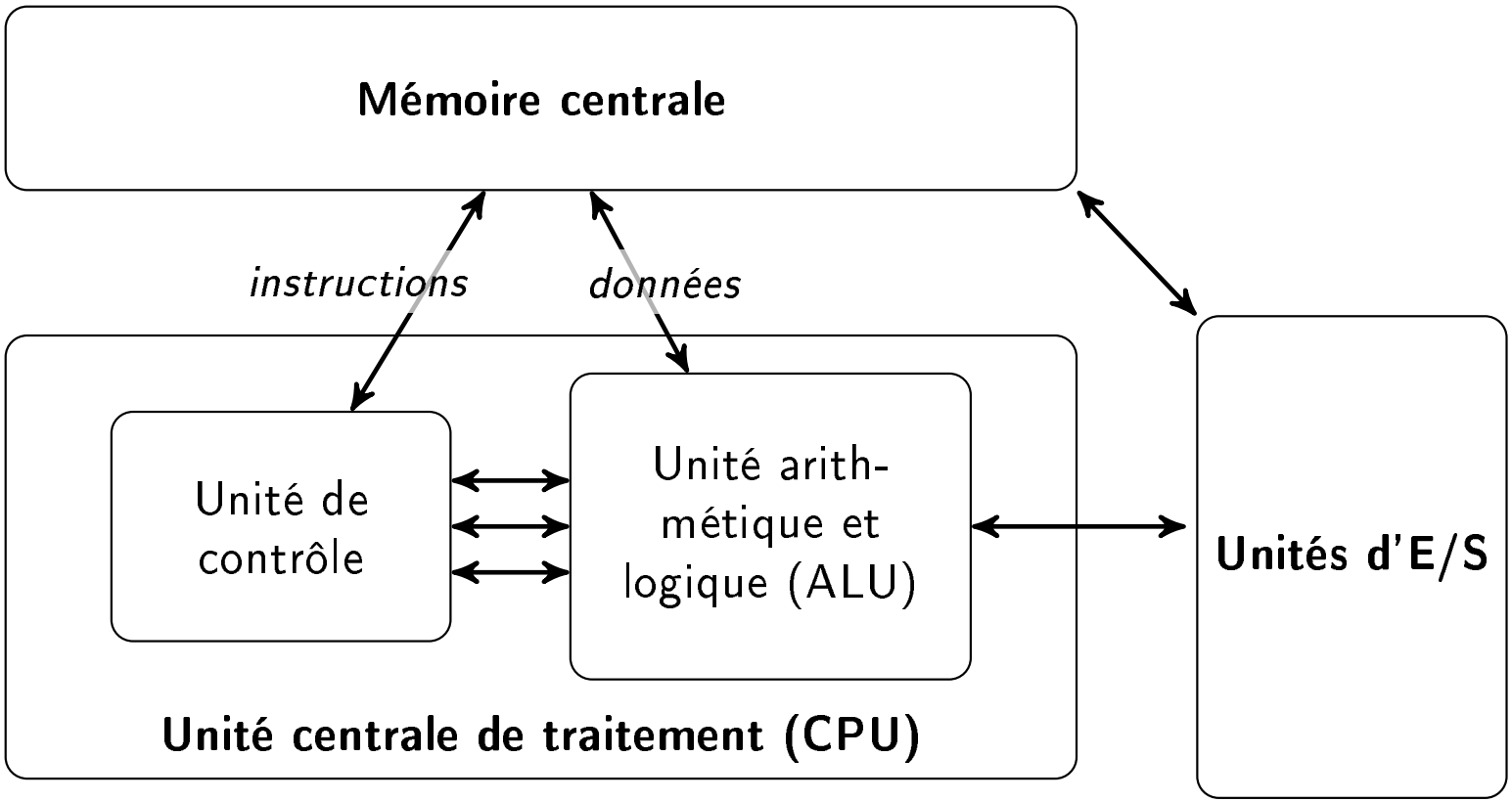


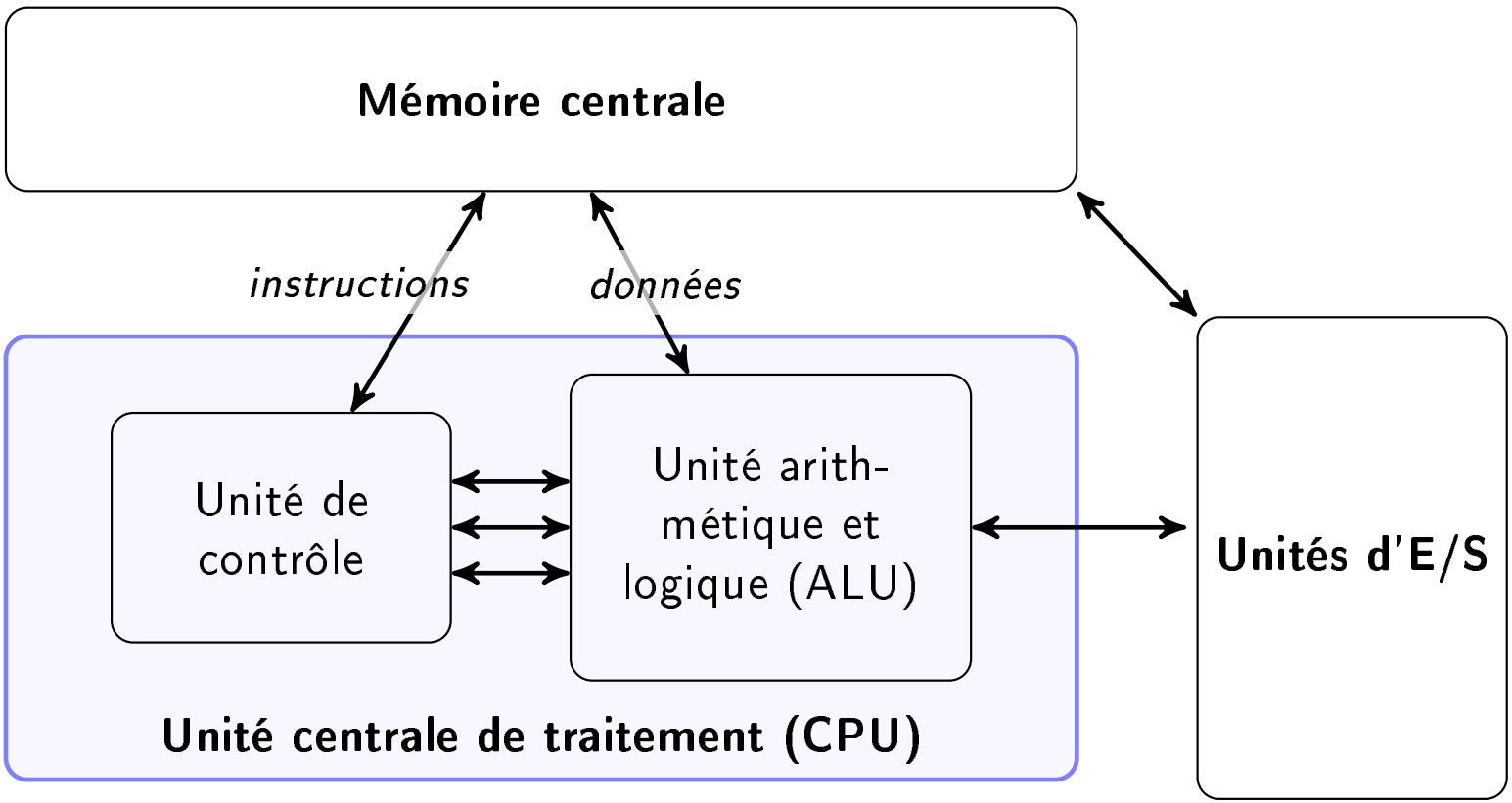
Note sur le passage de base 2 en base 16 et inversement:



B2	B10	B16
1111	15	F
1110	14	E
1101	13	D
1100	12	C
1011	11	B
1010	10	A
1001	9	9
1000	8	8
0111	7	7
0110	6	6
0101	5	5
0100	4	4
0011	3	3
0010	2	2
0001	1	1
0000	0	0

Plan


Mémoire centrale
instructions
données
Unité de contrôle
Unité arith-métique et logique (ALU)
Unité centrale de traitement (CPU)
Unités d'E/S


Mémoire centrale
instructions
données
Unité de contrôle
Unité arith-métique et logique (ALU)
Unité centrale de traitement (CPU)
Unités d'E/S

Plan

Plusieurs familles de micro-processeurs

- x86 (Intel, AMD) : PC et compatibles
- PowerPC (IBM) : équipent les Wii, PS3, XBox 360
- 68x00 (Motorola)
- SPARC, MIPS, ARM, etc.

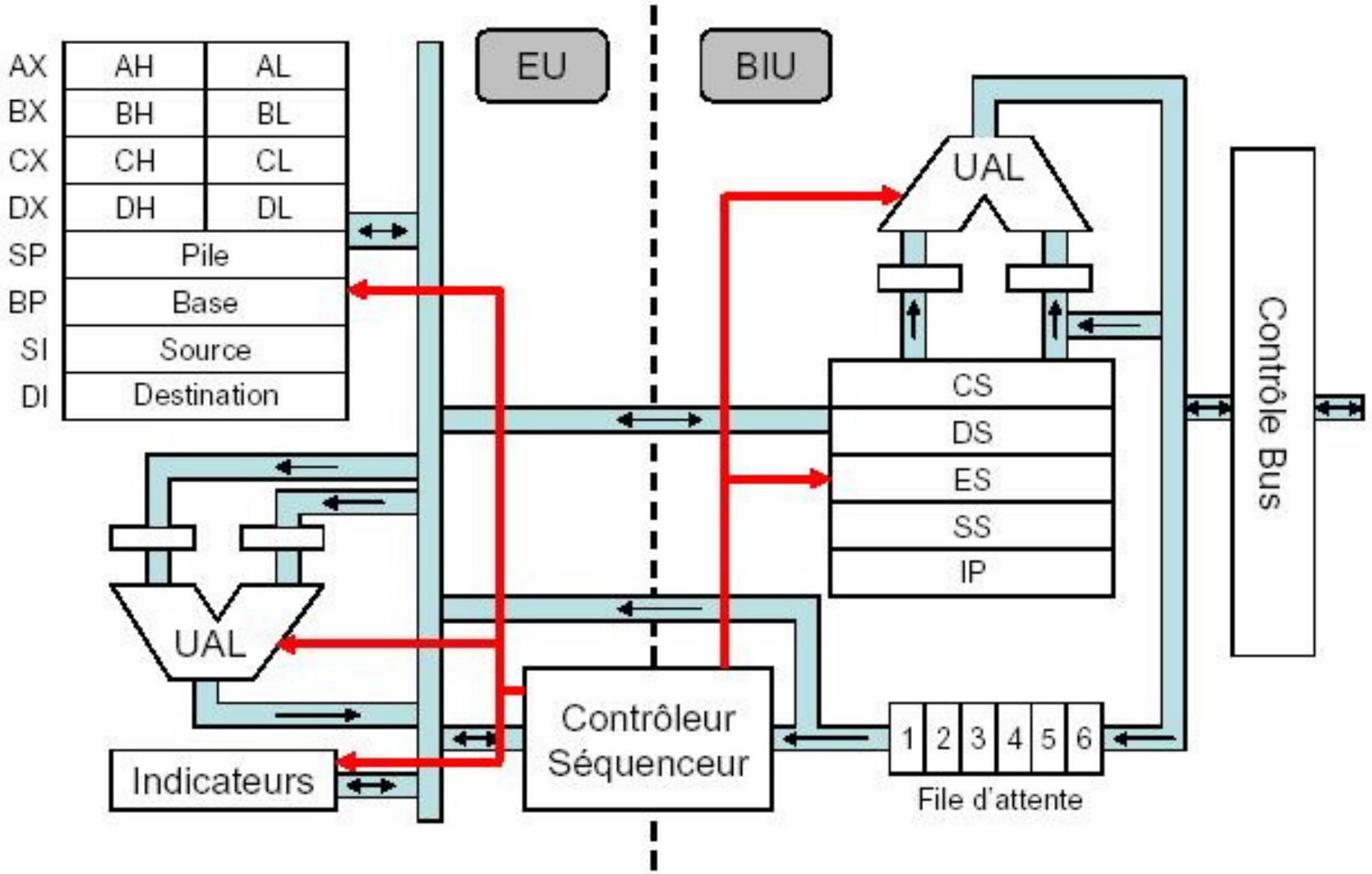
x86 : famille de microprocesseurs compatibles avec le jeu d'instructions de l'Intel 8086

- Evolution :
  - début 80 : 8086, microprocesseur 16 bits
  - puis 80286, 80386, 80486, Pentium, etc.
  - augmentation de la fréquence d'horloge, et de la largeur des bus d'adresses et de données
  - ajout de nouvelles instructions et registres
- Compatibilité ascendante
  - un programme écrit dans le langage machine du 286 peut s'exécuter sur un 386

x86 : famille de microprocesseurs compatibles avec le jeu d'instructions de l'Intel 8086

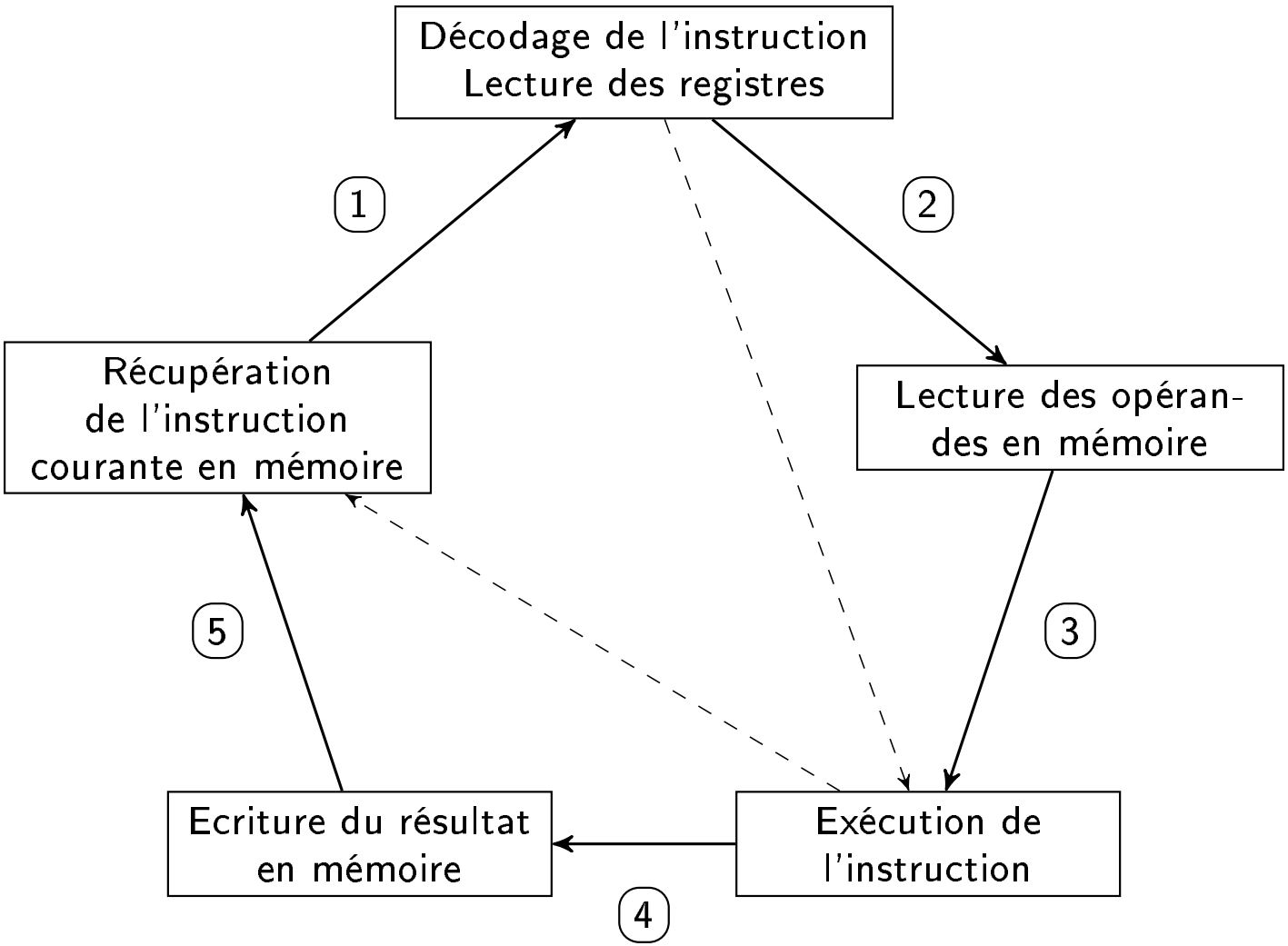
- Evolution :
  - début 80 : 8086, microprocesseur 16 bits
  - puis 80286, 80386, 80486, Pentium, etc.
  - augmentation de la fréquence d'horloge, et de la largeur des bus d'adresses et de données
  - ajout de nouvelles instructions et registres
- Compatibilité ascendante
  - un programme écrit dans le langage machine du 286 peut s'exécuter sur un 386

NB : en TP, on utilisera un assembleur 8086 sous linux, et on exécutera le programme sur des Pentium


AX
AH
AL
BX
BH
BL
CX
CH
CL
DX
DH
DL
SP
Pile
BP
Base
SI
Source
DI
Destination
EU
BIU
UAL
CS
DS
ES
SS
IP
Contrôle Bus
UAL
Contrôleur
Séquenceur
1
2
3
4
5
6
File d'attente
Indicateurs

- Bus : nappes de fils transportant l'info
  (données sur 16 bits, adresses sur 20 bits, infos de contrôle)
- Registres : emplacements de stockage à accès rapide
- UAL : unité de calcul (entiers et booléens)
- Unité de contrôle : interprète les instructions
- Horloge : cadence les instructions

Cycle de fonctionnement de l'unité de contrôle



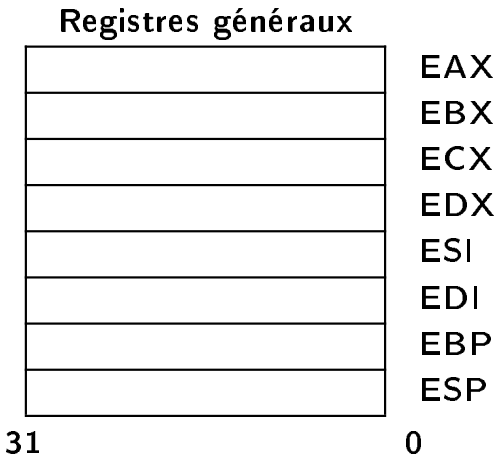
Sur une architecture 80x86, 32 bits :
modèle mode réel, modèle segmenté

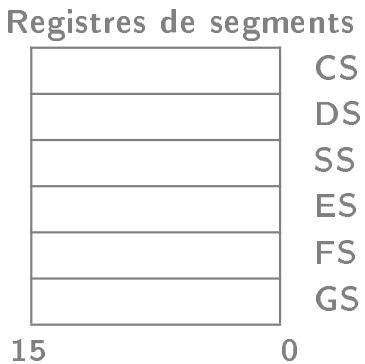
Registres présents sur une architecture 80x86, 32 bits :

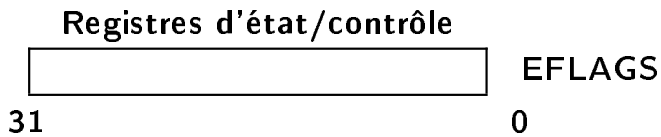
- 16 registres de base
- + d'autres registres

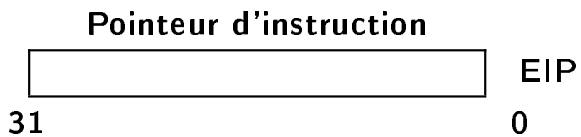
Registres de base classés en 4 catégories

1. Registres généraux (8, pour stocker adresses et opérandes)
2. Registres de segments
3. EFLAGS (registre d'état et de contrôle)
4. EIP (Extended Instruction Pointer)
   (pointe sur la prochaine instruction à exécuter)


Registres généraux
EAX
EBX
ECX
EDX
ESI
EDI
EBP
ESP
31
0


Registres de segments
CS
DS
SS
ES
FS
GS
15
0


Registres d'état/contrôle
EFLAGS
31
0


Pointeur d'instruction
EIP
31
0

Utilité des registres :

- stocker des opérandes lors d'opérations logiques ou arithmétiques
- stocker des opérandes pour des calculs d'adresses
- stocker des pointeurs (adresses mémoire)

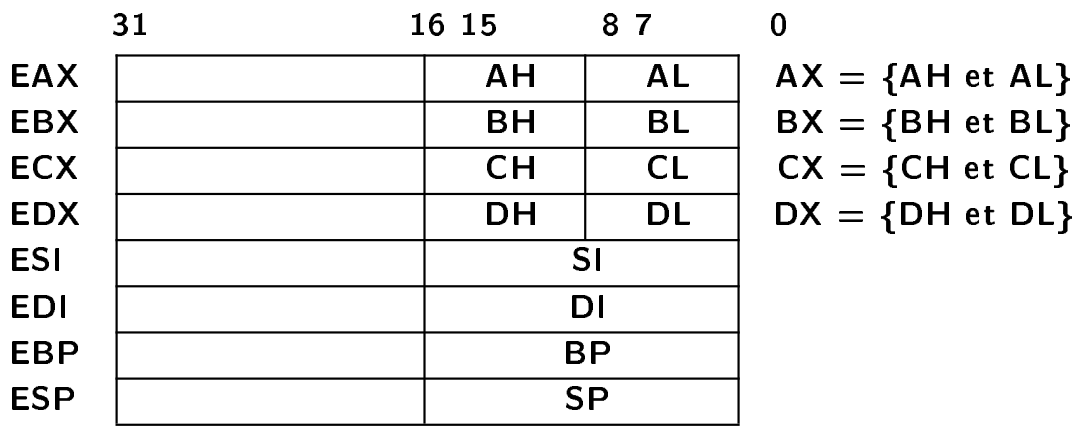
- certaines instructions imposent les registres à utiliser

Utilisations particulières :

- **EAX** : accumulateur ou valeur de retour de fonction
- **EBX** : pointeur vers les données dans le segment DS
- **ECX** : compteur de boucles
- **EDX** : accumulateur aux. et pointeur pour les entrées/sorties
- **ESI** : pointeur source pour la manipulation de caractères
- **EDI** : pointeur destination pour la manipulation de caractères
- **ESP** : pointeur de début de cadre de pile (segment SS)
- **EBP** : pointeur de bas de pile (segment SS)

- premiers processeurs : registres 16 bits (AX, BX, etc.)
- à partir du 386 : 32 bits (EAX, EBX, etc.)

Manipulation des registres 32, 16 et 8 bits :



EFLAGS : Registre 32 bits

- Indicateurs d'état : résultat d'instructions arithmétiques (mis à jour après chaque instruction)
  - *Carry Flag* **CF** : retenue en excédent
  - *Parity Flag* **PF** : nombre pair de '1'
  - *Zero Flag* **ZF** : vaut 1 si résultat nul
  - *Sign Flag* **SF** : vaut 0 si bit de poids fort = 0
  - *Overflow Flag* **OF** : 0 si le résultat peut être stocké dans l'opérande destination
- Indicateur **DF** : sens de l'auto-incrémentation (manipulation de chaînes)
- Indicateurs système

- Chaque processeur possède son propre jeu d'instructions machine directement compréhensibles
- Chaque instruction possède un identifiant numérique (*opcode*)
- Programmation directe du processeur avec les instructions machines:
  - difficile et long
  - quasi-impossible à comprendre

Exemple de programme en langage machine:

A1 01 10 03 06 01 12 A3 01 14

Ce programme additionne le contenu de deux cases mémoires et range le résultat dans une troisième...

Exemple de programme en langage machine:

A1 01 10 03 06 01 12 A3 01 14

Ce programme additionne le contenu de deux cases mémoires et range le résultat dans une troisième...

opcode	Symbole	octets	
A1	MOV AX, [*adr*]	3	AX $\leftarrow$ contenu de l'adresse *adr*
03 06	ADD AX, [*adr*]	4	AX $\leftarrow$ AX + contenu de l'adresse *adr*
A3	MOV [*adr*], AX	3	range AX à l'adresse *adr*

Exemple de programme en langage machine:

A1 01 10 03 06 01 12 A3 01 14

Ce programme additionne le contenu de deux cases mémoires et range le résultat dans une troisième...

opcode	Symbole	octets	
A1	MOV AX, [*adr*]	3	AX $\leftarrow$ contenu de l'adresse *adr*
03 06	ADD AX, [*adr*]	4	AX $\leftarrow$ AX + contenu de l'adresse *adr*
A3	MOV [*adr*], AX	3	range AX à l'adresse *adr*

Transcription du programme :

A1 01 10	MOV AX, [0110]	Charge AX avec le contenu de 0110
03 06 01 12	ADD AX, [0112]	Ajouter le contenu de 0112 à AX
A3 01 14	MOV [01 14], AX	Ranger AX à l'adresse 0114

- Programmation directe du processeur avec les instructions machines:
  - difficile et long
  - quasi-impossible à comprendre

- Programmation directe du processeur avec les instructions machines:
  - difficile et long
  - quasi-impossible à comprendre

Utilisation d'un langage de plus haut niveau : l'assembleur

Plan

Langage haut niveau

(Ada, C++, etc.)

```
c:= c + 3;
...
```

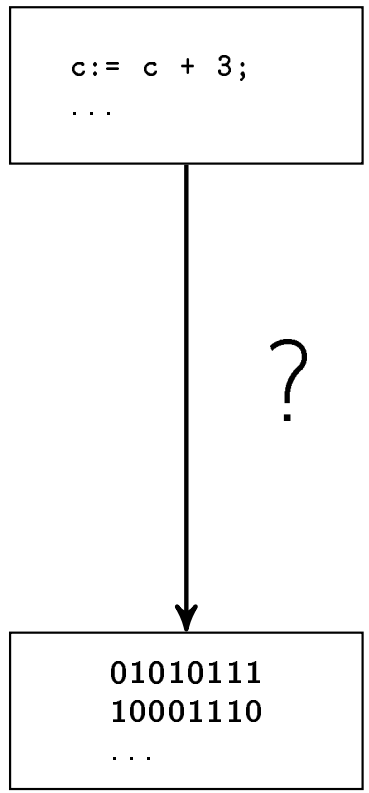
Langage machine

Binaire en mémoire qui forme un exécutable

```
01010111
10001110
...
```

Langage haut niveau

(Ada, C++, etc.)



Langage machine

Binaire en mémoire qui forme un exécutable

Langage haut niveau

(Ada, C++, etc.)

```
c:= c + 3;
...
```

Langage d'assemblage

Mnémoniques associées au langage machine

```
mov eax, [esi]
add eax, 3
...
```

Langage machine

Binaire en mémoire qui forme un exécutable

```
01010111
10001110
...
```

Langage haut niveau

(Ada, C++, etc.)

```
c:= c + 3;
...
```

Compilation

Langage d'assemblage

Mnémoniques associées au langage machine

```
mov eax, [esi]
add eax, 3
...
```

Langage machine

Binaire en mémoire qui forme un exécutable

```
01010111
10001110
...
```

Langage haut niveau

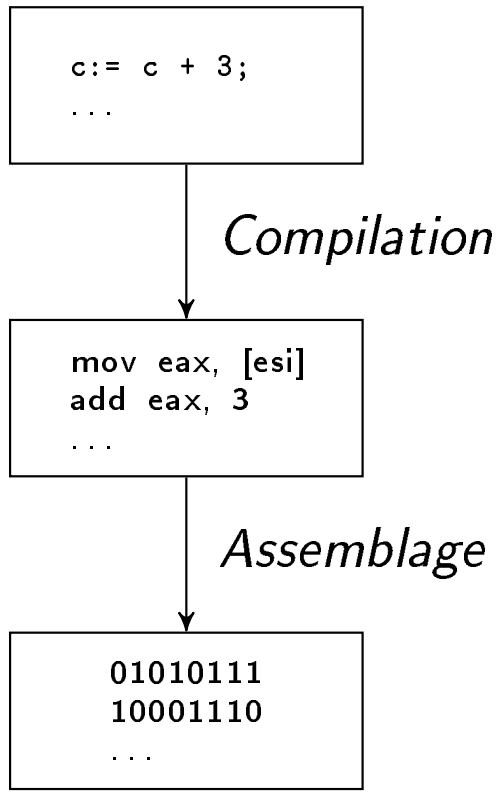
(Ada, C++, etc.)

Langage d'assemblage

Mnémoniques associées au langage machine

Langage machine

Binaire en mémoire qui forme un exécutable



Langage haut niveau

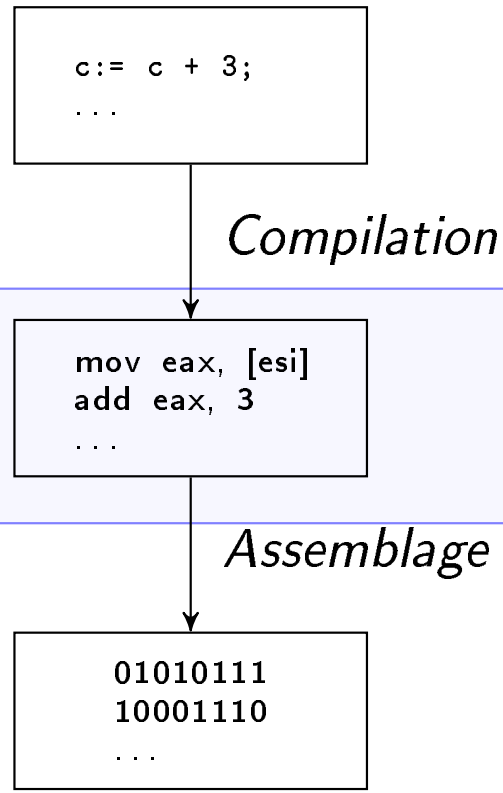
(Ada, C++, etc.)

Langage d'assemblage

Mnémoniques associées au langage machine

Langage machine

Binaire en mémoire qui forme un exécutable



Avantages :

- Accès à toutes les possibilités de la machine
- Vitesse d'exécution du code
- Faible taille du code généré
- Meilleure connaissance du fonctionnement de la machine

Inconvénients :

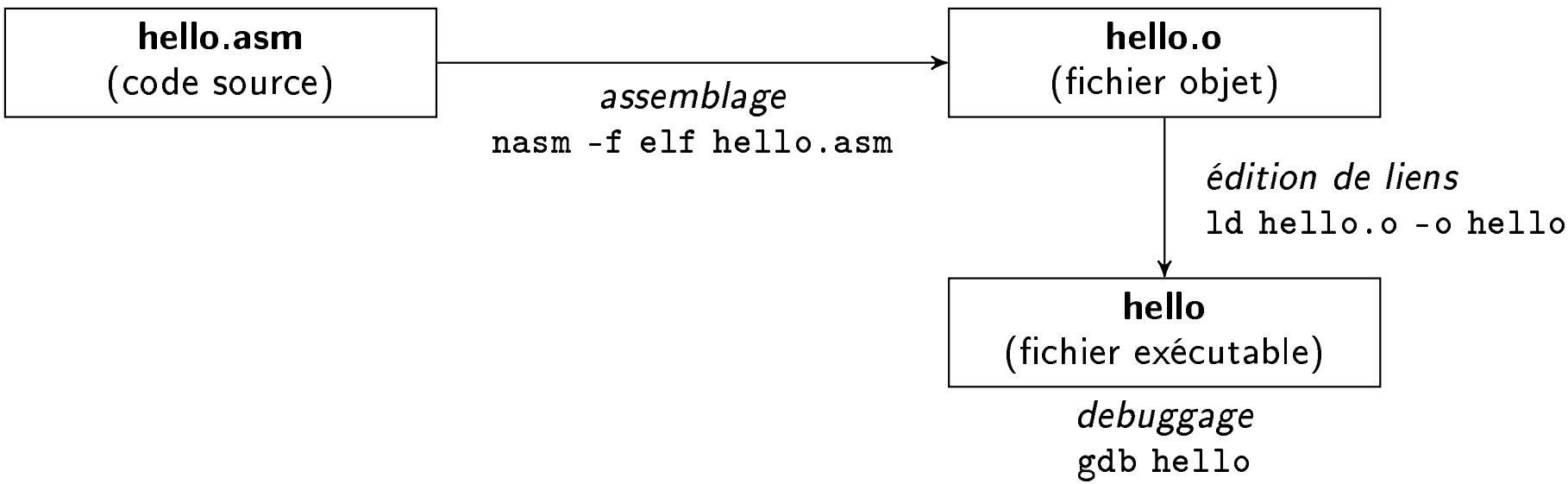
- Temps de codage plus long
- Fastidieux
- Pas de structures évoluées
- Garde-fous minimaux
- Absence de portabilité

Pourquoi apprendre l'assembleur ?

- Mieux comprendre la façon dont fonctionne un ordinateur
- Mieux comprendre comment fonctionnent compilateurs et langages haut niveau

En 3 phases:

1. Saisie du code assembleur avec un éditeur de texte
2. Compilation du code (assemblage)
3. Edition des liens



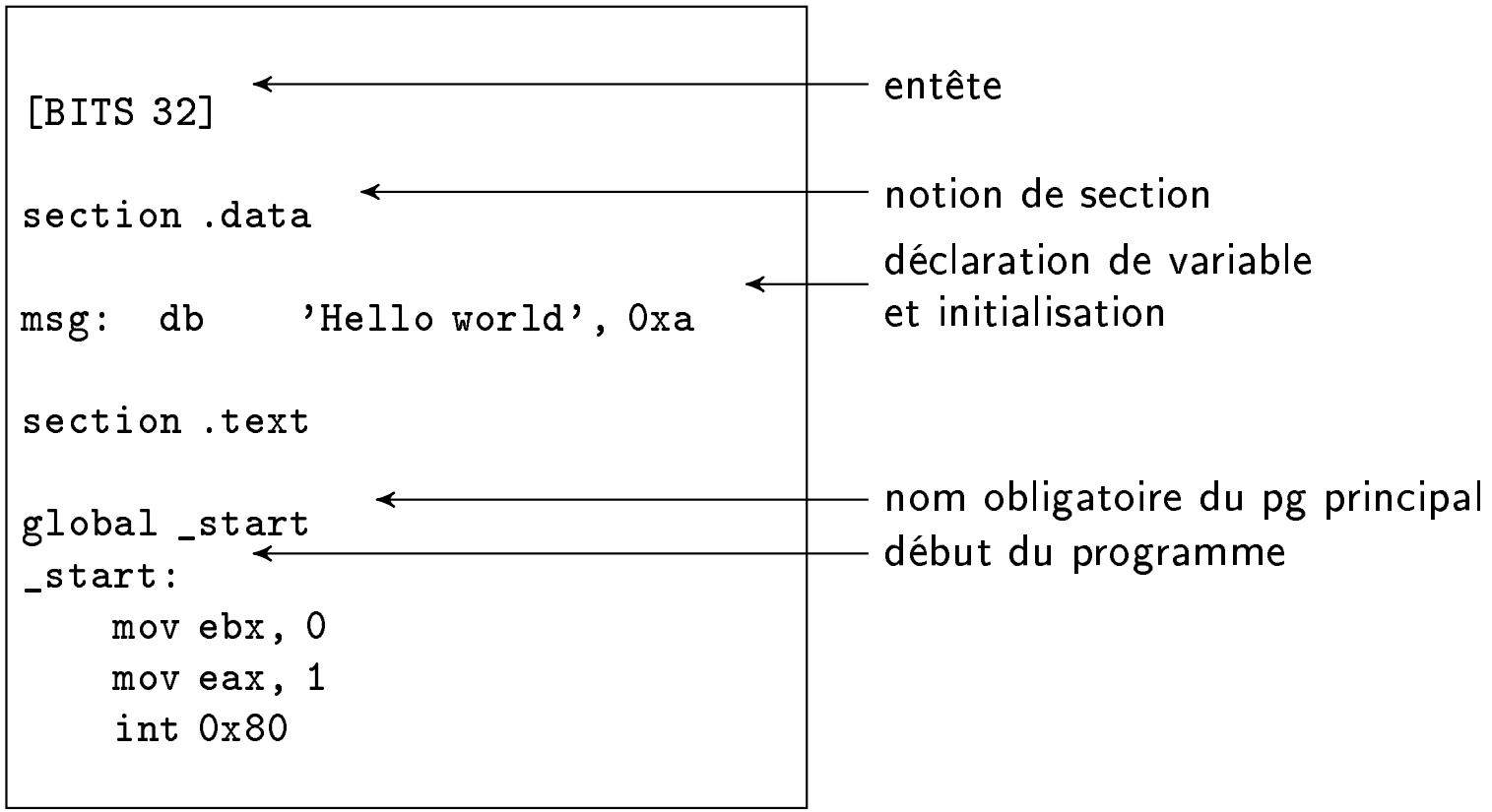
gdb = outil indispensable de debuggage !

Quelques commandes utiles :

- Lancement : `gdb <nom prg>`
- `break <label>` : définition d'un point d'arrêt (parmi les étiquettes du code asm du prog.)
- `info break` : liste des points d'arrêt
- `run` : exécution du programme jusqu'au premier point d'arrêt
- `c` (continue) : exécution jusqu'au prochain point d'arrêt

- `info registers` : valeur des registres du CPU
- `info variables` : liste des variables avec leur adresse
- `x <@ var>` : valeur contenue à l'adresse de la variable
- `set disassembly-flavor intel` : indique que l'on veut voir le code assembleur selon le type Intel
- `disassemble <label>`: affiche le code assembleur à partir de l'étiquette jusqu'à la suivante

Plan


[BITS 32]
section .data
msg: db 'Hello world', 0xa
section .text
global _start
_start:
mov ebx, 0
mov eax, 1
int 0x80
entête
notion de section
déclaration de variable
et initialisation
nom obligatoire du pg principal
début du programme

Structure d'un programme assembleur

- **Entête** ([BITS 16] ou [BITS 32]) : les instructions seront interprétées par défaut selon ces modes

- **section .data** : zone de déclaration des données initialisées
- **section .bss** : zone de déclaration des données non initialisées
- **section .text** : contient le code

- programme principal **global** nommé **_start**
- commentaires commencent par **;**

[label:] mnémonique $\mathbf{arg}_1$, $\mathbf{arg}_2$, $\mathbf{arg}_3$

- **label** ou étiquette : identificateur suivi d'un ':'
  identifie une position en mémoire (code ou données)
- **mnémonique**: nom correspondant à l'*opcode* de l'opérateur
  (l'opération à réaliser)
- **$\mathbf{arg}_i$**: entre 0 et 3 opérandes (les données manipulées)
  - notation Intel : destination = opérande de gauche

- Syntaxe d'un label :
  - mot composé de lettres, chiffres, _ $ # @ ~ . ?
  - commançant par une lettre, un chiffre, un point, '_' ou '?'

- Constantes numériques:
  - entiers positifs postfixés par leur base (b, h, q)

```
100     ; décimale
0a2h    ; hexadécimal (ou 0x0a2)
777q    ; octal
100b    ; binaire
```

- Caractères et chaînes :

```
'h'     ; caractère 'h'
'hello' ; chaîne 'hello'
```

Déclaration de **variables initialisées** (dans la section .data) :

<label> : d<type> <valeur>

Exemples:

```
section .data
; déclaration d'un octet et
; initialisation à 34
var1 : db 34
; déclaration d'un tableau (2, 5, 31)
t : dw 2, 4, 0x1F
; déclaration d'un double-mot (4 octets)
x : dd 364
; déclaration d'un tableau
t2 : times 5 db 0
```

Types		
octet	db	(**b**yte)
mot	dw	(**w**ord)
double-mot	dd	(**d**ouble-word)
quad-mot	dq	(**q**uad-word)

la 2ème lettre indique
la taille à réserver

Déclaration de **variables non initialisées** (dans la section .bss) :

<label> : res<type> <valeur>

Exemples:

```
section .bss

x : resw          ; réservation d'un mot

tab : resb 32     ; réservation d'un tableau de 32 octets
```

Déclaration de **constantes** (pas de ':') :

<label> equ <valeur>

Exemple :

```
max equ 15
```

Constantes spéciales :

- **$** : adresse courante dans la déclaration des variables (segment de données)
- **$$** : adresse du début du segment courant

Exemple d'utilisation:

```
chaine : db 'hello'
taille equ $ - chaine
```

Une instruction spécifie :

- l'opération à réaliser (opérateur)
- les données à manipuler (opérandes)

Les données (opérandes) peuvent être de plusieurs types :

- **Registre** : dans un registre du processeur
- **Mémoire** : correspond à une zone de la mémoire identifiée par son adresse logique
- **Immédiat** : constante numérique stockée dans le code
- **Implicite**: non indiquée, mais implicite du fait de l'opérateur (ex.: incrémentation, le 1 est implicite)

Plan

Illustration avec l'instruction de transfert

$$\textbf{mov} <\textbf{op}_1>, <\textbf{op}_2>$$

- avec
  - $\textbf{op}_1$: la destination (registre, adresse mémoire)
  - $\textbf{op}_2$: la donnée à transférer (valeur, registre, adresse, etc.)
- action : le contenu de op_2 est transféré dans op_1

1. L'adressage **immédiat** (ou **par valeur**):

mov ax, 12

- Le contenu du registre est initialisé par une valeur immédiate
- La valeur fait partie de l'instruction
- Le nombre de bits du registre (op_1) détermine le nombre de bits impliqués dans l'opération:

```
mov bh, 5          ; bh := 5   (transfert de 8 bits)
mov bx, 5          ; bx := 5   (transfert de 16 bits)
mov eax, 0x5A32    ; eax := 0x5A32   (transfert de 32 bits)
```

2. L'adressage **par registre** :

mov eax, ebx

- Le contenu du registre eax est initialisé par le contenu du registre ebx
- Pas d'accès mémoire
- La taille des registres doivent être équivalentes

```
mov eax, ebp        ; eax := ebp    (transfert de 32 bits)
mov bh, al          ; bh := al      (transfert de 8 bits)
```

3. L'adressage **direct** :

mov eax, <adresse_mémoire>

- L'adresse de la donnée fait partie de l'instruction
- La taille du registre destination détermine le nombre de bits lus en mémoire

```
id1 :  db 231      ; déclaration et initialisation d'une variable
...

mov ebx, id1       ; ebx contient l'adresse mémoire de la variable id1
mov eax, [id1]     ; eax contient la valeur de la zone mémoire identifiée
                   ; par la var. id1 (soit 231)
mov edx, [0x2000]  ; edx contient la valeur de la zone mémoire
                   ; située à l'adresse 0x2000
```

4. L'adressage **indirect** :

mov eax, [ebx]

- L'adresse de la donnée est contenue dans un registre (ebx, ebp, esi, ou edi)
- La taille du registre destination détermine le nombre de bits lus en mémoire

Exemples d'adressages basés, ou indexés simples :

```
mov ebx, 0x2000 ; ebx := 0x2000  (adresse mémoire)
mov eax, [ebx]  ; eax := contenu de la zone mémoire située à l'adresse 0x2000
mov dx, [esi]   ; dx := 16 bits situés dans la zone mémoire
                ; dont l'adresse est contenue dans esi
mov dx, [esi+2]
```

4. L'adressage **indirect** :

$$\text{Offset} = \text{base} + (\text{index*echelle}) + \text{déplacement}$$

$$Offset = \begin{bmatrix} EAX \\ EBX \\ ECX \\ EDX \\ ESP \\ EBP \\ ESI \\ EDI \end{bmatrix} + \left[\begin{pmatrix} EAX \\ EBX \\ ECX \\ EDX \\ EBP \\ ESI \\ EDI \end{pmatrix} * \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} \right] + \begin{bmatrix} rien \\ 8\,bits \\ 16\,bits \\ 32\,bits \end{bmatrix}$$

4. L'adressage **indirect** :

Exemples d'adressages basés et indexés :

```
tab: dw 5, 10, 34    ; déclaration d'un tableau
...
mov ebx, tab         ; bx contient l'adresse du tableau
mov esi, 2           ; si := 2
mov ax, [ebx+esi]    ; ax := contenu de la 2ème case du tableau (10)
                     ; (i.e. adresse mémoire du tableau, décalée de 2 octets)
mov dx, [ebx+4]      ; dx := contenu de la 3ème case du tableau (34)
```

La taille du registre de destination détermine le nombre de bits lus en mémoire

- Certaines instructions sont ambigües concernant la taille de la donnée manipulée
- Exemple:

```
mov [ebx], 5    ; écrit '5' à l'adresse mémoire contenue dans ebx
                ; mais sur combien d'octets ??
```

- Pour spécifier la taille :

```
mov byte [ebx], 5     ; écriture sur 1 octet
mov word [ebx], 5     ; écriture sur 2 octets
mov dword [ebx], 5    ; écriture sur 4 octets
```

Plan

- Instruction de transferts de données / affectation
  (mov, etc.)
- Instructions arithmétiques
  (add, sub, dec, inc, neg, ...)
- Instructions logiques
  (and, or, etc.)
- Instructions de contrôle de flot
  (jmp, loop, call, etc.)

mov dest, source

- **source** : immédiat, registre général, adresse mémoire
- **dest** : registre général, adresse mémoire
- opérandes de **même taille** !

Exemples:

```
mov ebx, eax          ; ebx := eax
mov eax, [edx]        ; eax := 32 bits lus en mémoire à l'adresse contenue
                      ; dans edx
mov word [ebx], 74    ; écrit de 74 sur 2 octets à l'adresse mémoire
                      ; contenue dans ebx
```

add x, y

- addition binaire, valable en signés et non-signés
- équivaut à $x := x + y$
- si le résultat "déborde" la représentation binaire, alors affectation du CARRY Flag (registre EFLAGS)
  Exemple :
  - $11110011 + 10001000 = 101111011$
  - 8 bits + 8 bits $\rightarrow$ 9 bits max : 8 bits + CARRY flag (CF)
- Les autres flags affectés :
  - SF = 0 si le bit de poids fort = 0
  - ZF = 1 si le résultat est nul
  - OF = 0 si le résultat peut être stocké dans la destination

sub x, y

- Mêmes remarques que pour l'addition
- Nombres négatifs = complément à deux
  - pour calculer le complément à deux : **neg x**
  - attention aux flags :
    - $4 + (-1) = 3$
    - $00000100 + 11111111 = [1]00000011 \quad (3)$
    - il faut ignorer le flag CF !

inc x

- équivaut à $x := x + 1$

dec x

- équivaut à $x := x - 1$

mul *x*

- Multiplication de **eax**, **ax**, ou **al** par l'opérande *x* (zone mémoire ou registre)
- Le résultat tient dans un nombre de bits égal à deux fois la taille de l'opérande *x*

```
mul bl      ; ax := al * bl
mul bx      ; dx:ax := ax * bx
mul ecx     ; edx:eax := eax * ecx
```

div *x*

- Division de **edx**:**eax**, **dx**:**ax**, ou **ax** par l'opérande *x* (zone mémoire ou registre)
- Le nombre de bits divisés est égal à deux fois la taille de l'opérande *x*

```
div bl     ; al := ax / bl          et  ah := ax mod bl  (reste)
div bx     ; ax := dx:ax / bx       et  dx := dx:ax mod bx
div ecx    ; eax := edx:eax / ecx   et  edx := edx:eax mod ecx
```

Notes concernant **mul** *x* et **div** *x* :

- C'est la taille de l'opérande *x* qui spécifie les registres impliqués
  - il faut parfois spécifier la taille de l'opérande *x*
  - ex: mul byte [bx]
- Il n'est pas possible d'utiliser des constantes ou des valeurs immédiates
- Pour la multiplication/division avec des nombres négatifs, voir **imul** et **idiv**

Notes concernant **mul** *x* et **div** *x* :

- Il n'est pas possible de diviser un octet par un octet, ou 2 octets par 2 octets
  Exemple : un octet *k* divisé par un octet *l*

```
mov al, k
cbw          ; complète le registre ah avec des zéros
mov bl, l
div bl       ; divise ax par bl
```

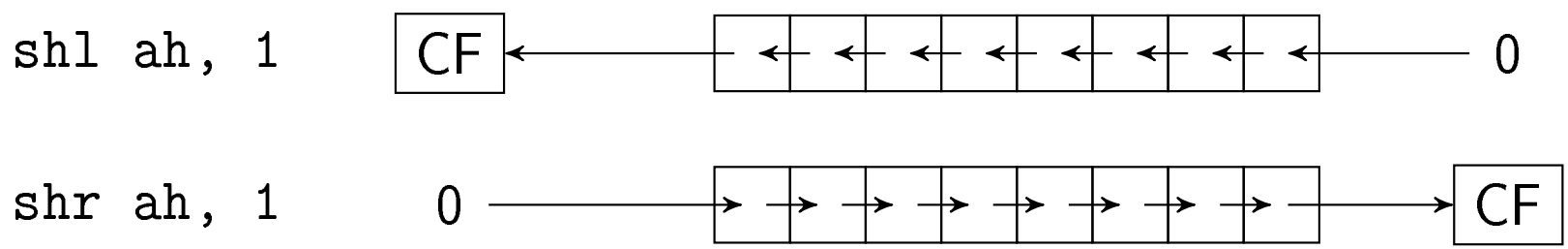
- Instructions de conversion disponibles :
  - **cbw** (*Convert Byte to Word*) : étend à 16 bits l'octet situé en **al**, et place le résultat en **ax**
  - **cwd** (*Convert Word to Double-word*) : étend à 32 bits les 2 octets situés en **ax**, et place le résultat dans **dx**:**ax**
  - **cwde** (*Convert Word to Extended Doubleword*) : convertit le double-mot situé en **eax** et place le résultat dans **edx**:**eax**

Décalage des bits d'un registre vers la droite ou vers la gauche

- pour décoder des données bit à bit
- pour diviser/multiplier rapidement par une puissance de 2
  - décaler **ax** de n bits vers la gauche revient à le multiplier par 2^n
  - décaler **ax** de n bits vers la droite revient à le diviser par 2^n

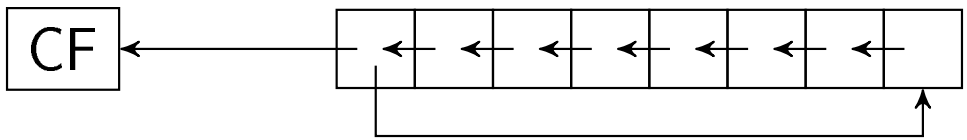
shl *registre*, *n*
shr *registre*, *n*

- **shl** (*Shift Left*) : décale les bits du registre de *n* positions vers la gauche
- **shr** (*Shift Right*) : décale les bits du registre de *n* positions vers la droite
- le bit de gauche (ou de droite pour **shr**) est transféré dans CF
- les bits introduits sont à zéro

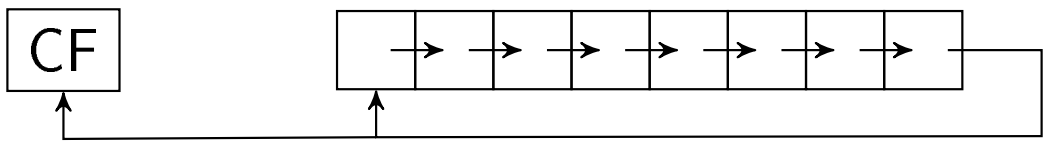


- **sal** : décalage arithmétique gauche
  - identique à **shl**
- **sar** : décalage arithmétique droit
  - si bit de poids fort = 1, alors le bit inséré est 1
  - sinon, 0

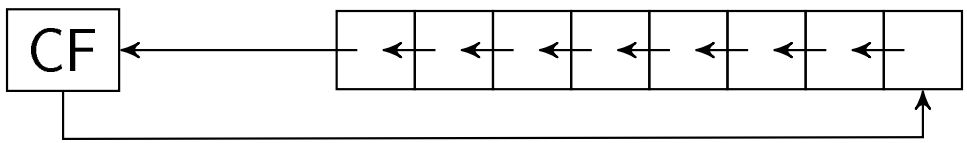
- **rol** *registre*, **1** (*Rotate Left*)
  - rotation vers la gauche : le bit de poids fort passe à droite, et est aussi copié dans CF
  - les autres bits sont décalés d'une position



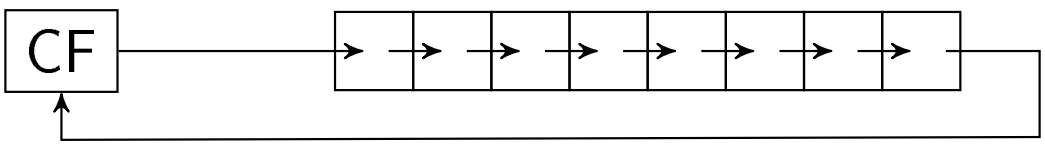
- **ror** *registre*, **1** (*Rotate Right*)
  - comme **rol**, mais à droite



- **rcl** *registre*, **1** (*Rotate Carry Left*)
  - rotation vers la gauche en passant par CF



- **rcr** *registre*, **1** (*Rotate Carry Right*)
  - comme **rcl**, mais à droite



- Opérateurs logiques : **and**, **or**, **xor** et **not**
  - calcul **bit à bit** : pas de propagation de retenue
- sous la forme :

or *dest*, *source*

  - *dest*: registre ou emplacement mémoire
  - *source*: valeur immédiate, registre ou adresse mémoire

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

NOT	0	1
	1	0

Utilisations :

- mise à zéro d'un registre:
  - and ax, 0x0000
  - xor ax, ax
- forcer certains bits à 1:
  - or ax, 0xff00
    l'octet de poids fort de ax vaut 0xff, l'octet de poids faible reste inchangé
- forcer certains bits à 0:
  - and ax, 0xff00
    l'octet de poids faible vaut 00, l'octet de poids fort reste inchangé
- inverser certains bits:
  - xor ax, 0xffff ; inverse tous les bits de ax

- Le processeur : fonctionnement séquentiel
  - exécute une instruction en mémoire, puis la suivante
  - cycle de fonctionnement:
    1. lire et décoder l'instruction à l'adresse EIP
    2. EIP $\leftarrow$ EIP + *taille de l'instruction*
    3. exécuter l'instruction
  - le registre EIP contient l'adresse de la prochaine instruction à exécuter

- Comment
  - répéter une suite d'instruction ??
  - déclencher une action en fonction d'un test ??

Solution :

- Utilisation d'une instruction de branchement ou saut
  - on indique au processeur l'adresse de la prochaine instruction (modifie la valeur de EIP)
- Deux catégories de sauts :
  - **saut inconditionnel** : toujours effectué
  - **saut conditionnel** : effectué seulement si une condition est vérifiée

jmp *dest*

- **saut inconditionnel**
  - *dest* : label, registre, ou adresse

- Exemple:

```
0x0100   B8 00 00    mov ax, 0          ; ax ← 0
0x0103   A3 01 40    mov [0x0140], ax   ; écrit le contenu de ax à l'adresse 0x140
0x0106   EB FC       jmp 0x0103         ; branchement à l'adresse 0x103
0x0107   B8 00 10    mov ax, 0x10       ; jamais exécuté !
```

saut conditionnel : **j***cc*

- basé sur la valeur des indicateurs EFLAGS (ZF, CF, SF, OF)
- pour comparer 2 valeurs : instruction **cmp ax, bx**
  - compare les deux opérandes par soustraction
  - modifie les indicateurs EFLAGS

Exemple :

```
cmp eax, 5
js titi       ; saut à titi si SF = 1
jz tata       ; saut à tata si ZF = 1
jae toto      ; saut à toto si eax >= 5
```

Tests conditionnels:

Test	Signé	Non signé
$=$	je	je
$\neq$	jne	jne
$<$	jl/jnge	jb/jnae
$\leq$	jle/jng	jbe/jna
$>$	jg/jnle	ja/jnbe
$\geq$	jge/jnl	jae/jnb

jz (*jump if zero*), jo (*jump if overflow*), js (*jump if sign*)

Traduction du **si alors sinon** (*if*) :

```
si eax=1
alors
        ebx ← 10
sinon
        ecx ← 20
finsi
```

```
si_:    cmp eax, 1
        jne sinon
alors:  mov ebx, 10
        jmp finsi
sinon:  mov ecx, 20
finsi:  ...
```

Traduction du **pour** (*for*) :

```
pour i=0 jusqu'à i=10
     ...
finpour
```

```
          mov ecx, 0
pour:     cmp ecx, 10
          je finpour
          ...
          inc ecx
          jmp pour
finpour:  ...
```

Traduction du **tantque** (*while*) :

```
ebx ← 5
tant que ebx>0 faire
     ebx ← ebx - 1
     ...
fin tant que
```

```
                mov ebx, 5
tantque:        cmp ebx, 0
                jle fintantque
                dec ebx
                ...
                jmp tantque
fintantque:     ...
```

Traduction du **cas** (*switch*) :

```
cas où
    bx=1:   ...
    bx=2:   ...
    bx=3:   ...
    sinon:   ...
fin cas
```

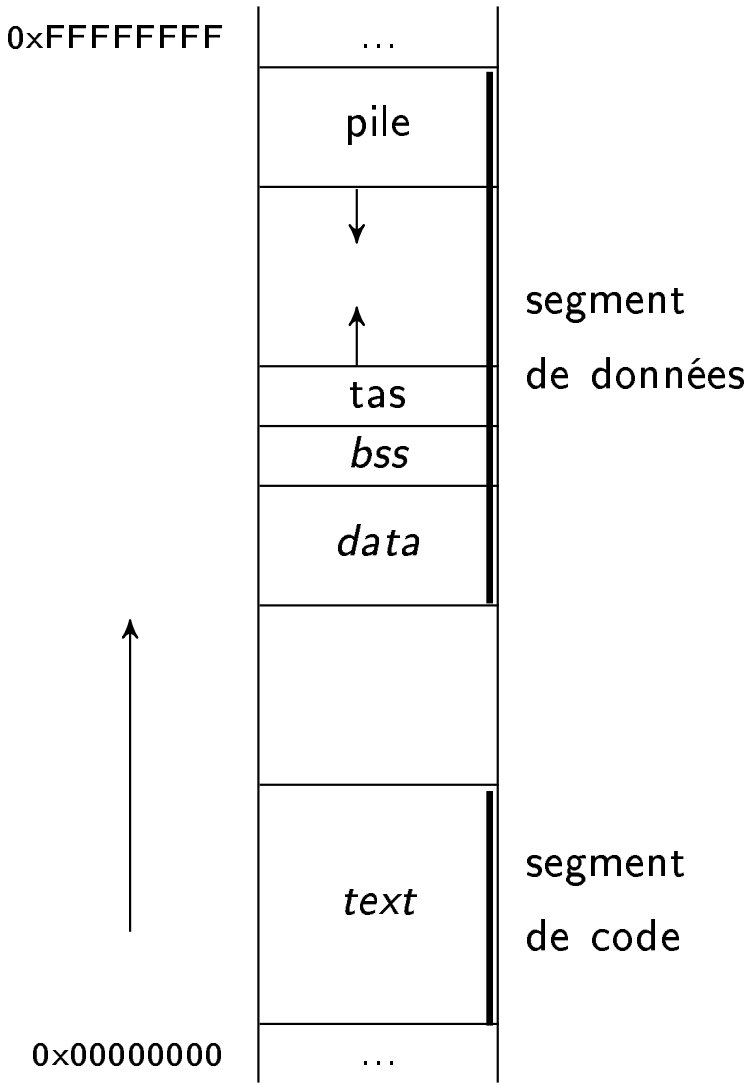
```
cas:
cas1:     cmp bx, 1
          jne cas2
          ...
          jmp fincas
cas2:     cmp bx, 2
          jne cas3
          ...
          jmp fincas
cas3:     cmp bx, 3
          jne sinon
          ...
          jmp fincas
sinon:    ...
fincas:   ...
```

Plan

La pile

- La pile est une structure (et une zone mémoire) qui sert à stocker temporairement des informations
- Utilisée lors d'appel de procédures (passage de paramètres)
- Fonctionne en mode LIFO (*Last In First Out*)

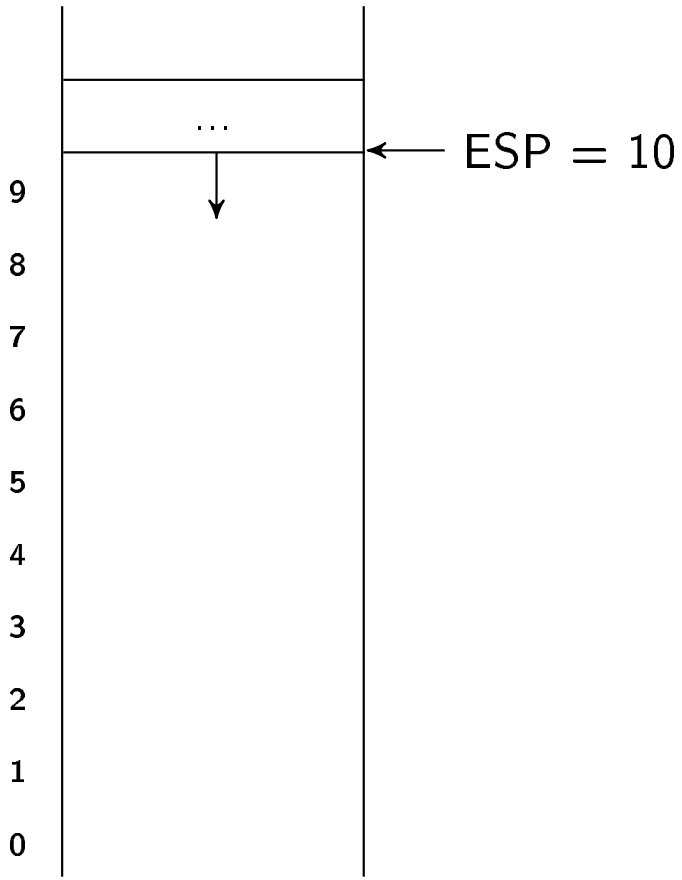
La pile



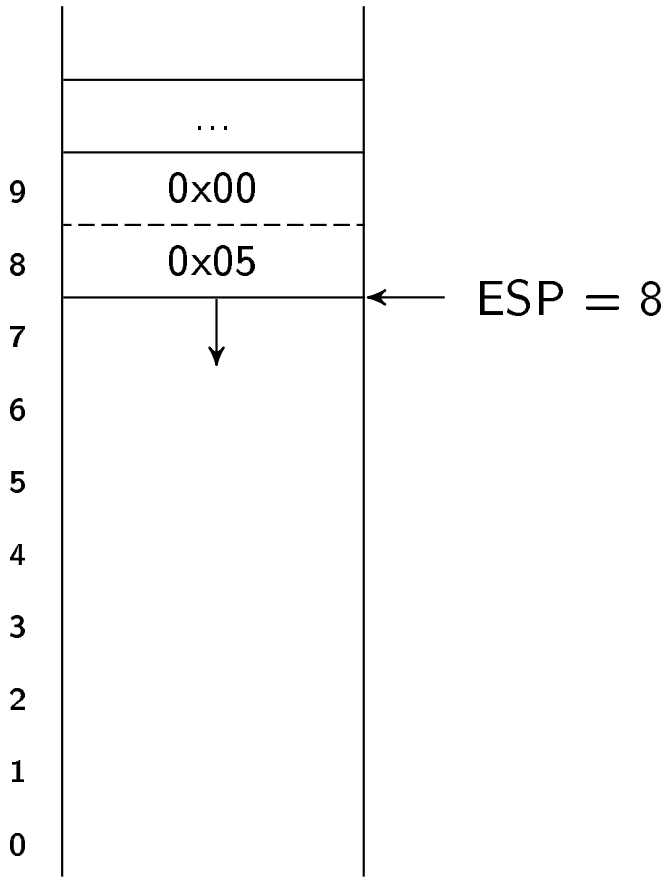
- la pile “grandit” vers le bas
- EBP = adresse du “fond” de la pile
- ESP = adresse du sommet de la pile (dernier élément empilé)

- **push** *arg* : empiler *arg* (reg., case mémoire, valeur)
- **pop** *arg* : dépiler dans *arg* (reg., case mémoire)

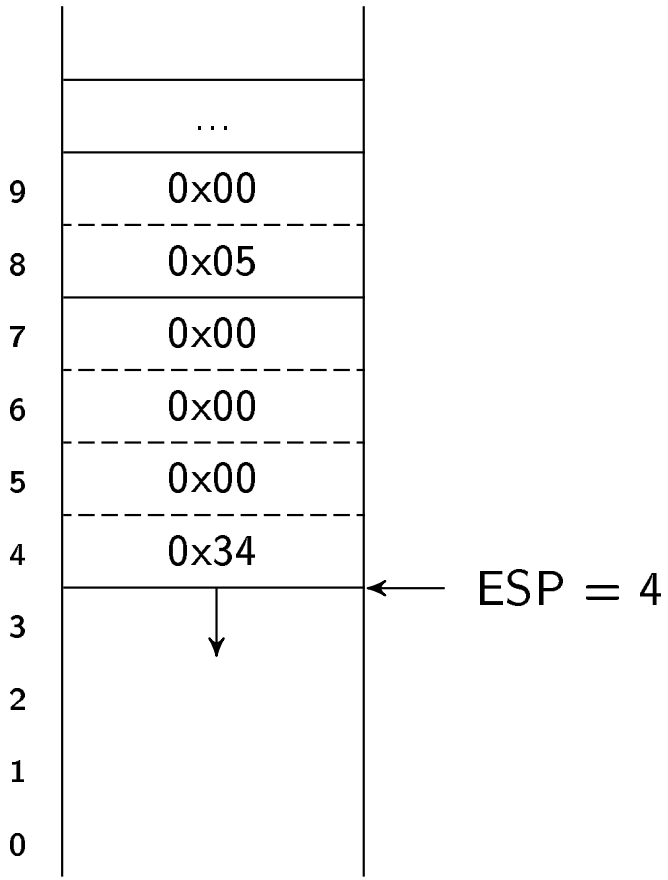
- empilement de mots (16bits) ou double-mots (32bits)
- c'est l'argument qui détermine la taille
- parfois nécessaire d'indiquer la taille



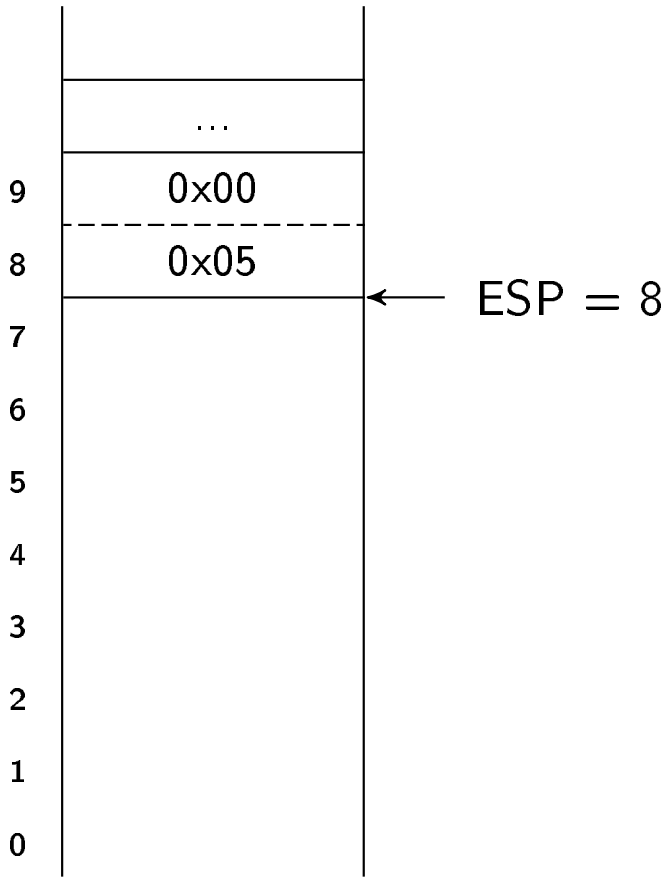

```
mov ax, 5
mov ebx, 0x34
push ax
push ebx
pop ecx
pop [edx]
```



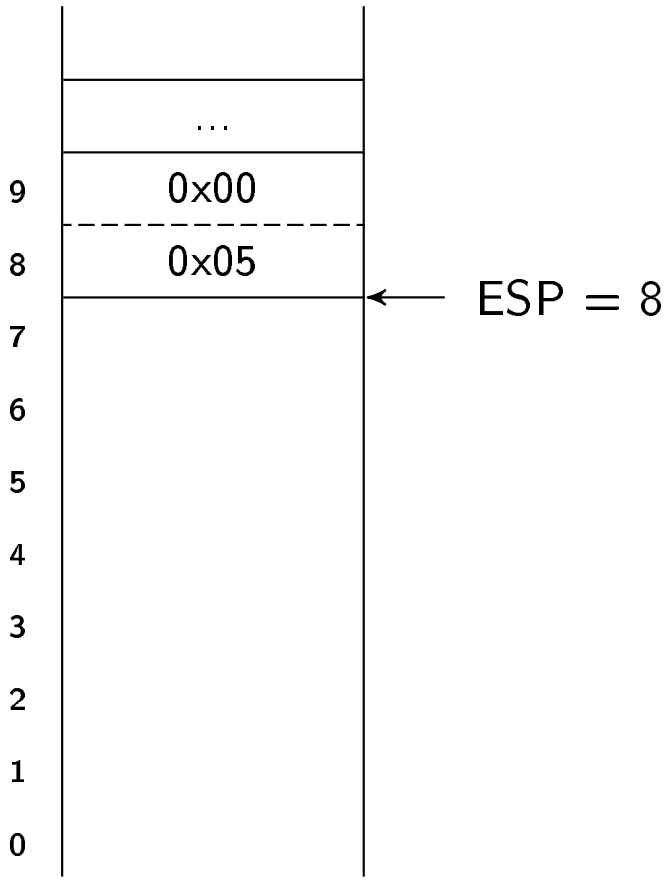

```
mov ax, 5
mov ebx, 0x34
push ax
push ebx
pop ecx
pop [edx]
```



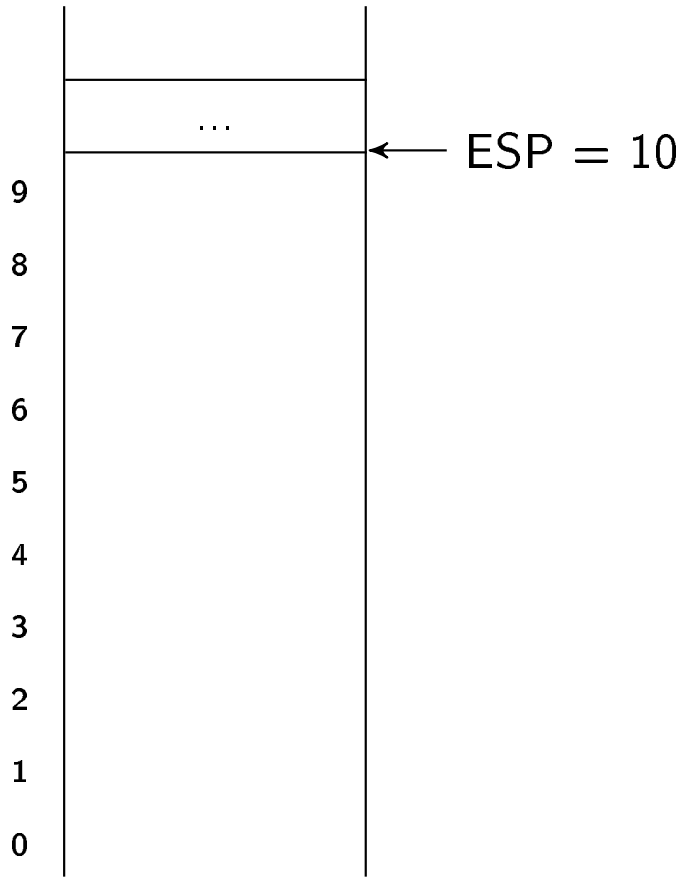

```
mov ax, 5
mov ebx, 0x34
push ax
push ebx
pop ecx
pop [edx]
```




```
mov ax, 5
mov ebx, 0x34
push ax
push ebx
pop ecx    ; ecx ← 0x34
pop [edx]
```




```
mov ax, 5
mov ebx, 0x34
push ax
push ebx
pop ecx    ; ecx ← 0x34
pop [edx]
```




```
mov ax, 5
mov ebx, 0x34
push ax
push ebx
pop ecx    ; ecx ← 0x34
pop word [edx]; [edx] ← 0x05
```

Une procédure

- est une suite d'instructions
- possède un **début** et une **fin**
  - début signalé par un *label*
  - fin signalée par l'instruction **ret**

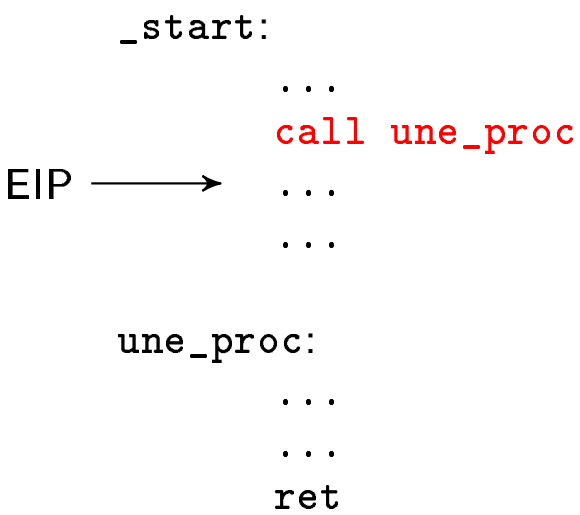
Appel d'une procédure avec l'instruction **call** <*label*>

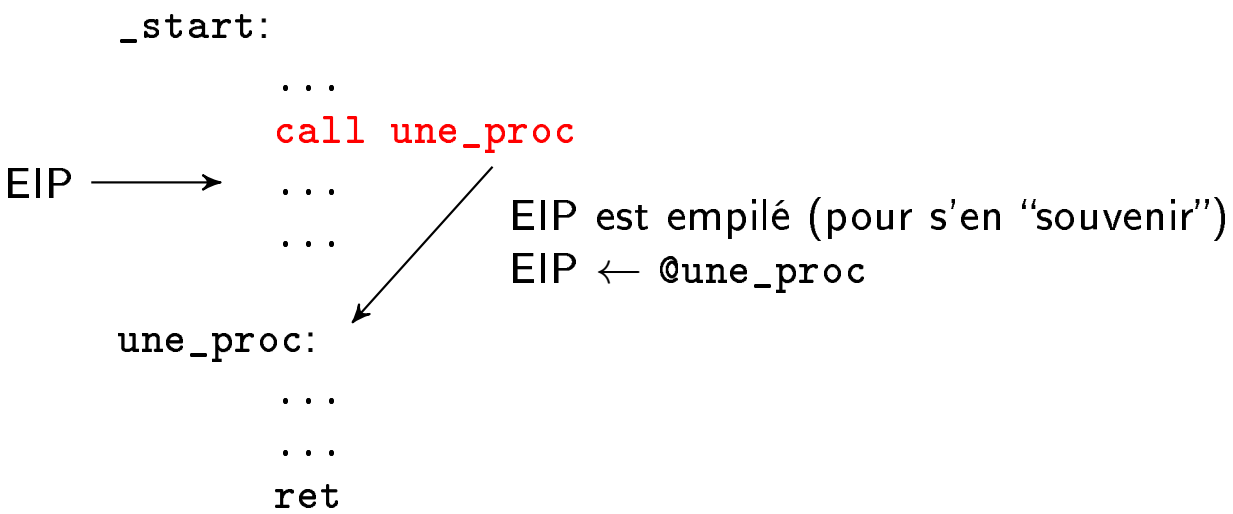
Attention :

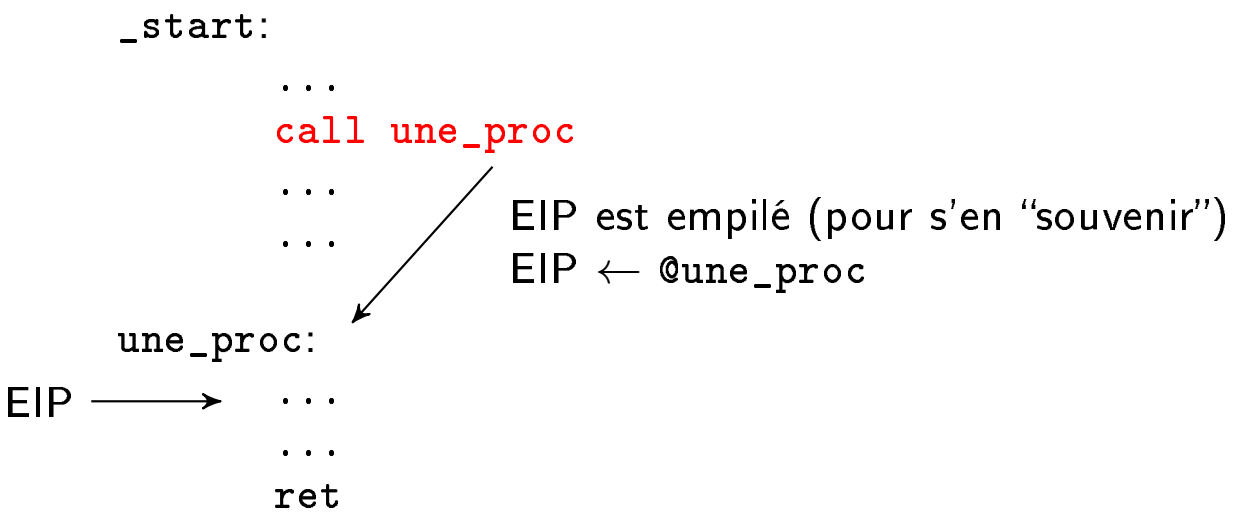
- il faut que la pile soit dans le même état à la fin de la procédure qu'avant son appel
- EIP doit être en sommet de pile (instruction **ret**)

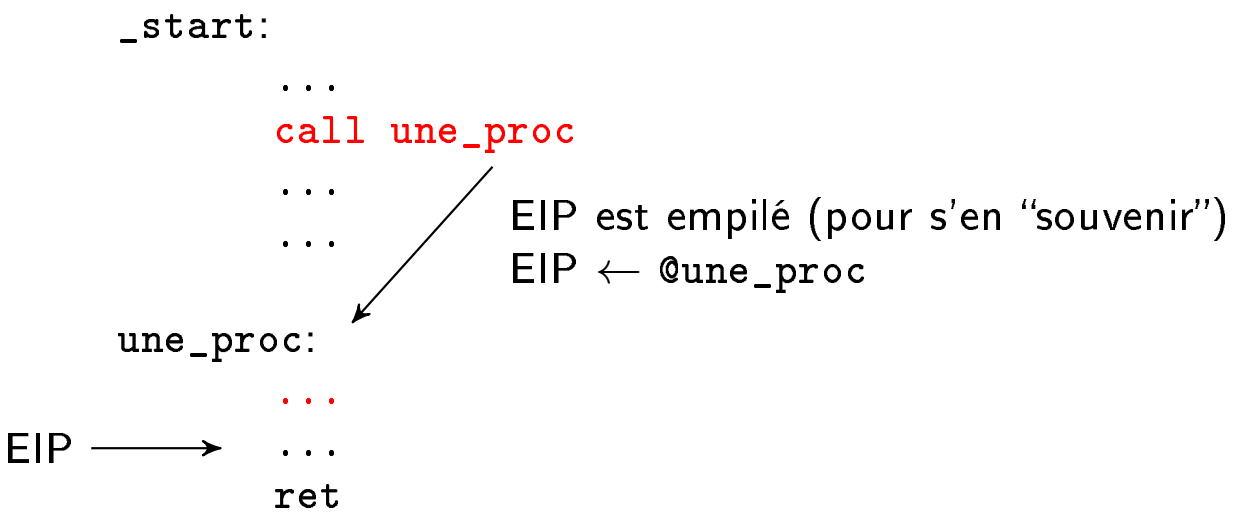
Exemple

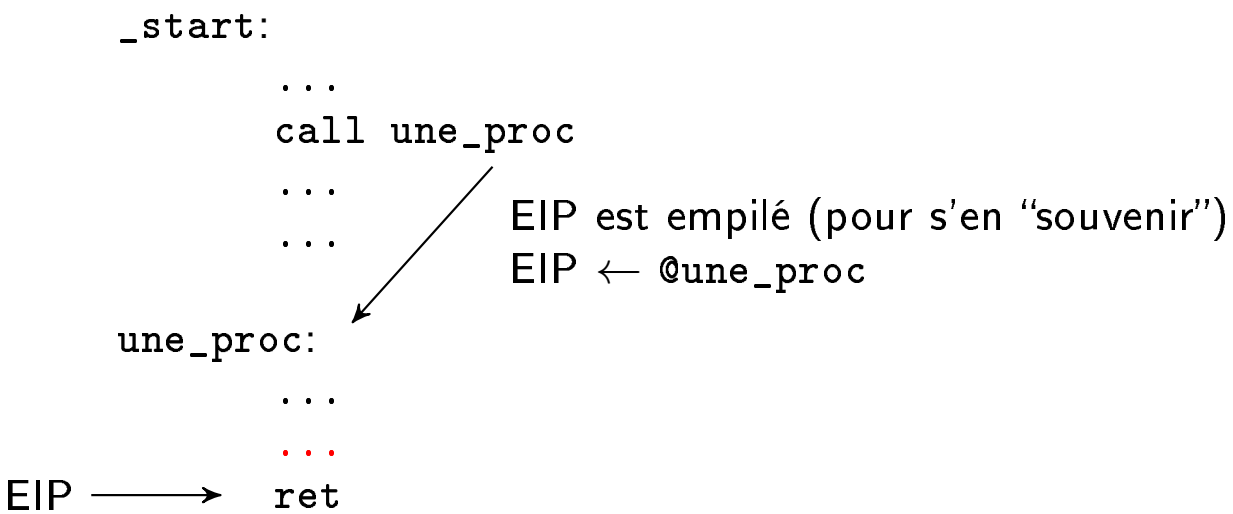
```
_start:
        ...
        call une_proc
        ...
        ...

une_proc:
        ...
        ...
        ret
```

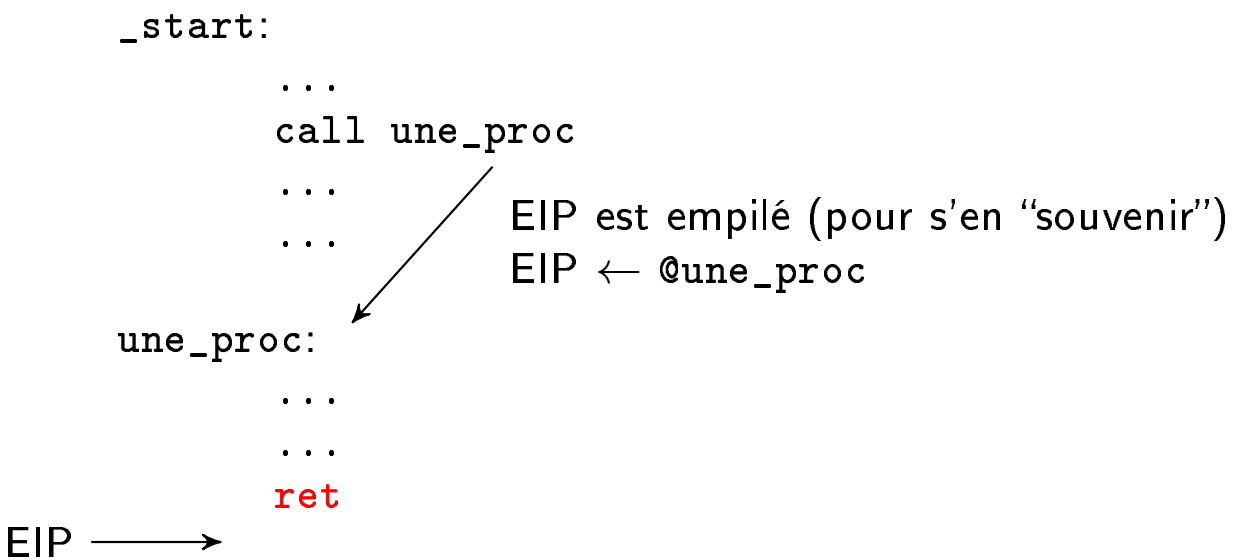

_start:
...
call une_proc
EIP
...
...
une_proc:
...
...
ret

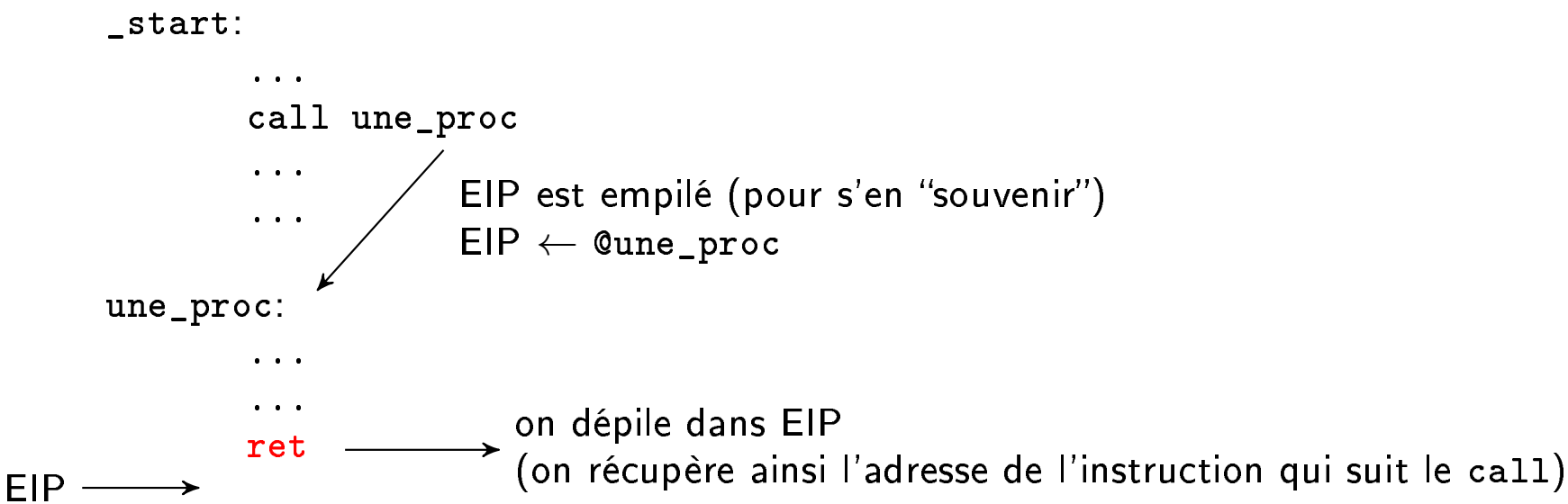

_start:
...
call une_proc
EIP
...
...
EIP est empilé (pour s'en "souvenir")
EIP ← @une_proc
une_proc:
...
...
ret

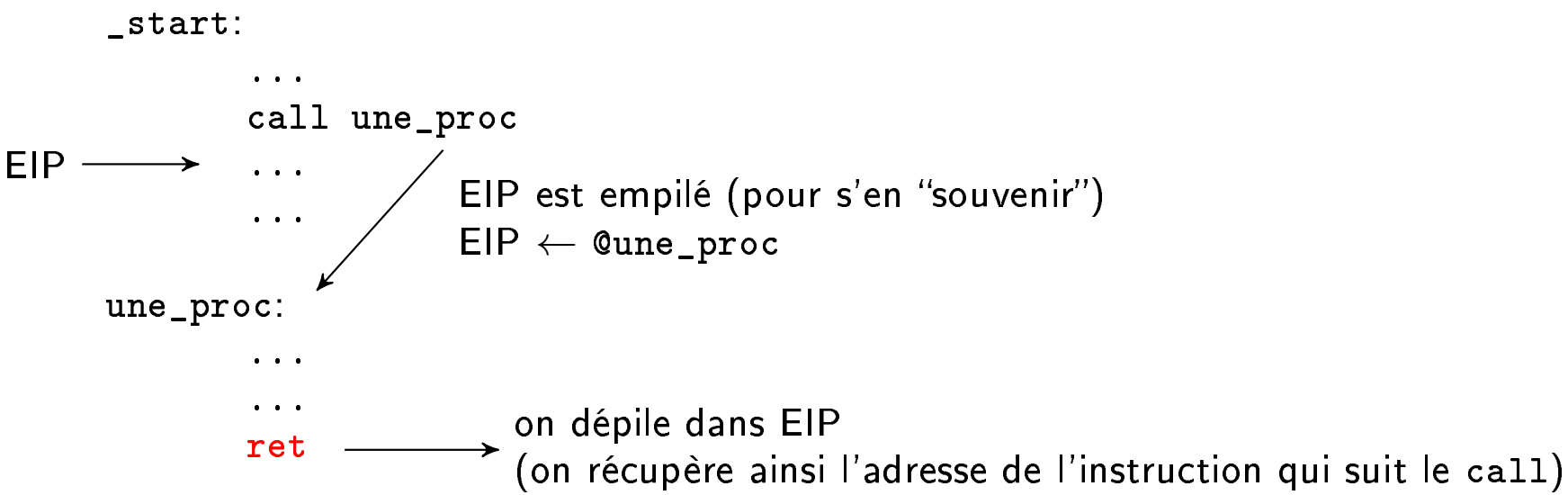

_start:
...
call une_proc
...
...
une_proc:
EIP
...
...
ret
EIP est empilé (pour s'en "souvenir")
EIP ← @une_proc


_start:
...
call une_proc
...
...
une_proc:
...
EIP
...
ret
EIP est empilé (pour s'en "souvenir")
EIP ← @une_proc


_start:
...
call une_proc
...
...
une_proc:
...
...
ret
EIP
EIP est empilé (pour s'en "souvenir")
EIP ← @une_proc


_start:
...
call une_proc
...
...
une_proc:
...
...
ret
EIP
EIP est empilé (pour s'en "souvenir")
EIP ← @une_proc


_start:
...
call une_proc
...
...
une_proc:
...
...
ret
EIP
EIP est empilé (pour s'en "souvenir")
EIP ← @une_proc
on dépile dans EIP
(on récupère ainsi l'adresse de l'instruction qui suit le call)


_start:
...
call une_proc
EIP
...
...
EIP est empilé (pour s'en "souvenir")
EIP ← @une_proc
une_proc:
...
...
ret
on dépile dans EIP
(on récupère ainsi l'adresse de l'instruction qui suit le call)

Une procédure est identifiée uniquement par un *label*

$\rightarrow$ **Comment passer des paramètres ?**

Une procédure est identifiée uniquement par un *label*

$\rightarrow$ **Comment passer des paramètres ?**

Solutions :

- grâce à des variables (globales!) $\rightarrow$ NON !

Une procédure est identifiée uniquement par un *label*

$\rightarrow$ **Comment passer des paramètres ?**

Solutions :

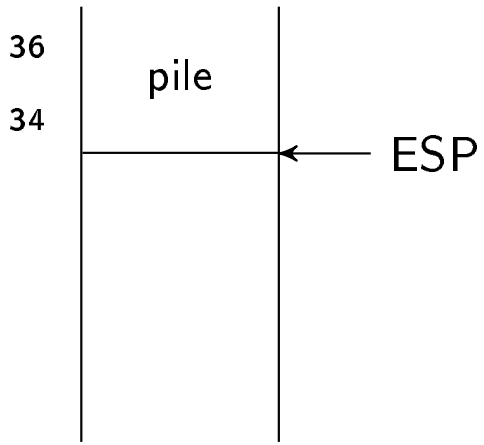
- grâce à des variables (globales!) $\rightarrow$ NON !
- grâce aux registres $\rightarrow$ très limité

Une procédure est identifiée uniquement par un *label*

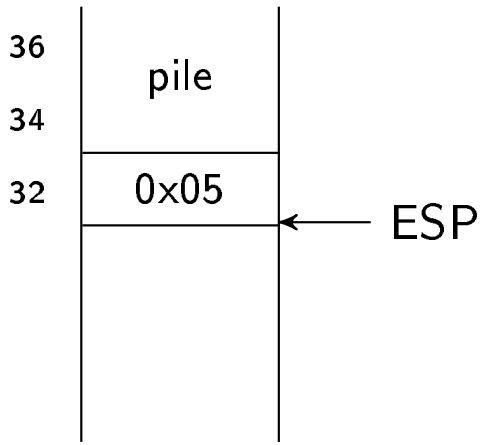
$\rightarrow$ **Comment passer des paramètres ?**

Solutions :

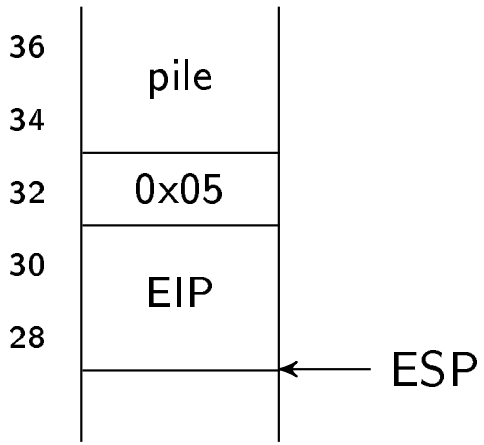
- grâce à des variables (globales!) $\rightarrow$ NON !
- grâce aux registres $\rightarrow$ très limité
- grâce à la pile $\rightarrow$ pas de limitation (sauf la taille du segment)
  1. on empile les paramètres
  2. on appelle la procédure



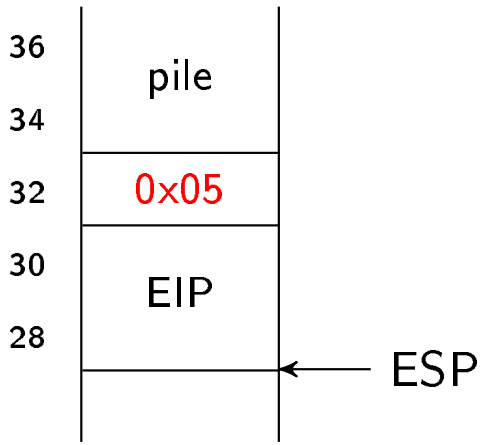

```
_start:
    ...
    push word 5
    call maproc
    ...
maproc :
    ...
    ret
```




```
_start:
    ...
    push word 5
    call maproc
    ...
maproc :
    ...
    ret
```




```
_start:
    ...
    push word 5
    call maproc
    ...
maproc :
    ...
    ret
```



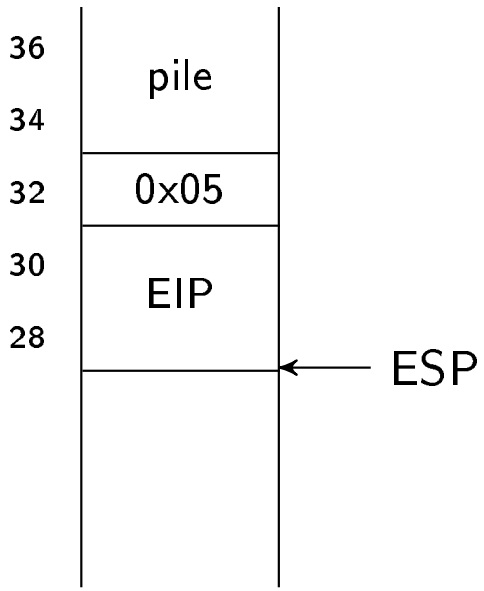

```
_start:
    ...
    push word 5
    call maproc
    ...
maproc :
    ...
    ret
```

Dans la procédure `maproc`,
comment récupérer le paramètre 5 ?

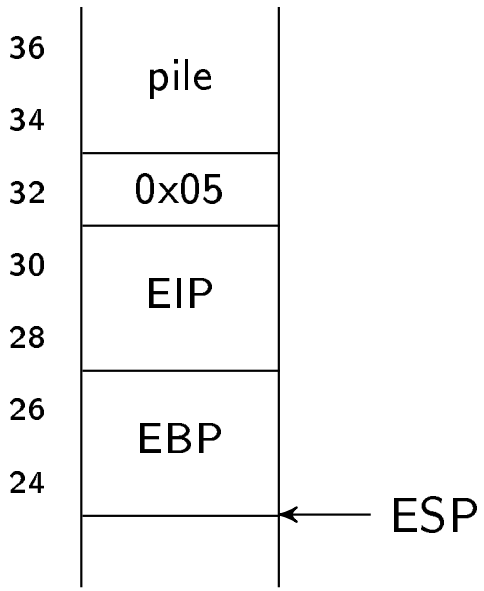
- On ne peut lire les paramètres qu'au moyen du registre **ebp** (pointeur de base de la pile)

Solution:

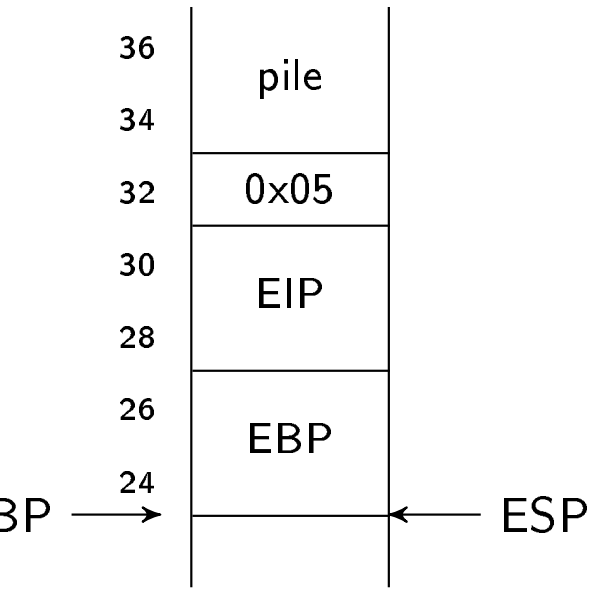
$\rightarrow$ On initialise **ebp** avec la valeur de **esp**
(en ayant pris soin de sauvegarder *avant* la valeur de **ebp**)



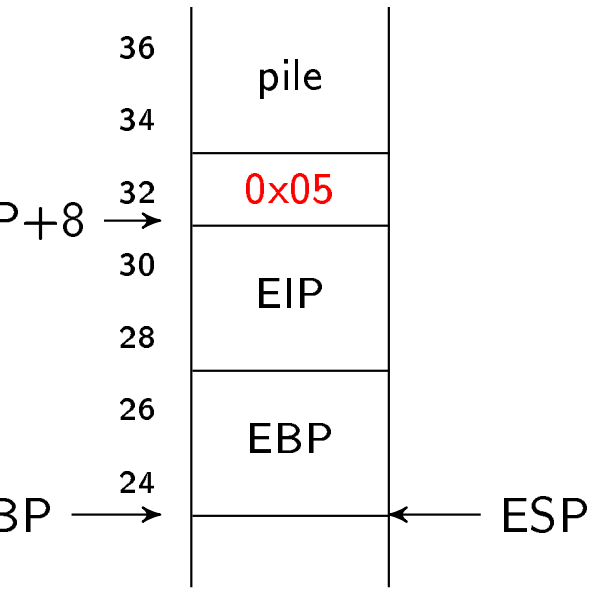

```
_start:
    ...
    push word 5
    call maproc
    ...
maproc :
    push ebp    ; on mémorise ebp
    mov ebp, esp
    mov ax, [ebp+8]
    ...

    ret
```



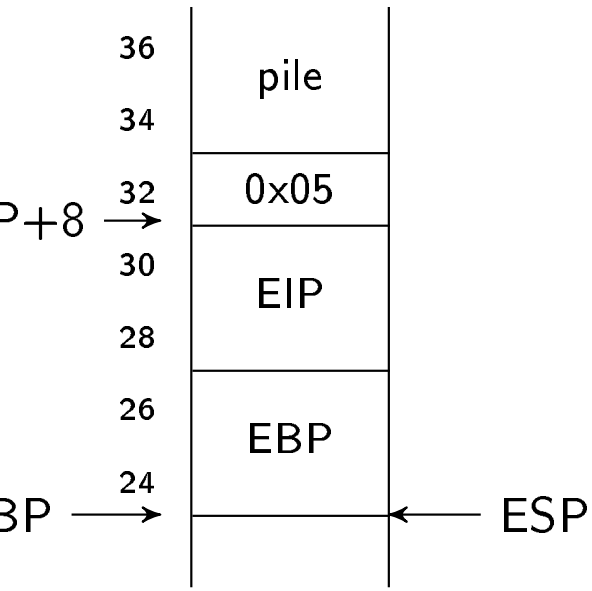

```
_start:
   ...
   push word 5
   call maproc
   ...
maproc :
   push ebp    ; on mémorise ebp
   mov ebp, esp
   mov ax, [ebp+8]
   ...

   ret
```



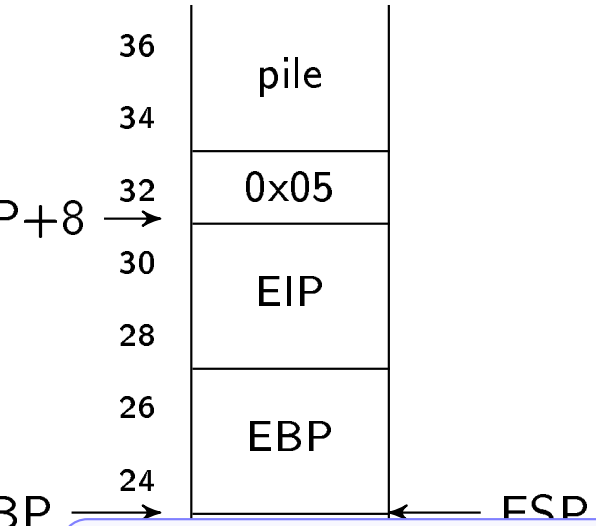

```
_start:
    ...
    push word 5
    call maproc
    ...
maproc :
    push ebp    ; on mémorise ebp
    mov ebp, esp
    mov ax, [ebp+8]
    ...

    ret
```




```
_start:
   ...
   push word 5
   call maproc
   ...
maproc :
   push ebp    ; on mémorise ebp
   mov ebp, esp
   mov ax, [ebp+8]
   ...

   ret
```



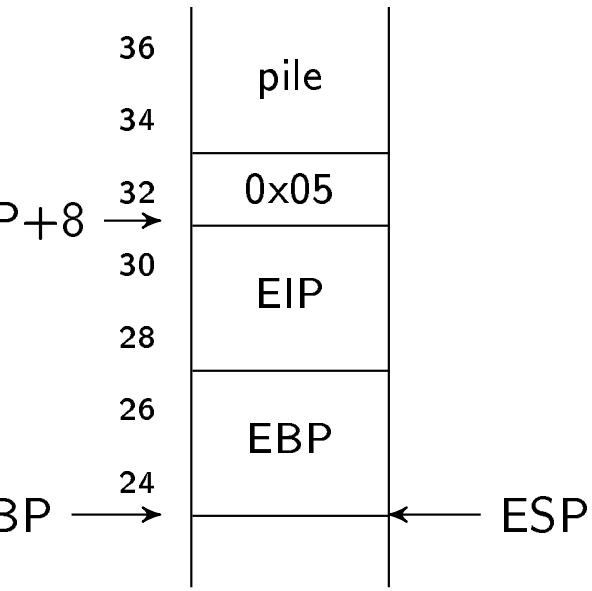

```
_start:
   ...
   push word 5
   call maproc
   ...
maproc :
   push ebp    ; on mémorise ebp
   mov ebp, esp
   mov ax, [ebp+8]
   ...

   ret
```



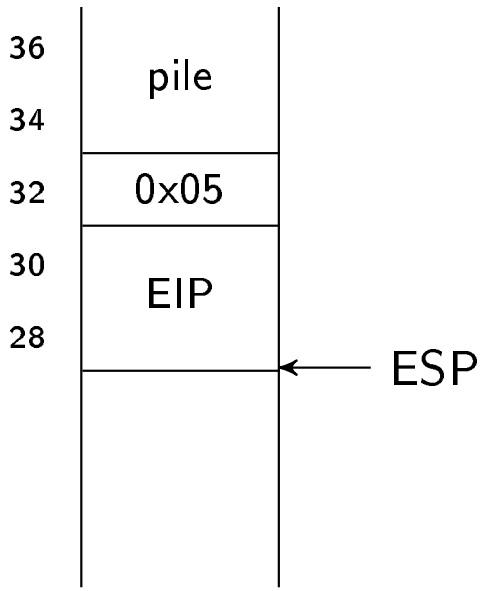

```
_start:
    ...
    push word 5
    call maproc
    ...
maproc :
    push ebp    ; on mémorise ebp
    mov ebp, esp
```

Aïe !

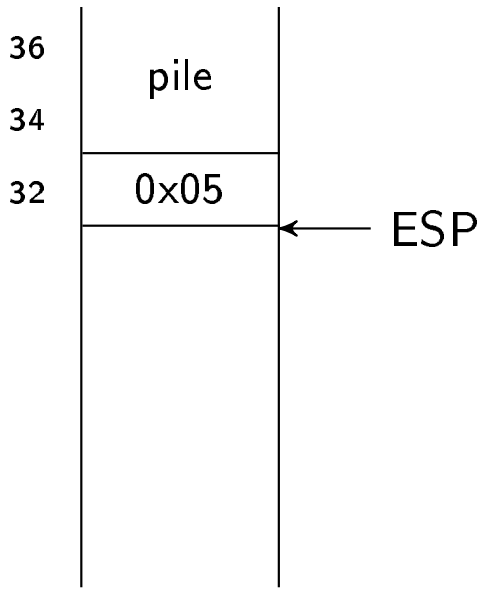
- **L'instruction *ret* s'attend à trouver EIP en sommet de pile !**
- **Une procédure doit rendre la pile et les registres dans l'état où elle les a trouvé**




```
_start:
   ...
   push word 5
   call maproc
   ...
maproc :
   push ebp    ; on mémorise ebp
   mov ebp, esp
   mov ax, [ebp+8]
   ...
   pop ebp
   ret
```




```
_start:
    ...
    push word 5
    call maproc
    ...
maproc :
    push ebp    ; on mémorise ebp
    mov ebp, esp
    mov ax, [ebp+8]
    ...
    pop ebp
    ret
```




```
_start:
    ...
    push word 5
    call maproc
    ...
maproc :
    push ebp    ; on mémorise ebp
    mov ebp, esp
    mov ax, [ebp+8]
    ...
    pop ebp
    ret
```

Plan

- Déclaration :
  - `texte1 db 'ceci est un texte', 0`
  - Tableau d'octets
  - Equivaut à :
    `texte1 db 'c','e','c','i',' ','e','s','t',' ','u','n',' ','t','e','x','t','e', 0`

- Instructions spécialisées pour la manipulation de chaînes (plus généralement les tableaux)

Les instructions de tableaux/chaînes :

- Utilisent des registres d'index dédiés
  - **esi** (*Source Index*) : adresse source
  - **edi** (*Destination Index*) : adresse destination
- Effectuent une opération impliquant généralement
  - **eax** (ou **ax** ou **al**) : registre qui contient la donnée
- Puis incrémentent ou décrémentent les registres d'index (traitement "caractère par caractère")

- Sens de parcours de la chaîne/du tableau :
  - défini par le flag DF (*Direction Flag*)
- Instructions pour spécifier le sens :
  - **cld** (*Clear Direction flag*) : incrémentation **esi**/**edi**
  - **std** (*Set Direction flag*) : décrémentation **esi**/**edi**

Ne pas oublier de commencer par **cld** ou **std** !

Lire en mémoire :

- **lodsb**
  - Remplit le registre **al** avec l'octet pointé par **esi**
  - Incrémente/décrémente **esi**
- **lodsw** : remplit **ax**, **esi**± 2
- **lodsd** : remplit **eax**, **esi**± 4
- b=byte (1 octet), w=word (2 octets), d=double-word (4 octets)

Ecrire en mémoire :

- **stosb**
  - Le contenu de **al** est copié dans la case pointée par **edi**
  - Incrémente/décrémente **edi**
- **stosw** : contenu de **ax**, **edi**± 2
- **stosd** : contenu de **eax**, **edi**± 4

Transfert d'un emplacement mémoire à un autre :

- **movsb**
  - l'octet de la zone mémoire pointée par **esi** est copié dans la zone mémoire pointée par **edi**
  - Incrémente/décrémente **esi** et **edi**

- **movsw** : transfert de 2 octets, **esi** et **edi** ± 2
- **movsd** : transfert de 4 octets, **esi** et **edi** ± 4

Le préfixe d'instruction **rep** :

- Pour répéter une instruction de chaîne un certain nombre de fois
- Le nombre d'itérations est fixé par **ecx**

Exemple:

```
mov ecx, 5
rep movsb    ; répète 5 fois l'instruction movsb
```

Pour récupérer la longueur d'une chaîne :

- au moment de la déclaration :

```
ch1 db 'un texte', 0
long_ch1 equ $ - ch1
```

Comparaison de chaînes de caractères :

- **cmpsb**
  - compare l'octet en **esi** avec l'octet en **edi**
  - affecte le flag ZF
    - ZF = 1 si [esi] = [edi]
    - ZF = 0 sinon
  - Incrémente/décrémente **esi** et **edi**
- **cmpsw** : compare 2 octets, **esi** et **edi** ± 2
- **cmpsd** : compare 4 octets, **esi** et **edi** ± 4

Recherche d'un caractère dans une chaîne :

- **scasb**
  - compare le contenu du registre **al** avec l'octet en **edi**
  - affecte le flag ZF
    - ZF = 1 si al = [edi]
    - ZF = 0 sinon
  - Incrémente/décrémente **edi**

- **scasw** : recherche **ax**, **edi** ± 2
- **scasd** : recherche **eax**, **edi** ± 4

Variantes du préfixe **rep**:

- **repe**
  Répète l'instruction qui suit tant que ZF est allumé, ou au maximum **ecx** fois
- **repne**
  Répète l'instruction qui suit tant que ZF est éteint, ou au maximum **ecx** fois

Plan

- BIOS (*Basic Input Output System*) : partie de l'OS du plus bas niveau
  - opérations très primitives : lecture clavier, affichage écran en mode texte, accès simplifié lecteurs et disque dur, accès au port série et parallèle
- Le programme du BIOS se trouve en mémoire morte (ROM)
- Chaque modèle de PC est vendu avec une version du BIOS adaptée à sa configuration matérielle

- Le BIOS $\simeq$ librairie de fonctions
  - Chaque fonction exécute une tâche bien précise
    (par exemple, afficher un caractère donné sur l'écran)
- L'appel de l'une de ces fonctions = **appel système** (*syscall*)
- Fonctions du BIOS $\neq$ procédures appelées avec **call**
  - Le code du BIOS en ROM est non modifiable
  - Il a été prévu de pouvoir modifier le comportement du BIOS en cours d'utilisation (ex: gestion d'un nouveau périphérique)
  - Le BIOS étant différent d'un ordinateur à l'autre, les adresses des fonctions changent...

- Le BIOS $\simeq$ librairie de fonctions
  - Chaque fonction exécute une tâche bien précise
    (par exemple, afficher un caractère donné sur l'écran)
- L'appel de l'une de ces fonctions = **appel système** (*syscall*)
- Fonctions du BIOS $\neq$ procédures appelées avec **call**
  - Le code du BIOS en ROM est non modifiable
  - Il a été prévu de pouvoir modifier le comportement du BIOS en cours d'utilisation (ex: gestion d'un nouveau périphérique)
  - Le BIOS étant différent d'un ordinateur à l'autre, les adresses des fonctions changent...

Comment ça marche ?

Solution :

- Utilisation d'une table d'indirection : la table des **vecteurs d'interruptions**
  - table placée en RAM
  - contient les adresses (en ROM ou RAM) des fonctions du BIOS
  - implantée à partir de l'adresse 0x00000000 (première case mémoire)
  - initialisée par le BIOS lui-même au moment du démarrage

0x00000008	...
0x00000004	adresse de la deuxième fonction du BIOS
0x00000000	adresse de la première fonction du BIOS

- Chaque élément de la table occupe 4 octets (adresse 32 bits)
  - table à 256 éléments ($\rightarrow$ 1Ko)
  - exemple : si l'on sait que la fonction du BIOS qui affiche un caractère est la $33^{\text{ème}}$, on va l'appeler en lisant la $33^{\text{ème}}$ ligne de la table, puis en allant exécuter les instructions à l'adresse trouvée

- Chaque élément de la table occupe 4 octets (adresse 32 bits)
  - table à 256 éléments ($\rightarrow$ 1Ko)
  - exemple : si l'on sait que la fonction du BIOS qui affiche un caractère est la $33^{\text{ème}}$, on va l'appeler en lisant la $33^{\text{ème}}$ ligne de la table, puis en allant exécuter les instructions à l'adresse trouvée

...

0x00000084 0xF123A508 (car $4 \times 33 = 0x84$)

...

- Chaque élément de la table occupe 4 octets (adresse 32 bits)
  - table à 256 éléments ($\rightarrow$ 1Ko)
  - exemple : si l'on sait que la fonction du BIOS qui affiche un caractère est la $33^{\text{ème}}$, on va l'appeler en lisant la $33^{\text{ème}}$ ligne de la table, puis en allant exécuter les instructions à l'adresse trouvée

...

0x00000084 0xF123A508 (car $4 \times 33 = 0x84$)

...

- La table des vecteurs d'interruptions contient des valeurs différentes pour chaque version de BIOS
  - peut être modifiée pour pointer sur du code en mémoire principale, modifiant ainsi le BIOS existant

- L'Unité de Traitement exécute séquentiellement les instructions ou effectue des sauts programmés (call, jmp)
- Il existe des situations où l'UT est "déroutée" de son fonctionnement (= **interruptions**)
  - *reset* : signal envoyé au processeur pour un redémarrage
  - Interruptions physiques (externes) : appels d'autres périphériques
  - Exceptions (interruptions internes) : débordement de pile, dépassement de capacité mémoire, division par zéro, etc.
  - Appels systèmes (interruptions logicielles) : appels du programme lui-même (`int 0x80`, etc.)

- L'Unité de Traitement exécute séquentiellement les instructions ou effectue des sauts programmés (call, jmp)
- Il existe des situations où l'UT est "déroutée" de son fonctionnement (= **interruptions**)
  - *reset* : signal envoyé au processeur pour un redémarrage
  - Interruptions physiques (externes) : appels d'autres périphériques
  - Exceptions (interruptions internes) : débordement de pile, dépassement de capacité mémoire, division par zéro, etc.
  - Appels systèmes (interruptions logicielles) : appels du programme lui-même (`int 0x80`, etc.)

int *n*

- Permet d'appeler la $n^{\text{ième}}$ fonction de la table des vecteurs d'interruptions
  - *n* est compris entre 0 et 255
- Similaire à l'instruction **call**
  - sauf que l'adresse de destination est donnée par la table des vecteurs d'interruptions
  - les indicateurs sont automatiquement sauvegardés sur la pile
  - l'adresse de retour *complète* est empilée (car le code de la fonction n'est pas nécessairement dans le même segment de code que le programme)

- Le système Linux repose sur le BIOS :

  il appelle des fonctions du BIOS pour interagir avec le matériel
- Les fonctions de Linux s'utilisent comme celles du BIOS, via des vecteurs d'interruptions
  - fonctions de plus haut niveau que celles du BIOS (e/s, ouvertures fichiers, etc.)
- Les fonctions de Linux s'appellent à l'aide du vecteur 0x80
  - La valeur du registre **eax** permet d'indiquer quelle est la fonction Linux appelée :

```
mov eax, n    ; n = numéro de la fonction Linux
int 0x80
```

Fonction n°1 : exit (fin de programme)

eax : 1

ebx : code de sortie

Fonction n°4 : write (ex: afficher à l'écran)

- **eax** : 4
- **ebx** : descripteur de fichier (flux de sortie standard : 1)
- **ecx** : adresse du buffer à écrire
- **edx** : nombre d'octets à écrire

Fonction n°3 : read (ex: saisie au clavier)

eax : 3

ebx : descripteur de fichier (flux d'entrée standard : 0)

ecx : adresse du buffer où les caractères seront stockés

edx : nombre d'octets à lire

Fonction n°5 : open

eax : 5

ebx : adresse du tableau contenant le nom du fichier à ouvrir

ecx : bits d'accès (ex: 02 read-write)

edx : bits de permissions (00200 : le créateur est le propriétaire)

En retour, **eax** = descripteur de fichier

Plan

Plan

Au delà des instructions classiques, Intel/AMD/... ont introduits des instructions spécialisées

MMX (3DNow ! chez AMD)

- MMX (`MultiMedia eXtension` ou `Matrix Math eXtension`) est un jeu d'instructions pour le calcul matriciel (addition de 4 entiers sur 16 bits en une seule instruction par ex) ;
- en 1996 avec le Pentium ;
- ajout de 8 registres sur 64 bits (MM0 à MM7) ;
- 57 instructions de calcul

SSE/SSE2/SSE3/SSE4

- SSE (Streaming SIMD Extensions)
- en 1999 avec le Pentium 3 ;
- 8 nouveaux registres sur 128 bits (XMM0 à XMM7)
- 70 instructions de calcul, de déplacement, de conversion, ...

Plan

Le cache contient une copie des données lorsqu'elles sont coûteuses (en termes de temps d'accès)

L'utilisation des données peut être réalisée en accédant à la copie **en cache** plutôt qu'en récupérant les données.

Fonctionnement

- 1. le processeur demande une information (une lecture, par exemple) ;
- 2. le cache vérifie s'il possède cette information :
  - s'il la possède, il la retransmet au processeur ;
  - s'il ne la possède pas, il la demande à la mémoire principale
- 3. le processeur traite la demande et renvoie la réponse au cache ;
- 4. le cache la stocke pour utilisation ultérieure et la retransmet au processeur au besoin.

Pourquoi ça marche ?

Principes

- principe de localité spatiale : l'accès à une donnée située à une adresse X va probablement être suivi d'un accès à une zone tout proche de X ;
- principe de localité temporelle : l'accès à une zone mémoire à un instant donné a de fortes chances de se reproduire dans la suite immédiate du programme.

Plan

Comment est exécutée une instruction ?

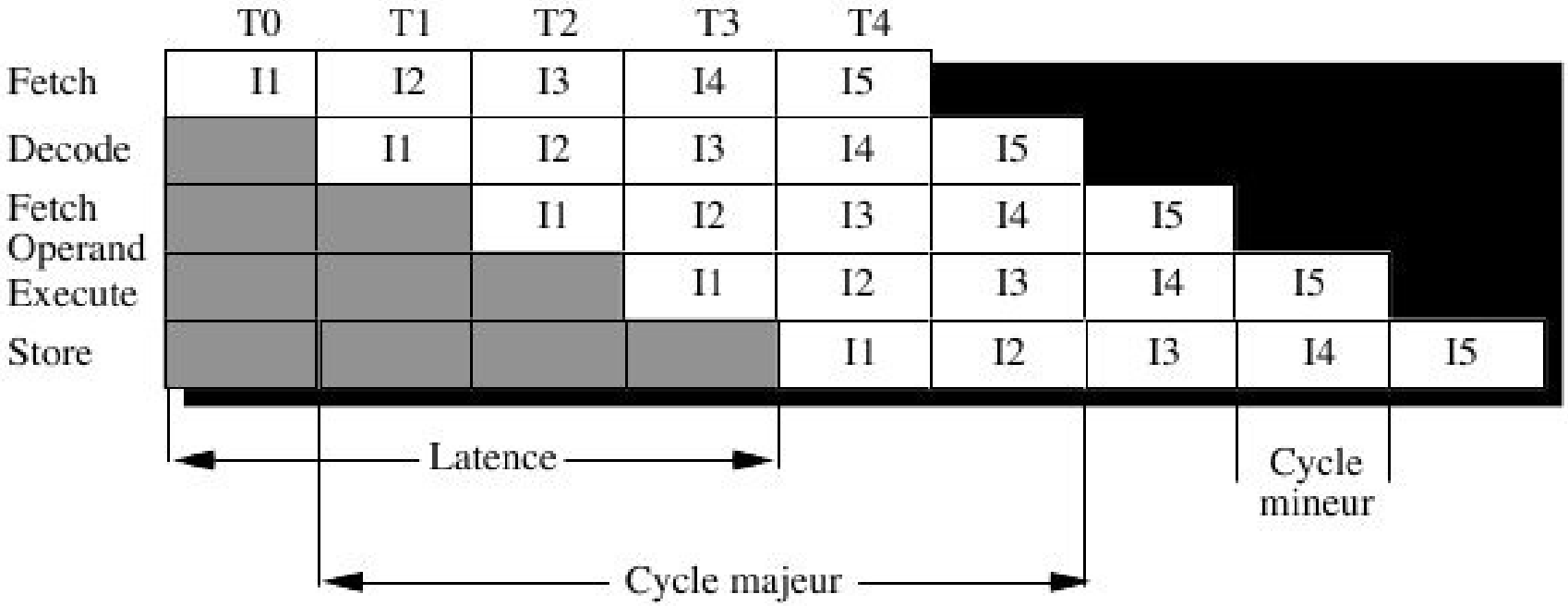
- 1. Lire une instruction en mémoire (`fetch`).
- 2. Comprendre l'instruction (`decode`).
- 3. Sélectionner l'opération à effectuer dans "unité de traitement".
- 4. Charger les données si nécessaire (`fetch operands`).
- 5. Effectuer le calcul (`execute`).
- 6. Ranger le résultat du calcul (`store`).
- 7. Aller à 1.

- l'exécution d'une instruction est décomposée par l'unité de commande en plusieurs phases "disjointes" ;
- chacune de ces phases sont prises en charge par un module ;
- ces modules travaillent en parallèle avec les autres.

On commence ainsi le traitement d'une nouvelle instruction avant que la précédente ne soit terminée

Recouvrement

On ne peut en effet jamais être sûr qu'une instruction est terminée avant que la suivante ne s'exécute.



- cycle mineur : temps nécessaire à une phase
- cycle majeur : temps total pour l'exécution d'une instruction

Longueur du pipeline

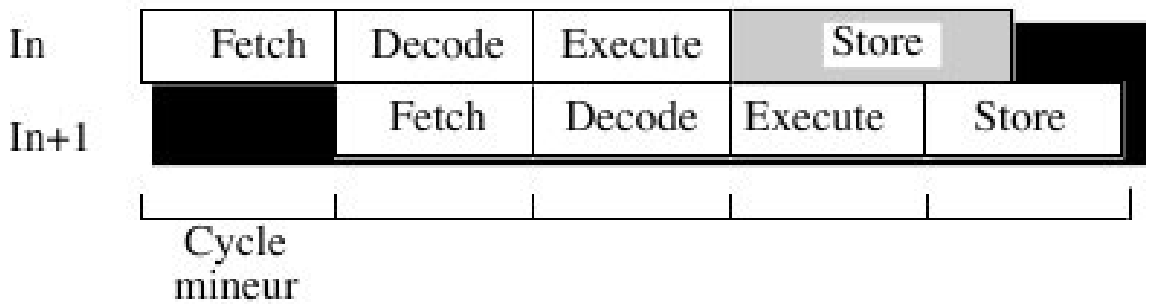
- Pentium : 5
- Pentium 3 : 10
- Pentium 4 (Northwood) : 20
- Pentium 4 (Prescott) : 31
- AMD Athlon : 10
- AMD Athlon 64 : 12

Les difficultés

- les accès concurrents à la mémoire (recherche instruction et recherche opérande simultanés),
- les branchements,
- les différentes durées passées dans chaque étage :
  - aspect structurel : certaines opérations sont plus longues que d'autres (une addition dure plus qu'un décalage),
  - aspect dynamique : un accès à la mémoire mov ax, [mem] plus long qu'un transfert entre registres mov ax,bx
- les problèmes de dépendance (ou de précédence) : accès simultanés aux mêmes opérandes par des instructions différentes,
- les interruptions.

La même instruction sur des opérandes différents

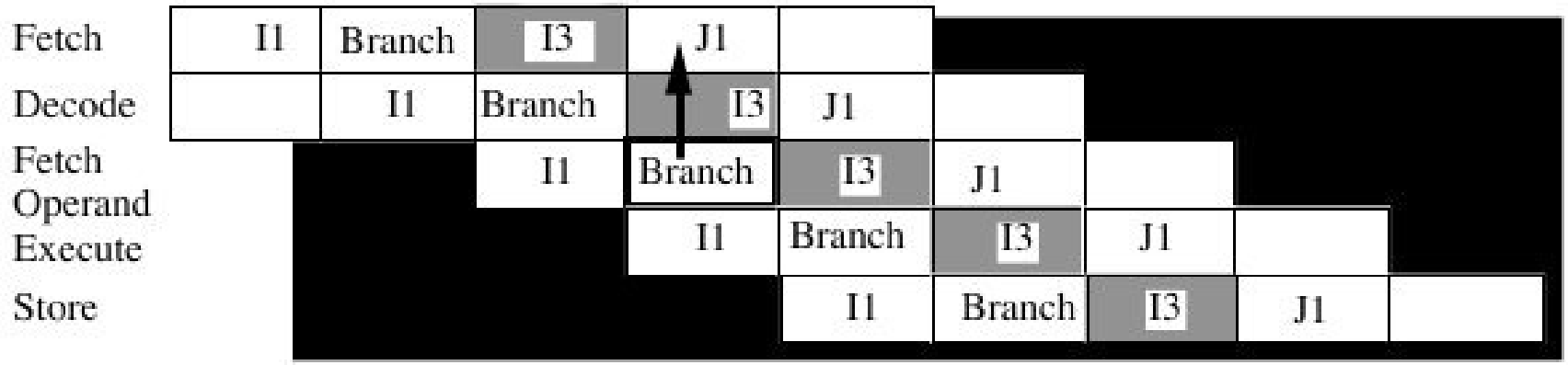
- I_N : MOV AX, [mem]
- I_{N+1} : MOV AX, BX



- problème : il s'agit ici du problème plus général des accès à la mémoire.
- solution : introduire des délais pour synchroniser les instructions du pipeline sur la plus lente (qui sont les accès mémoire)

Que faire dans le cas du déroutement vers l'adresse cible alors que l'unité fetch a déja été chercher l'instruction suivante ?

- I_1 : CMP AX, 0
- I_2 : JMP J1
- I_3 : ADD BX, CX
- ...
- I_n : J1: ADD AX, CX



- on appelle pénalité le nombre de cycles perdus si on suspend le pipeline jusqu'à obtention de l'instruction suivante.
- dans l'exemple, on doit annuler l'instruction I_3

Une solution : prédiction dynamique

Branch target buffer (BTB)

- utilisation d'un cache associatif utilisé par l'étage `fetch` :
  - pour les n dernières adresses d'instructions de branchement, l'adresse de l'instruction suivante qui a été choisie.
- lorsque le pipeline décode une instruction de branchement :
  - il vérifie si elle n'est pas dans le BTB,
  - si elle y est, alors, le module `fetch` va chercher l'instruction suivante à l'adresse associée,
  - sinon, le pipeline est suspendu pendant le cacul de l'instruction suivante, puis l'adresse de ce branchement et celle de l'instruction suivante effective sont rangées dans le BTB.

Le problème de dépendance : une dépendance entre deux instructions I_i et I_{i+k} apparaissent lorsque l'instruction I_{i+k} travaille sur les mêmes données que I_i.

Trois classes :

- RAW (Read After Write) : I_{i+k} utilise (lecture) un opérande mis à jour (écriture) par I_i et doit donc attendre la fin de I_i,
- WAR (Write After Read) : I_{i+k} modifie (écriture) un opérande qui est utilisé (lecture) par I_i,
- WAW (Write After Write) : I_{i+j} et I_i mettent à jour (écriture) une information commune dans un ordre qui doit être respecté.

Les solutions

- introduction de délais,
- solution logicielle : le compilateur se charge d'engendrer du code évitant le conflit.

Réarrangement du code : I = J + K + L

Première version

```
mov  ax ,  [ j ]
mov  bx ,  [ k ]
mov  cx ,  [ l ]
add  bx ,  cx
add  ax ,  bx
```

L'affectation du registre cx par le contenu de la variable l est réalisée trop tardivement !

Deuxième version

```
mov  cx ,  [ l ]
mov  bx ,  [ k ]
mov  ax ,  [ j ]
add  bx ,  cx
add  ax ,  bx
```

Plan

Une autre architecture de processeur : RISC

- de manière générale :
  - moins de 20 % instructions utilisées représentent plus de 80 % du temps
- simplication des processeurs :
  - instructions courantes seulement
  - plus simples, mieux optimisées, plus efficaces
  - puce moins grosse, moins gourmandes en énergie, ...
- processeurs à architecture RISC (Reduced Instruction Set Computer)

Conséquences

- instructions de taille fixe
- pipelines plus efficaces
- ... mais compilateur plus compliqué

CISC

- $>$ 200 instructions
- taille variable des instructions
- instructions : les simples sont câblées mais les complexes sont sous forme de micro-code (nécessité d'un automate de séquencement)
- accès mémoire pour toutes instructions
- vitesse typique : une instruction dure entre 3 et 10 cycles

RISC

- $<$ 100 instructions
- format fixe
- instructions câblées
- accès mémoire pour instructions de chargement
- vitesse typique : 1 instruction dure 1 cycle

CISC

- avantages :
  - développement plus simple
  - très répandu
- inconvénients :
  - puce plus grosse et plus compliquée
  - évolutions limitées
  - plusieurs cycles par instruction

RISC

- avantages :
  - puce plus petite
  - fréquence plus élevée
  - exécution plus rapide
- inconvénients :
  - compilateur complexe
  - programmes longs (nécessitent plus de mémoire)

Quelques exemples de processeurs

- ARM7TDMI, Samsung ARM1176JZF : les processeurs des iPod, iPhone, . . . d'Apple
- la famille des Marvell XScale : embarqués dans des PDAs
- l'architecture Power d'IBM et de Freescale : consoles de jeux WII, Xbox 360, . . .
- PowerPC
- SPARC