Assembleur Intel x86

Alexis Nasr
Carlos Ramisch
Manon Scholivet
Franck Dary

Compilation – L3 Informatique
Département Informatique et Interactions
Aix Marseille Université

Rappels d'architecture

Un ordinateur se compose principalement

- d'un processeur (CPU),
- de mémoire.

On y attache ensuite des périphériques, mais ils sont optionnels.

données	: disque dur, etc
entrée	utilisateur : clavier, souris
sortie	utilisateur : écran, imprimante
processeur	supplémentaire : GPU

Le processeur

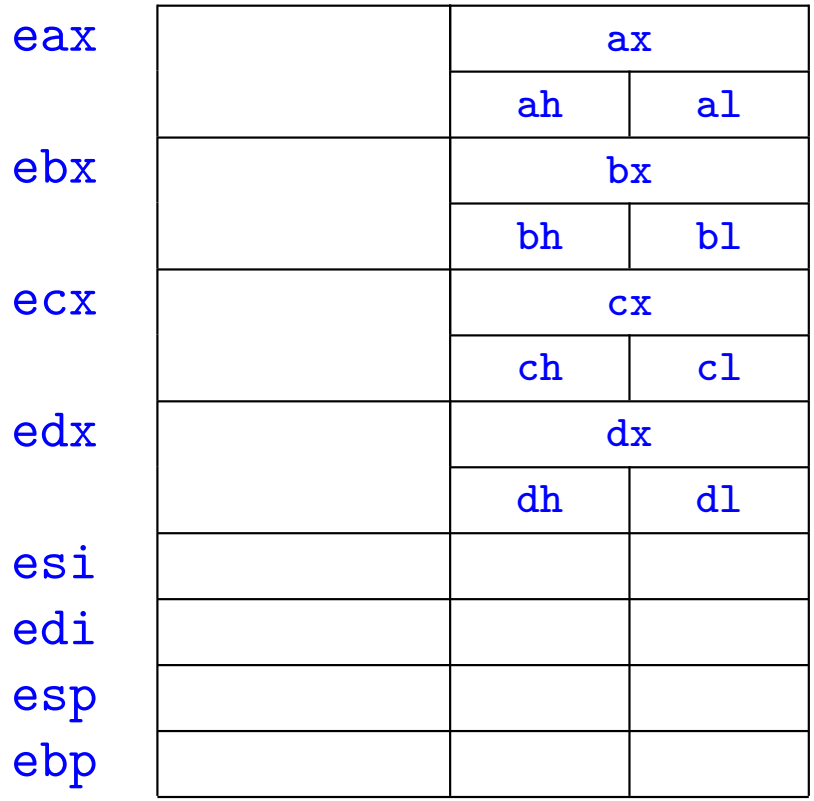
- Le processeur lit et écrit des informations en **mémoire**
- Il peut de plus effectuer des **opérations** arithmétiques et logiques
- Chaque action qu'il peut effectuer est appelée **instruction**
- Les instructions effectuées par le processeur sont stockées dans la mémoire.
- Il dispose d'un petit nombre d'emplacements mémoire d'accès plus rapide, les **registres**.
- Un registre spécial nommé `eip` (extended instruction pointer) contient à tout moment l'adresse de la prochaine instruction à exécuter
- De façon répétée le processeur :
  1. lit l'instruction stockée à l'adresse contenue dans `eip`
  2. l'interprète ce qui peut modifier certains registres (dont `eip`) et la mémoire

X86

- X86 est un jeu d'instruction commun à plusieurs processeurs
- Le nom X86 provient des processeurs Intel utilisant ce jeu d'instructions (8086, 80186, 80286, 80386, 80486)
- jeu d'instruction des processeurs équipant les ordinateurs personnels

Registres

- Les processeurs X86 (à partir du 386) ont 8 registres de 4 octets



- `eax` peut être découpé en trois registres : un registre de deux octets : `ax` composé de deux registres d'un octet : `ah` et `al`.
- `esp` (*extended stack pointer*) pointe sur le sommet de la pile
- `ebp` (*extended base pointer*) pointe sur l'adresse de base de l'espace local (trame de pile de la fonction active)

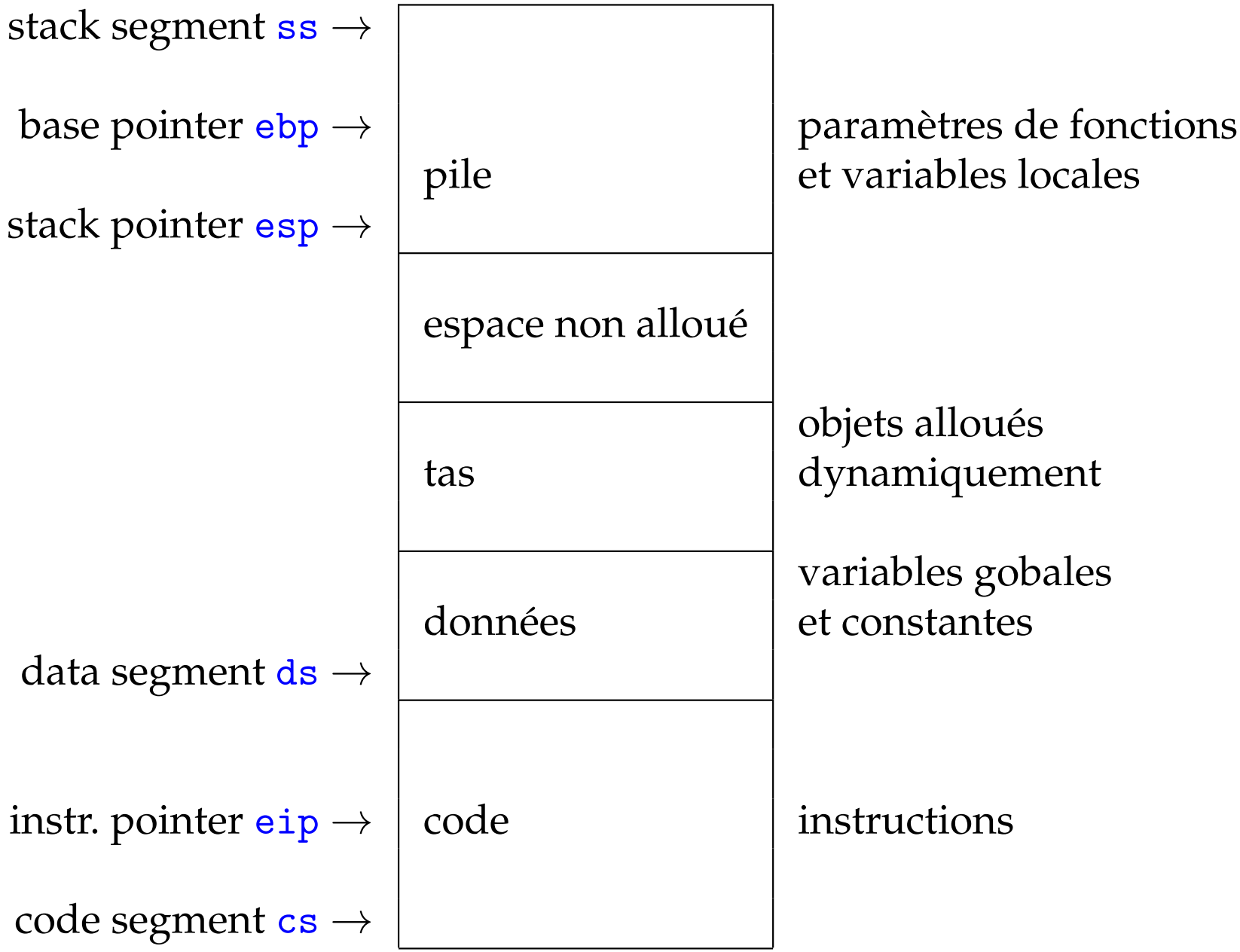
Segmentation de la mémoire

- La mémoire est divisée en **segments** indépendants.
- L'adresse de début de chaque segment est stockée dans un registre.
- Chaque segment contient un type particulier de données.
  - le **segment de données** permet de stocker les variables globales et les constantes. La taille de ce segment n'évolue pas au cours de l'exécution du programme (il est statique).
  - le **segment de code** permet de stocker les instructions qui composent le programme
  - la **pile** permet de stocker les variables locales, paramètres de fonctions et certains résultats intermédiaires de calcul
- L'organisation de la mémoire en segments est conventionnelle
- En théorie tous les segments sont accessibles de la même manière

Registres liés aux segments

- Segment de code
  - `cs` (Code Segment) adresse de début du segment de code
  - `eip` (Instruction Pointer) adresse relative de la prochaine instruction à effectuer
    `cs` + `eip` est l'adresse absolue de la prochaine instruction à effectuer
- Segment de données
  - `ds` (Data Segment) adresse de début du segment de données
- Pile
  - `ss` (Stack Segment) adresse de la base de la pile
  - `esp` (Stack Pointer) adresse relative du sommet de pile
    `ss` + `esp` est l'adresse absolue du sommet de pile
  - `ebp` (Base Pointer) registre utilisé pour le calcul d'adresses de variables locales et de paramètres

Segmentation de la mémoire



Flags

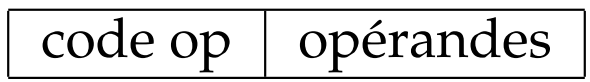
- Les flags sont des variables booléennes (stockées sur un bit) qui donnent des informations sur le déroulement d'une opération et sur l'état du processeur.
- 32 flags sont définis, ils sont stockés dans le registre `eflags`, appelé registre d'état.
- Valeur de quelques flags après une opération :
  - `CF` : Carry Flag.
    Indique une retenue (`CF`=VRAI) sur les entiers non signés.
  - `PF` : Parity Flag.
    Indique que le résultat est pair (`PF`=VRAI) ou impair (`PF`=FAUX).
  - `ZF` : Zero Flag.
    Indique si le résultat est nul (`ZF`=VRAI) ou non nul (`ZF`=FAUX).
  - `SF` : Sign Flag.
    Indique si le résultat est positif (`SF`=FAUX) ou négatif (`SF`=VRAI).
  - `OF` : Overflow Flag.
    Indique un débordement (`OF`=VRAI) sur les entiers signés.

Langage machine

Une instruction de langage machine correspond à une instruction possible du processeur.

Elle contient :

- un code correspondant à opération à réaliser,
- les arguments de l'opération : valeurs directes, numéros de registres, adresses mémoire.



Langage machine

Si on ouvre un fichier exécutable avec un éditeur (hexadécimal), on obtient

```
...
01ebe8140637274737475666662e6305f5f43544f525f4c
5f05f5f44544f525f4c4953545f5f05f5f4a43525f4c49
53545f5f05f5f646f5f676c6f62616c5f64746f72735f6
75780636f6d706c657465642e3635333106474 6f725f69
...
```

Langage ~~machine~~ lisible

Si on ouvre un fichier exécutable avec un éditeur (hexadécimal), on obtient

```
...
01ebe8140637274737475666 62e6305f5f43544f525f4c
5f05f5f44544f525f4c4953545f5f05f5f4a43525f4c49
53545f5f05f5f646f5f676c6f62616c5f64746f72735f6
75780636f6d706c657465642e3635331064746f725f69
...
```

C'est une suite d'instructions comme `01ebe814`, que l'on peut traduire directement de façon plus lisible :

```
mov     eax, ebx
```

C'est ce qu'on appelle l'*assembleur*.

- L'assembleur est donc une *représentation* du langage machine.
- Il y a autant d'assembleurs que de type de processeurs différents.

NASM : exemple

```
section .data
const     dw     123

section     .bss
var     resw     1

section     .text
global _start
_start:
     call    main
     mov     eax, 1
     int     0x80
main:
     push    ebp
     mov     ebp, esp
     mov     word [var], const
     pop     ebp
     ret
```

Sections

Un programme NASM est composé de trois sections :

- `.data`
  Déclaration de constantes (leur valeur ne changera pas durant l'execution)
- `.bss` (Block Started by Symbol)
  Déclaration de variables
- `.text`
  Instructions qui composent le programme

La section `data`

- La section `data` permet de définir des constantes
- Elle commence par

```
section .data
```

- Elle est constituée de lignes de la forme
  `etiquette pseudo-instruction valeur-initiale`
- Les pseudo instructions sont les suivantes :

`db`	define byte	déclare un octet
`dw`	define word	déclare deux octets
`dd`	define doubleword	déclare quatre octets
`dq`	define quadword	déclare huit octets
`dt`	define tenbytes	déclare dix octets

- Exemples :

```
consta  db 1
constb  dw 123
```

- les variables déclarées en séquence sont disposées les unes à côté des autres en mémoire

La section bss

- La section `bss` permet de définir des variables
- Elle commence par

```
section .bss
```

- Elle est constituée de lignes de la forme
  `etiquette pseudo-instruction nb`
- Les pseudo instructions sont les suivantes :

`resb`	reserve byte	déclare un octet
`resw`	reserve word	déclare deux octets
`resd`	reserve doubleword	déclare quatre octets
`resq`	reserve quadword	déclare huit octets
`rest`	reserve tenbytes	déclare dix octets

- `nb` représente le nombre d'octets (pour `resb`) de mots (pour `resw`) ... à réserver
- Exemples :

```
buffer      resb    64      ; reserve 64 octets
wordvar     resw    1       ; reserve un mot (deux octets)
realarray   resq    10      ; reserve 10 * 8 octets
```

La section `text`

- La section `text` contient les instructions correspondant au programme
- Elle commence par

```
section .text
```

- Elle est constituée de lignes de la forme

```
[étiquette:] nom_instr [opérandes] [;commentaires]
```

  les parties entre crochets sont optionnelles
- une étiquette correspond à une adresse (l'adresse dans laquelle est stockée l'instruction)
- une opérande peut être :
  - un *registre* (p. ex. `eax`),
  - une *adresse mémoire* (p. ex. l'étiquette `buffer`),
  - une *constante* ou valeur *immédiate* (p. ex. `365`),
  - une *expression* limitée à certains opérateurs (p. ex. `2 * 4`).

Accès à la mémoire

- Si `adr` est une adresse mémoire, alors `[adr]` représente le contenu de l'adresse `adr`
- C'est comme l'opérateur de déréférencement `*` du langage C
- La taille de l'objet référencé peut être spécifiée si nécessaire [1]
  - `byte [adr]` un octet
  - `word [adr]` deux octets
  - `dword [adr]` quatre octets
- `adr` peut être :
  - une *constante* ou valeur *immédiate* (p. ex. `[123]`),
  - une *étiquette* (p. ex. `[var]`),
  - un *registre* (p. ex. `[eax]`),
  - une *expression* limitée à certains opérateurs (p. ex. `[2*eax+var+1]`).

1. Notamment quand l'instruction ne comporte pas de registre à taille fixe.

Instructions

- instructions de transfert : registres $\leftrightarrow$ mémoire
  - Copie : `mov`
  - Gestion de la pile : `push`, `pop`
- instructions de calcul
  - Arithmétique : `add`, `sub`, `mul`, `div`
  - Logique : `and`, `or`
  - Comparaison : `cmp`
- instructions de saut
  - sauts inconditionnels : `jmp`
  - sauts conditionnels : `je`, `jne`, `jg`, `jl`, `jle`, `jge`
  - appel et retour de procédure : `call`, `ret`
- appels système

Copie - mov

- Syntaxe :

```
mov destination source
```

- Copie `source` vers `destination`
- `source` : un registre, une adresse ou une constante
- `destination` : un registre ou une adresse
- Les copies registre - registre sont possibles, mais pas les copies mémoire - mémoire
- Exemples :

```
mov eax, ebx          ; reg reg
mov eax, [var]        ; reg mem
mov ebx, 12           ; reg constante
mov [var], eax        ; mem re
```

Nombre d'octets copiés

- Lorsqu'on copie vers un registre ou depuis un registre , c'est la taille du registre qui indique le nombre d'octets copiés
- lorsqu'on copie une constante en mémoire, il faut préciser le nombre d'octets à copier, à l'aide des mots clefs
  - `byte` un octet
  - `word` deux octets
  - `dword` quatre octets
- Exemples :

```
mov eax, ebx          ; reg reg
mov eax, [var]        ; reg mem
mov ebx, 12           ; reg constante
mov [var], eax        ; mem reg
mov dword [var], 1     ; mem constante
```

Empile – push

- Syntaxe :

```
push source
```

- Copie le contenu de `source` au somme de la pile.
- Commence par décrémenter `esp` de 4 puis effectue la copie
- `source` : adresse, constante ou registre
- Exemples

```
push 1      ; empile la constante 1
push eax    ; empile le contenu de eax
push [var]  ; empile la valeur se trouvant
            ; a l'adresse var
```

Dépile – pop

- Syntaxe :

```
pop destination
```

- Copie les 4 octets qui se trouvent au somme de la pile dans `destination`.
- Commence par effectuer la copie puis incrémente `esp` de 4.
- `destination` est une adresse ou un registre
- Exemples :

```
pop eax   ; depile dans le registre eax
pop [var] ; depile a l'adresse var
```

Addition - add

- Syntaxe :

  `add destination source`

- Effectue `destination = destination + source`
- `source` : un registre, une adresse ou une constante
- `destination` : un registre ou une adresse
- modifie éventuellement les flags overflow (`OF`) et carry (`CF`)
- Les opérations registre - registre sont possibles, mais pas les opérations mémoire -mémoire
- Exemples :

```
add eax, ebx        ; reg reg
add eax, [var]      ; reg mem
add eax, 12         ; reg const
add [var], eax      ; mem reg
add [var], 1        ; mem const
```

Soustraction - sub

- Syntaxe :

  `sub destination source`

- Effectue `destination = destination - source`
- `source` : un registre, une adresse ou une constante
- `destination` : un registre ou une adresse
- modifie éventuellement les flags overflow (`OF`) et carry (`CF`)
- Les opérations registre - registre sont possibles, mais pas les opérations mémoire -mémoire
- Exemples :

```
sub eax, ebx        ; reg reg
sub eax, [var]      ; reg mem
sub eax, 12         ; reg const
sub [var], eax      ; mem reg
sub [var], 1        ; mem const
```

Multliplication – `imul`

- Syntaxe :

```
imul destination source
```

- Effectue `destination = destination * source`
- `source` : un registre, une adresse ou une constante
- `destination` : un registre ou une adresse
- Les opérations registre - registre sont possibles, mais pas les opérations mémoire -mémoire
- Exemples :

```
imul eax, ebx        ; reg reg
imul eax, [var]      ; reg mem
imul eax, 12         ; reg const
imul [var], eax      ; mem reg
imul [var], 1        ; mem const
```

Division – `idiv`

- Syntaxe :

```
idiv source
```

- Effectue la division entière : `edx:eax / source`
- Le quotient est mis dans `eax`
- Le reste est mis dans `edx`
- `source` : adresse, constante ou registre

Opérations logiques

```
and   destination   source
or    destination   source
xor   destination   source
not   destination
```

- Effectue les opérations logiques correspondantes bit à bit
- Le résultat se trouve dans `destination`
- opérandes :
  - `source` peut être : une adresse, un registre ou une constante
  - `destination` peut être : une adresse ou un registre

Comparaisons – cmp

- Syntaxe :

```
cmp destination, source
```

- Effectue l'opération `destination` - `source`
- le résultat n'est pas stocké
- `destination` : registre ou adresse
- `source` : constante, registre ou adresse
- les valeurs des flags `ZF` (zero flag), `SF` (sign flag) et `PF` (parity flag) sont éventuellement modifiées
- si `destination = source`, `ZF` vaut `VRAI`
- si `destination` $<$ `source`, `SF` vaut `VRAI`,

Saut inconditionnel – `jmp`

- Syntaxe :

```
jmp adr
```

- va à l'adresse `adr`

Saut conditionnel – `je`

- Syntaxe :

```
je adr
```

- `je` veut dire *jump equal*
- Si `ZF` vaut `VRAI` va à l'adresse `adr`

Autres sauts conditionnels – `jne`, `jg`, `jl`, ...

Instruction	Description	Flags testés
`jne`	jump not equal	`ZF`
`jg`	jump greater	`OF`, `SF`, `ZF`
`jl`	jump less	`OF`, `SF`
`jle`	jump less or equal	`OF`, `SF`, `ZF`
`jge`	jump greater or equal	`OF`, `SF`

Appel de procédure - `call`

- Syntaxe :

```
call adr
```

- empile `eip` (instruction pointer)
- va à l'adresse `adr`
- utilisé dans les appel de procédure : va à l'adresse où se trouve les instructions de la procédure et sauvegarde la prochaine instruction à effectuer au retour de l'appel.

Retour de procédure - `ret`

- Syntaxe :

```
ret
```

- dépile `eip`
- utilisé en fin de procédure
- à utiliser avec `call`

Appels système

- Syntaxe :

```
int 0x80
```

- NASM permet de communiquer avec le système grâce à la commande `int 0x80`.
- La fonction réalisée est déterminée par la valeur de `eax`

eax	Name	ebx	ecx	edx
1	sys_exit	int		
3	sys_read	unsigned int	char *	size_t
4	sys_write	unsigned int	const char *	size_t

Assembler un programme

- Un programme en assembleur x86 n'est pas exécutable
- Il s'agit d'un fichier textuel `.asm` contenant le code-source qui doit être assemblé
- Pour transformer le code-source assembleur `X86`, il faut les transformer en langage machine :
  `nasm -f elf -g -F dwarf test.asm -o test.o`
- puis faire l'édition de liens pour générer l'exécutable :
  `ld -m elf_i386 -o test test.o`

Références

- Cours d'architecture de Peter Niebert :

  `http://www.cmi.univ-mrs.fr/~niebert/archi2012.php`
- Cours de compilation de François Pottier :

  `http://www.enseignement.polytechnique.fr/informatique/INF564/`