Architecture IA-32 et Assembleur

Intel architecture 32 bits : architecture des Pentium.

Aussi appelée x86 (architecture de l'ensemble des processeurs Intel à partir du 8086).

Assembleur = programme convertissant les instructions du langage d'assemblage en micro-instructions.

Remarque : compilateur = programme similaire pour les langages de haut niveau (C, Java, ...).

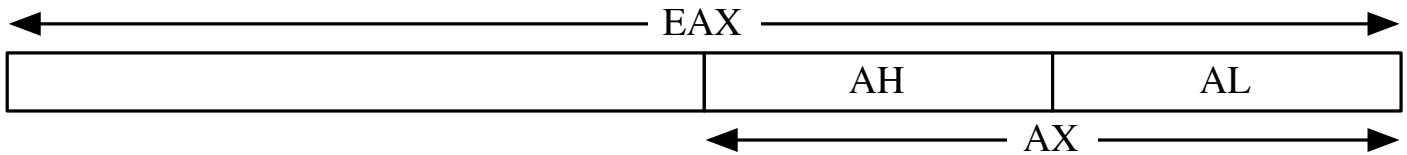
Chaque type de processeur a son propre langage machine $\Rightarrow$ il a également son propre langage d'assemblage.

En TP : NASM (Netwide Assembler)

Registres généraux (32 bits)

- EAX : registre accumulateur (*accumulator reg.*)
  pour les opérations arithmétiques et le stockage de la valeur de retour des appels systèmes.
- ECX : registre compteur (*counter reg.*)
- EBX : registre de base (*base reg.*)
- EDX : registre de données (*data reg.*)
  pour les opérations arithmétiques et les opérations d'E/S.

- AX : 16 bits de poids faible de EAX (idem BX, CX, DX)
- AL : octet de poids faible de AX (idem BL, CL, DL)
- AH : octet de poids fort de AX (idem BH, CH ,DH)



Registres spécialisés (32 bits)

▷ Registres d'adresses

- ESI : pointeur source (*Extended Source Index*)
- EDI : pointeur destination (*Extended Destination Index*)
- EBP : pointeur de base (*Extended Base Pointer*)
- ESP : pointeur de pile (*Extended Stack Pointeur*)

▷ Autres registres

- EIP : pointeur d'instruction
- EFLAGS : registre d'états (drapeaux)
- CS, SS, DS, ES, FS, GS : registres de segment (16 bits) : adresses et données de programme

Drapeaux

- Zero Flag (ZF)
  résultat de la dernière opération de comparaison (1 si égalité).
- Overflow Flag (OF)
  1 si la dernière opération a provoqué un overflow.
- Carry Flag (CF)
  1 si la dernière opération a provoqué une retenue.
- Sign Flag (SF)
  1 si la dernière opération a générée un résultat négatif, 0 s'il est positif ou nul.
- Parity Flag (PF)
  1 si la dernière opération a générée un résultat impair, 0 s'il est pair.
- Interrupt Flag (IF)
  1 si les interruptions sont autorisées, à 0 sinon.

Format d'instruction et d'opérandes

INSTRUCTION = OPÉRATION suivi d'OPÉRANDES (de 0 à 3)

▷ une opérandes est :

- soit une donnée brute :
  - *adressage immédiat* : valeur binaire, décimale ou hexadécimale
    Exemples : `mov eax, 16` (décimal)
    `mov eax, 0b11` (binaire)
    `mov eax, 0xff` (hexadécimal)
  - *adressage implicite* : pas spécifié explicitement, par exemple l'incrémentation (le 1 est implicite)
- soit une adresse : avec différents *modes d'adressage*.

! les types d'opérandes autorisés dépendent de l'opération effectuée.

Notation : **A : adresse A** $\neq$ **[A] : donnée à l'adresse A**

Mode d'adressage

Les opérandes peuvent avoir les types suivants :

- *adressage direct* : l'opérande est une adresse (32 bits) en mémoire. désigne toujours le même emplacement, mais la valeur correspondante peut changer (ex : variable globale)
  Exemple : `mov eax, [0x0000f13a]`
- *adressage par registre* : l'opérande est un registre.
  mode le plus courant (+ efficace)
  Exemple : `mov eax, ebx`
- *adressage indirect par registre* : l'opérande une adresse mémoire contenue dans un registre (qui sert de pointeur)
  Exemples : `mov eax, [esp]` (eax ← **sommet** de la pile)
  !! `mov eax, esp` (eax ← **adresse** du sommet)
- *adressage indexé* : l'opérande une adresse mémoire contenue dans un registre associée à un décalage
  Exemple : `mov eax, [esp+4]`

Instructions x86

Grandes catégories d'opérations :

- Opérations de transfert :
  entre la mémoire et les registres ; opérations sur la pile.
- Opérations arithmétiques
- Opérations logiques
- Opérations de décalage et rotation
  multiplications et divisions rapides
- Opérations de branchement
  sauts, boucles, appels de fonctions
- Opérations sur les chaînes de caractères

Instructions de transfert

▷ Copie de données entre mémoire et registres : `mov`

Le 1er argument est toujours la *destination* et le 2nd la *source*

Restriction sur le type d'opérandes :

```
mov registre, mémoire
mov mémoire, registre
mov registre, registre
```
(registres de même taille !)

```
mov type mémoire, immédiate
```
(type=byte, word, dword)

```
mov registre, immédiate
```

~~`mov mémoire, mémoire`~~ impossible !!

▷ Instruction spéciale pour échanger les contenus de 2 registres ou d'un registre et d'une case mémoire : `xchg`

Exemples :
```
xchg eax, ebx
xchg eax, [0xbgfffeedc]
```

Instructions de transfert (suite)

▷ Opérations de pile : `push` et `pop`

adressage immédiat ou par registre

Exemples : `push eax`

`pop ebx`

!! la pile est à l'envers :

si `esp` est l'adresse du sommet de la pile,
alors l'élément *en dessous* est `[esp+4]`

Résumé

Instruction			Description
`mov`	*dst*	*src*	déplace *src* dans *dst*
`xchg`	*ds*1	*ds*2	échange *ds*1 et *ds*2
`push`	*src*		place *src* au sommet de la pile
`pop`	*dst*		supprime le sommet de la pile et le place dans *dst*

Instructions arithmétiques

▷ Addition entière (en cplt à 2) : `add`

2 opérandes : destination et source : valeurs, registres ou mémoire (au moins 1 reg.)

positionne les FLAGS : Carry (CF) et Overflow (OF)

Exemples : `add ah, bl`

~~`add ax, bl`~~ opérandes imcompatibles !

▷ Addition avec retenue : `adc`

additionne les 2 opérandes et la retenue positionnée dans CF

Exemple : `adc eax, ebx` (`eax` ← `eax`+`ebx`+CF)

▷ Multiplication entière positive : `mul`

1 seule opérande : multiplication par `eax`

le résultat est stocké dans deux registres : `edx` pour les bits de poids fort et `eax` pour les bits de poids faible

Exemple : `mul ebx` (`edx|eax` ← `eax`·`ebx`)

Instructions arithmétiques (suite)

▷ Multiplication entière en cplt à 2 : `imul`

mêmes caractéristiques que `mul` avec des entiers relatifs

Résumé

`add`	*dst*	*src*	ajoute *src* à *dst*
`adc`	*dst*	*src*	ajoute *src* à *dst* avec retenue
`sub`	*dst*	*src*	soustrait *src* à *dst*
`sbb`	*dst*	*src*	soustrait *src* à *dst* avec retenue
`mul`	*src*		multiplie `eax` par *src* (résultat dans `edx`|`eax`)
`imul`	*src*		multiplie `eax` par *src* (cplt à 2)
`div`	*src*		divise `edx`|`eax` par *src* (`eax`=quotient, `edx`=reste)
`idiv`	*src*		divise `edx`|`eax` par *src* (cplt à 2)
`inc`	*dst*		$1 + dst$
`dec`	*dst*		$dst - 1$
`neg`	*dst*		$-dst$

Instructions logiques

Les opérations logiques sont des opérations bit à bit.

▷ Et logique : `and`

2 opérandes : destination et source

Exemple : Utilisation d'un masque pour l'extraction des 4 bits de poids faible de `ax` : `and ax, 0b00001111`

Résumé

`not`	*dst*		place (not *dst*) dans *dst*
`and`	*dst*	*src*	place (*src* AND *dst*) dans *dst*
`or`	*dst*	*src*	place (*src* OR *dst*) dans *dst*
`xor`	*dst*	*src*	place (*src* XOR *dst*) dans *dst*

Instructions de décalage/rotation

2 opérandes : un registre suivi d'un nombre de décalages nb.

▷ Décalage logique à gauche : `shl`

Insertion de nb 0 au niveau du bit de poids faible.
Permet d'effectuer efficacement la multiplication par 2^{nb}.
Exemple : `shl al, 4` (ex : 01100111 → 01110000)

▷ Décalage arithmétique à droite : `sar`

Insertion de nb copies du bit de poids fort à gauche.
Permet la division rapide d'un entier relatif par 2^{nb}.
Exemple : `sar al, 4` (ex : 10011110 → 11111001)

▷ Rotation à gauche : `rol`

rotation de nb bits vers la gauche : les bits sortants à gauche sont immédiatement réinjectés à droite.
exemple : `rol al, 3` (ex : 10011111 → 11111100)

Instructions de décalage/rotation (suite)

▷ Rotation à droite avec retenue : `rcr`

Rotation de nb bits vers la droite en passant par la retenue : lors d'un décalage, le bit sortant à droite est mémorisé dans la retenue qui est elle-même réinjectée à gauche.

Exemple : `rcr al, 3` (ex : $11111101, c = 0 \rightarrow 01011111, c = 1$)

Résumé

`sal`	*dst*	*nb*	décalage arithmétique à gauche de *nb* bits de *dst*
`sar`	*dst*	*nb*	décalage arithmétique à droite de *nb* bits de *dst*
`shl`	*dst*	*nb*	décalage logique à gauche de *nb* bits de *dst*
`shr`	*dst*	*nb*	décalage logique à droite de *nb* bits de *dst*
`rol`	*dst*	*nb*	rotation à gauche de *nb* bits de *dst*
`ror`	*dst*	*nb*	rotation à droite de *nb* bits de *dst*
`rcl`	*dst*	*nb*	rotation à gauche de *nb* bits de *dst* avec retenue
`rcr`	*dst*	*nb*	rotation à droite de *nb* bits de *dst* avec retenue

Instructions de branchement

▷ Instruction de comparaison : `cmp`

Effectue une soustraction (comme `sub`), mais ne stocke pas le résultat : seuls les drapeaux sont modifiés.

Exemple : `cmp eax, ebx` (si eax=ebx, alors ZF=1)

▷ Saut conditionnel vers l'étiquette spécifiée : `jxx`

- `je`, `jne` : jump if (resp. if not) equal
  saute si le drapeau d'égalité (positionné par `cmp`) est à 1 (resp. à 0).
- `jge`, `jnge` : jump if (resp. if not) greater or equal
  saute si le résultat de `cmp` est (resp. n'est pas) *plus grand ou égal à.*
- `jl`, `jnl` : jump if (resp. if not) less than
  saute si le résultat de `cmp` est (resp. n'est pas) *stt plus petit que.*
- `jo`, `jno` : jump if (resp. if not) overflow
- `jc`, `jnc` : jump if (resp. if not) carry
- `jp`, `jnp` : jump if (resp. if not) parity
- `jcxz`, `jecxz` jump if cx (resp. ecx) is null
  sautent quand le registre cx (resp. ecx) est nul

Instructions de branchement (suite)

▷ Boucle fixe : `loop`

`ecx` ← `ecx`-1 et saute vers l'étiquette sauf si `ecx` = 0.

▷ Boucle conditionnelle : `loope`

`ecx` ← `ecx`-1 et saute vers l'étiquette si ZF = 1 et `ecx` ≠ 0.

▷ Boucle conditionnelle : `loopne`

`ecx` ← `ecx`-1 et saute vers l'étiquette si ZF = 0 et `ecx` ≠ 0.

Résumé

`cmp`	*sr*1	*sr*2	compare *sr*1 et *sr*2
`jmp`	*adr*		saut vers l'adresse *adr*
`jxx`	*adr*		saut conditionné par xx vers l'adresse *adr*
`loop`	*adr*		répétition de la boucle *nb* de fois (*nb* dans `ecx`)
`loopx`	*adr*		répétition de la boucle conditionnée par x