Écriture en mémoire

L'instruction

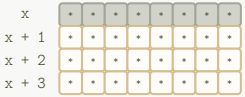
```
mov DT [ADR], VAL
```

écrit dans la mémoire à partir de l'adresse `ADR` la valeur `VAL`.

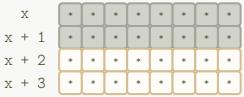
Le champ `DT` est un descripteur de taille qui permet de préciser la taille de `VAL` :

Descripteur de taille	Taille (en octets)
`byte`	1
`word`	2
`dword`	4

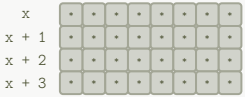
Si `x` est une adresse accessible en mémoire et `val` une valeur, les parties foncées de la mémoire sont celles qui sont modifiées :



`mov byte [x], val`



`mov word [x], val`



`mov dword [x], val`

Écriture en mémoire — tailles

Le champ `val` peut être un (sous-)registre. Dans ce cas, il n'est pas nécessaire de préciser le descripteur de taille (le nombre d'octets écrits en mémoire dépend de la taille du (sous-)registre).

– Exemples –

Les instructions suivantes **ne sont pas correctes** (ici, `x` est une adresse accessible en mémoire) :

- `mov byte [x], eax`

  Le registre `eax` occupe 4 octets, ce qui est contradictoire avec le descripteur `byte` (1 octet).

- `mov word [x], bl`

  Le sous-registre `bl` occupe 1 octet, ce qui est contradictoire avec le descripteur `word` (2 octets).

- `mov byte [x], 0b010010001`

  La donnée à écrire tient sur au moins 2 octets (9 bits), ce qui est contradictoire avec le descripteur `byte` (1 octet).

- `mov [x], -125`

  La taille de la donnée à écrire n'est pas connue.

Écriture en mémoire — tailles

– Exemples –

En revanche, les instructions suivantes **sont correctes** :

- `mov [x], eax`

  Le registre `eax` occupe implicitement 4 octets.

- `mov dword [x], eax`

  Ceci est correct, bien que pléonastique.

- `mov word [x], 0b010010001`

  La donnée à écrire est vue sur 16 bits, en ajoutant des `0` à gauche.

- `mov dword [x], 0b010010001`

  La donnée à écrire est vue sur 32 bits, en ajoutant des `0` à gauche.

- `mov word [x], -125`

  La donnée à écrire est vue sur 2 octets, en ajoutant des `1` à gauche car elle est négative.

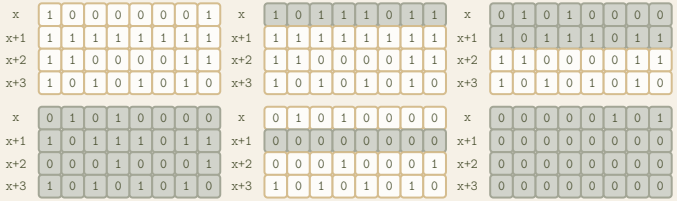
Écriture en mémoire

– Exemple –

Observons l'effet des instructions avec la mémoire dans l'état suivant :

x	1	0	0	0	0	0	0	1
x + 1	1	1	1	1	1	1	1	1
x + 2	1	1	0	0	0	0	1	1
x + 3	1	0	1	0	1	0	1	0

```
mov eax, 0xAA11BB50
mov [x], ah
mov [x], ax
mov [x], eax
mov byte [x + 1], 0
mov dword [x], 0x5
```



Section de données initialisées

La section de données est la partie (facultative) du programme qui regroupe des définitions de données pointées par des adresses. Elle commence par `section .data`.

On **définit une donnée** par

```
ID: DT VAL
```

où `ID` est un identificateur (appelé étiquette), `VAL` une valeur et `DT` un descripteur de taille parmi

Descripteur de taille	Taille (en octets)
`db`	1
`dw`	2
`dd`	4
`dq`	8

Ceci place en mémoire à l'adresse `ID` la valeur `VAL`, dont la taille est spécifiée par `DT`.

La valeur de l'adresse `ID` est attribuée par le système.

Section de données initialisées

– Exemples –

- `entier: dw 55`

  Créé à l'adresse `entier` un entier sur 2 octets, initialisé à $(55)_{\text{dix}}$.

- `x: dd 0xFFE05`

  Créé à l'adresse `x` un entier sur 4 octets, initialisé à $(\texttt{000FFE05})_{\text{hex}}$.

- `y: db 0b11001100`

  Créé à l'adresse `y` un entier sur 1 octet initialisé à $(\texttt{11001100})_{\text{deux}}$.

- `c: db 'a'`

  Créé à l'adresse `c` un entier sur 1 octet dont la valeur est le code ASCII du caractère `'a'`.

- `chaine: db 'Test', 0`

  Créé à partir de l'adresse `chaine` une suite de 5 octets contenant successivement les codes ASCII des lettres `'T'`, `'e'`, `'s'`, `'t'` et du marqueur de fin de chaîne.

  De plus, à l'adresse `chaine + 2` figure le code ASCII du caractère `'s'`.

Section de données initialisées — définitions multiples

On peut **définir plusieurs données** de manière concise par

```
ID: times NB DT VAL
```

où `ID` est un identificateur, `VAL` une valeur, `DT` un descripteur de taille et `NB` une valeur positive.

Ceci place en mémoire, à partir de l'adresse `ID`, `NB` occurrences de la valeur `VAL`, dont la taille est spécifiée par `DT`.

– Exemples –

- `suite: times 85 dd 5`

  Créé à partir de l'adresse `suite` une suite de 85×4 octets, où chaque double mot est initialisé à la valeur $(5)_{\text{dix}}$.

  L'adresse du 1er double mot est `suite`. L'adresse du 7e double mot est `suite + (6 * 4)`.

- `chaine: times 9 db 'a'`

  Créé à partir de l'adresse `chaine` une suite de 9 octets tous initialisés par le code ASCII du caractère `'a'`. L'adresse du 3e octet est `chaine + 2`.

Section de données non initialisées

La section de données non initialisées est la partie (facultative) du programme qui regroupe des déclarations de données non initialisées pointées par des adresses. Elle commence par `section .bss`.

On **déclare une donnée non initialisée** par

```
ID: DT NB
```

où `ID` est un identificateur, `NB` une valeur positive et `DT` un descripteur de taille parmi

Descripteur de taille	Taille (en octets)
`resb`	1
`resw`	2
`resd`	4
`resq`	8

Ceci réserve une zone de mémoire commençant à l'adresse `ID` et pouvant accueillir `NB` données dont la taille est spécifiée par `DT`.

Section de données non initialisées

– Exemples –

La déclaration

```
x: resw 120
```

réserve, à partir de l'adresse `x`, une suite de 120×2 octets non initialisés.

- L'adresse de la `i`$^{\text{e}}$ donnée à partir de `x` est `x + ((i - 1) * 2)`.
- Pour écrire la valeur $(\texttt{0xEF01})_{\text{hex}}$ en 4$^{\text{e}}$ position, on utilise l'instruction

```
mov word [x + (3 * 2)], 0xEF01
```

- Pour lire la valeur située en 7$^{\text{e}}$ position à partir de `x`, on utilise les instructions

```
mov eax, 0
mov ax, [x + (6 * 2)]
```

Attention : il ne faut jamais rien supposer sur la valeur initiale d'une donnée non initialisée.

Section d'instructions

La section des instructions est la partie du programme qui regroupe les instructions. Elle commence par `section .text`.

Pour définir le point d'entrée du programme, il faut définir une **étiquette de code** et faire en sorte de la rendre visible depuis l'extérieur.

Pour cela, on écrit

```
section .text
global main
main:
    INSTR
```

où `INSTR` dénote la suite des instructions du programme. Ici, `main` est une étiquette et sa valeur est l'adresse de la 1re instruction constituant `INSTR`.

La ligne `global main` sert à rendre l'étiquette `main` visible pour l'édition des liens.

Interruptions — généralités

L'exécution d'un programme se fait instruction par instruction. Dès qu'une instruction est traitée, le processeur s'occupe de la suivante.

Cependant, certaines instructions ont besoin d'**interrompre l'exécution** pour être menées à bien. Parmi celles-ci, nous avons

- l'écriture de texte sur la sortie standard ;
- la lecture d'une donnée sur l'entrée standard ;
- l'écriture d'une donnée sur le disque ;
- la gestion de la souris ;
- la communication via le réseau ;
- la sollicitation de l'unité graphique ou sonore.

Dans ce but, il existe des instruction particulières appelées interruptions.

Interruptions — instruction

L'instruction

```
int 0x80
```

permet d'**appeler une interruption** dont le traitement est délégué au système (Linux).

La tâche à réaliser est spécifiée par un code lu depuis le registre `eax`. Voici les principaux :

Code	Rôle
`1`	Arrêt et fin de l'exécution
`3`	Lecture sur l'entrée standard
`4`	Écriture sur la sortie standard

Les autres registres de travail `ebx`, `ecx` et `edx` jouent le rôle d'arguments à la tâche en question.

Attention : le traitement d'une interruption peut modifier le contenu des registres. Il faut sauvegarder leur valeur dans la mémoire si besoin est.

– Exemples –

- Pour **stopper l'exécution** d'un programme, on utilise

```
mov ebx, 0
mov eax, 1
int 0x80
```

Le registre `ebx` contient la valeur de retour de l'exécution.

- Pour **afficher un caractère** sur la sortie standard, on utilise

```
mov ebx, 1
mov ecx, x
mov edx, 1
mov eax, 4
int 0x80
```

La valeur de `ebx` spécifie que l'on écrit sur la sortie standard.
Le registre `ecx` contient l'adresse `x` du caractère à afficher.
La valeur de `edx` signifie qu'il y a un unique caractère à afficher.

- Pour **lire un caractère** sur l'entrée standard, on utilise

```
mov ebx, 1
mov ecx, x
mov edx, 1
mov eax, 3
int 0x80
```

La valeur de `ebx` spécifie que l'on lit sur l'entrée standard.
Le registre `ecx` contient l'adresse `x` à laquelle le code ASCII du caractère lu sera écrite.
La valeur de `edx` signifie qu'il y a un unique caractère à lire.

Directives

Une directive est un élément d'un programme qui n'est pas traduit en langage machine mais qui sert à informer l'assembleur, entre autre, de

- la définition d'une constante ;
- l'inclusion d'un fichier.

Pour **définir une constante**, on se sert de

```
%define NOM VAL
```

Ceci fait en sorte que, dans le programme, le symbole `NOM` est remplacé par le symbole `VAL`.

Pour **inclure un fichier** (assembleur `.asm` ou en-tête `.inc`), on se sert de

```
%include CHEM
```

Ceci fait en sorte que le fichier de chemin relatif `CHEM` soit inclus dans le programme. Il est ainsi possible d'utiliser son code dans le programme appelant.

Assemblage

Pour assembler un programme `PRGM.asm`, on utilise la commande

```
nasm -f elf32 PRGM.asm
```

Ceci créé un fichier objet nommé `PRGM.o`.

On obtient un exécutable par l'**édition des liens**, en utilisant la commande

```
ld -o PRGM -e main PRGM.o
```

Ceci créé un exécutable nommé `PRGM`.

L'option `-e main` spécifie que le point d'entrée du programme est l'instruction à l'adresse `main`.

Remarque : sur un système 64 bits, on ajoute pour l'édition des liens l'option `-melf_i386`, ce qui donne donc la commande

```
ld -o PRGM -melf_i386 -e main PRGM.o
```

.

Exemple complet de programme

```
; Def. de donnees
section .data

chaine_1:
  db 'Caractere ? ',0

chaine_2:
  db 'Suivant : ',0

; Decl. de donnees
section .bss

car: resb 1
```

```
; Instructions
section .text

global main

main:
  ; Aff. chaine_1
  mov ebx, 1
  mov ecx, chaine_1
  mov edx, 13
  mov eax, 4
  int 0x80
```

```
  ; Lect. car.
  mov ebx, 1
  mov ecx, car
  mov edx, 1
  mov eax, 3
  int 0x80

  ; Incr. car.
  mov eax, [car]
  add eax, 1
  mov [car], al

  ; Aff. chaine_2
  mov ebx, 1
  mov ecx, chaine_2
  mov edx, 11
  mov eax, 4
  int 0x80
```

```
  ; Aff. car.
  mov ebx, 1
  mov ecx, car
  mov edx, 1
  mov eax, 4
  int 0x80

  ; Sortie
  mov ebx, 0
  mov eax, 1
  int 0x80
```

Ceci lit un caractère sur l'entrée standard et affiche le caractère suivant de la table ASCII.

Étiquettes d'instruction

Les **instructions** d'un programme sont des données comme des autres. Elles ont donc une **adresse**.

Tout comme pour les données, il est possible de disposer des étiquettes dans un programme, dont les valeurs sont des adresses d'instructions.

Ceci se fait par

```
ETIQ: INSTR
```

où `ETIQ` est le nom de l'étiquette et `INSTR` une instruction.

– Exemple –

```
mov eax, 0
e1: mov ebx, 1
add eax, ebx
e2: sub eax, 0x12A
```

L'étiquette `e1` pointe vers l'instruction `mov ebx, 1`.

L'étiquette `e2` pointe vers l'instruction `sub eax, 0x12A`.

Remarque : nous avons déjà rencontré l'étiquette `main`. Il s'agit d'une étiquette d'instruction. Sa valeur est l'adresse de la 1re instruction du programme.

Pointeur d'instruction et exécution

À chaque instant de l'exécution d'un programme, le registre `eip`, appelé pointeur d'instruction, contient l'**adresse de la prochaine instruction** à exécuter.

L'exécution d'un programme s'organise selon l'algorithme suivant :

1. répéter, tant que l'exécution n'est pas interrompue :
   1.1 charger l'instruction I d'adresse `eip` ;
   1.2 mettre à jour `eip` ;
   1.3 traiter l'instruction I.

Par défaut, après le traitement d'une instruction (en tout cas de celles que nous avons vues pour le moment), `eip` est mis à jour de sorte à contenir l'adresse de l'instruction suivante en mémoire.

Il est impossible d'intervenir directement sur la valeur de `eip`.

Sauts inconditionnels

Ainsi, par défaut, l'exécution d'un programme se fait instruction par instruction, dans l'ordre dans lequel elles sont écrites.

Néanmoins, il est possible de rompre cette ligne d'exécution en réalisant des sauts. Ils consistent, étant donné un point de départ, à poursuivre l'exécution du programme vers un point cible.

Pour cela, on se sert de l'instruction

```
jmp ETIQ
```

où `ETIQ` est une étiquette d'instruction. Cette instruction **saute** à l'endroit du code pointé par `ETIQ`.

Elle agit en **modifiant** de manière adéquate **le pointeur d'instruction** `eip`.

Sauts inconditionnels

– Exemples –

```
mov ebx, 0xFF
jmp e1
mov ebx, 0
e1:
    mov eax, 1
```

L'instruction `mov ebx, 0` n'est pas exécutée puisque l'instruction `jmp e1` qui la précède fait en sorte que l'exécution passe à l'étiquette `e1`.

```
mov eax, 0
debut:
    add eax, 1
    jmp debut
```

L'exécution de ces instructions provoque une boucle infinie. Le saut inconditionnel vers l'étiquette `debut` précédente provoque la divergence.

```
e1:
    mov eax, 1
    mov ebx, 1
    e2:
        mov eax, 1
        jmp e1
```

L'étiquette `e2` est présente mais n'est pas utilisée ici.
Noter l'indentation du code à chaque création d'étiquette.

Le registre `flags`

À tout moment de l'exécution d'un programme, le registre de drapeaux `flags` contient des informations sur la dernière instruction exécutée.

Comme son nom l'indique, il fonctionne comme un drapeau : chacun de ses bits code une information du type oui (bit à `1`) / non (bit à `0`).

Voici certaines des informations qu'il contient. Le bit

- `CF`, « *Carry Flag* », vaut `1` si l'instruction produit une retenue de sortie et `0` sinon ;
- `ZF`, « *Zero Flag* », vaut `1` si l'instruction produit un résultat nul et `0` sinon ;
- `SF`, « *Sign Flag* », vaut `1` si l'instruction produit un résultat négatif et `0` sinon ;
- `OF`, « *Overflow Flag* », vaut `1` si l'instruction produit un dépassement de capacité et `0` sinon.

Instruction de comparaison

L'**instruction de comparaison** `cmp` s'utilise par

```
cmp VAL_1, VAL_2
```

et permet de **comparer** les valeurs `VAL_1` et `VAL_2` en mettant à jour le registre `flags`.

Cette instruction calcule la différence `VAL_1` − `VAL_2` et modifie `flags` de la manière suivante :

- si `VAL_1` − `VAL_2` $= 0$, alors `ZF` prend pour valeur `1` ;
- si `VAL_1` − `VAL_2` > 0, alors `ZF` prend pour valeur `0` et `CF` prend pour valeur `0` ;
- si `VAL_1` − `VAL_2` < 0, alors `ZF` prend pour valeur `0` et `CF` prend pour valeur `1`.

Un descripteur de taille (`byte`, `word`, `dword`) peut être utilisé pour préciser la taille des valeurs.

– Exemple –

```
mov ebx, 5
cmp dword 21, ebx
```

Cette comparaison fait que `ZF` et `CF` prennent pour valeur `0`.

Sauts conditionnels

Un saut conditionnel est un saut qui n'est réalisé que si une condition impliquant le registre `flags` est vérifiée ; si celle-ci n'est pas vérifiée, l'exécution se poursuit en l'instruction qui suit le saut conditionnel.

Pour ce faire, on adopte le schéma

```
cmp VAL_1, VAL_2
SAUT ETIQ
```

`VAL_1` et `VAL_2` sont des valeurs, `ETIQ` est une étiquette d'instruction et `SAUT` est une instruction de saut conditionnel.

Les différentes instruction de saut conditionnel diffèrent sur la condition qui provoque le saut :

Instruction	Condition de saut
`je`	`VAL_1` $=$ `VAL_2`
`jne`	`VAL_1` $\neq$ `VAL_2`
`jl`	`VAL_1` $<$ `VAL_2`
`jle`	`VAL_1` $\leqslant$ `VAL_2`
`jg`	`VAL_1` $>$ `VAL_2`
`jge`	`VAL_1` $\geqslant$ `VAL_2`

Sauts conditionnels

– Exemples –

```
cmp eax, ebx
jl inferieur
jmp fin
inferieur:
    mov eax, ebx
fin:
```

Ceci saute à l'étiquette `inferieur` si la valeur de `eax` est strictement inférieure à celle de `ebx`.

Ceci fait en sorte que `eax` vaille $\max($`eax`, `ebx`$)$.

```
mov ecx, 15
debut:
    cmp ecx, 0
    je fin
    sub ecx, 1
    jmp debut
fin:
```

Ceci forme une boucle

Tant que la valeur de `ecx` est différente de `0`, `ecx` est décrémenté et un tour de boucle est réalisé.

Quinze tours de boucle sont effectués avant de rejoindre l'étiquette `fin`.

Simulation du `if`

L'équivalent du pseudo-code

```
Si a = b
    BLOC
FinSi
```

est

```
cmp eax, ebx
je then
jmp end_if
then:
    BLOC
end_if:
```

L'équivalent du pseudo-code

```
Si a = b
    BLOC_1
Sinon
    BLOC_2
FinSi
```

est

```
cmp eax, ebx
jne else
BLOC_1
    jmp end_if
else:
    BLOC_2
end_if:
```

Simulation du `while` et du `do while`

L'équivalent du pseudo-code

```
TantQue a = b
    BLOC
FinTantQue
```

est

```
while:
    cmp eax, ebx
    jne end_while
    BLOC
    jmp while
end_while:
```

L'équivalent du pseudo-code

```
Faire
    BLOC
TantQue a = b
```

est

```
do:
    BLOC
    cmp eax, ebx
    je do
```

Simulation du `for`

L'équivalent du pseudo-code

```
Pour a = 1 à b
    BLOC
FinPour
```

est

```
mov eax, 1
for:
    cmp eax, ebx
    jg end_for
    BLOC
    add eax, 1
    jmp for
end_for:
```

On peut simuler ce pseudo-code de manière plus compacte grâce à l'instruction

```
loop ETIQ
```

Celle-ci décrémente `ecx` et saute vers l'étiquette d'instruction `ETIQ` si `ecx` est non nul.

On obtient la suite d'instructions suivante :

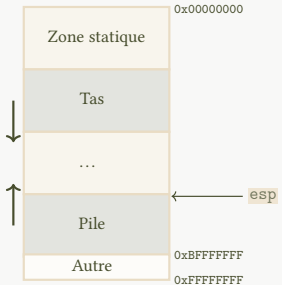
```
mov ecx, ebx
boucle:
    BLOC
    loop boucle
```

Pile

La pile est une zone de la mémoire dans laquelle des données peuvent être empilée ou dépilées.

La zone qu'elle occupe en mémoire est de taille variable mais elle se situe toujours avant l'adresse `0xBFFFFFFF`.

La pile est de type **LIFO** : les données sont dépilées de la plus récente à la plus ancienne.



On place et on lit dans la pile uniquement des **doubles mots**.

Le registre `esp` contient l'adresse de la **tête de pile**.

Le registre `ebp` est utilisé pour sauvegarder une position dans la pile (lorsque `esp` est susceptible de changer).