L'hexadécimal

Un entier codé en hexadécimal est un entier écrit en base seize.

Ce codage utilise seize symboles : les chiffres `0`, `1`, …, `9` et les lettres `A`, `B`, `C`, `D`, `E`, `F`.

Pour exprimer une **suite de bits** u en hexadécimal, on remplace, en partant de la droite, chaque groupe de quatre bits de u par un chiffre hexadécimal au moyen de la table suivante :

$(\texttt{0000})_{\text{deux}} \to (\texttt{0})_{\text{hex}}$	$(\texttt{0100})_{\text{deux}} \to (\texttt{4})_{\text{hex}}$	$(\texttt{1000})_{\text{deux}} \to (\texttt{8})_{\text{hex}}$	$(\texttt{1100})_{\text{deux}} \to (\texttt{C})_{\text{hex}}$
$(\texttt{0001})_{\text{deux}} \to (\texttt{1})_{\text{hex}}$	$(\texttt{0101})_{\text{deux}} \to (\texttt{5})_{\text{hex}}$	$(\texttt{1001})_{\text{deux}} \to (\texttt{9})_{\text{hex}}$	$(\texttt{1101})_{\text{deux}} \to (\texttt{D})_{\text{hex}}$
$(\texttt{0010})_{\text{deux}} \to (\texttt{2})_{\text{hex}}$	$(\texttt{0110})_{\text{deux}} \to (\texttt{6})_{\text{hex}}$	$(\texttt{1010})_{\text{deux}} \to (\texttt{A})_{\text{hex}}$	$(\texttt{1110})_{\text{deux}} \to (\texttt{E})_{\text{hex}}$
$(\texttt{0011})_{\text{deux}} \to (\texttt{3})_{\text{hex}}$	$(\texttt{0111})_{\text{deux}} \to (\texttt{7})_{\text{hex}}$	$(\texttt{1011})_{\text{deux}} \to (\texttt{B})_{\text{hex}}$	$(\texttt{1111})_{\text{deux}} \to (\texttt{F})_{\text{hex}}$

– Exemple –

$(\texttt{10 1111 0101 1011 0011 1110})_{\text{deux}} = (\texttt{2F5B3E})_{\text{hex}}$.

La conversion dans l'autre sens se réalise en appliquant la même idée.

Ce codage est **concis** : un octet est codé par seulement deux chiffres hexadécimaux.

Les nombres réels

Nombres entiers *vs* nombres réels :

- il y a une infinité de nombre entiers mais il y a un **nombre fini** d'entiers dans tout intervalle $[\![a, b]\!]$ où $a \leqslant b \in \mathbb{Z}$;

- il y a également une infinité de nombre réels mais il y a cette fois un **nombre infini** de réels dans tout intervalle $[\alpha, \beta]$ où $\alpha < \beta \in \mathbb{R}$.

Conséquence : il n'est possible de représenter qu'un sous-ensemble de nombres réels d'un intervalle donné.

Les nombres représentables sont appelés nombre flottants. Ce sont des approximations des nombres réels.

Représentation à virgule fixe

Le codage à virgule fixe consiste à représenter un nombre à virgule en deux parties :

1. sa **troncature**, sur n bits ;
2. sa **partie décimale**, sur m bits ;

où les entiers n et m sont fixés.

La troncature est codée en utilisant la représentation en complément à deux.

Chaque bit de la partie décimale correspond à l'inverse d'une puissance de deux.

– Exemple –

Avec $n = 4$ et $m = 5$,

$$\begin{aligned}(0110.01100)_{\text{vf}} &= 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} \\ &= (6.375)_{\text{dix}}\end{aligned}$$

Représentation à virgule fixe

Soient $b \geqslant 2$ un entier et $0 < x < 1$ un nombre rationnel. Pour écrire x en **base** b, on procède comme suit :

1. $L \leftarrow [\,]$
2. Tant que $x \neq 0$:
   - 2.1 $x \leftarrow x \times b$
   - 2.2 $L \leftarrow L \cdot \lfloor x \rfloor$
   - 2.3 $x \leftarrow x - \lfloor x \rfloor$
3. Renvoyer L.

Ici, $[\,]$ est la liste vide, $\times$ est la multiplication, $\cdot$ désigne la concaténation des listes et $\lfloor - \rfloor$ est l'opérateur de partie entière inférieure.

Représentation à virgule fixe

– Exemple –

Avec $x := (0.375)_{\text{dix}}$ et $b := 2$, on a

x	$x \times 2$	$x - \lfloor x \rfloor$	L
$(0.375)_{\text{dix}}$	$(\mathbf{0}.75)_{\text{dix}}$	$(0.75)_{\text{dix}}$	$[\mathbf{0}]$
$(0.75)_{\text{dix}}$	$(\mathbf{1}.5)_{\text{dix}}$	$(0.5)_{\text{dix}}$	$[0, \mathbf{1}]$
$(0.5)_{\text{dix}}$	$(\mathbf{1}.0)_{\text{dix}}$	$(0.0)_{\text{dix}}$	$[0, 1, \mathbf{1}]$
$(0)_{\text{dix}}$	–	–	$[0, 1, 1]$

Ainsi, $(0.375)_{\text{dix}} = (0.011)_{\text{vf}}$.

Limites de la représentation à virgule fixe

Appliquons l'« algorithme » précédent sur les entrées $x := (0.1)_{\text{dix}}$ et $b := 2$. On a

x	$x \times 2$	$x - \lfloor x \rfloor$	L
$(0.1)_{\text{dix}}$	$(\mathbf{0}.2)_{\text{dix}}$	$(0.2)_{\text{dix}}$	$[\mathbf{0}]$
$(\mathbf{0}.2)_{\text{dix}}$	$(\mathbf{0}.4)_{\text{dix}}$	$(0.4)_{\text{dix}}$	$[0, \mathbf{0}]$
$(0.4)_{\text{dix}}$	$(\mathbf{0}.8)_{\text{dix}}$	$(0.8)_{\text{dix}}$	$[0, 0, \mathbf{0}]$
$(0.8)_{\text{dix}}$	$(\mathbf{1}.6)_{\text{dix}}$	$(0.6)_{\text{dix}}$	$[0, 0, 0, \mathbf{1}]$
$(0.6)_{\text{dix}}$	$(\mathbf{1}.2)_{\text{dix}}$	$(0.2)_{\text{dix}}$	$[0, 0, 0, 1, \mathbf{1}]$
$(\mathbf{0}.2)_{\text{dix}}$	$(\mathbf{0}.4)_{\text{dix}}$	$(0.4)_{\text{dix}}$	$[0, 0, 0, 1, 1, \mathbf{0}]$
…	…	…	…

Ainsi,

$$(0.1)_{\text{dix}} = (0.000110\ 0110\ 0110 \ldots)_{\text{vf}}.$$

Ce rationnel n'admet pas d'écriture finie en représentation à virgule fixe.

Pour éviter ce comportement, il suffit de borner la taille de la partie décimale calculée (et donc réaliser au plus m tours de boucle).

Représentation IEEE 754

Le codage IEEE 754 consiste à représenter un nombre à virgule x par trois données :

1. un **bit de signe** s ;
2. un **exposant** e ;
3. une **mantisse** m.

Principaux formats (tailles en bits) :

Nom	Taille totale	Bit de signe	Exposant	Mantisse
half	16	1	5	10
single	32	1	8	23
double	64	1	11	52
quad	128	1	15	112

Représentation IEEE 754

Le **bit de signe** s renseigne sur le signe : si $s = 0$, le nombre codé est positif, sinon il est négatif.

L'**exposant** e code un entier en représentation avec biais avec un biais donné par la la table suivante.

Nom	**Biais** B
half	15
single	127
double	1023
quad	16383

On note $\mathrm{vale}(e)$ la valeur ainsi codée.

La **mantisse** m code la partie décimale d'un nombre de la forme $(1.m)_{\mathrm{vf}}$. On note $\mathrm{valm}(m)$ la valeur ainsi codée.

Le nombre à virgule codé par s, e et m est

$$x = (-1)^s \times \mathrm{valm}(m) \times 2^{\mathrm{vale}(e)}.$$

– Exemple –

Codons le nombre $x := (-23.375)_{\text{dix}}$ en single.

1. Comme x est négatif, $s := 1$.
2. On écrit $|x|$ en représentation à virgule fixe. On obtient $|x| = (\texttt{10111.011})_{\text{vf}}$.
   Dans $\mathbf{1}\texttt{0111.011}$, la virgule est placée 4 positions à droite du bit à 1 de poids le plus fort. Ainsi, $|x| = \text{valm}(\texttt{0111011}) \times 2^4$.
3. On en déduit $\text{vale}(e) = 4$ et ainsi, $e = (\texttt{10000011})_{\text{biais}=127}$.
4. On en déduit par ailleurs que $m = \texttt{01110110000000000000000}$.

Finalement, $x = (\texttt{1 10000011 01110110000000000000000})_{\text{IEEE 754 single}}$.

Représentation IEEE 754

– Exemple –

Codons le nombre $x := (0.15625)_{\text{dix}}$ en single.

1. Comme x est positif, $s := 0$.
2. On écrit $|x|$ en représentation à virgule fixe. On obtient $|x| = (0.00101)_{\text{vf}}$.
   Dans $0.00\mathbf{1}01$, la virgule est placée 3 positions à gauche du bit à 1 de poids le plus fort. Ainsi, $|x| = \text{valm}(01) \times 2^{-3}$.
3. On en déduit $\text{vale}(e) = -3$ et ainsi, $e = (01111100)_{\text{biais}=127}$.
4. On en déduit par ailleurs que $m = 01000000000000000000000$.

Finalement, $x = (0\ 01111100\ 01000000000000000000000)_{\text{IEEE 754 single}}$.

Représentation IEEE 754

Il existe plusieurs **représentations spéciales** pour coder certains éléments.

Valeur	s	e	m
Zéro	0	0 ... 0	000 ... 0
+Infini	0	1 ... 1	000 ... 0
−Infini	1	1 ... 1	000 ... 0
NaN	0	1 ... 1	010 ... 0

« NaN » signifie « Not a Number ». C'est un code qui permet de représenter une valeur mal définie provenant d'une opération qui n'a pas de sens (p.ex., $\frac{0}{0}$, $\infty - \infty$ ou $0 \times \infty$).

Le code ASCII

ASCII est l'acronyme de **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange. Ce codage des caractères fut introduit dans les années 1960.

Un caractère occupe **un octet** dont le bit de poids fort vaut 0.

La correspondance octet (en héxa.) / caractère est donnée par la table

	0x	1x	2x	3x	4x	5x	6x	7x
x0	NUL	DLE	esp.	0	@	P	‘	p
x1	SOH	DC1	!	1	A	Q	a	q
x2	STX	DC2	"	2	B	R	b	r
x3	ETX	DC3	#	3	C	S	c	s
x4	EOT	DC4	$	4	D	T	d	t
x5	ENQ	NAK	%	5	E	U	e	u
x6	ACK	SYN	&	6	F	V	f	v
x7	BEL	ETB	’	7	G	W	g	w
x8	BS	CAN	(	8	H	X	h	x
x9	HT	EM	)	9	I	Y	i	y
xA	LF	SUB	*	:	J	Z	j	z
xB	VT	ESC	+	;	K	[	k	{
xC	FF	FS	,	<	L	\	l	|
xD	CR	GS	-	=	M	]	m	}
xE	SO	RS	.	>	N	^	n	~
xF	SI	US	/	?	O	_	o	DEL

Le code ISO 8859-1

Ce codage est également appelé Latin-1 ou Europe occidentale et fut introduit en 1986.

Un caractère occupe **un octet**. La valeur du bit de poids fort n'est plus fixée.

À la différence du code ASCII, ce codage permet en plus de représenter, entre autres, des lettres accentuées.

Ce codage des caractères est aujourd'hui (2021) de moins en moins utilisé.

Le code Unicode

Le codage Unicode est une version encore étendue du code ASCII qui fut introduite en 1991.

Chaque caractère est représenté sur **deux octets** [1].

Il permet ainsi de représenter une large variété de caractères : caractères latins (accentués ou non), grecs, cyrilliques, *etc.*

Il existe des extensions où plus d'octets encore par caractère sont utilisés.

Problème : un texte encodé en Unicode prend plus de place qu'en ASCII.

1. trois dans la version 2.0 de 1996

Le code UTF-8

Le codage UTF-8 apporte une réponse satisfaisante au problème précédent.

Voici comment un caractère c se représente selon le codage UTF-8 :

- si c est un caractère qui peut être représenté par le codage ASCII, alors c est représenté par son code ASCII ;
- sinon, c peut être représenté par le codage Unicode. Il est codé par **deux**, **trois** ou **quatre octets**

où

est la suite des deux octets du code Unicode de c.

Lorsqu'un texte contient principalement des caractères ASCII, son codage est en général moins coûteux en place que le codage Unicode.

De plus, il est **rétro-compatible** avec le codage ASCII.

Textes

Un texte est une suite de caractères.

Chacun des codages de caractères vus précédemment produit un codage de texte. En effet, un texte est représenté par la suite de bits obtenue en remplaçant chacun de ses caractères par son codage.

Un codage de texte est un code si toute suite de bits se décode en au plus un texte (il n'y a pas d'ambiguïté sur l'interprétation d'une suite de bits).

Exercice : vérifier que les codages ASCII, Latin-1, Unicode et UTF-8 sont bien des codes.

Axe 2 : programmation en assembleur

Programmation

Langages bas niveau

Un langage de programmation bas niveau est un langage qui dépend fortement de la structure matérielle de la machine.

Cette caractéristique offre des avantages et des inconvénients divers.

Elle permet, entre autres,

- d'avoir un bon contrôle des **ressources matérielles** ;
- d'avoir un contrôle très fin sur la **mémoire** ;
- souvent, de produire du code dont l'exécution est très **rapide**.

En revanche, elle ne permet pas

- d'utiliser des techniques de programmation abstraites ;
- de programmer rapidement et facilement.

Langage machine

Un langage machine est un langage compris directement par le processeur en vue d'une exécution.

C'est un langage binaire : ses seules lettres sont les bits 0 et 1.

Chaque modèle de processeur possède son propre langage machine. Étant donnés deux modèles de processeurs P_1 et P_2, on dit que P_1 est compatible avec P_2 si toute instruction formulée pour P_2 peut être comprise et exécutée par P_1.

Dans la plupart des langages machine, une instruction commence par un opcode, une suite de bits qui porte la nature de l'instruction. Celui-ci est suivi des suites de bits codant les opérandes de l'instruction.

– Exemple –

La suite

```
01101010  00010101
```

est une instruction dont le opcode est 01101010 et l'opérande est 00010101. Elle ordonne de placer la valeur $(21)_{\text{dix}}$ en tête de la pile.

Langages d'assemblage

Un langage d'assemblage (ou assembleur) est un langage qui se trouve à mi-distance entre le programmeur et la machine en terme de facilité d'accès.

En effet, d'un côté, la machine peut convertir presque immédiatement un programme en assembleur vers du langage machine. De l'autre, l'assembleur est un langage assez facile à manipuler pour un humain.

Les opcodes sont codés via des mnémoniques, mots-clés bien plus manipulables pour le programmeur que les suites binaires associées.

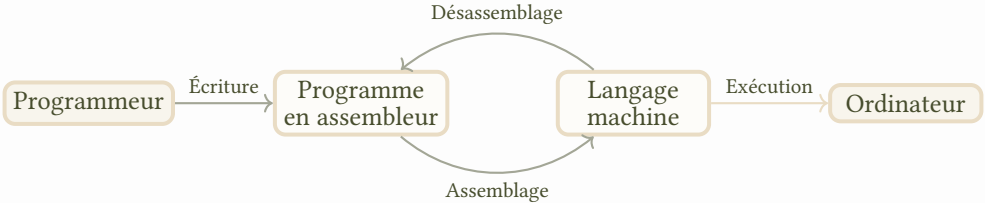
Du fait qu'un langage d'assemblage est spécifiquement dédié à un processeur donné, il existe presque autant de langages d'assemblage qu'il y a de modèles de processeurs.

Langages d'assemblage et assembleurs

L'assemblage est l'action d'un programme nommé **assembleur** qui consiste à traduire un programme en assembleur vers du langage machine.

Le désassemblage, réalisé par un **désassembleur**, est l'opération inverse. Elle permet de retrouver, à partir d'un programme en langage machine, un programme en assembleur qui lui correspond.

Voici le schéma opérationnel liant tout ceci :



L'assembleur NASM

Pour des raisons pédagogiques, nous choisissons de travailler sur une architecture **x86** en 32 bits. C'est une architecture dont le modèle de processeur est compatible avec le modèle Intel 8086.

Nous utiliserons l'**assembleur** NASM (Netwide Assembler).

Pour programmer, il faudra disposer :

1. d'un ordinateur (moderne) ;
2. d'un système Linux en 32 bits (réel ou virtuel) ou 64 bits ;
3. d'un éditeur de textes ;
4. du programme `nasm` (assembleur) ;
5. du programme `ld` (lieur) ou `gcc` ;
6. du programme `gdb` (débogueur), non indispensable à ce stade.

Généralités

Un programme assembleur est un fichier texte d'extension `.asm`.

Il est constitué de plusieurs parties dont le rôle est

1. d'invoquer des **directives** ;
2. de définir des **données initialisées** ;
3. de réserver de la mémoire pour des **données non initialisées** ;
4. de contenir une suite **instructions**.

```
% include io.asm ; directive

section .data
c1: db 'A'      ; donnée initialisée

section .bss
ta: resd 10     ; donnée reservée

section .text
main:
    push c1     ; instruction
    add eax, 4
```

Nous allons étudier chacune de ces parties.

Avant cela, nous avons besoin de nous familiariser avec trois ingrédients de base dans la programmation assembleur : les **valeurs**, les **registres** et la **mémoire**.

Exprimer des valeurs

Il y a plusieurs manières d'exprimer des valeurs en assembleur.

Les **entiers** sont représentés en interne selon le codage en complément à deux. Ils s'écrivent

- directement en base dix, p.ex., `0`, `10020`, `-91` ;
- en hexadécimal, avec le préfixe `0x`, p.ex., `0xA109C`, `-0x98` ;
- en binaire, avec le préfixe `0b`, p.ex., `0b001`, `0b11101`.

Les **caractères** sont représentés en interne par leur code ASCII. Ils s'écrivent

- directement, p.ex., `'a'`, `'9'` ;
- par leur code ASCII, p.ex., `10`, `120`.

Les **chaînes de caractères** sont des suites de caractères. Elles s'écrivent

- directement, p.ex., `'abbaa', 0` ;
- caractère par caractère, p.ex., `'a', 46, 36, 0`.

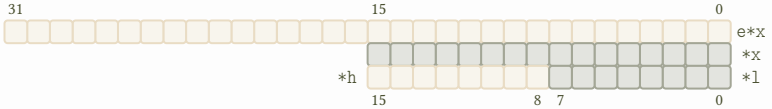
Le code ASCII du marqueur de fin de chaîne est `0` (à ne pas oublier).

Registres et sous-registres

Un registre est un emplacement de 32 bits interne au processeur.

On peut le considérer comme une **variable globale**.

Il y a quatre registres de travail : `eax`, `ebx`, `ecx` et `edx`. Il sont subdivisés en sous-registres selon le schéma suivant :



– Exemple –

`bh` désigne le 2^{e} octet de `ebx` et `cx` désigne les deux 1ers octets de `ecx`.

Il est possible d'écrire / lire dans chacun de ces (sous-)registres.

Attention : toute modification d'un sous-registre entraîne bien évidemment une modification du registre tout entier (et réciproquement).

Registres de diverses sortes

Il existe d'autres registres. Parmi ceux-ci, il y a

- le pointeur d'instruction `eip`, contenant l'adresse de la prochaine instruction à exécuter ;
- le pointeur de tête de pile `esp`, contenant l'adresse de la tête de la pile ;
- le pointeur de pile `ebp`, utilisé pour contenir l'adresse d'une donnée de la pile dans les fonctions ;
- le registre de drapeaux `flags`, utilisé pour contenir des informations sur le résultat d'une opération qui vient d'être réalisée.

Attention : ce ne sont pas des registres de travail, leurs rôles sont fixés. Il est possible pour certains de ces registres d'y écrire ou lire des valeurs explicitement.

L'opération `mov` — définition

L'instruction

```
mov REG, VAL
```

permet de **recopier** la valeur `VAL` dans le (sous-)registre `REG`.

– Exemples –

Voici les effets de quelques instructions :

- `mov eax, 0`

  eax = 

- `mov ebx, 0xFFFFFFFF`

  ebx = 

- `mov eax, 0b101`

  eax = 

- `mov al, 5`

  eax = 

  où le symbole * dénote une valeur non modifiée.

L'opération `mov` — respect des tailles

Il est important, pour que l'instruction `mov REG, VAL` soit correcte, que la taille en octets de la valeur `VAL` soit la même que celle du (sous-)registre `REG`.

– Exemples –

Les instructions suivantes **ne sont pas correctes** :

- `mov al, 0xA5A5` Le sous-registre `al` occupe 1 octet alors que la valeur `0xA5A5` en occupe 2.
- `mov ax, ebx` Le sous-registre `ax` occupe 2 octets alors que la valeur contenue dans `ebx` en occupe 4.
- `mov eax, cx` Le sous-registre `eax` occupe 4 octets alors que la valeur contenue dans `cx` en occupe que 2.

– Exemples –

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

- `mov al, 0xA5` La valeur `0xA5` occupe bien 1 octet, tout comme `al`.
- `mov ax, 0xF` Le sous-registre `ax` occupe 2 octets alors que la valeur `0xF` n'en occupe que 1. Néanmoins, cette valeur est étendue sur 2 octets sans perte d'information en `0x000F`.

Opérations sur les registres — arithmétique

Opérations **arithmétiques** :

- `add REG, VAL` incrémente le (sous-)registre `REG` de la valeur `VAL` ;
- `sub REG, VAL` décrémente le (sous-)registre `REG` de la valeur `VAL` ;
- `mul REG` multiplie la valeur contenue dans `eax` et le (sous-)registre `REG` et place le résultat dans `edx:eax`. Cette notation désigne la suite de 64 bits dont les bits de `edx` sont ceux de poids forts et ceux de `eax` ceux de poids faibles) ;
- `div REG` place le quotient de la division de `edx:eax` par la valeur portée par le (sous-)registre `REG` dans `eax` et le reste dans `edx`.

– Exemple –

```
mov eax, 20   ; eax = 20
add eax, 51   ; eax = 71
add eax, eax   ; eax = 142

mov ebx, 3   ; ebx = 3
mul ebx   ; eax = 426

mov edx, 0   ; edx = 0
sub ebx, 1   ; ebx = 2
div ebx   ; eax = 213
```

Opérations sur les registres — logique

Opérations **logiques** :

- `not REG` place dans le (sous-)registre `REG` la valeur obtenue en réalisant le *non* bit à bit de sa valeur ;
- `and REG, VAL` place dans le (sous-)registre `REG` la valeur du *et* logique bit à bit entre les suites contenues dans `REG` et `VAL` ;
- `or REG, VAL` place dans le (sous-)registre `REG` la valeur du *ou* logique bit à bit entre les suites contenues dans `REG` et `VAL` ;
- `xor REG, VAL` place dans le (sous-)registre `REG` la valeur du *ou exclusif* logique bit à bit entre les suites contenues dans `REG` et `VAL`.

Opérations sur les registres — bit à bit

Opérations **bit à bit** :

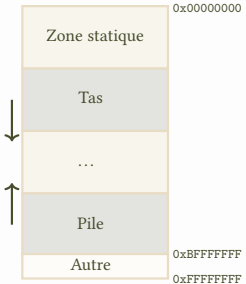
- `shl REG, NB` décale les bits du (sous-)registre `REG` à gauche de `NB` places et complète à droite par des `0` ;
- `shr REG, NB` décale les bits du (sous-)registre `REG` à droite de `NB` places et complète à gauche par des `0` ;
- `rol REG, NB` réalise une rotation des bits du (sous-)registre `REG` à gauche de `NB` places ;
- `ror REG, NB` réalise une rotation des bits du (sous-)registre `REG` à droite de `NB` places.

Mémoire

Les registres n'offrent pas assez de mémoire pour construire des programmes élaborés. C'est pour cette raison que l'on utilise la **mémoire**.

La mémoire est segmentée en plusieurs parties :

- la zone statique qui contient le code et les données statiques ;
- le tas, de taille variable au fil de l'exécution ;
- la pile, de taille variable au fil de l'exécution.



Il y a d'autres zones (non représentées ici).

En mode protégé, chaque programme en exécution possède son propre environnement de mémoire. Les adresses y sont relatives et non absolues.

De cette manière, un programme en exécution ne peut empiéter sur la mémoire d'un autre.

La lecture / écriture en mémoire suit la convention *little-endian*.

Lecture en mémoire

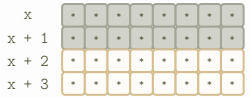
L'instruction

```
mov REG, [ADR]
```

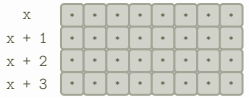
place dans le (sous-)registre `REG` un, deux ou quatre octets en fonction de la taille de `REG`, **lus** à partir de l'adresse `ADR` **dans la mémoire**.

En supposant que `x` soit une adresse accessible en mémoire, les parties foncées sont celles qui sont lues et placées dans le (sous-)registre opérande de l'instruction :

`mov ah, [x]` ou `mov al, [x]`



`mov ax, [x]`



`mov eax, [x]`

Lecture en mémoire

– Exemple –

Observons l'effet des instructions suivantes sur `eax` 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, 0
mov al, [x]
mov ah, [x + 1]
mov ax, [x]
mov eax, [x]
mov ah, [x + 2]
mov ax, [x + 2]
```

Écriture en mémoire

L'instruction

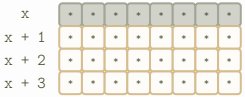
```
mov DT [ADR], VAL
```

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

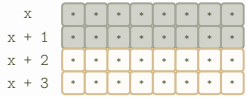
Le champ `DT` est un descripteur de taille qui permet de préciser la taille de `VAL` :

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

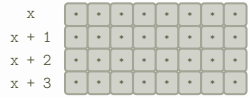
Si `x` est une adresse accessible en mémoire et `val` une valeur, les parties foncées de la mémoire sont celles qui sont modifiées :



`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.