Représentation IEEE 754

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

Représentation IEEE 754

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

1. Comme x est positif, $s := 0$.

Représentation IEEE 754

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

$$|x| = (0.00101)_{\text{vf}}$$

et donc,

$$|x| = \text{valm}(01) \times 2^{-3}.$$

Représentation IEEE 754

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
$$|x| = (0.00101)_{\text{vf}}$$
et donc,
$$|x| = \text{valm}(01) \times 2^{-3}.$$
3. On en déduit $\text{vale}(e) = -3$ et ainsi,
$$e = (01111100)_{\text{biais}=127}.$$

Représentation IEEE 754

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
$$|x| = (0.00101)_{\text{vf}}$$
et donc,
$$|x| = \text{valm}(01) \times 2^{-3}.$$
3. On en déduit $\text{vale}(e) = -3$ et ainsi,
$$e = (01111100)_{\text{biais}=127}.$$
4. On en déduit par ailleurs que
$$m = 01000000000000000000000.$$

Représentation IEEE 754

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
$$|x| = (0.00101)_{\text{vf}}$$
et donc,
$$|x| = \text{valm}(01) \times 2^{-3}.$$
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	0...0
+Infini	0	1...1	0...0
−Infini	1	1...1	0...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 $0 \times \infty$).

Plan

Le code ASCII

ASCII est l'acronyme de American Standard Code for Information Interchange. Ce codage des caractères fut introduit dans les années 1960.

Le code ASCII

ASCII est l'acronyme de American Standard Code for Information Interchange. Ce codage des caractères fut introduit dans les années 1960.

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

Le code ASCII

ASCII est l'acronyme de American Standard Code for Information Interchange. 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.

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.

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.

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 (2016) de moins en moins utilisé.

Le code Unicode

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

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**.

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**.

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

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**.

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.

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**.

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.

Le code UTF-8

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

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 ;

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 **trois octets**

où

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

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 **trois 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.

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.

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).

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.

Plan

Plan

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.

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.

P.ex., elle permet

- 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**.

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.

P.ex., elle permet

- 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

Le langage machine est un langage compris directement par un processeur donné en vue d'une exécution.

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

Langage machine

Le langage machine est un langage compris directement par un processeur donné 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 x_1 et x_2, on dit que x_1 est **compatible** avec x_2 si toute instruction formulée pour x_2 peut être comprise et exécutée par x_1.

Langage machine

Le langage machine est un langage compris directement par un processeur donné 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 x_1 et x_2, on dit que x_1 est **compatible** avec x_2 si toute instruction formulée pour x_2 peut être comprise et exécutée par x_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.

Langage machine

Le langage machine est un langage compris directement par un processeur donné 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 x_1 et x_2, on dit que x_1 est **compatible** avec x_2 si toute instruction formulée pour x_2 peut être comprise et exécutée par x_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.

P.ex., 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.

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.

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.

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.

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.

Langages d'assemblage et assembleurs

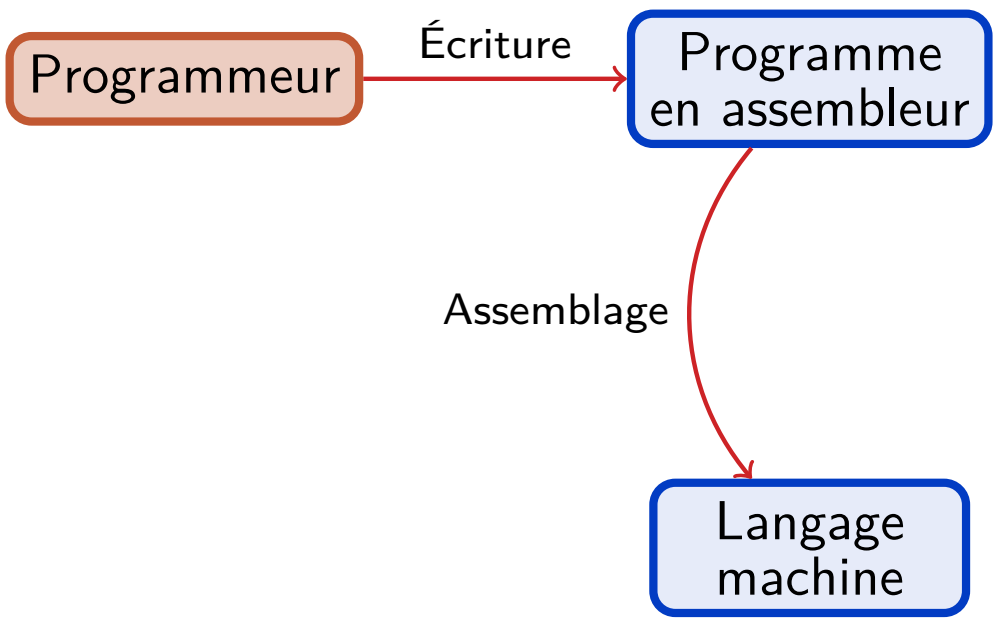
L'assemblage est l'action d'un programme nommé **assembleur** qui consiste à traduire un programme en assembleur vers du langage machine.

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.

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.

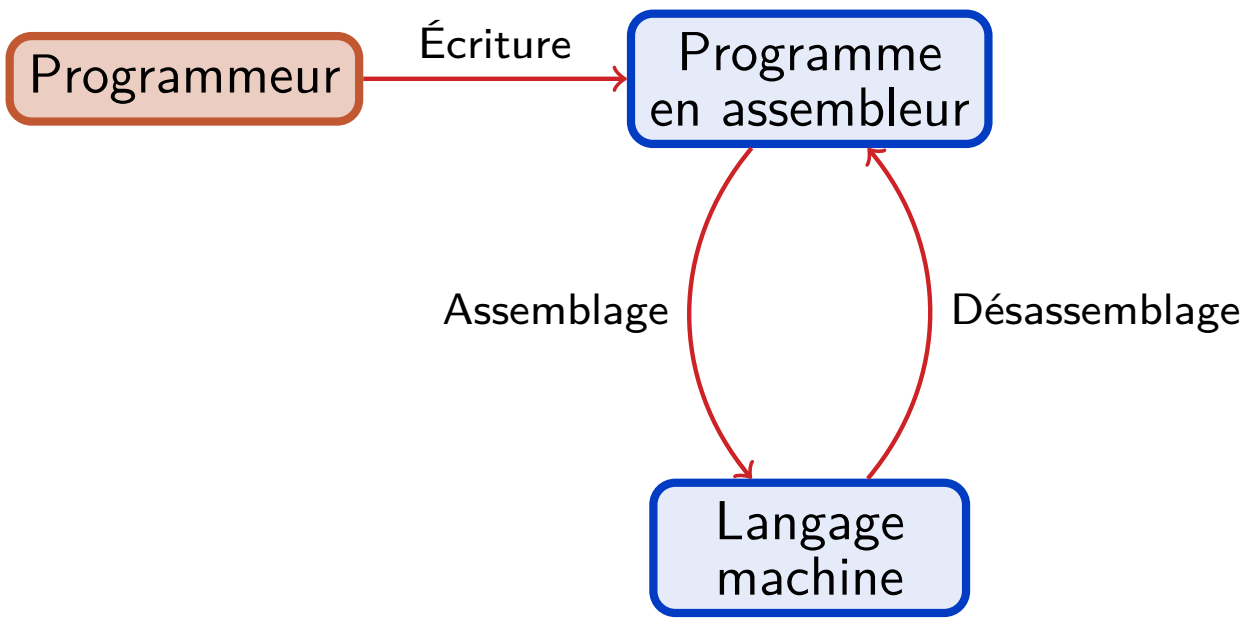
Le désassemblage, réalisé par un **désassembleur**, est l'opération inverse. Elle permet de retrouver, à partir d'un programme en langage machine, un programme en assembleur qui lui correspond.



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.

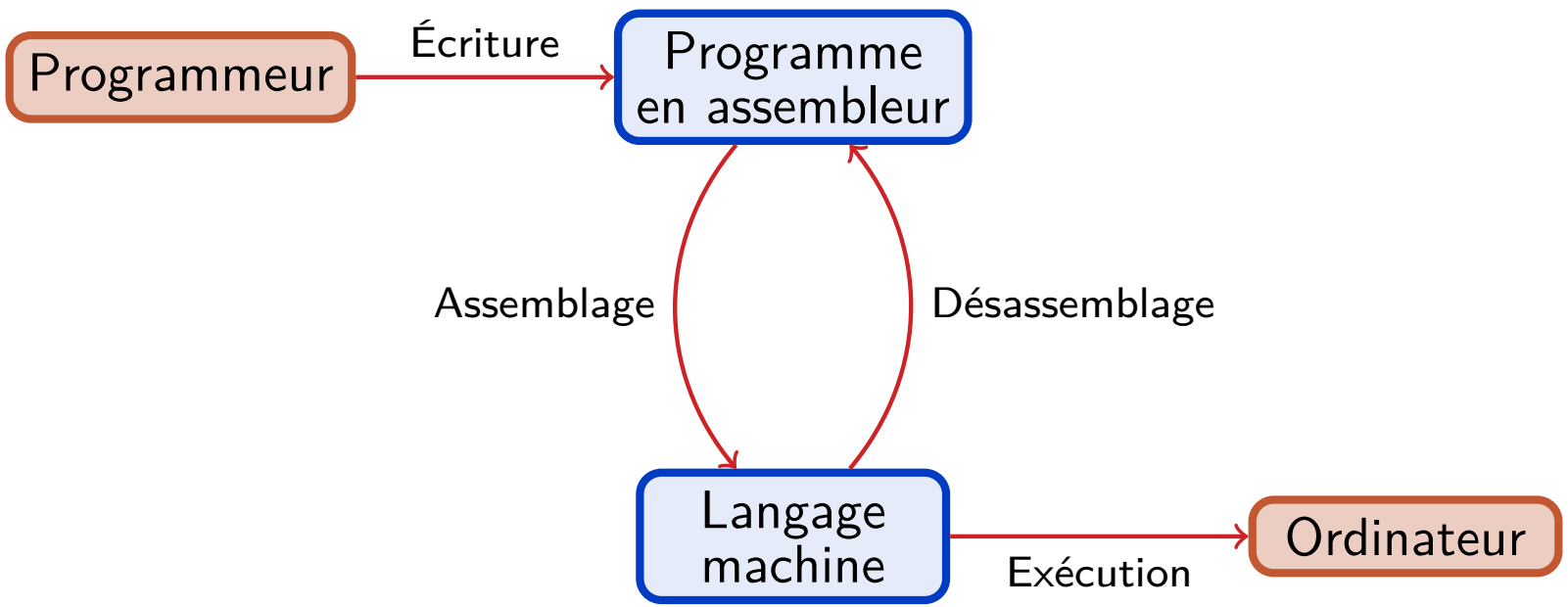
Le désassemblage, réalisé par un **désassembleur**, est l'opération inverse. Elle permet de retrouver, à partir d'un programme en langage machine, un programme en assembleur qui lui correspond.



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.



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.

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).

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).

Généralités

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

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**.

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**.

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**.

Plan

Exprimer des valeurs

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

Exprimer des valeurs

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

On peut exprimer des entiers (représentés par leurs suites de bits obtenues en repr. complément à deux) :

- 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`.

Exprimer des valeurs

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

On peut exprimer des entiers (représentés par leurs suites de bits obtenues en repr. complément à deux) :

- 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`.

On peut exprimer des caractères (en repr. ASCII) :

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

Exprimer des valeurs

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

On peut exprimer des entiers (représentés par leurs suites de bits obtenues en repr. complément à deux) :

- 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`.

On peut exprimer des caractères (en repr. ASCII) :

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

On peut exprimer des chaînes de caractères (comme suites de carac.) :

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

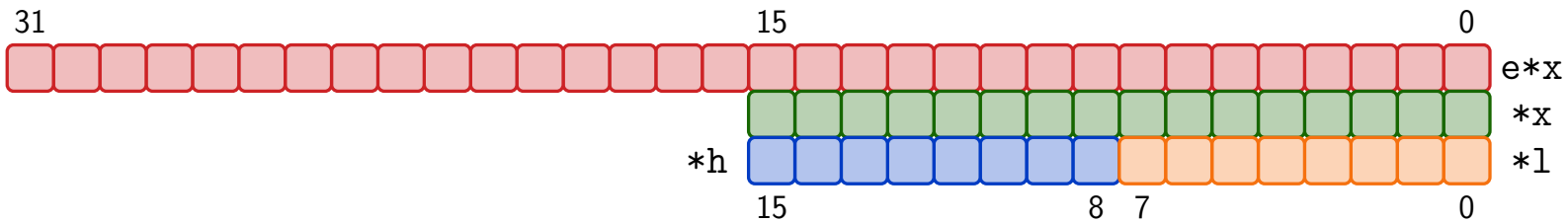
On peut le considérer comme une **variable globale**.

Registres et sous-registres

Un registre est un emplacement de 32 bits.

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 :



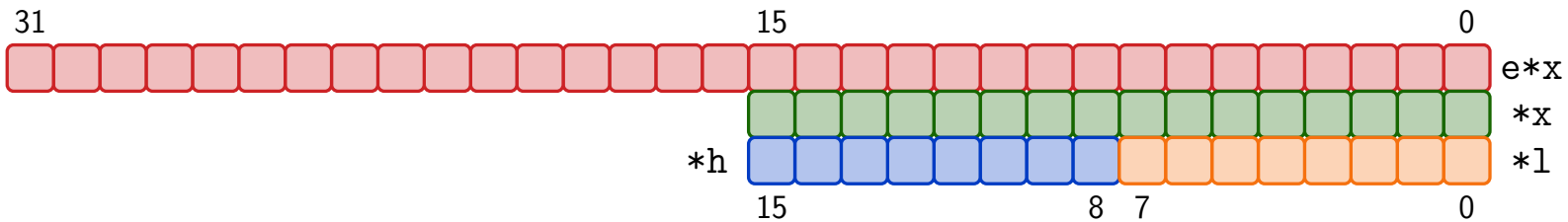
P.ex., `bh` désigne le 2^{e} octet de `ebx` et `cx` désigne les deux 1ers octets de `ecx`.

Registres et sous-registres

Un registre est un emplacement de 32 bits.

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 :



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

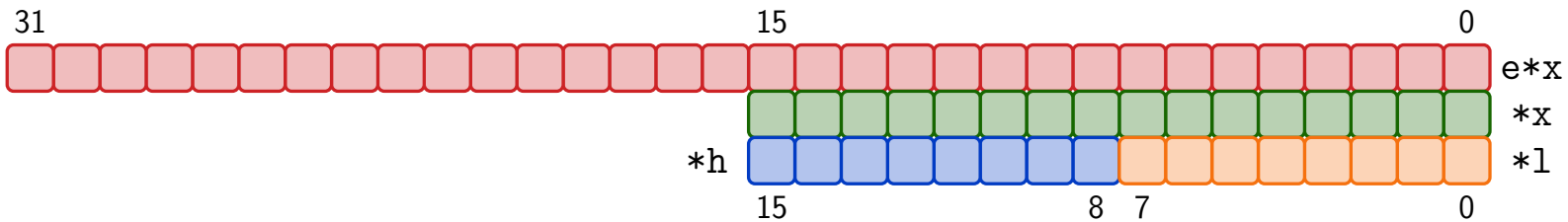
Il est possible d'écrire/lire dans chacun de ces (sous-)registres.

Registres et sous-registres

Un registre est un emplacement de 32 bits.

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 :



P.ex., `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 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 ;

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 ;

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 ;

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.

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, leur rôle est fixé. Même s'il est possible pour certains d'y écrire / lire explicitement, il faut essayer, pour la plupart, de le faire le moins possible.

L'opération `mov` — définition

L'instruction

```
mov REG, VAL
```

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

L'opération `mov` — définition

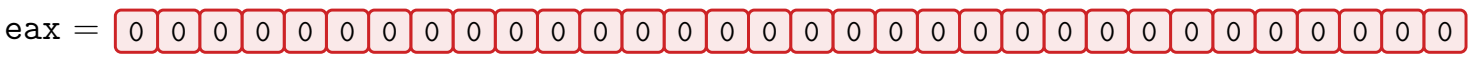
L'instruction

```
mov REG, VAL
```

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

P.ex., voici les effets de quelques instructions :

- `mov eax, 0`



L'opération `mov` — définition

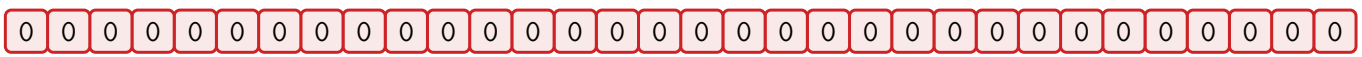
L'instruction

```
mov REG, VAL
```

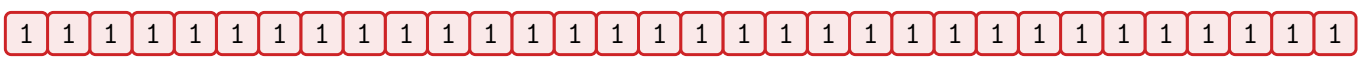
permet de **recopier** la valeur `VAL` dans le (sous-)registre `REG`.

P.ex., voici les effets de quelques instructions :

- `mov eax, 0`

  eax = 

- `mov ebx, 0xFFFFFFFF`

  ebx = 

L'opération `mov` — définition

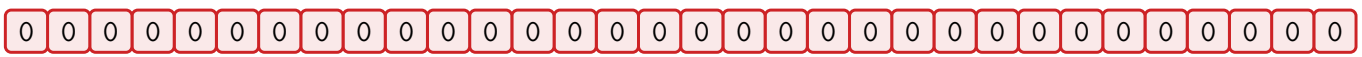
L'instruction

```
mov REG, VAL
```

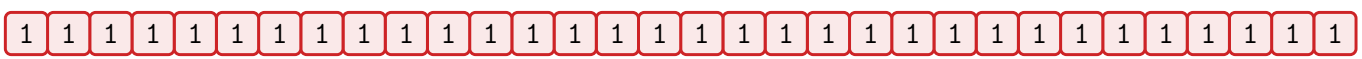
permet de **recopier** la valeur `VAL` dans le (sous-)registre `REG`.

P.ex., voici les effets de quelques instructions :

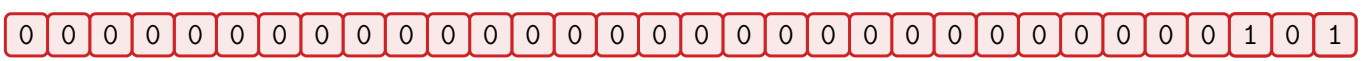
- `mov eax, 0`

  eax = 

- `mov ebx, 0xFFFFFFFF`

  ebx = 

- `mov eax, 0b101`

  eax = 

L'opération `mov` — définition

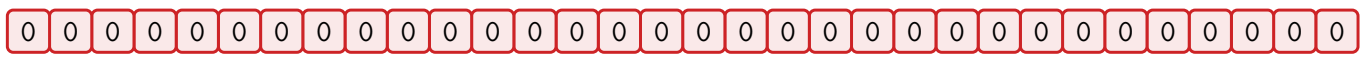
L'instruction

```
mov REG, VAL
```

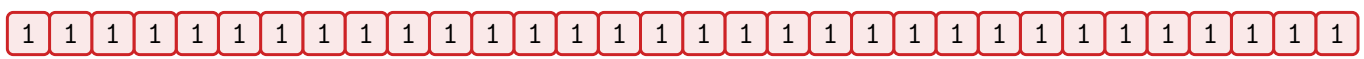
permet de **recopier** la valeur `VAL` dans le (sous-)registre `REG`.

P.ex., voici les effets de quelques instructions :

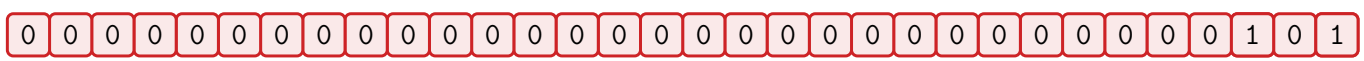
- `mov eax, 0`

  eax = 

- `mov ebx, 0xFFFFFFFF`

  ebx = 

- `mov eax, 0b101`

  eax = 

- `mov al, 5` Le symbole * dénote une valeur non modifiée.

  eax = 

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`.

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`.

P.ex., les instructions suivantes **ne sont pas correctes** :

- `mov al, 0xA5A5`

  Le ss-registre `al` occupe 1 octet alors que la valeur `0xA5A5` en occupe 2.

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`.

P.ex., les instructions suivantes **ne sont pas correctes** :

- `mov al, 0xA5A5`

  Le ss-registre `al` occupe 1 octet alors que la valeur `0xA5A5` en occupe 2.

- `mov ax, eax`

  Le ss-registre `ax` occupe 2 octets alors que la valeur contenue dans `eax` en occupe 4.

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`.

P.ex., les instructions suivantes **ne sont pas correctes** :

- `mov al, 0xA5A5`

  Le ss-registre `al` occupe 1 octet alors que la valeur `0xA5A5` en occupe 2.

- `mov ax, eax`

  Le ss-registre `ax` occupe 2 octets alors que la valeur contenue dans `eax` en occupe 4.

- `mov eax, ax`

  Le ss-registre `eax` occupe 4 octets alors que la valeur contenue dans `ax` en occupe que 2.

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`.

P.ex., les instructions suivantes **ne sont pas correctes** :

- `mov al, 0xA5A5`

  Le ss-registre `al` occupe 1 octet alors que la valeur `0xA5A5` en occupe 2.

- `mov ax, eax`

  Le ss-registre `ax` occupe 2 octets alors que la valeur contenue dans `eax` en occupe 4.

- `mov eax, ax`

  Le ss-registre `eax` occupe 4 octets alors que la valeur contenue dans `ax` en occupe que 2.

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

- `mov al, 0xA5`

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`.

P.ex., les instructions suivantes **ne sont pas correctes** :

- `mov al, 0xA5A5`

  Le ss-registre `al` occupe 1 octet alors que la valeur `0xA5A5` en occupe 2.

- `mov ax, eax`

  Le ss-registre `ax` occupe 2 octets alors que la valeur contenue dans `eax` en occupe 4.

- `mov eax, ax`

  Le ss-registre `eax` occupe 4 octets alors que la valeur contenue dans `ax` en occupe que 2.

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

- `mov al, 0xA5`

- `mov ax, 0xF`

  Le ss-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` ;

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` ;

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 VAL`
  multiplie la valeur contenue dans `eax` et `VAL` et place le résultat dans `edx:eax` (c.-à-d. la suite de 64 bits dont les bits de `edx` sont ceux de poids forts et ceux de `eax` ceux de poids faibles) ;

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 VAL`
  multiplie la valeur contenue dans `eax` et `VAL` et place le résultat dans `edx:eax` (c.-à-d. la suite de 64 bits dont les bits de `edx` sont ceux de poids forts et ceux de `eax` ceux de poids faibles) ;
- `div VAL`
  place le quotient de la division de `edx:eax` par la valeur `VAL` dans `eax` et le reste dans `edx`.

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 VAL`
  multiplie la valeur contenue dans `eax` et `VAL` et place le résultat dans `edx:eax` (c.-à-d. la suite de 64 bits dont les bits de `edx` sont ceux de poids forts et ceux de `eax` ceux de poids faibles) ;
- `div VAL`
  place le quotient de la division de `edx:eax` par la valeur `VAL` dans `eax` et le reste dans `edx`.

P.ex.,

```
mov eax, 20
add eax, 51
add eax, eax
```

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 VAL`
  multiplie la valeur contenue dans `eax` et `VAL` et place le résultat dans `edx:eax` (c.-à-d. la suite de 64 bits dont les bits de `edx` sont ceux de poids forts et ceux de `eax` ceux de poids faibles) ;
- `div VAL`
  place le quotient de la division de `edx:eax` par la valeur `VAL` dans `eax` et le reste dans `edx`.

P.ex.,

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

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 ;

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` ;

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` ;

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` ;

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` ;

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 ;

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`

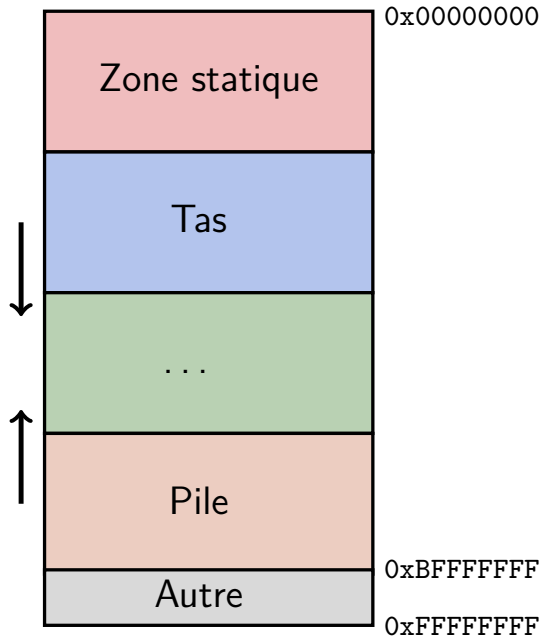
  réalise une rotation des bits du (sous-)registre `REG` à gauche de `NB` places ;

- `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.

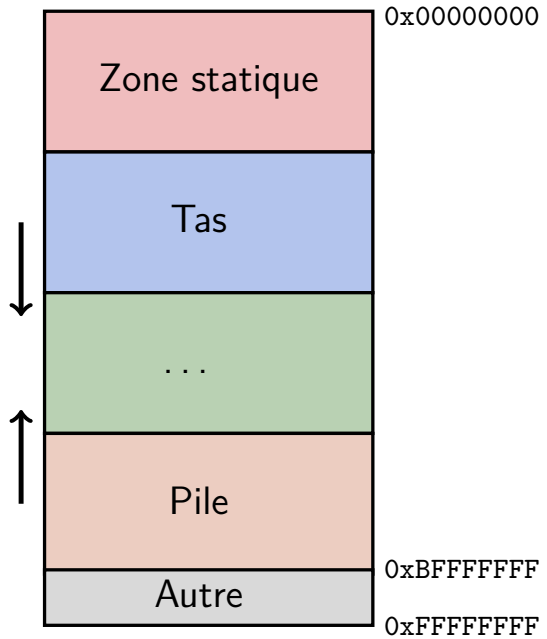


Mémoire

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 ;

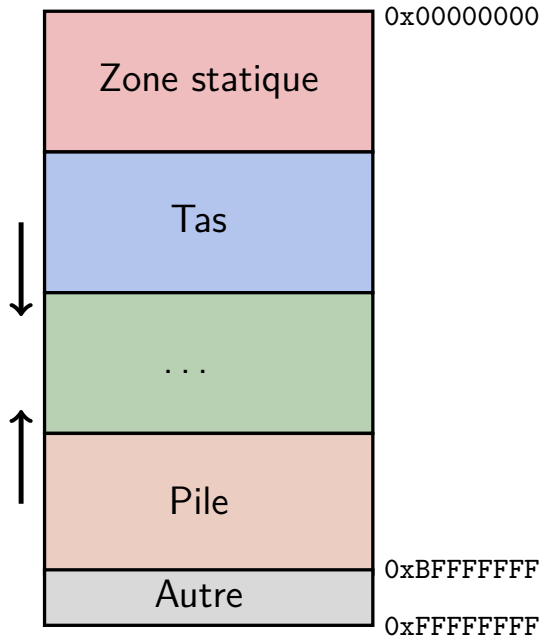


Mémoire

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 ;

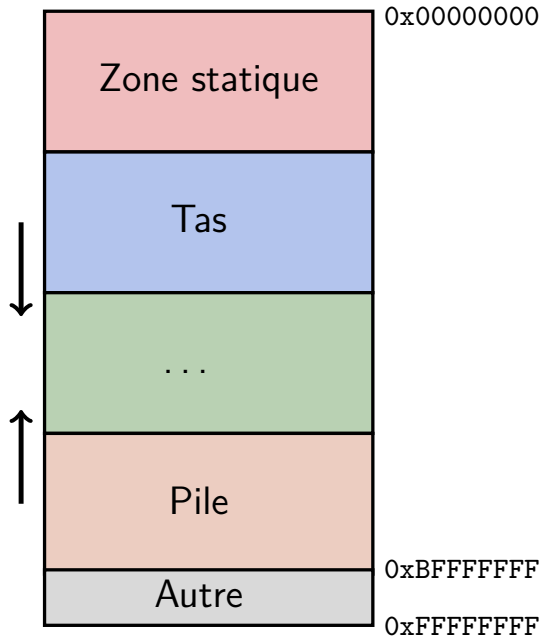


Mémoire

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.



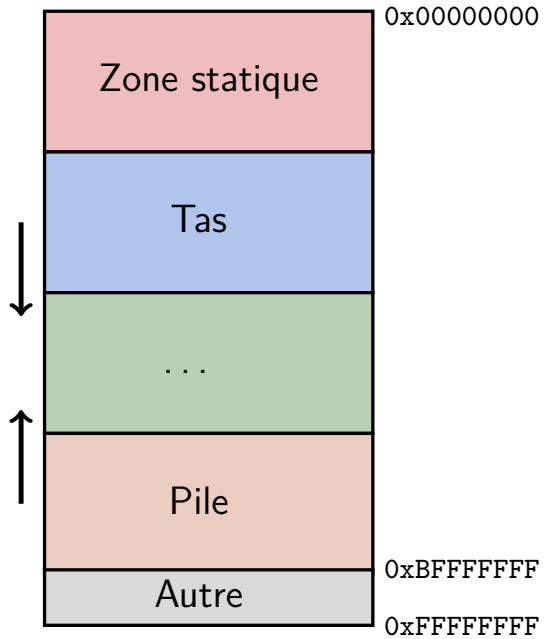
Mémoire

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 repr. ici).



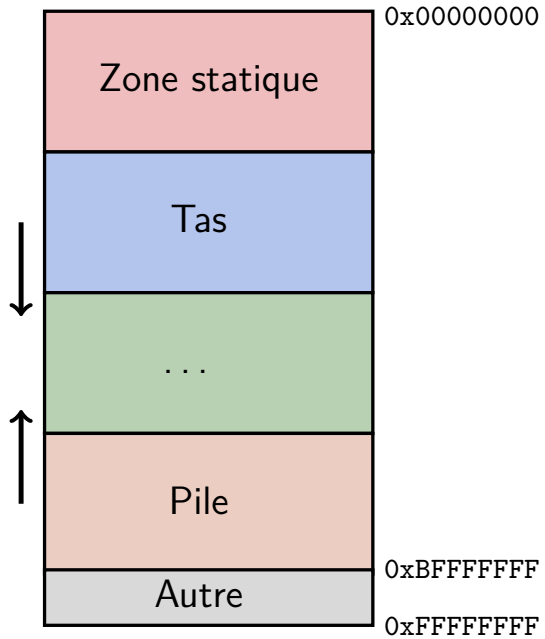
Mémoire

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 repr. 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.

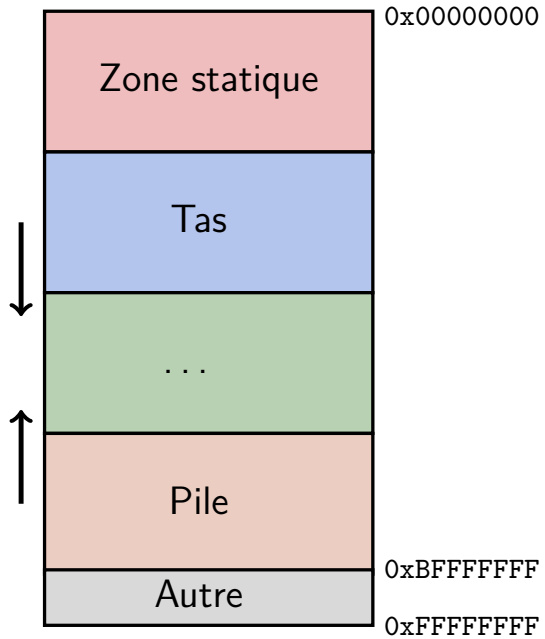
Mémoire

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 repr. 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**.

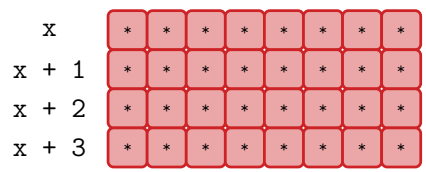
Lecture en mémoire

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 rouges sont celles qui sont lues et placées dans le (sous-)registre opérande de l'instruction :




```
mov eax, [x]
```

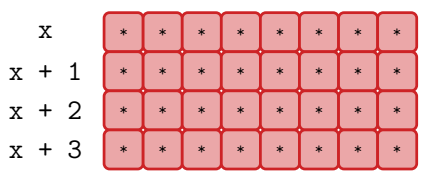
Lecture en mémoire

L'instruction

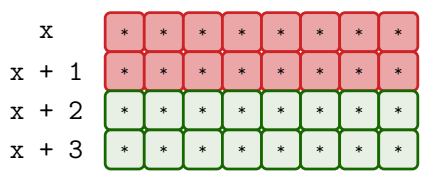
```
mov REG, [ADR]
```

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 rouges sont celles qui sont lues et placées dans le (sous-)registre opérande de l'instruction :




```
mov eax, [x]
```




```
mov ax, [x]
```

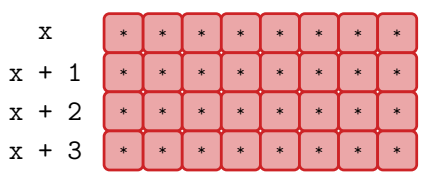
Lecture en mémoire

L'instruction

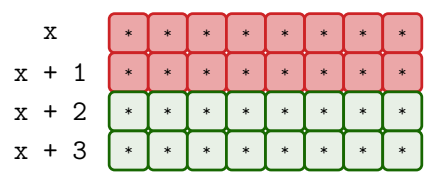
```
mov REG, [ADR]
```

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**.

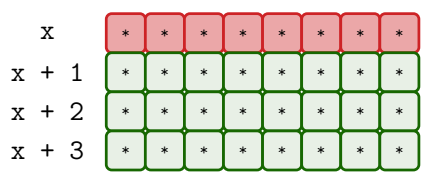
En supposant que `x` soit une adresse accessible en mémoire, les parties rouges sont celles qui sont lues et placées dans le (sous-)registre opérande de l'instruction :




```
mov eax, [x]
```




```
mov ax, [x]
```



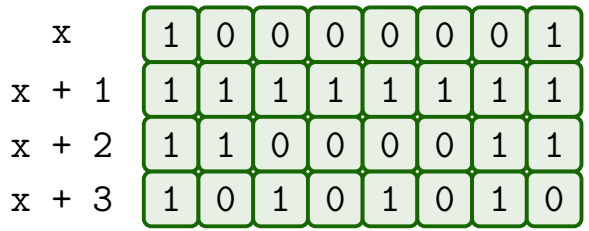

```
mov ah, [x]
```

ou

```
mov al, [x]
```

Lecture en mémoire — exemple

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




```
mov eax, 0
mov al, [x]
mov ah, [x + 1]
mov ax, [x]
mov eax, [x]
mov ah, [x + 2]
mov ax, [x + 2]
```

Lecture 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, 0
mov al, [x]
mov ah, [x + 1]
mov ax, [x]
mov eax, [x]
mov ah, [x + 2]
mov ax, [x + 2]
```

`eax` = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Lecture 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, 0
mov al, [x]
mov ah, [x + 1]
mov ax, [x]
mov eax, [x]
mov ah, [x + 2]
mov ax, [x + 2]
```

eax = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

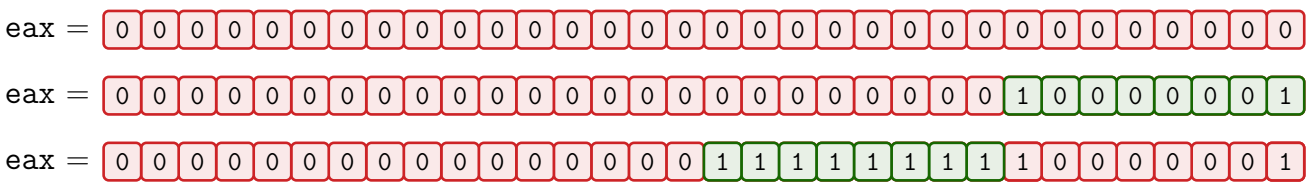
eax = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1

Lecture 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, 0
mov al, [x]
mov ah, [x + 1]
mov ax, [x]
mov eax, [x]
mov ah, [x + 2]
mov ax, [x + 2]
```

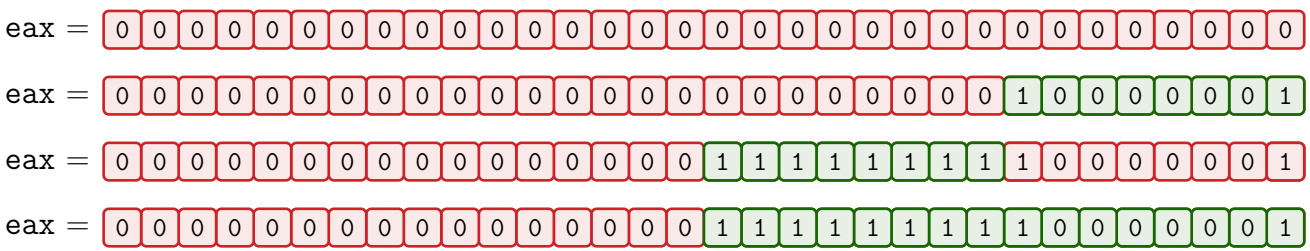


Lecture en mémoire — exemple

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, 0
mov al, [x]
mov ah, [x + 1]
mov ax, [x]
mov eax, [x]
mov ah, [x + 2]
mov ax, [x + 2]
```



Lecture 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, 0
mov al, [x]
mov ah, [x + 1]
mov ax, [x]
mov eax, [x]
mov ah, [x + 2]
mov ax, [x + 2]
```

eax = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

eax = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1

eax = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 1

eax = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 1

eax = 1 0 1 0 1 0 1 0 1 1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 1

Lecture 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, 0
mov al, [x]
mov ah, [x + 1]
mov ax, [x]
mov eax, [x]
mov ah, [x + 2]
mov ax, [x + 2]
```

eax = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

eax = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1

eax = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 1

eax = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 1

eax = 1 0 1 0 1 0 1 0 1 1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 1

eax = 1 0 1 0 1 0 1 0 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 0 0 0 0 0 0 1

Lecture 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, 0
mov al, [x]
mov ah, [x + 1]
mov ax, [x]
mov eax, [x]
mov ah, [x + 2]
mov ax, [x + 2]
```

eax = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

eax = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1

eax = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 1

eax = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 1

eax = 1 0 1 0 1 0 1 0 1 1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 1

eax = 1 0 1 0 1 0 1 0 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 0 0 0 0 0 0 1

eax = 1 0 1 0 1 0 1 0 1 1 0 0 0 0 1 1 1 0 1 0 1 0 1 0 1 1 0 0 0 0 1 1

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` en octet selon la table suivante :

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

Écriture en mémoire

L'instruction

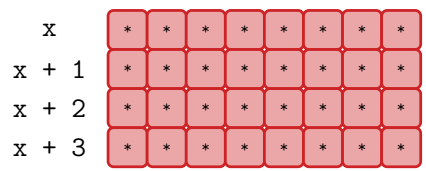
```
mov DT [ADR], VAL
```

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` en octet selon la table suivante :

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

En supposant que `x` soit une adresse accessible en mémoire et `val` une valeur, les parties rouges de la mémoire sont celles qui sont modifiées :




```
mov dword [x], val
```

Écriture en mémoire

L'instruction

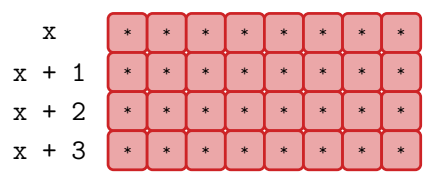
```
mov DT [ADR], VAL
```

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

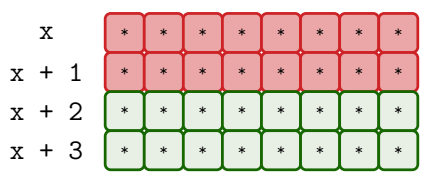
Le champ `DT` est un descripteur de taille qui permet de préciser la taille de `VAL` en octet selon la table suivante :

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

En supposant que `x` soit une adresse accessible en mémoire et `val` une valeur, les parties rouges de la mémoire sont celles qui sont modifiées :



`mov dword [x], val`



`mov word [x], val`

Écriture en mémoire

L'instruction

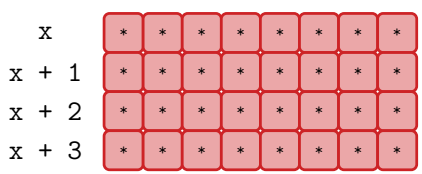
```
mov DT [ADR], VAL
```

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

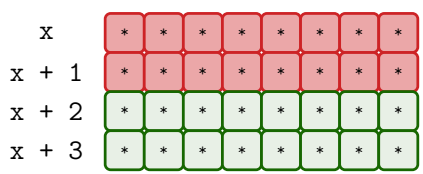
Le champ `DT` est un descripteur de taille qui permet de préciser la taille de `VAL` en octet selon la table suivante :

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

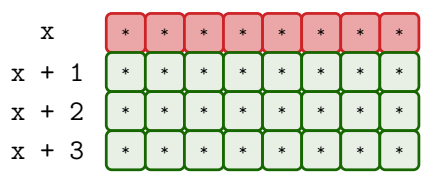
En supposant que `x` soit une adresse accessible en mémoire et `val` une valeur, les parties rouges de la mémoire sont celles qui sont modifiées :



`mov dword [x], val`



`mov word [x], val`



`mov byte [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).

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).

P.ex., 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).

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).

P.ex., 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).

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).

P.ex., 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).

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).

P.ex., 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

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

- `mov [x], eax`

  Le registre `eax` occupe implicitement 4 octets.

Écriture en mémoire — tailles

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.

Écriture en mémoire — tailles

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.

Écriture en mémoire — tailles

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.

Écriture en mémoire — tailles

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

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
```

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

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
```

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

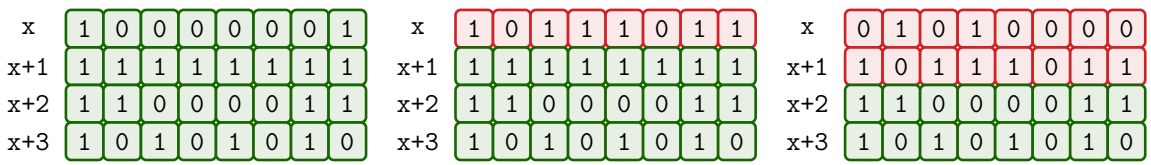
x	1	0	1	1	1	0	1	1
x+1	1	1	1	1	1	1	1	1
x+2	1	1	0	0	0	0	1	1
x+3	1	0	1	0	1	0	1	0

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
```



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
```

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

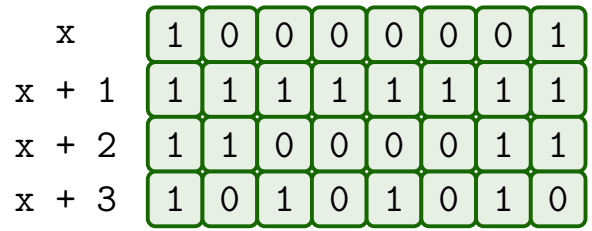
x	1	0	1	1	1	0	1	1
x+1	1	1	1	1	1	1	1	1
x+2	1	1	0	0	0	0	1	1
x+3	1	0	1	0	1	0	1	0

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

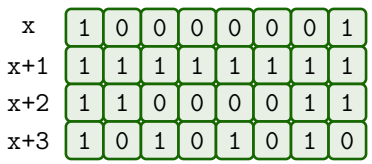
x	0	1	0	1	0	0	0	0
x+1	1	0	1	1	1	0	1	1
x+2	0	0	0	1	0	0	0	1
x+3	1	0	1	0	1	0	1	0

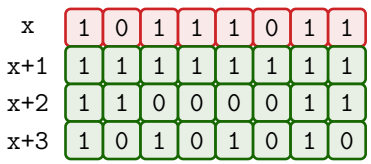
Écriture en mémoire — exemple

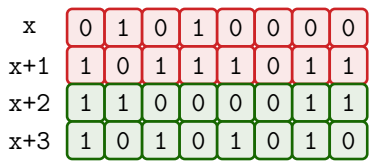
Observons l'effet des instructions avec la mémoire dans l'état suivant :

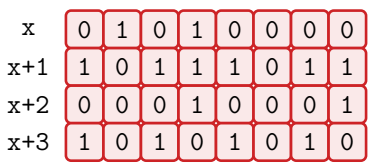


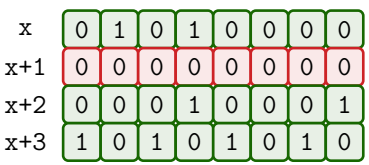

```
mov eax, 0xAA11BB50
mov [x], ah
mov [x], ax
mov [x], eax
mov byte [x + 1], 0
```









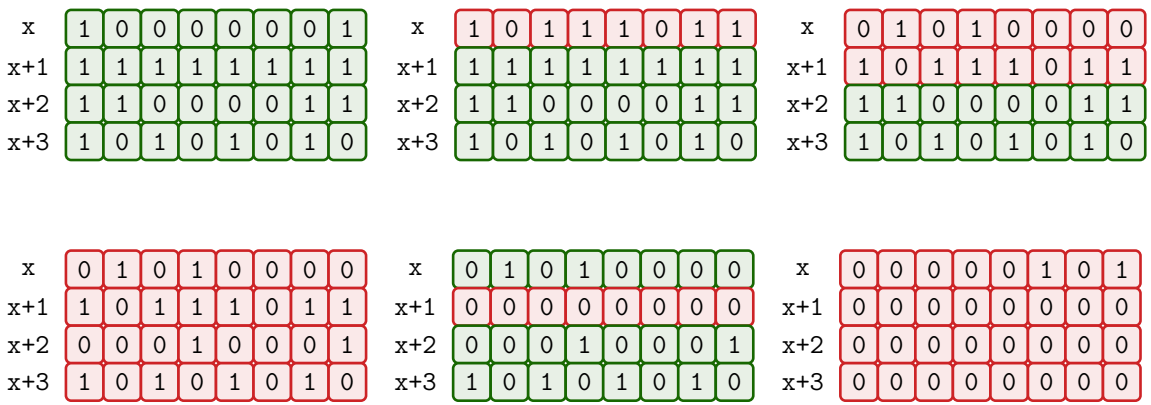


Écriture en mémoire — exemple

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

Section de données initialisées

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

Elle commence par `section .data`.

Section de données initialisées

La section `.data` 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 les suivants :

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`.

Section de données initialisées

La section `.data` 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 les suivants :

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

Quelques exemples de définitions de données initialisées :

- `entier: dw 55`

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

Section de données initialisées — exemples

Quelques exemples de définitions de données initialisées :

- `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 $(000\texttt{FFE05})_{\text{hex}}$.

Section de données initialisées — exemples

Quelques exemples de définitions de données initialisées :

- `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 $(000\text{FFE05})_{\text{hex}}$.

- `y: db 0b11001100`

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

Section de données initialisées — exemples

Quelques exemples de définitions de données initialisées :

- `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 $(000\text{FFE05})_{\text{hex}}$.

- `y: db 0b11001100`

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

- `c: db 'a'`

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

Section de données initialisées — exemples

Quelques exemples de définitions de données initialisées :

- `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 $(000\text{FFE05})_{\text{hex}}$.

- `y: db 0b11001100`

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

- `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.

Section de données initialisées — exemples

Quelques exemples de définitions de données initialisées :

- `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 $(11001100)_{\text{deux}}$.

- `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'`.