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.

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`.

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`.

P.ex.,

- `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 $(5)_{\text{dix}}$.

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`.

P.ex.,

- `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 1^{er} double mot est `suite`.

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`.

P.ex.,

- `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)`.

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`.

P.ex.,

- `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'`.

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`.

P.ex.,

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

Section de données non initialisées

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

Elle commence par `section .bss`.

Section de données non initialisées

La section `.bss` 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 les suivants :

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

P.ex., l'instruction

```
x: resw 120
```

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

Section de données non initialisées — exemples

P.ex., l'instruction
`x: resw 120`

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

L'adresse de la $\texttt{i}^{\text{e}}$ donnée à partir de `x` est `x + ((i - 1) * 2)`.

Section de données non initialisées — exemples

P.ex., l'instruction
`x: resw 120`

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

L'adresse de la $\texttt{i}^{\text{e}}$ donnée à partir de `x` est `x + ((i - 1) * 2)`.

Pour écrire la valeur $(\texttt{0xEF01})_{\text{hex}}$ en 4^{e} position, on utilise l'instruction
`mov word [x + (3 * 2)], 0xEF01`

Section de données non initialisées — exemples

P.ex., l'instruction
`x: resw 120`

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

L'adresse de la $\texttt{i}^{\text{e}}$ donnée à partir de `x` est `x + ((i - 1) * 2)`.

Pour écrire la valeur $(\texttt{0xEF01})_{\text{hex}}$ en 4^{e} position, on utilise l'instruction
`mov word [x + (3 * 2)], 0xEF01`

Pour lire la valeur située en 7^{e} position à partir de `x`, on utilise les instructions
```
mov eax, 0
mov ax, [x + (6 * 2)]
```

Section de données non initialisées — exemples

P.ex., l'instruction
`x: resw 120`
réserve, à partir de l'adresse `x`, une suite de 120×2 octets non initialisés.

L'adresse de la $\texttt{i}^{\text{e}}$ donnée à partir de `x` est `x + ((i - 1) * 2)`.

Pour écrire la valeur $(\texttt{0xEF01})_{\text{hex}}$ en 4^{e} position, on utilise l'instruction
`mov word [x + (3 * 2)], 0xEF01`

Pour lire la valeur située en 7^{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 .text est la partie du programme qui regroupe les instructions.

Elle commence par `section .text`.

Section d'instructions

La section .text 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.

Section d'instructions

La section .text 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.

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 p.ex.,

- l'écriture de texte sur la sortie standard ;
- la lecture d'une donnée sur la sortie 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.

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 p.ex.,

- l'écriture de texte sur la sortie standard ;
- la lecture d'une donnée sur la sortie 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).

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

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.

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.

Interruptions — 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.

Interruptions — 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 et la valeur de `edx` signifie qu'il y a un unique caractère à afficher.

Interruptions — exemples

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 la sortie standard. Le registre `ecx` contient l'adresse `x` à laquelle le code ASCII du caractère lu sera enregistré et la valeur de `edx` signifie qu'il y a un unique caractère à lire.

Interruptions — exemples

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 la sortie standard. Le registre `ecx` contient l'adresse `x` à laquelle le code ASCII du caractère lu sera enregistré et la valeur de `edx` signifie qu'il y a un unique caractère à lire.

Il est bien entendu possible, pour les interruptions commandant l'écriture et l'affichage de caractères, de placer d'autres valeurs dans `edx` pour pouvoir écrire/lire plus de caractères.

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.

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`.

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`.

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`.

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`.

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`.

Astuce : 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
```

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
```

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:
```

Exemple complet de programme

```
; Def. de donnees
section .data
chaine_1:
  db 'Caractere? ',0
chaine_2:
  db 'Suivant : ',0
```

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

```
; Decl. de donnees
section .bss
car: resb 1
```

```
; Instructions
section .text
global main
main:
```

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
```

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
```

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
```

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
```

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
```

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
```

Ce programme lit un caractère sur l'entrée standard et affiche le caractère suivant dans la table ASCII.

Plan

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.

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.

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.

```
mov eax, 0
instr_2: mov ebx, 1
add eax, ebx
```

P.ex., ici l'étiquette `instr_2` pointe vers l'instruction `mov ebx, 1`.

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.

```
mov eax, 0
instr_2: mov ebx, 1
add eax, ebx
```

P.ex., ici l'étiquette `instr_2` pointe vers l'instruction `mov ebx, 1`.

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.

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 en l'algorithme suivant :

1. Répéter, tant que l'exécution n'est pas interrompue :
   1. charger l'instruction *I* d'adresse `eip` ;
   2. mettre à jour `eip` ;
   3. traiter l'instruction *I*.

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 en l'algorithme suivant :

1. Répéter, tant que l'exécution n'est pas interrompue :
   1. charger l'instruction *I* d'adresse `eip` ;
   2. mettre à jour `eip` ;
   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.

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.

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

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

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

Sauts inconditionnels

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

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

```
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.

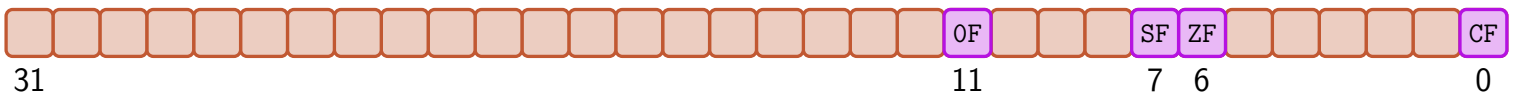
Le registre `flags`

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

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.

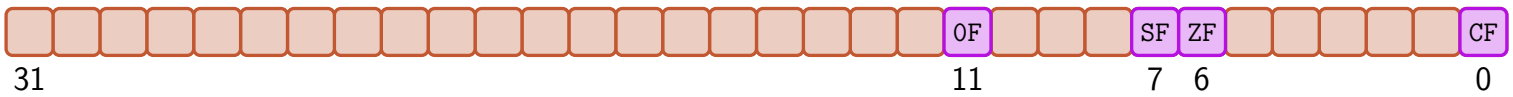
Comme son nom l'indique, il fonctionne comme un drapeau : chacun de ses bits code une information du type oui (bit à 1) / non (bit à 0).



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`.

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`.

Plus précisément, cette instruction calcule la différence `VAL_1` − `VAL_2` et modifie `flags` de la manière suivante :

- si `VAL_1` − `VAL_2` $= 0$, alors `ZF` est positionné à `1` ;
- si `VAL_1` − `VAL_2` > 0, alors `ZF` est positionné à `0` et `CF` est positionné à `0` ;
- si `VAL_1` − `VAL_2` < 0, alors `ZF` est positionné à `0` et `CF` est positionné à `1`.

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`.

Plus précisément, cette instruction calcule la différence `VAL_1` − `VAL_2` et modifie `flags` de la manière suivante :

- si `VAL_1` − `VAL_2` = 0, alors `ZF` est positionné à `1` ;
- si `VAL_1` − `VAL_2` > 0, alors `ZF` est positionné à `0` et `CF` est positionné à `0` ;
- si `VAL_1` − `VAL_2` < 0, alors `ZF` est positionné à `0` et `CF` est positionné à `1`.

On peut préciser la taille des valeurs à comparer à l'aide d'un descripteur de taille (`dbyte`, `word`, `dword`) si besoin est.

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`.

Plus précisément, cette instruction calcule la différence `VAL_1` − `VAL_2` et modifie `flags` de la manière suivante :

- si `VAL_1` − `VAL_2` = 0, alors `ZF` est positionné à `1` ;
- si `VAL_1` − `VAL_2` > 0, alors `ZF` est positionné à `0` et `CF` est positionné à `0` ;
- si `VAL_1` − `VAL_2` < 0, alors `ZF` est positionné à `0` et `CF` est positionné à `1`.

On peut préciser la taille des valeurs à comparer à l'aide d'un descripteur de taille (`dbyte`, `word`, `dword`) si besoin est.

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

P.ex., cette comparaison fait que `ZF` et `CF` sont positionnés à `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.

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.

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.

Il y a plusieurs instructions de saut conditionnel. Elles diffèrent sur la condition qui provoque le saut :

Instruction	Saute si
`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

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

Ceci saute à `inferieur` si la valeur de `eax` est strict. inf. à celle de `ebx`.

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

Sauts conditionnels

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

Ceci saute à `inferieur` si la valeur de `eax` est strict. inf. à celle de `ebx`.

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

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

Ceci est une boucle. Tant que la valeur de `ecx` est diff. 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:
```

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:
```

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:
```

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 saute vers l'étiquette d'instruction `ETIQ` si `ecx` est non nul et décrémente ce dernier.

On obtient la suite d'instructions suivante :

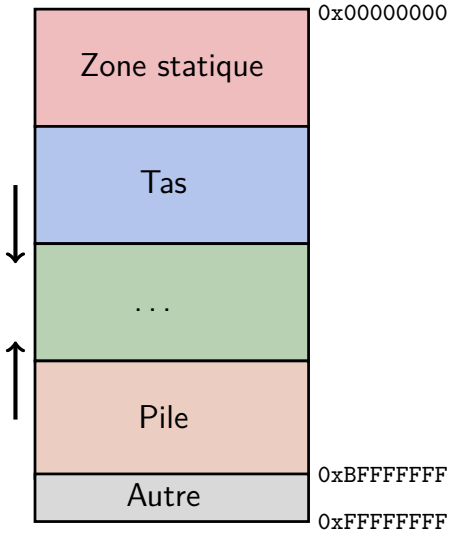
```
mov ecx, ebx
boucle:
    BLOC
    loop boucle
```

Plan

Pile

La pile est une zone de la mémoire dans laquelle on peut empiler et dépiler des données.

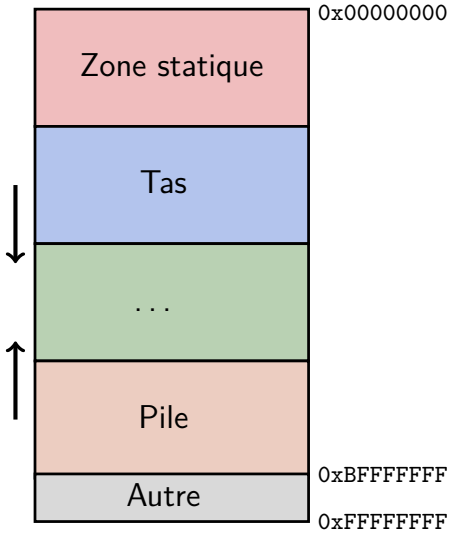
La zone qu'elle occupe en mémoire est de taille variable mais elle se situe toujours avant l'adresse `0xBFFFFFFF`.



Pile

La pile est une zone de la mémoire dans laquelle on peut empiler et dépiler des données.

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

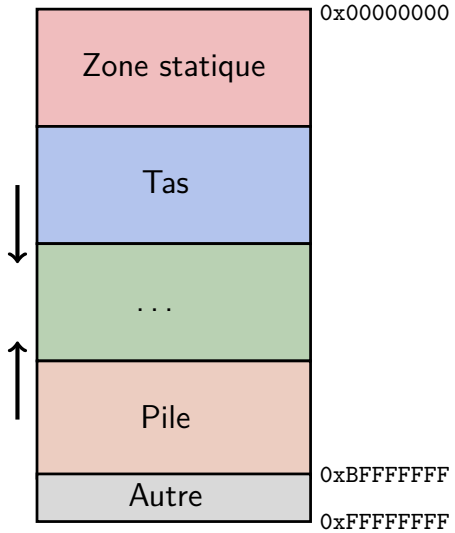


La pile est de type LIFO : les données sont dépilées de la plus récente à la plus ancienne.

Pile

La pile est une zone de la mémoire dans laquelle on peut empiler et dépiler des données.

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



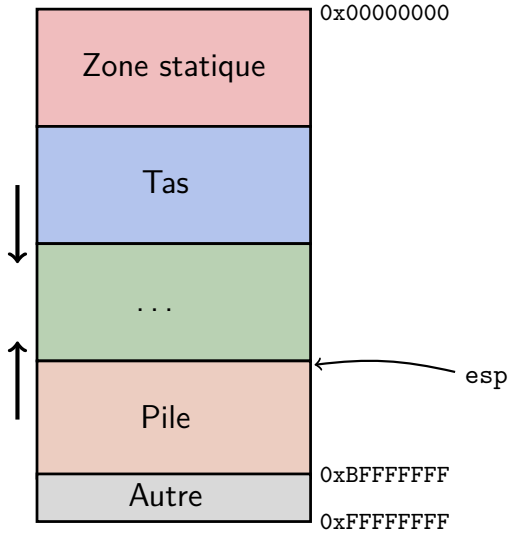
La pile est de type LIFO : les données sont dépilées de la plus récente à la plus ancienne.

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

Pile

La pile est une zone de la mémoire dans laquelle on peut empiler et dépiler des données.

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.

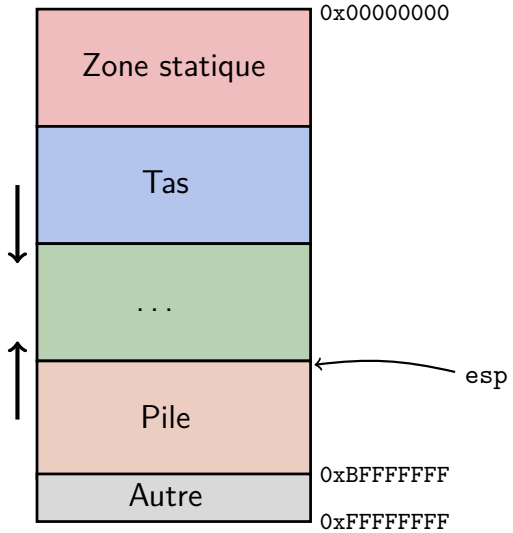
On place et on lit dans la pile uniquement des **doubles mots**.

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

Pile

La pile est une zone de la mémoire dans laquelle on peut empiler et dépiler des données.

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**.

On utilise le registre `ebp` pour sauvegarder une position dans la pile (lorsque `esp` est susceptible de changer).

Pile

On dispose de deux opérations pour manipuler la pile :

1. empiler une valeur ;
2. dépiler une valeur.

Pile

On dispose de deux opérations pour manipuler la pile :

1. empiler une valeur ;
2. dépiler une valeur.

Pour **empiler** une valeur `VAL` à la pile, on utilise

```
push VAL
```

Ceci décrémente `esp` de 4 et écrit à l'adresse `esp` la valeur `VAL`.

Pile

On dispose de deux opérations pour manipuler la pile :

1. empiler une valeur ;
2. dépiler une valeur.

Pour **empiler** une valeur `VAL` à la pile, on utilise

```
push VAL
```

Ceci décrémente `esp` de 4 et écrit à l'adresse `esp` la valeur `VAL`.

Pour **dépiler** vers le registre `REG` la valeur située en tête de pile, on utilise

```
pop REG
```

Ceci recopie les 4 octets à partir de l'adresse `esp` vers `REG` et incrémente `esp` de 4.

Pile

On dispose de deux opérations pour manipuler la pile :

1. empiler une valeur ;
2. dépiler une valeur.

Pour **empiler** une valeur `VAL` à la pile, on utilise

```
push VAL
```

Ceci décrémente `esp` de 4 et écrit à l'adresse `esp` la valeur `VAL`.

Pour **dépiler** vers le registre `REG` la valeur située en tête de pile, on utilise

```
pop REG
```

Ceci recopie les 4 octets à partir de l'adresse `esp` vers `REG` et incrémente `esp` de 4.

Attention : l'ajout d'éléments dans la pile fait décroître la valeur de `esp` et la suppression d'éléments fait croître sa valeur, ce qui est peut-être contre-intuitif.

Pile

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

Adresses	Pile
1000	0x01010101
1004	0x20202020
1008	0xAA55AA55
1012	0xFFFFFFFF
1016	0x00000000

esp = 1008

Pile

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

Adresses	Pile
1000	0x01010101
1004	0x20202020
1008	0xAA55AA55
1012	0xFFFFFFFF
1016	0x00000000

esp = 1008

```
push 0x3
```

0x01010101
0x00000003
0xAA55AA55
0xFFFFFFFF
0x00000000

esp = 1004

Pile

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

Adresses	Pile
1000	0x01010101
1004	0x20202020
1008	0xAA55AA55
1012	0xFFFFFFFF
1016	0x00000000

esp = 1008

```
push 0x3
pop eax
```

0x01010101
0x00000003
0xAA55AA55
0xFFFFFFFF
0x00000000

esp = 1004

0x01010101
0x00000003
0xAA55AA55
0xFFFFFFFF
0x00000000

esp = 1008

eax = 0x3

Pile

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

Adresses	Pile
1000	0x01010101
1004	0x20202020
1008	0xAA55AA55
1012	0xFFFFFFFF
1016	0x00000000

esp = 1008

```
push 0x3
pop eax
pop ebx
```

0x01010101
0x00000003
0xAA55AA55
0xFFFFFFFF
0x00000000

esp = 1004

0x01010101
0x00000003
0xAA55AA55
0xFFFFFFFF
0x00000000

esp = 1008

eax = 0x3

0x01010101
0x00000003
0xAA55AA55
0xFFFFFFFF
0x00000000

esp = 1012

ebx = 0xA...5

Pile

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

Adresses	Pile
1000	0x01010101
1004	0x20202020
1008	0xAA55AA55
1012	0xFFFFFFFF
1016	0x00000000

esp = 1008

```
push 0x3
pop eax
pop ebx
push eax
```

0x01010101
0x00000003
0xAA55AA55
0xFFFFFFFF
0x00000000

esp = 1004

0x01010101
0x00000003
0xAA55AA55
0xFFFFFFFF
0x00000000

esp = 1008

eax = 0x3

0x01010101
0x00000003
0xAA55AA55
0xFFFFFFFF
0x00000000

esp = 1012

ebx = 0xA...5

0x01010101
0x00000003
0x00000003
0xFFFFFFFF
0x00000000

esp = 1008

Instruction `call`

On souhaite maintenant établir un mécanisme pour pourvoir **écrire des fonctions et les appeler**.

Instruction `call`

On souhaite maintenant établir un mécanisme pour pourvoir **écrire des fonctions et les appeler**.

L'un des ingrédients pour cela est l'instruction

```
call ETIQ
```

Elle permet de sauter à l'étiquette d'instruction `ETIQ`.

Instruction `call`

On souhaite maintenant établir un mécanisme pour pourvoir **écrire des fonctions et les appeler**.

L'un des ingrédients pour cela est l'instruction

```
call ETIQ
```

Elle permet de sauter à l'étiquette d'instruction `ETIQ`.

La différence avec l'instruction `jmp ETIQ` réside dans le fait que `call ETIQ` **empile**, avant le saut, l'**adresse de l'instruction qui la suit** dans le programme.

Instruction `call`

On souhaite maintenant établir un mécanisme pour pourvoir **écrire des fonctions et les appeler**.

L'un des ingrédients pour cela est l'instruction

```
call ETIQ
```

Elle permet de sauter à l'étiquette d'instruction `ETIQ`.

La différence avec l'instruction `jmp ETIQ` réside dans le fait que `call ETIQ` **empile**, avant le saut, l'**adresse de l'instruction qui la suit** dans le programme.

Ainsi, les deux suites d'instructions suivantes sont équivalentes :

```
call cible
suite:
...
```

```
push suite
jmp cible
suite:
...
```

Instruction `ret`

L'intérêt d'enregistrer l'adresse de l'instruction qui suit un `call ETIQ` repose sur le fait que l'exécution peut **revenir** à cette instruction.

Instruction `ret`

L'intérêt d'enregistrer l'adresse de l'instruction qui suit un `call ETIQ` repose sur le fait que l'exécution peut **revenir** à cette instruction.

Ceci est offert par l'instruction (sans opérande)

```
ret
```

Elle fonctionne en dépilant la donnée en tête de pile et en sautant à l'adresse spécifiée par cette valeur.

Instruction `ret`

L'intérêt d'enregistrer l'adresse de l'instruction qui suit un `call ETIQ` repose sur le fait que l'exécution peut **revenir** à cette instruction.

Ceci est offert par l'instruction (sans opérande)

`ret`

Elle fonctionne en dépilant la donnée en tête de pile et en sautant à l'adresse spécifiée par cette valeur.

Ainsi, les deux suites d'instructions suivantes sont équivalentes (excepté pour la valeur de `eax` qui est modifiée dans la seconde) :

```
call cible
suite:
...
cible:
...
ret
```

```
push suite
jmp cible
suite:
...
cible:
...
pop eax
jmp eax
```

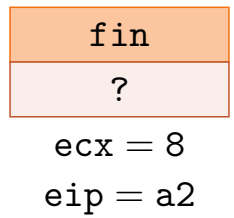
Exemple d'utilisation `call` / `ret`

Considérons la suite d'instructions suivante et observons l'état de la pile et de l'exécution.

```
a1 | mov ecx, 8
a2 | call loin
a3 | suiv: add ecx, 24
a4 | loin: add ecx, 16
a5 | ret
   | ...
a6 | fin:
```

Exemple d'utilisation `call` / `ret`

Considérons la suite d'instructions suivante et observons l'état de la pile et de l'exécution.

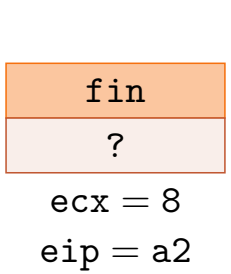


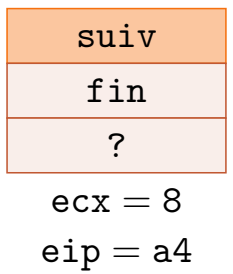

```
→   a1 | mov ecx, 8
    a2 | call loin
    a3 | suiv: add ecx, 24
    a4 | loin: add ecx, 16
    a5 | ret
       | ...
    a6 | fin:
```

Exemple d'utilisation `call` / `ret`

Considérons la suite d'instructions suivante et observons l'état de la pile et de l'exécution.

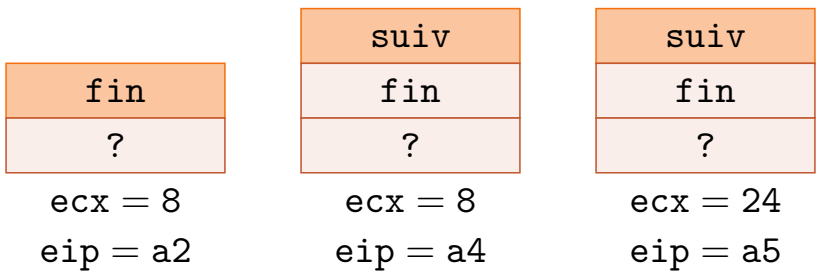
```
      a1 | mov ecx, 8
→     a2 | call loin
      a3 | suiv: add ecx, 24
      a4 | loin: add ecx, 16
      a5 | ret
         | ...
      a6 | fin:
```





Exemple d'utilisation `call` / `ret`

Considérons la suite d'instructions suivante et observons l'état de la pile et de l'exécution.

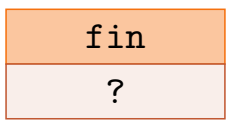



```
      a1 | mov ecx, 8
      a2 | call loin
      a3 | suiv: add ecx, 24
→     a4 | loin: add ecx, 16
      a5 | ret
         | ...
      a6 | fin:
```

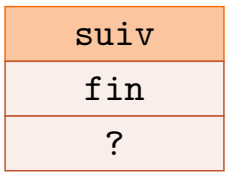
Exemple d'utilisation `call` / `ret`

Considérons la suite d'instructions suivante et observons l'état de la pile et de l'exécution.

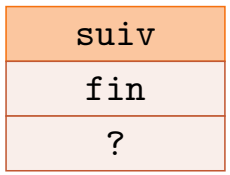
```
      a1 | mov ecx, 8
      a2 | call loin
      a3 | suiv: add ecx, 24
      a4 | loin: add ecx, 16
→     a5 | ret
         | ...
      a6 | fin:
```



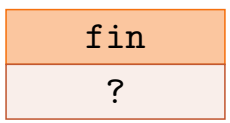
ecx = 8
eip = a2



ecx = 8
eip = a4



ecx = 24
eip = a5

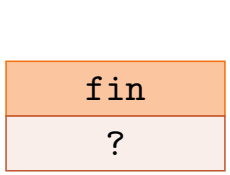


ecx = 24
eip = a3

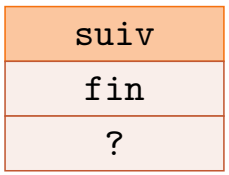
Exemple d'utilisation `call` / `ret`

Considérons la suite d'instructions suivante et observons l'état de la pile et de l'exécution.

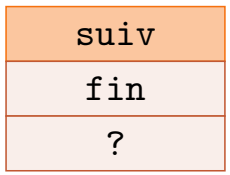
```
      a1 | mov ecx, 8
      a2 | call loin
→     a3 | suiv: add ecx, 24
      a4 | loin: add ecx, 16
      a5 | ret
         | ...
      a6 | fin:
```



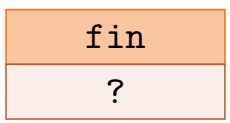
ecx = 8
eip = a2



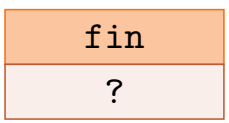
ecx = 8
eip = a4



ecx = 24
eip = a5



ecx = 24
eip = a3

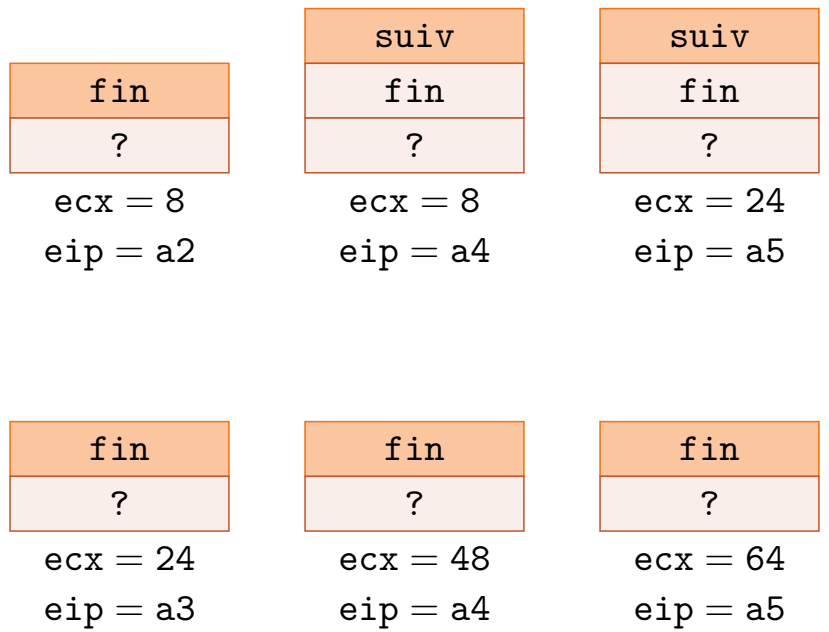


ecx = 48
eip = a4

Exemple d'utilisation `call` / `ret`

Considérons la suite d'instructions suivante et observons l'état de la pile et de l'exécution.

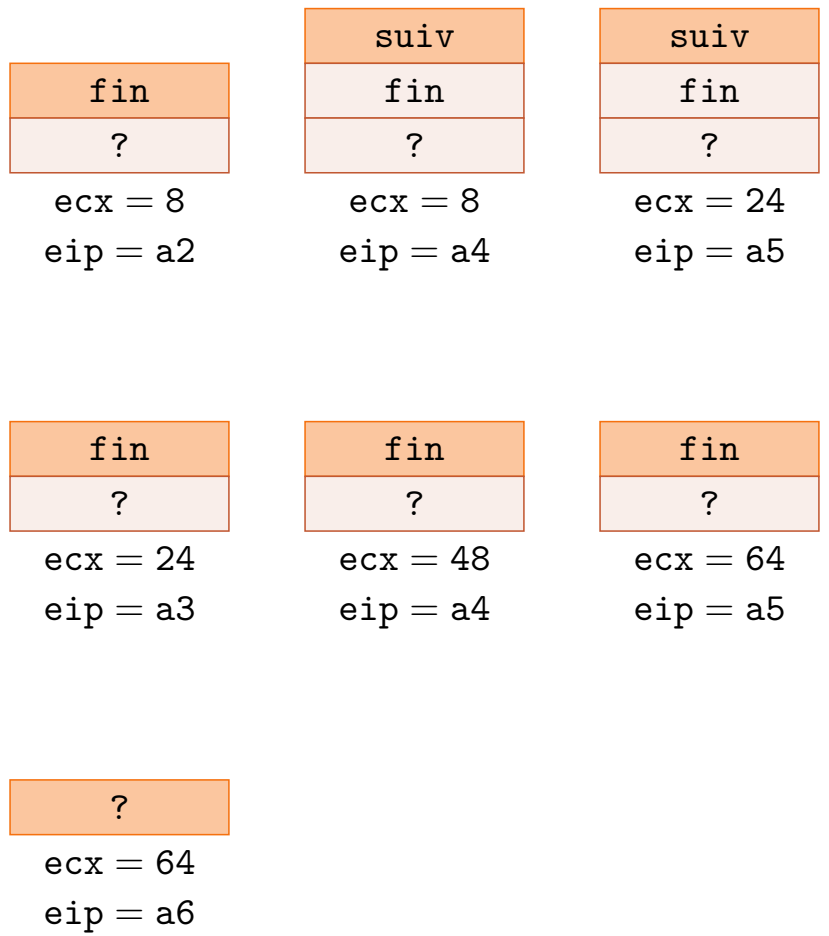
```
      a1 | mov ecx, 8
      a2 | call loin
      a3 | suiv: add ecx, 24
→     a4 | loin: add ecx, 16
      a5 | ret
         | ...
      a6 | fin:
```



Exemple d'utilisation `call` / `ret`

Considérons la suite d'instructions suivante et observons l'état de la pile et de l'exécution.

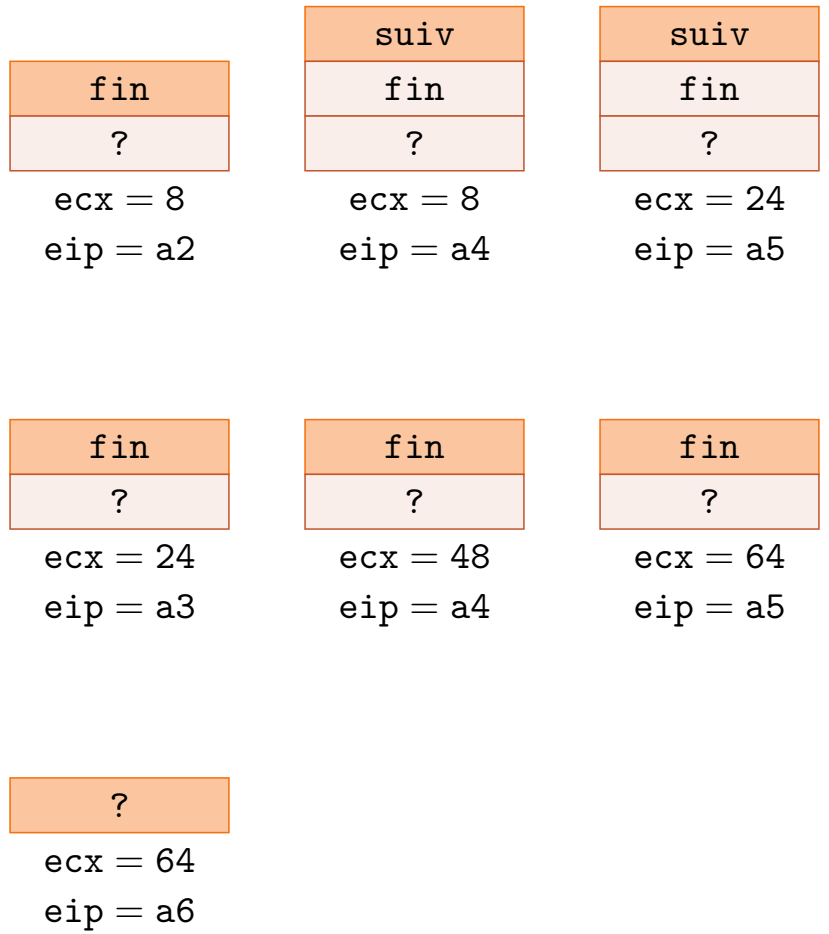
```
      a1 | mov ecx, 8
      a2 | call loin
      a3 | suiv: add ecx, 24
      a4 | loin: add ecx, 16
→     a5 | ret
         | ...
      a6 | fin:
```



Exemple d'utilisation `call` / `ret`

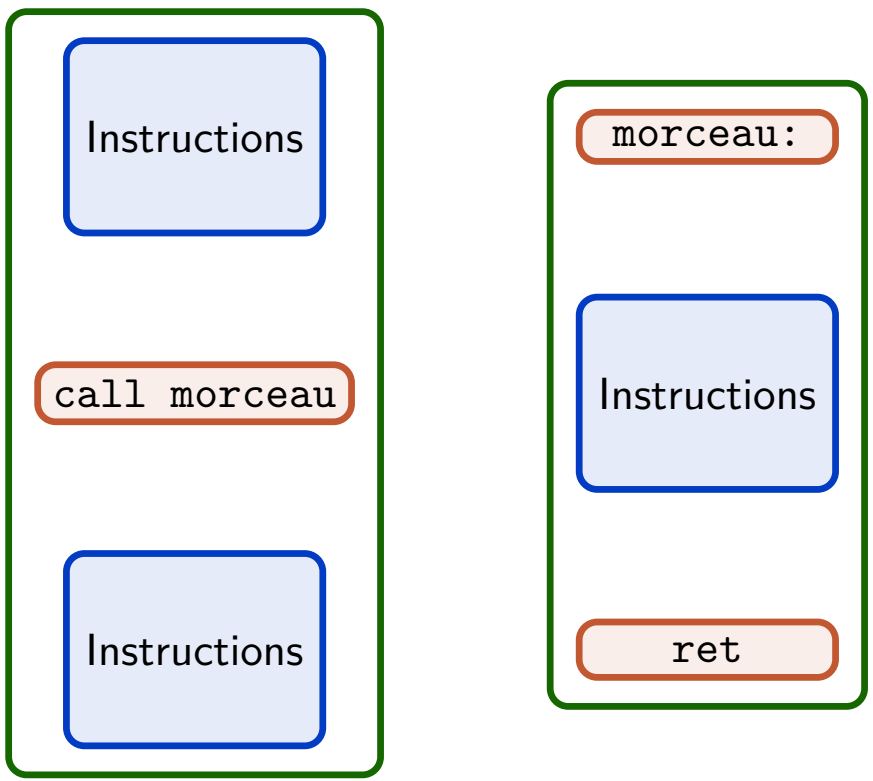
Considérons la suite d'instructions suivante et observons l'état de la pile et de l'exécution.

```
      a1 | mov ecx, 8
      a2 | call loin
      a3 | suiv: add ecx, 24
      a4 | loin: add ecx, 16
      a5 | ret
         | ...
→     a6 | fin:
```



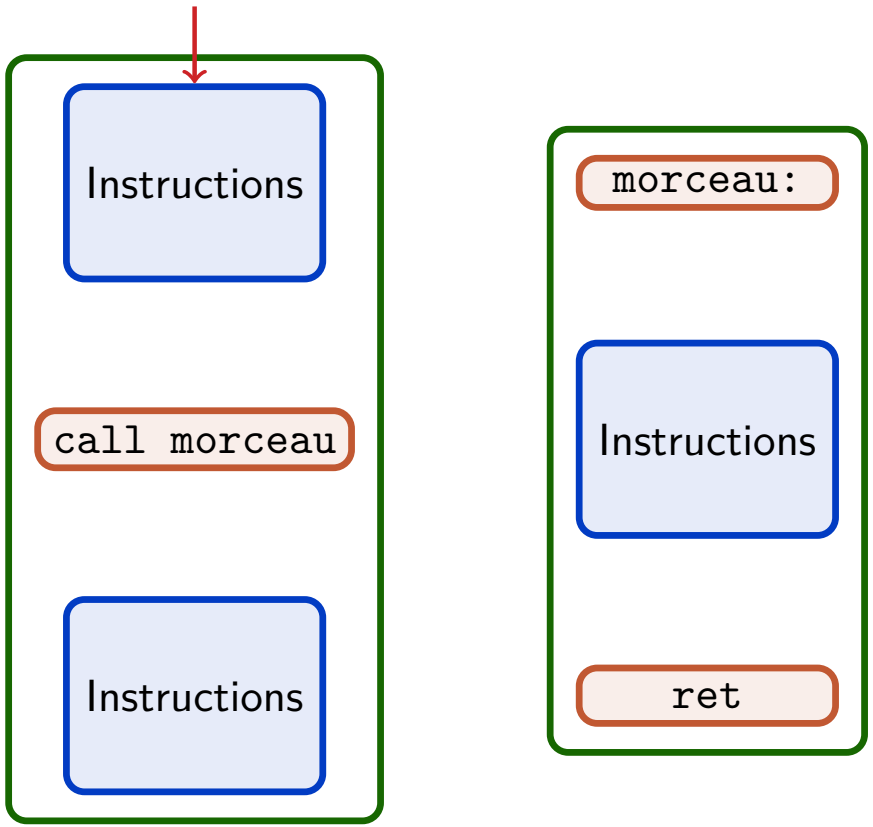
Exemple d'utilisation `call` / `ret`

Schématiquement, voici l'action produite sur l'exécution d'un programme par le couple `call` / `ret` :



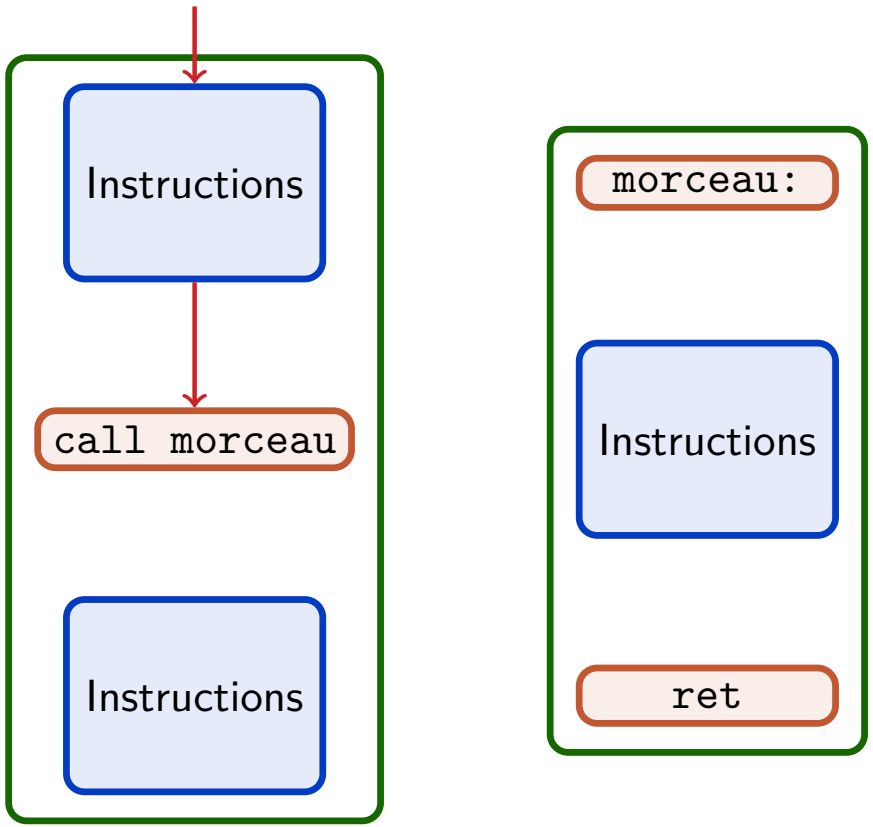
Exemple d'utilisation `call` / `ret`

Schématiquement, voici l'action produite sur l'exécution d'un programme par le couple `call` / `ret` :



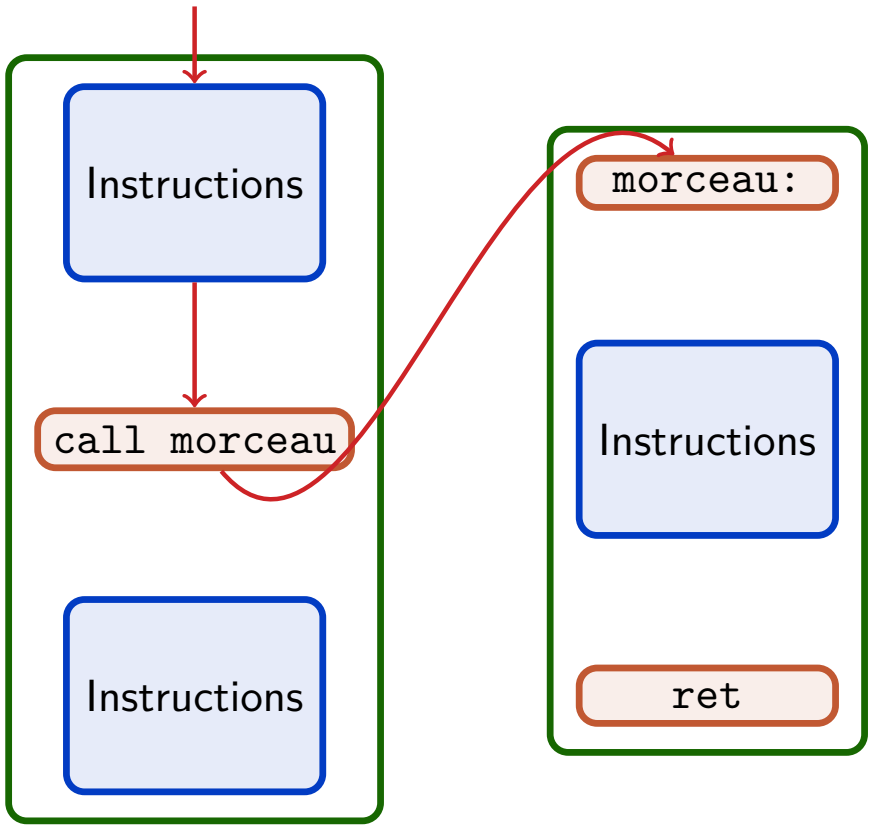
Exemple d'utilisation `call` / `ret`

Schématiquement, voici l'action produite sur l'exécution d'un programme par le couple `call` / `ret` :



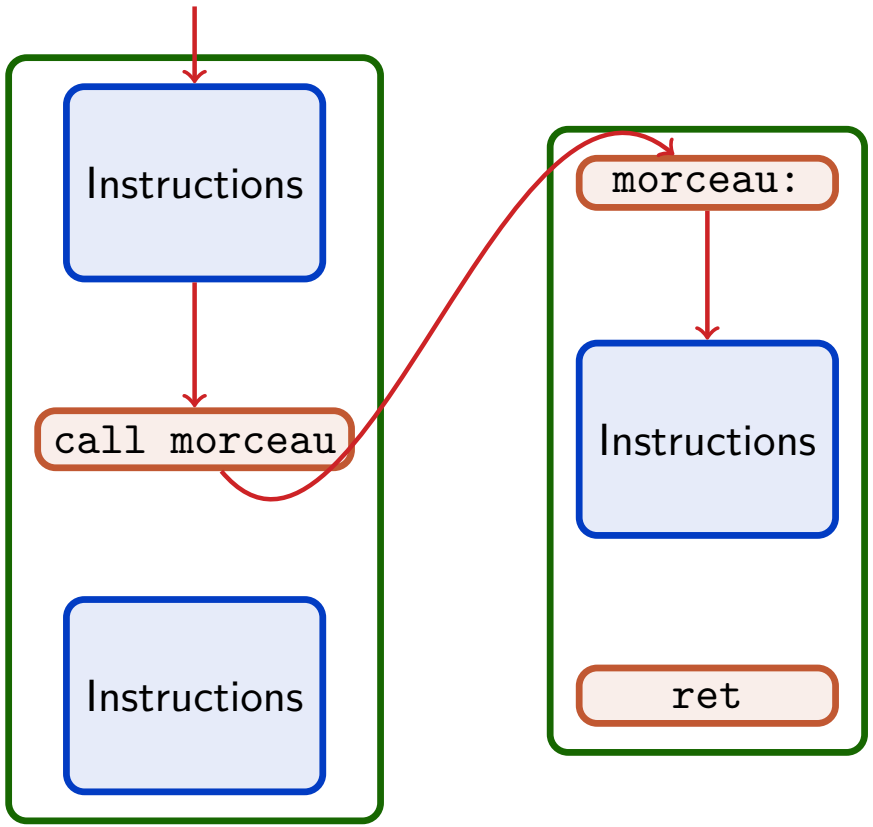
Exemple d'utilisation `call` / `ret`

Schématiquement, voici l'action produite sur l'exécution d'un programme par le couple `call` / `ret` :



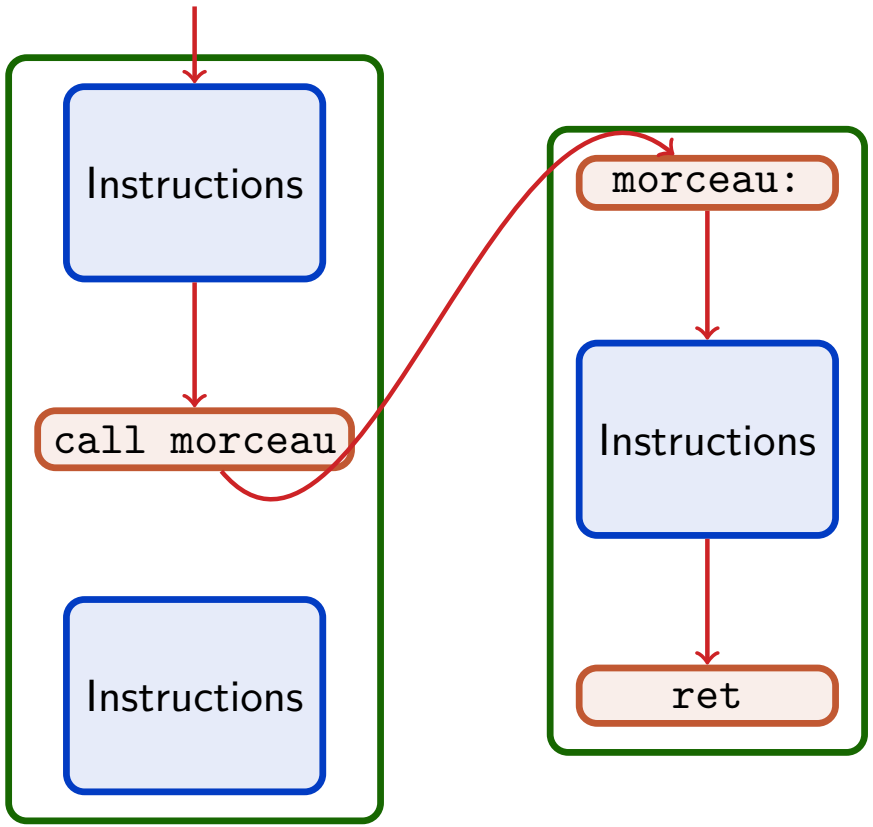
Exemple d'utilisation `call` / `ret`

Schématiquement, voici l'action produite sur l'exécution d'un programme par le couple `call` / `ret` :



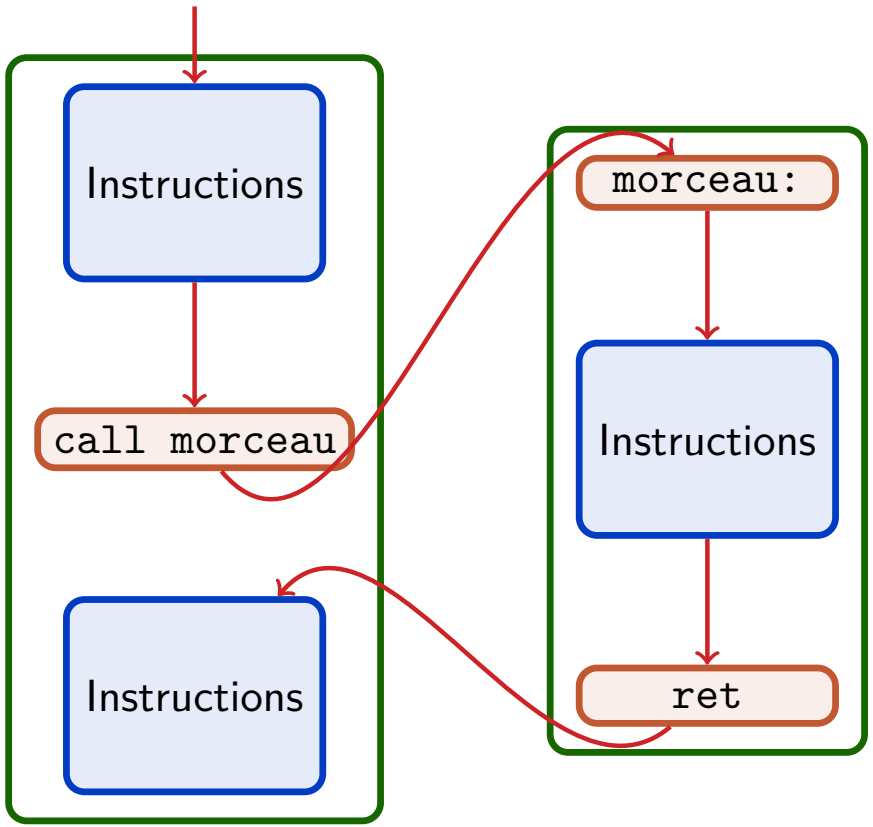
Exemple d'utilisation `call` / `ret`

Schématiquement, voici l'action produite sur l'exécution d'un programme par le couple `call` / `ret` :



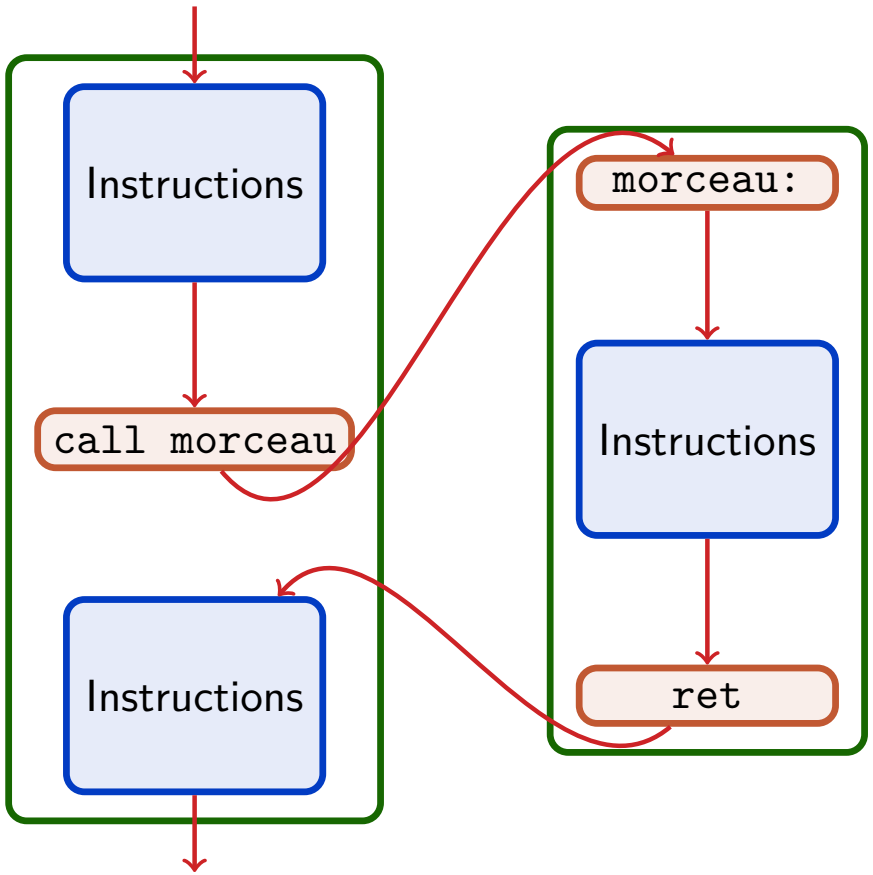
Exemple d'utilisation `call` / `ret`

Schématiquement, voici l'action produite sur l'exécution d'un programme par le couple `call` / `ret` :



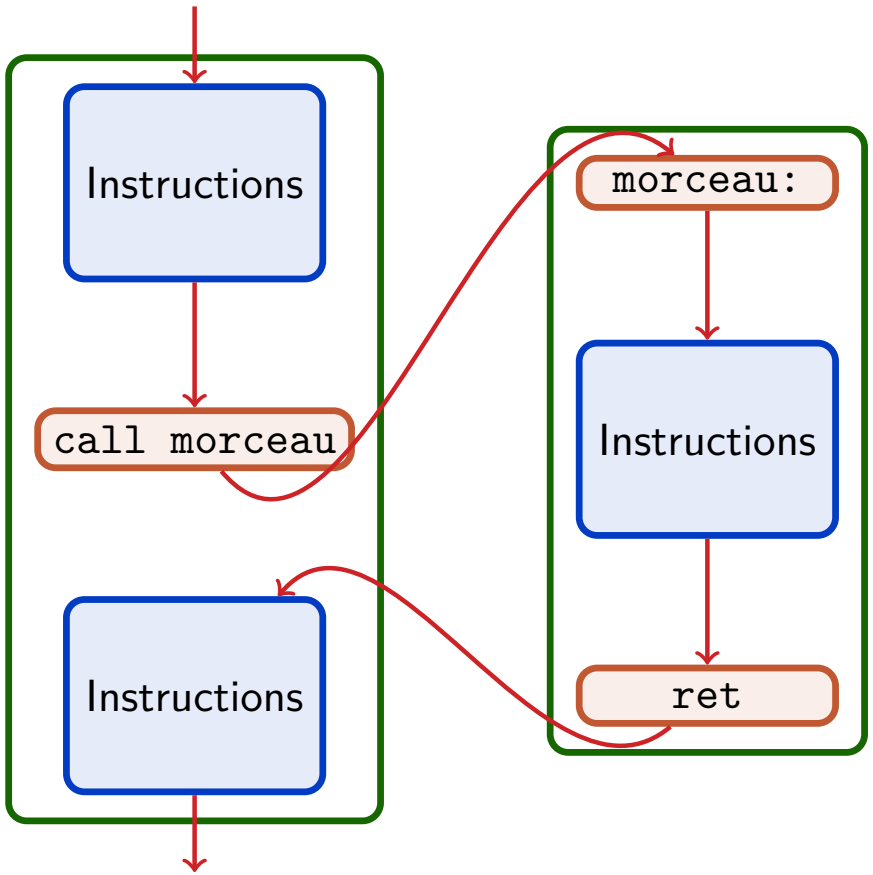
Exemple d'utilisation `call` / `ret`

Schématiquement, voici l'action produite sur l'exécution d'un programme par le couple `call` / `ret` :



Exemple d'utilisation `call` / `ret`

Schématiquement, voici l'action produite sur l'exécution d'un programme par le couple `call` / `ret` :



Attention : le retour à l'endroit du code attendu par l'instruction `ret` n'est correct que si l'état de la pile à l'étiquette `morceau` est le même que celui juste avant le `ret`.