Fonctions — conventions d'appel du C

L'écriture de fonctions respecte des conventions, dites conventions d'appel du C.

Fonctions — conventions d'appel du C

L'écriture de fonctions respecte des conventions, dites conventions d'appel du C. Ceci consiste en le respect des points suivants :

1. les **arguments** d'une fonction sont passés, avant son appel, dans la **pile** en les empilant ;

Fonctions — conventions d'appel du C

L'écriture de fonctions respecte des conventions, dites conventions d'appel du C. Ceci consiste en le respect des points suivants :

1. les **arguments** d'une fonction sont passés, avant son appel, dans la **pile** en les empilant ;
2. le **résultat** d'une fonction est renvoyé en l'écrivant dans le registre `eax` ;

Fonctions — conventions d'appel du C

L'écriture de fonctions respecte des conventions, dites conventions d'appel du C. Ceci consiste en le respect des points suivants :

1. les **arguments** d'une fonction sont passés, avant son appel, dans la **pile** en les empilant ;
2. le **résultat** d'une fonction est renvoyé en l'écrivant dans le registre `eax` ;
3. les **valeurs des registres de travail** `ebx`, `ecx` et `edx` doivent être dans le **même état** avant l'appel et après l'appel d'une fonction ;

Fonctions — conventions d'appel du C

L'écriture de fonctions respecte des conventions, dites conventions d'appel du C. Ceci consiste en le respect des points suivants :

1. les **arguments** d'une fonction sont passés, avant son appel, dans la **pile** en les empilant ;
2. le **résultat** d'une fonction est renvoyé en l'écrivant dans le registre `eax` ;
3. les **valeurs des registres de travail** `ebx`, `ecx` et `edx` doivent être dans le **même état** avant l'appel et après l'appel d'une fonction ;
4. la **pile** doit être dans le **même état** avant l'appel et après l'appel d'une fonction. Ceci signifie que l'état des pointeurs `esp` et `ebp` sont conservés et que le contenu de la pile qui suit l'adresse `esp` est également conservé.

Fonctions — conventions d'appel du C

L'écriture de fonctions respecte des conventions, dites conventions d'appel du C. Ceci consiste en le respect des points suivants :

1. les **arguments** d'une fonction sont passés, avant son appel, dans la **pile** en les empilant ;
2. le **résultat** d'une fonction est renvoyé en l'écrivant dans le registre `eax` ;
3. les **valeurs des registres de travail** `ebx`, `ecx` et `edx` doivent être dans le **même état** avant l'appel et après l'appel d'une fonction ;
4. la **pile** doit être dans le **même état** avant l'appel et après l'appel d'une fonction. Ceci signifie que l'état des pointeurs `esp` et `ebp` sont conservés et que le contenu de la pile qui suit l'adresse `esp` est également conservé.

Note : le point 3 n'est pas obligatoire à suivre.

Fonctions — écriture et appel

L'**écriture** d'une fonction suit le squelette

```
NOM_FCT:
    push ebp
    mov ebp, esp
    INSTR
    pop ebp
    ret
```

Ici, `NOM_FCT` est une étiquette d'instruction qui fait d'office de nom pour la fonction. De plus, `INSTR` est un bloc d'instructions.

Il est primordial que `INSTR` **conserve l'état de la pile**.

Fonctions — écriture et appel

L'**écriture** d'une fonction suit le squelette

```
NOM_FCT:
    push ebp
    mov ebp, esp
    INSTR
    pop ebp
    ret
```

Ici, `NOM_FCT` est une étiquette d'instruction qui fait d'office de nom pour la fonction. De plus, `INSTR` est un bloc d'instructions.

Il est primordial que `INSTR` **conserve l'état de la pile**.

L'**appel** d'une fonction se fait par

```
push ARG_N
...
push ARG_1

call NOM_FCT

add esp, 4 * N
```

Ici, `NOM_FCT` est le nom de la fonction à appeler. Elle admet `N` arguments qui sont **empilés du dernier au premier**.

Après l'appel, on incrémente `esp` pour dépiler d'un coup les `N` arguments de la fonction.

Fonctions — appel et état de la pile

Analysons l'état de la pile, étape par étape, lors de l'appel d'une fonction.

```
push ARG_N
...
push ARG_1
call NOM_FCT
ADD esp, 4 * N
```

```
NOM_FCT:
    push ebp
    mov ebp, esp
    INSTR
    pop ebp
    ret
```

Légende : en orange, la zone de la pile pointée par esp, en violet, la zone de la pile pointée par ebp, a_ret est l'adresse de l'instr. qui suit le call.

Fonctions — appel et état de la pile

Analysons l'état de la pile, étape par étape, lors de l'appel d'une fonction.

→

```
    push ARG_N
    ...
    push ARG_1
    call NOM_FCT
    ADD esp, 4 * N
```

```
NOM_FCT:
    push ebp
    mov ebp, esp
    INSTR
    pop ebp
    ret
```

Légende : en orange, la zone de la pile pointée par esp, en violet, la zone de la pile pointée par ebp, a_ret est l'adresse de l'instr. qui suit le call.

?

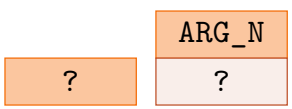
Fonctions — appel et état de la pile

Analysons l'état de la pile, étape par étape, lors de l'appel d'une fonction.

```
→   push ARG_N
    ...
    push ARG_1
    call NOM_FCT
    ADD esp, 4 * N
```

```
NOM_FCT:
    push ebp
    mov ebp, esp
    INSTR
    pop ebp
    ret
```

Légende : en orange, la zone de la pile pointée par esp, en violet, la zone de la pile pointée par ebp, a_ret est l'adresse de l'instr. qui suit le call.



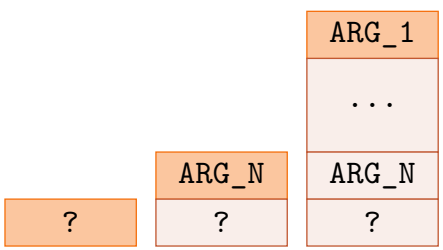
Fonctions — appel et état de la pile

Analysons l'état de la pile, étape par étape, lors de l'appel d'une fonction.

```
        push ARG_N
        ...
→       push ARG_1
        call NOM_FCT
        ADD esp, 4 * N
```

```
NOM_FCT:
    push ebp
    mov ebp, esp
    INSTR
    pop ebp
    ret
```

Légende : en orange, la zone de la pile pointée par esp, en violet, la zone de la pile pointée par ebp, a_ret est l'adresse de l'instr. qui suit le call.



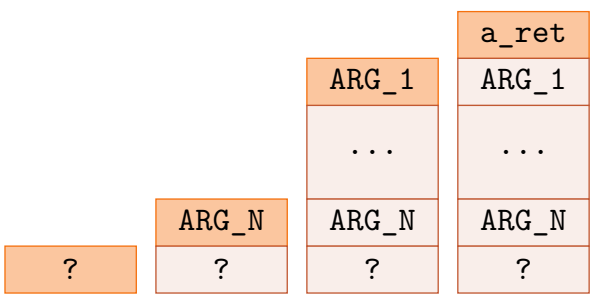
Fonctions — appel et état de la pile

Analysons l'état de la pile, étape par étape, lors de l'appel d'une fonction.

```
        push ARG_N
        ...
        push ARG_1
→       call NOM_FCT
        ADD esp, 4 * N
```

```
NOM_FCT:
    push ebp
    mov ebp, esp
    INSTR
    pop ebp
    ret
```

Légende : en orange, la zone de la pile pointée par esp, en violet, la zone de la pile pointée par ebp, a_ret est l'adresse de l'instr. qui suit le call.

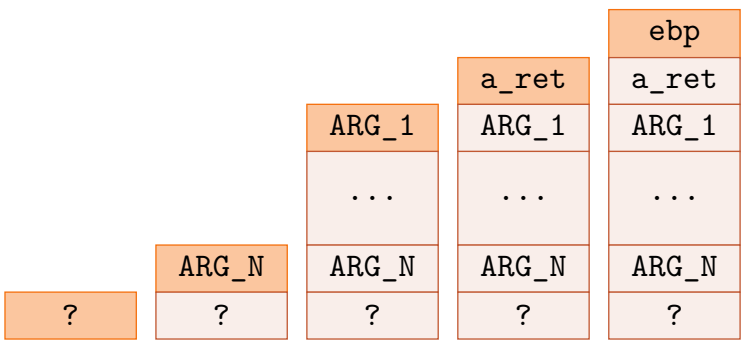


Fonctions — appel et état de la pile

Analysons l'état de la pile, étape par étape, lors de l'appel d'une fonction.

```
push ARG_N                    →    NOM_FCT:
...                                    push ebp
push ARG_1                             mov ebp, esp
call NOM_FCT                           INSTR
ADD esp, 4 * N                         pop ebp
                                       ret
```

Légende : en orange, la zone de la pile pointée par esp, en violet, la zone de la pile pointée par ebp, a_ret est l'adresse de l'instr. qui suit le call.



Fonctions — appel et état de la pile

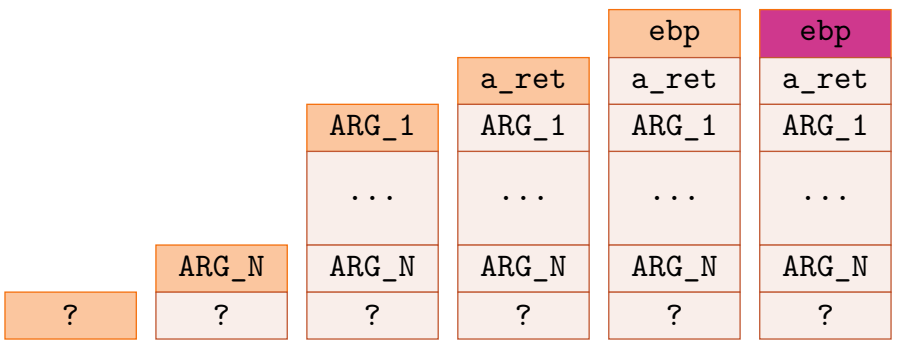
Analysons l'état de la pile, étape par étape, lors de l'appel d'une fonction.

```
push ARG_N
...
push ARG_1
call NOM_FCT
ADD esp, 4 * N
```

→

```
NOM_FCT:
    push ebp
    mov ebp, esp
    INSTR
    pop ebp
    ret
```

Légende : en orange, la zone de la pile pointée par esp, en violet, la zone de la pile pointée par ebp, a_ret est l'adresse de l'instr. qui suit le call.



Fonctions — appel et état de la pile

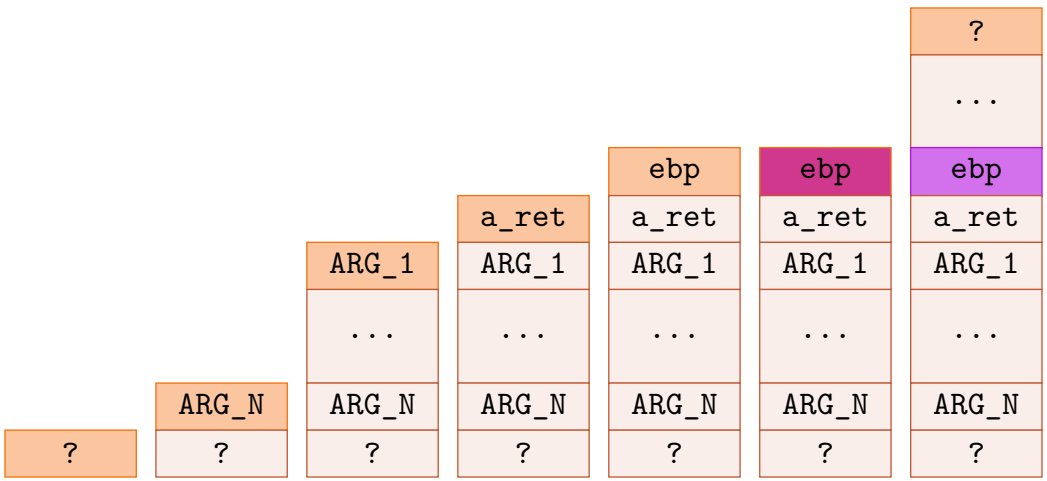
Analysons l'état de la pile, étape par étape, lors de l'appel d'une fonction.

```
push ARG_N
...
push ARG_1
call NOM_FCT
ADD esp, 4 * N
```

→

```
NOM_FCT:
    push ebp
    mov ebp, esp
    INSTR
    pop ebp
    ret
```

Légende : en orange, la zone de la pile pointée par esp, en violet, la zone de la pile pointée par ebp, a_ret est l'adresse de l'instr. qui suit le `call`.



Fonctions — appel et état de la pile

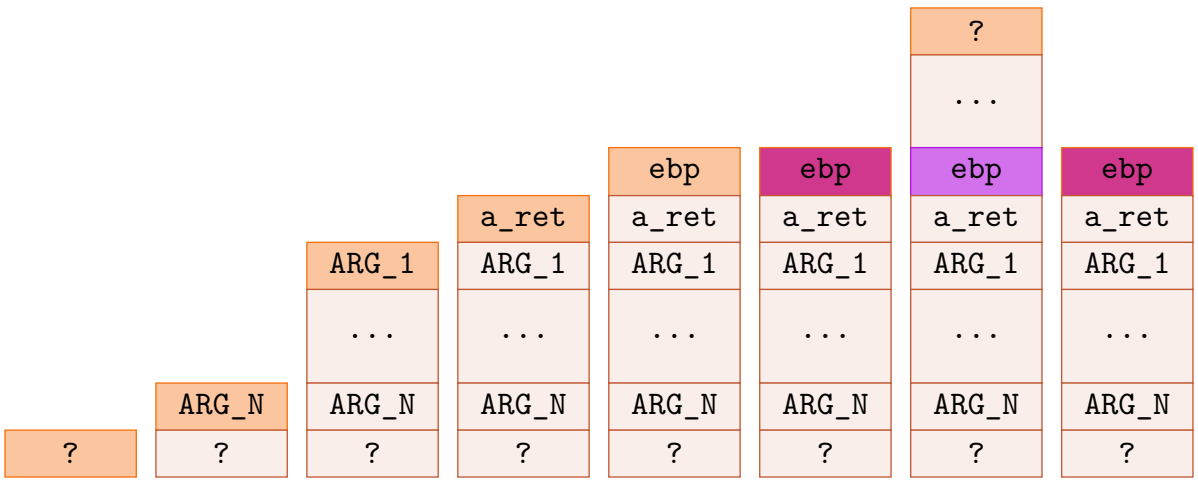
Analysons l'état de la pile, étape par étape, lors de l'appel d'une fonction.

```
push ARG_N
...
push ARG_1
call NOM_FCT
ADD esp, 4 * N
```

→

```
NOM_FCT:
    push ebp
    mov ebp, esp
    INSTR
    pop ebp
    ret
```

Légende : en orange, la zone de la pile pointée par esp, en violet, la zone de la pile pointée par ebp, a_ret est l'adresse de l'instr. qui suit le call.



Fonctions — appel et état de la pile

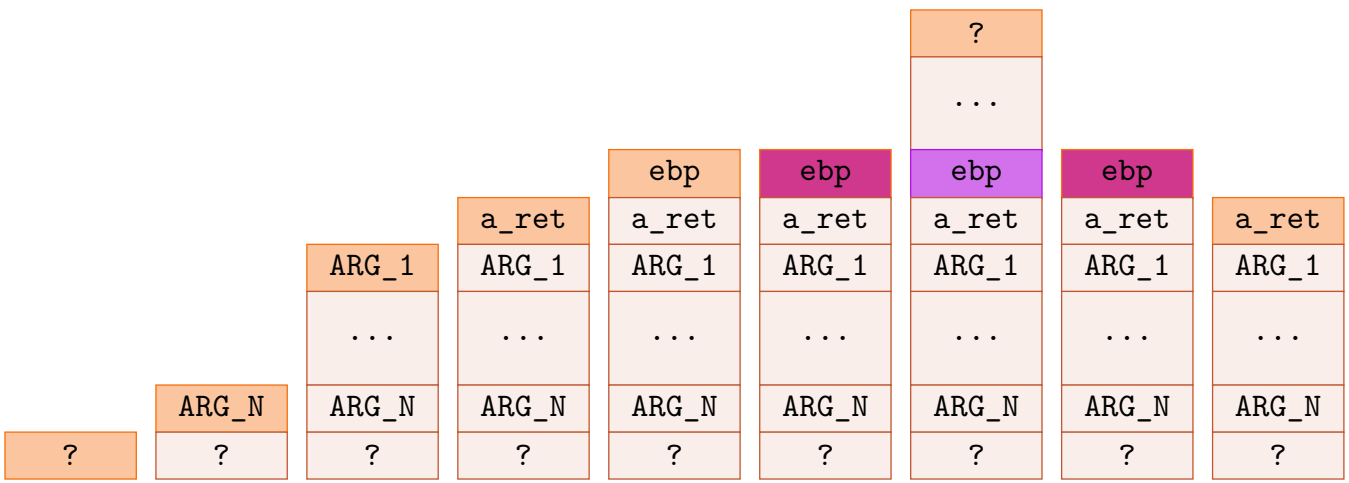
Analysons l'état de la pile, étape par étape, lors de l'appel d'une fonction.

```
push ARG_N
...
push ARG_1
call NOM_FCT
ADD esp, 4 * N
```

→

```
NOM_FCT:
    push ebp
    mov ebp, esp
    INSTR
    pop ebp
    ret
```

Légende : en orange, la zone de la pile pointée par esp, en violet, la zone de la pile pointée par ebp, a_ret est l'adresse de l'instr. qui suit le call.



Fonctions — appel et état de la pile

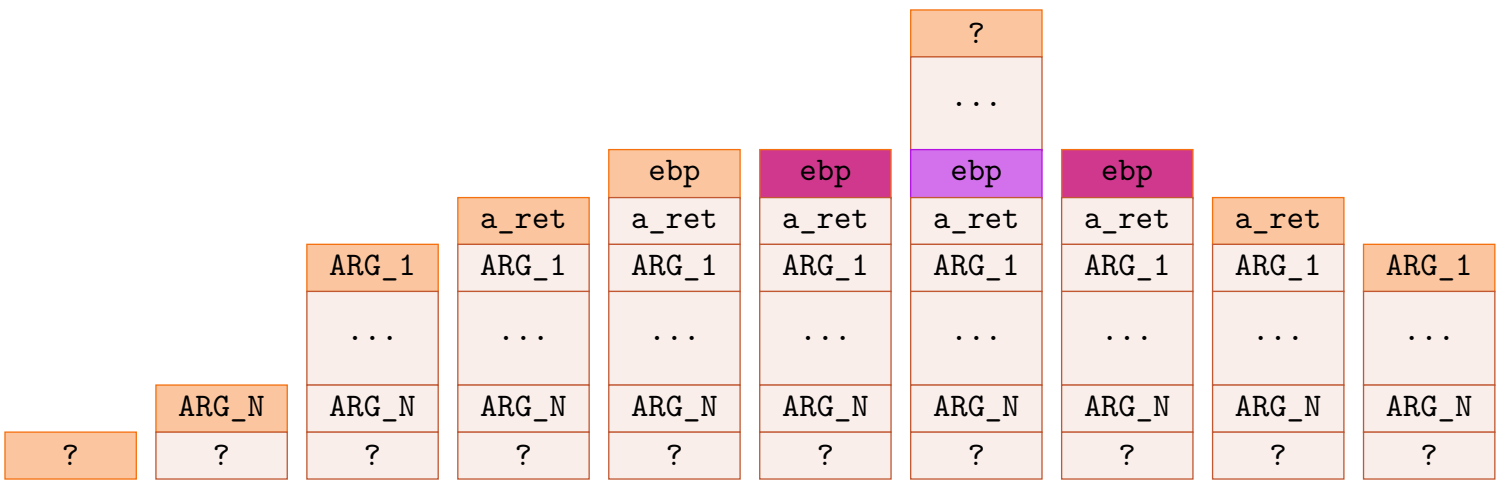
Analysons l'état de la pile, étape par étape, lors de l'appel d'une fonction.

```
push ARG_N
...
push ARG_1
call NOM_FCT
ADD esp, 4 * N
```

→

```
NOM_FCT:
    push ebp
    mov ebp, esp
    INSTR
    pop ebp
    ret
```

Légende : en orange, la zone de la pile pointée par esp, en violet, la zone de la pile pointée par ebp, a_ret est l'adresse de l'instr. qui suit le call.



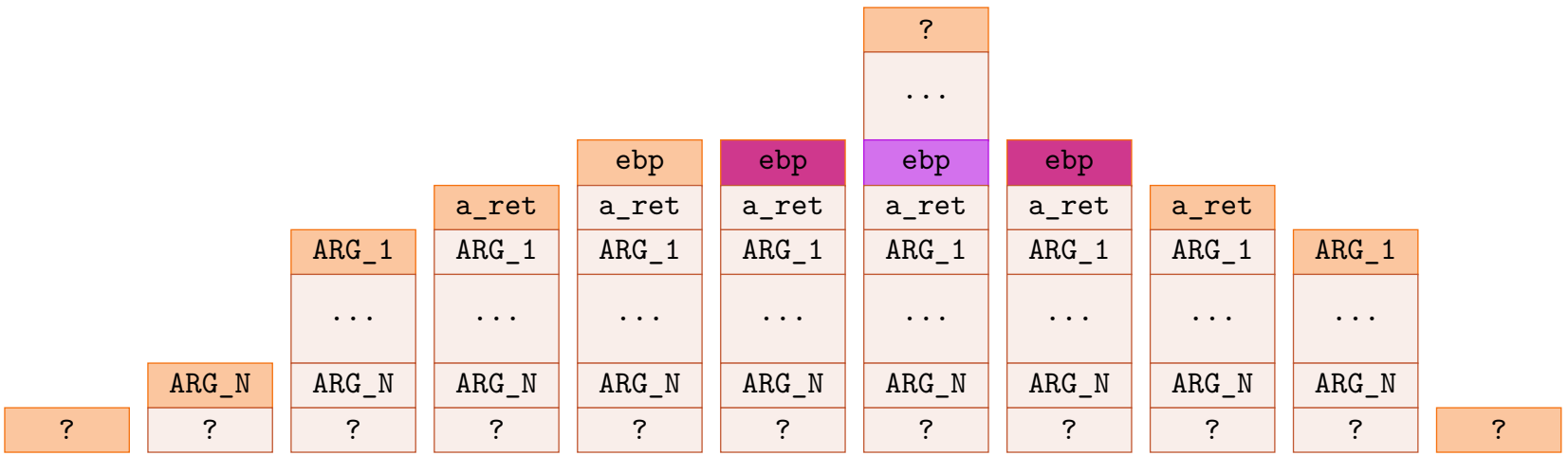
Fonctions — appel et état de la pile

Analysons l'état de la pile, étape par étape, lors de l'appel d'une fonction.

```
        push ARG_N
        ...
        push ARG_1
        call NOM_FCT
→       ADD esp, 4 * N
```

```
NOM_FCT:
    push ebp
    mov ebp, esp
    INSTR
    pop ebp
    ret
```

Légende : en orange, la zone de la pile pointée par esp, en violet, la zone de la pile pointée par ebp, a_ret est l'adresse de l'instr. qui suit le `call`.



Fonctions — préservation de l'état de la pile

L'état de la pile doit être préservé par le bloc d'instructions `INSTR`.

Fonctions — préservation de l'état de la pile

L'état de la pile doit être préservé par le bloc d'instructions `INSTR`.

Cela signifie que l'état de la pile juste avant d'exécuter `INSTR` et son état juste après son exécution sont les mêmes. En d'autres termes,

- `esp` doit posséder la même valeur ;
- toutes les données de la pile d'adresses plus grandes que `esp` ne doivent pas été modifiées.

Fonctions — préservation de l'état de la pile

L'état de la pile doit être préservé par le bloc d'instructions `INSTR`.

Cela signifie que l'état de la pile juste avant d'exécuter `INSTR` et son état juste après son exécution sont les mêmes. En d'autres termes,

- `esp` doit posséder la même valeur ;
- toutes les données de la pile d'adresses plus grandes que `esp` ne doivent pas été modifiées.

C'est bien le cas si

- `INSTR` contient autant de `push` que de `pop` ;
- à tout instant, il y a au moins autant de `push` que de `pop` qui ont été exécutés.

Fonctions — préservation de l'état de la pile

L'état de la pile doit être préservé par le bloc d'instructions `INSTR`.

Cela signifie que l'état de la pile juste avant d'exécuter `INSTR` et son état juste après son exécution sont les mêmes. En d'autres termes,

- `esp` doit posséder la même valeur ;
- toutes les données de la pile d'adresses plus grandes que `esp` ne doivent pas été modifiées.

C'est bien le cas si

- `INSTR` contient autant de `push` que de `pop` ;
- à tout instant, il y a au moins autant de `push` que de `pop` qui ont été exécutés.

Il faut bien respecter ces deux conditions dans la pratique.

Fonctions — rôle de ebp

La valeur de esp est susceptible de changer dans INSTR. C'est pour cela que l'on sauvegarde sa valeur, à l'entrée de la fonction, dans ebp.

Fonctions — rôle de ebp

La valeur de esp est susceptible de changer dans INSTR. C'est pour cela que l'on sauvegarde sa valeur, à l'entrée de la fonction, dans ebp.

Le registre ebp sert ainsi, dans INSTR, à **accéder** aux **arguments**. En effet, l'adresse du 1er argument est `ebp + 8`, celle du 2e est `ebp + 12` et plus généralement, celle du `i`e argument est

```
ebp + 4 * (i + 1)
```

Fonctions — rôle de ebp

La valeur de esp est susceptible de changer dans INSTR. C'est pour cela que l'on sauvegarde sa valeur, à l'entrée de la fonction, dans ebp.

Le registre ebp sert ainsi, dans INSTR, à **accéder** aux **arguments**. En effet, l'adresse du 1er argument est ebp + 8, celle du 2e est ebp + 12 et plus généralement, celle du i^{e} argument est

```
ebp + 4 * (i + 1)
```

On sauvegarde et on restaure tout de même, par un push ebp et pop ebp l'état de ebp à l'entrée et à la sortie de la fonction.

Fonctions — rôle de ebp

La valeur de esp est susceptible de changer dans `INSTR`. C'est pour cela que l'on sauvegarde sa valeur, à l'entrée de la fonction, dans ebp.

Le registre ebp sert ainsi, dans `INSTR`, à **accéder** aux **arguments**. En effet, l'adresse du 1er argument est `ebp + 8`, celle du 2e est `ebp + 12` et plus généralement, celle du `i`e argument est

```
ebp + 4 * (i + 1)
```

On sauvegarde et on restaure tout de même, par un `push ebp` et `pop ebp` l'état de ebp à l'entrée et à la sortie de la fonction.

Ce même mécanisme peut être utilisé pour sauvegarder/restaurer l'état des registres de travail `eax` (sauf si la fonction renvoie une valeur), `ebx`, `ecx` et `edx`.

Fonctions — exemple 1

Voici un exemple de fonction :

```
; Fonction qui renvoie la somme de deux
; entiers
; Arguments :
;       (1) une valeur entière signée
;       sur 4 octets
;       (2) une valeur entière signée
;       sur 4 octets
; Renvoi : la somme des deux
; arguments
somme:
    push ebp
    mov ebp, esp

    mov eax, [ebp + 8]
    add eax, [ebp + 12]

    pop ebp
    ret
```

Fonctions — exemple 1

Voici un exemple de fonction :

```
; Fonction qui renvoie la somme de deux
; entiers
; Arguments :
;       (1) une valeur entière signée
;       sur 4 octets
;       (2) une valeur entière signée
;       sur 4 octets
; Renvoi : la somme des deux
; arguments
somme:
    push ebp
    mov ebp, esp

    mov eax, [ebp + 8]
    add eax, [ebp + 12]

    pop ebp
    ret
```

Pour calculer dans eax la somme de $(43)_{\text{dix}}$ et $(1996)_{\text{dix}}$, on utilise

```
push 1996
push 43
call somme
add esp, 8
```

Fonctions — exemple 1

Voici un exemple de fonction :

```
; Fonction qui renvoie la somme de deux
; entiers
; Arguments :
;       (1) une valeur entière signée
;       sur 4 octets
;       (2) une valeur entière signée
;       sur 4 octets
; Renvoi : la somme des deux
; arguments
somme:
    push ebp
    mov ebp, esp

    mov eax, [ebp + 8]
    add eax, [ebp + 12]

    pop ebp

    ret
```

Pour calculer dans eax la somme de $(43)_{\text{dix}}$ et $(1996)_{\text{dix}}$, on utilise

```
push 1996
push 43
call somme
add esp, 8
```

Rappel 1 : on empile les arguments dans l'ordre inverse de ce que la fonction attend.

Fonctions — exemple 1

Voici un exemple de fonction :

```
; Fonction qui renvoie la somme de deux
; entiers
; Arguments :
;       (1) une valeur entière signée
;       sur 4 octets
;       (2) une valeur entière signée
;       sur 4 octets
; Renvoi : la somme des deux
; arguments
somme:
    push ebp
    mov ebp, esp

    mov eax, [ebp + 8]
    add eax, [ebp + 12]

    pop ebp
    ret
```

Pour calculer dans eax la somme de $(43)_{\text{dix}}$ et $(1996)_{\text{dix}}$, on utilise

```
push 1996
push 43
call somme
add esp, 8
```

Rappel 1 : on empile les arguments dans l'ordre inverse de ce que la fonction attend.

Rappel 2 : on ajoute 8 a esp après l'appel pour dépiler d'un seul coup les deux arguments ($8 = 2 \times 4$).

Fonctions — exemple 2

```
; Fonction qui affiche un
; caractere
; Arguments :
;       (1) valeur du
;       caractere a afficher.
; Renvoi : rien.
```

Fonctions — exemple 2

```
; Fonction qui affiche un
; caractere
; Arguments :
;       (1) valeur du
;       caractere a afficher.
; Renvoi : rien.

print_char:
 ; Debut
    push ebp
    mov ebp, esp
```

```
; Fin
    pop ebp
    ret
```

Fonctions — exemple 2

```
; Fonction qui affiche un
; caractere
; Arguments :
;       (1) valeur du
;       caractere a afficher.
; Renvoi : rien.

print_char:
 ; Debut
    push ebp
    mov ebp, esp
```

```
; Affichage
    mov ebx, 1
    mov ecx, ebp
    add ecx, 8
    mov edx, 1
    mov eax, 4
    int 0x80
```

```
; Fin
    pop ebp
    ret
```

Fonctions — exemple 2

```
; Fonction qui affiche un
; caractere
; Arguments :
;       (1) valeur du
;       caractere a afficher.
; Renvoi : rien.

print_char:
 ; Debut
    push ebp
    mov ebp, esp

 ; Sauv. des reg.
    push eax
    push ebx
    push ecx
    push edx

 ; Affichage
    mov ebx, 1
    mov ecx, ebp
    add ecx, 8
    mov edx, 1
    mov eax, 4
    int 0x80

 ; Rest. des reg.
    pop edx
    pop ecx
    pop ebx
    pop eax

 ; Fin
    pop ebp
    ret
```

Fonctions — exemple 2 bis

```
; Fonction qui affiche un
; caractere
; Arguments :
;       (1) adresse du
;       caractere a afficher.
; Renvoi : rien.
```

Fonctions — exemple 2 bis

```
; Fonction qui affiche un
; caractere
; Arguments :
;       (1) adresse du
;       caractere a afficher.
; Renvoi : rien.

print_char_2:
 ; Debut
    push ebp
    mov ebp, esp
```

```
; Fin
    pop ebp
    ret
```

Fonctions — exemple 2 bis

```
; Fonction qui affiche un
; caractere
; Arguments :
;       (1) adresse du
;       caractere a afficher.
; Renvoi : rien.

print_char_2:
 ; Debut
    push ebp
    mov ebp, esp

; Affichage
    mov ebx, 1
    mov ecx, [ebp + 8]
    mov edx, 1
    mov eax, 4
    int 0x80

; Fin
    pop ebp
    ret
```

Fonctions — exemple 2 bis

```
; Fonction qui affiche un
; caractere
; Arguments :
;       (1) adresse du
;       caractere a afficher.
; Renvoi : rien.

print_char_2:
 ; Debut
    push ebp
    mov ebp, esp

 ; Sauv. des reg.
    push eax
    push ebx
    push ecx
    push edx

 ; Affichage
    mov ebx, 1
    mov ecx, [ebp + 8]
    mov edx, 1
    mov eax, 4
    int 0x80

 ; Rest. des reg.
    pop edx
    pop ecx
    pop ebx
    pop eax

 ; Fin
    pop ebp
    ret
```

Fonctions — exemples 2 et 2 bis

Les deux fonctions `print_char` et `print_char_2` ne s'utilisent pas de la même manière : la 1re prend son argument par valeur, tandis que la 2e prend son argument par adresse.

Fonctions — exemples 2 et 2 bis

Les deux fonctions `print_char` et `print_char_2` ne s'utilisent pas de la même manière : la 1re prend son argument par valeur, tandis que la 2e prend son argument par adresse.

Ainsi, pour afficher p.ex. le caractère `'W'`, on appelle `print_char` par

```
push 'W'
call print_char
add esp, 4
```

Fonctions — exemples 2 et 2 bis

Les deux fonctions `print_char` et `print_char_2` ne s'utilisent pas de la même manière : la 1re prend son argument par valeur, tandis que la 2e prend son argument par adresse.

Ainsi, pour afficher p.ex. le caractère `'W'`, on appelle `print_char` par

```
push 'W'
call print_char
add esp, 4
```

La fonction `print_char_2` s'appelle par

```
push c1
call print_char_2
add esp, 4
```

où `c1` est l'adresse d'un caractère en mémoire. Celle-ci a été définie par exemple par `c1 : db 'W'` dans la section de données.

Fonctions — exemple 3

```
; Fonction qui affiche une
; chaine de caracteres
; Arguments :
;       (1) adresse de la
;       chaine de caracteres.
; Renvoi : nbr carac. aff.
```

Fonctions — exemple 3

```
; Fonction qui affiche une
; chaine de caracteres
; Arguments :
;       (1) adresse de la
;       chaine de caracteres.
; Renvoi : nbr carac. aff.

print_string:
 ; Debut
    push ebp
    mov ebp, esp
```

```
; Fin
    pop ebp
    ret
```

Fonctions — exemple 3

```
; Fonction qui affiche une
; chaine de caracteres
; Arguments :
;       (1) adresse de la
;       chaine de caracteres.
; Renvoi : nbr carac. aff.

print_string:
 ; Debut
    push ebp
    mov ebp, esp



 ; Adresse debut chaine
    mov ebx, [ebp + 8]
```

```
; Fin
    pop ebp
    ret
```

Fonctions — exemple 3

```
; Fonction qui affiche une
; chaine de caracteres
; Arguments :
;       (1) adresse de la
;       chaine de caracteres.
; Renvoi : nbr carac. aff.

print_string:
 ; Debut
    push ebp
    mov ebp, esp



 ; Adresse debut chaine
    mov ebx, [ebp + 8]

 ; Init. compteur
    mov eax, 0
```

```
 ; Fin
    pop ebp
    ret
```

Fonctions — exemple 3

```
; Fonction qui affiche une
; chaine de caracteres
; Arguments :
;       (1) adresse de la
;       chaine de caracteres.
; Renvoi : nbr carac. aff.

print_string:
 ; Debut
    push ebp
    mov ebp, esp


 ; Adresse debut chaine
    mov ebx, [ebp + 8]

 ; Init. compteur
    mov eax, 0
```

```
; Boucle d'affichage
    boucle:
        cmp byte [ebx], 0
        je fin_boucle
        push dword [ebx]
        call print_char
        add esp, 4
        add ebx, 1
        add eax, 1
        jmp boucle
    fin_boucle:


; Fin
    pop ebp
    ret
```

Fonctions — exemple 3

```
; Fonction qui affiche une
; chaine de caracteres
; Arguments :
;       (1) adresse de la
;       chaine de caracteres.
; Renvoi : nbr carac. aff.

print_string:
; Debut
    push ebp
    mov ebp, esp

; Sauv. des reg.
    push ebx

; Adresse debut chaine
    mov ebx, [ebp + 8]

; Init. compteur
    mov eax, 0

; Boucle d'affichage
    boucle:
        cmp byte [ebx], 0
        je fin_boucle
        push dword [ebx]
        call print_char
        add esp, 4
        add ebx, 1
        add eax, 1
        jmp boucle
    fin_boucle:

; Fin
    pop ebp
    ret
```

Fonctions — exemple 3

```
; Fonction qui affiche une
; chaine de caracteres
; Arguments :
;       (1) adresse de la
;       chaine de caracteres.
; Renvoi : nbr carac. aff.

print_string:
; Debut
    push ebp
    mov ebp, esp

; Sauv. des reg.
    push ebx

; Adresse debut chaine
    mov ebx, [ebp + 8]

; Init. compteur
    mov eax, 0

; Boucle d'affichage
    boucle:
        cmp byte [ebx], 0
        je fin_boucle
        push dword [ebx]
        call print_char
        add esp, 4
        add ebx, 1
        add eax, 1
        jmp boucle
    fin_boucle:

; Rest. des reg.
    pop ebx

; Fin
    pop ebp
    ret
```

Fonctions — exemple 4

On souhaite écrire une fonction pour calculer récursivement le n^{e} nombre triangulaire $\mathrm{triangle}(n)$ défini par

$$\mathrm{triangle}(n) := \begin{cases} 0 & \text{si } n = 0, \\ n + \mathrm{triangle}(n-1) & \text{sinon.} \end{cases}$$

```
; Fonction de calcul de
; nombres triangulaires.
; Arguments :
;   (1) entier positif
; Renvoi : le nombre
; triangulaire de l'arg.
triangle:

    ; Debut
    push ebp
    mov ebp, esp

    ; Sauv. des reg.
    push ebx
    push ecx
    push edx

    ; Sauv. de l'arg.
    mov ebx, [ebp + 8]

    cmp ebx, 0
    je cas_terminal
    cas_non_terminal:

        ; Prepa. appel rec.
        mov ecx, ebx
        sub ecx, 1

        ; Appel rec.
        push ecx
        call triangle
        add esp, 4

        ; Calcul res.
        add eax, ebx

        jmp fin

    cas_terminal:
        mov eax, 0

    fin:
    ; Rest. des reg.
    pop edx
    pop ecx
    pop ebx

    ; Fin
    pop ebp
    ret
```

Fonctions — exemple 5

On souhaite écrire une fonction pour calculer récursivement la factorielle $\mathrm{fact}(n)$ de tout entier positif n. On rappelle que $\mathrm{fact}(n)$ est défini par

$$\mathrm{fact}(n) := \begin{cases} 1 & \text{si } n = 0, \\ n \times \mathrm{fact}(n-1) & \text{sinon.} \end{cases}$$

```
; Fonction de calcul de
; la factorielle
; Arguments :
;   (1) entier positif
; Renvoi : la factorielle
; de l'argument
fact:

    ; Debut
    push ebp
    mov ebp, esp

    ; Sauv. des reg.
    push ebx
    push ecx
    push edx

    ; Sauv. de l'arg.
    mov ebx, [ebp + 8]

    cmp ebx, 0
    je cas_terminal
    cas_non_terminal:

        ; Prepa. appel rec.
        mov ecx, ebx
        sub ecx, 1

        ; Appel rec.
        push ecx
        call fact
        add esp, 4

        ; Calcul res.
        mul ebx

        jmp fin

    cas_terminal:
        mov eax, 1

    fin:
    ; Rest. des reg.
    pop edx
    pop ecx
    pop ebx

    ; Fin
    pop ebp
    ret
```

Fonctions — exemple 6

On souhaite écrire une fonction pour calculer récursivement le n^{e} nombre de Fibonacci $\mathrm{fibo}(n)$. On rappelle que $\mathrm{fibo}(n)$ est défini par

$$\mathrm{fibo}(n) := \begin{cases} n & \text{si } n \leqslant 1, \\ \mathrm{fibo}(n-1) + \mathrm{fibo}(n-2) & \text{sinon.} \end{cases}$$

```
; Fonction de calcul de
; nombres de Fibonacci
; Arguments :
;    (1) entier positif
; Renvoi : le nombre de
; Fibonacci de l'argument
fibo:
    ; Debut
    push ebp
    mov ebp, esp
    ; Sauv. des reg.
    push ebx
    push ecx
    push edx

    ; Sauvegarde de l'arg.
    mov ebx, [ebp + 8]

    cmp ebx, 1
    jle cas_terminaux

cas_non_terminal :
    ; Prepa. ap. rec. 1
    mov ecx, ebx
    sub ecx, 1
    ; Appel rec. 1
    push ecx
    call fibo
    add esp, 4

    ; Sauv. res. 1 pile
    push eax

    ; Prepa. ap. rec. 2
    mov ecx, ebx
    sub ecx, 2
    ; Appel rec. 2
    push ecx
    call fibo
    add esp, 4

    ; Calcul res.
    pop ebx
    add eax, ebx

    jmp fin

cas_terminaux:
    mov eax, ebx

fin:

    ; Rest. des reg.
    pop edx
    pop ecx
    pop ebx

    pop ebp
    ret
```

Étiquettes locales

Dans un programme complet, il peut être difficile de gérer de nombreuses étiquettes de code.

Il existe pour cette raison la notion d'étiquettes locales, dont la syntaxe de définition et de référence est

```
.ETIQ:
```

Étiquettes locales

Dans un programme complet, il peut être difficile de gérer de nombreuses étiquettes de code.

Il existe pour cette raison la notion d'étiquettes locales, dont la syntaxe de définition et de référence est

`.ETIQ:`

P.ex.,

```
etiq_globale:
    .action:
        INSTR_1
        jmp .action

autre_etiq_globale:
    .action:
        INSTR_2
        jmp .action
```

déclare plusieurs étiquettes de code, dont deux du même nom et locales, `.action`.

Le 1er `jmp` saute à l'instruction correspondant au 1er `.action`, tandis que le 2^{e} `jmp` saute à l'instruction correspondant au 2^{e} `action`.

Étiquettes locales

Dans un programme complet, il peut être difficile de gérer de nombreuses étiquettes de code.

Il existe pour cette raison la notion d'étiquettes locales, dont la syntaxe de définition et de référence est

`.ETIQ:`

P.ex.,

```
etiq_globale:
    .action:
        INSTR_1
        jmp .action

autre_etiq_globale:
    .action:
        INSTR_2
        jmp .action
```

déclare plusieurs étiquettes de code, dont deux du même nom et locales, `.action`.

Le 1er `jmp` saute à l'instruction correspondant au 1er `.action`, tandis que le 2e `jmp` saute à l'instruction correspondant au 2e `action`.

Le noms absolus (`etiq_globale.action` et `autre_etiq_globale.action`) de ces étiquettes permettent de faire référence à la bonne si besoin est.

Programmation modulaire

Il est possible de réaliser des projets en assembleur sur plusieurs fichiers, découpés en modules.

Un module est constitué

- d'un **fichier source** d'extension `.asm` ;
- d'un **fichier d'en-tête** d'extension `.inc`.

Programmation modulaire

Il est possible de réaliser des projets en assembleur sur plusieurs fichiers, découpés en modules.

Un module est constitué

- d'un **fichier source** d'extension `.asm` ;
- d'un **fichier d'en-tête** d'extension `.inc`.

Seul le module qui contient la fonction principale `main` ne dispose pas de fichier d'en-tête.

Programmation modulaire

Il est possible de réaliser des projets en assembleur sur plusieurs fichiers, découpés en modules.

Un module est constitué

- d'un **fichier source** d'extension `.asm` ;
- d'un **fichier d'en-tête** d'extension `.inc`.

Seul le module qui contient la fonction principale `main` ne dispose pas de fichier d'en-tête.

Pour compiler un projet sur plusieurs fichiers, on se sert des commandes

```
nasm -f elf32 Main.asm
nasm -f elf32 M1.asm
...
nasm -f elf32 Mk.asm
ld -o Exec -melf_i386 -e main Main.o M1.o ... Mk.o
```

Programmation modulaire

Un module (non principal) contient une collection de fonctions destinées à **être utilisées depuis l'extérieur**.

Programmation modulaire

Un module (non principal) contient une collection de fonctions destinées à **être utilisées depuis l'extérieur**.

On autorise une étiquette d'instruction `ETIQ` à être visible depuis l'extérieur en ajoutant la ligne

```
global ETIQ
```

juste avant la définition de l'étiquette.

Programmation modulaire

Un module (non principal) contient une collection de fonctions destinées à **être utilisées depuis l'extérieur**.

On autorise une étiquette d'instruction `ETIQ` à être visible depuis l'extérieur en ajoutant la ligne

```
global ETIQ
```

juste avant la définition de l'étiquette.

De plus, on renseigne dans le fichier d'en-tête l'existence de la fonction par

```
extern ETIQ
```

Il est d'usage de documenter à cet endroit la fonction.

Programmation modulaire

Pour bénéficier des fonctions définies dans un module `M` dans un fichier `F.asm`, on invoque, au tout début de `F.asm`, la directive

```
%include "M.inc"
```

Uniquement les fonctions rendues visibles depuis l'extérieur de `M` peuvent être appelées dans `F.asm`.

Programmation modulaire

Pour bénéficier des fonctions définies dans un module `M` dans un fichier `F.asm`, on invoque, au tout début de `F.asm`, la directive

```
%include "M.inc"
```

Uniquement les fonctions rendues visibles depuis l'extérieur de `M` peuvent être appelées dans `F.asm`.

Voici p.ex., un module `ES` et son utilisation dans `Main.asm` :

```
; ES.inc

; Documentation...
extern print_char

; Documentation...
extern print_string
```

Programmation modulaire

Pour bénéficier des fonctions définies dans un module `M` dans un fichier `F.asm`, on invoque, au tout début de `F.asm`, la directive

```
%include "M.inc"
```

Uniquement les fonctions rendues visibles depuis l'extérieur de `M` peuvent être appelées dans `F.asm`.

Voici p.ex., un module `ES` et son utilisation dans `Main.asm` :

; ES.inc

```
; Documentation...
extern print_char

; Documentation...
extern print_string
```

; ES.asm

```
...
global print_char
print_char:
...
global print_string
print_string:
...
```

Programmation modulaire

Pour bénéficier des fonctions définies dans un module `M` dans un fichier `F.asm`, on invoque, au tout début de `F.asm`, la directive

```
%include "M.inc"
```

Uniquement les fonctions rendues visibles depuis l'extérieur de `M` peuvent être appelées dans `F.asm`.

Voici p.ex., un module `ES` et son utilisation dans `Main.asm` :

; ES.inc

```
; Documentation...
extern print_char

; Documentation...
extern print_string
```

; ES.asm

```
...
global print_char
print_char:
...
global print_string
print_string:
...
```

; Main.asm

```
%include "ES.inc"
...
call print_string
...
```

Plan

Plan

Étapes d'exécution d'une instruction

Pour des raisons pédagogiques, on se place sur une architecture simplifiée où toute instruction est exécutée selon le schéma suivant :

1. (**IF**, **I**nstruction **F**etch) chargement de l'instruction et mise à jour du pointeur d'instruction `eip` de sorte qu'il contienne l'adresse de la prochaine instruction à exécuter ;

Étapes d'exécution d'une instruction

Pour des raisons pédagogiques, on se place sur une architecture simplifiée où toute instruction est exécutée selon le schéma suivant :

1. (**IF**, **I**nstruction **F**etch) chargement de l'instruction et mise à jour du pointeur d'instruction `eip` de sorte qu'il contienne l'adresse de la prochaine instruction à exécuter ;
2. (**ID**, **I**nstruction **D**ecode) identification de l'instruction. Les arguments éventuels de l'instruction sont placés dans l'unité arithmétique et logique.

Étapes d'exécution d'une instruction

Pour des raisons pédagogiques, on se place sur une architecture simplifiée où toute instruction est exécutée selon le schéma suivant :

1. (**IF**, **I**nstruction **F**etch) chargement de l'instruction et mise à jour du pointeur d'instruction `eip` de sorte qu'il contienne l'adresse de la prochaine instruction à exécuter ;
2. (**ID**, **I**nstruction **D**ecode) identification de l'instruction. Les arguments éventuels de l'instruction sont placés dans l'unité arithmétique et logique.
3. (**EX**, **Ex**ecute) exécution de l'instruction par l'unité arithmétique et logique.

Étapes d'exécution d'une instruction

Pour des raisons pédagogiques, on se place sur une architecture simplifiée où toute instruction est exécutée selon le schéma suivant :

1. (**IF**, **I**nstruction **F**etch) chargement de l'instruction et mise à jour du pointeur d'instruction `eip` de sorte qu'il contienne l'adresse de la prochaine instruction à exécuter ;
2. (**ID**, **I**nstruction **D**ecode) identification de l'instruction. Les arguments éventuels de l'instruction sont placés dans l'unité arithmétique et logique.
3. (**EX**, **Ex**ecute) exécution de l'instruction par l'unité arithmétique et logique.
4. (**WB**, **W**rite **B**ack) écriture éventuelle dans les registres ou dans la mémoire.

Étapes d'exécution d'une instruction — exemple 1

Par exemple, l'instruction

```
add eax, [adr]
```

où `adr` est une adresse, est traitée de la manière suivante :

Étapes d'exécution d'une instruction — exemple 1

Par exemple, l'instruction

```
add eax, [adr]
```

où `adr` est une adresse, est traitée de la manière suivante :

1. (**IF**) l'instruction `add` est chargée et `eip` est incrémenté afin qu'il pointe vers la prochaine instruction ;

Étapes d'exécution d'une instruction — exemple 1

Par exemple, l'instruction

```
add eax, [adr]
```

où `adr` est une adresse, est traitée de la manière suivante :

1. (**IF**) l'instruction `add` est chargée et `eip` est incrémenté afin qu'il pointe vers la prochaine instruction ;
2. (**ID**) le système repère qu'il s'agit de l'instruction `add` et il lit les 4 octets situés à partir de l'adresse `adr` dans la mémoire, ainsi que la valeur du registre `eax` ;

Étapes d'exécution d'une instruction — exemple 1

Par exemple, l'instruction

```
add eax, [adr]
```

où `adr` est une adresse, est traitée de la manière suivante :

1. (**IF**) l'instruction `add` est chargée et `eip` est incrémenté afin qu'il pointe vers la prochaine instruction ;
2. (**ID**) le système repère qu'il s'agit de l'instruction `add` et il lit les 4 octets situés à partir de l'adresse `adr` dans la mémoire, ainsi que la valeur du registre `eax` ;
3. (**EX**) l'unité arithmétique et logique effectue l'addition entre les deux valeurs chargées ;

Étapes d'exécution d'une instruction — exemple 1

Par exemple, l'instruction

```
add eax, [adr]
```

où `adr` est une adresse, est traitée de la manière suivante :

1. (**IF**) l'instruction `add` est chargée et `eip` est incrémenté afin qu'il pointe vers la prochaine instruction ;
2. (**ID**) le système repère qu'il s'agit de l'instruction `add` et il lit les 4 octets situés à partir de l'adresse `adr` dans la mémoire, ainsi que la valeur du registre `eax` ;
3. (**EX**) l'unité arithmétique et logique effectue l'addition entre les deux valeurs chargées ;
4. (**WB**) le résultat ainsi calculé est écrit dans `eax`.

Étapes d'exécution d'une instruction — exemple 2

Par exemple, l'instruction

```
jmp adr
```

où `adr` est une adresse d'instruction, est traitée de la manière suivante :

Étapes d'exécution d'une instruction — exemple 2

Par exemple, l'instruction

```
jmp adr
```

où `adr` est une adresse d'instruction, est traitée de la manière suivante :

1. (**IF**) l'instruction `jmp` est chargée et `eip` est incrémenté afin qu'il pointe vers la prochaine instruction (il ne pointe donc pas encore forcément sur l'instruction d'adresse `adr` comme souhaité) ;

Étapes d'exécution d'une instruction — exemple 2

Par exemple, l'instruction

```
jmp adr
```

où `adr` est une adresse d'instruction, est traitée de la manière suivante :

1. (**IF**) l'instruction `jmp` est chargée et `eip` est incrémenté afin qu'il pointe vers la prochaine instruction (il ne pointe donc pas encore forcément sur l'instruction d'adresse `adr` comme souhaité) ;
2. (**ID**) le système repère qu'il s'agit de l'instruction `jmp` et il lit la valeur de l'adresse `adr` ;

Étapes d'exécution d'une instruction — exemple 2

Par exemple, l'instruction

```
jmp adr
```

où `adr` est une adresse d'instruction, est traitée de la manière suivante :

1. (**IF**) l'instruction `jmp` est chargée et `eip` est incrémenté afin qu'il pointe vers la prochaine instruction (il ne pointe donc pas encore forcément sur l'instruction d'adresse `adr` comme souhaité) ;
2. (**ID**) le système repère qu'il s'agit de l'instruction `jmp` et il lit la valeur de l'adresse `adr` ;
3. (**EX**) l'adresse d'instruction cible est calculée à partir de la valeur de `adr` ;

Étapes d'exécution d'une instruction — exemple 2

Par exemple, l'instruction

```
jmp adr
```

où `adr` est une adresse d'instruction, est traitée de la manière suivante :

1. (**IF**) l'instruction `jmp` est chargée et `eip` est incrémenté afin qu'il pointe vers la prochaine instruction (il ne pointe donc pas encore forcément sur l'instruction d'adresse `adr` comme souhaité) ;
2. (**ID**) le système repère qu'il s'agit de l'instruction `jmp` et il lit la valeur de l'adresse `adr` ;
3. (**EX**) l'adresse d'instruction cible est calculée à partir de la valeur de `adr` ;
4. (**WB**) l'adresse d'instruction cible est écrite dans `eip`.

Étapes d'exécution d'une instruction — horloge

L'horloge du processeur permet de rythmer ces étapes.

L'exécution de chacune de ces étapes demande au moins un cycle d'horloge.

Étapes d'exécution d'une instruction — horloge

L'horloge du processeur permet de rythmer ces étapes.

L'exécution de chacune de ces étapes demande au moins un cycle d'horloge.

Certaines opérations demandent plusieurs cycles d'horloge pour être traitées. La division (instruction `div`), par exemple, demande plusieurs dizaines de cycles d'horloge pour réaliser l'étape **EX**.

Étapes d'exécution d'une instruction — horloge

L'horloge du processeur permet de rythmer ces étapes.

L'exécution de chacune de ces étapes demande au moins un cycle d'horloge.

Certaines opérations demandent plusieurs cycles d'horloge pour être traitées. La division (instruction `div`), par exemple, demande plusieurs dizaines de cycles d'horloge pour réaliser l'étape **EX**.

La **vitesse d'horloge** d'un processeur est exprimée en hertz (Hz).

Un processeur dont la vitesse d'horloge est de 1 Hz évolue à 1 cycle d'horloge par seconde.

Étapes d'exécution d'une instruction — horloge

L'horloge du processeur permet de rythmer ces étapes.

L'exécution de chacune de ces étapes demande au moins un cycle d'horloge.

Certaines opérations demandent plusieurs cycles d'horloge pour être traitées. La division (instruction `div`), par exemple, demande plusieurs dizaines de cycles d'horloge pour réaliser l'étape **EX**.

La **vitesse d'horloge** d'un processeur est exprimée en hertz (Hz).

Un processeur dont la vitesse d'horloge est de 1 Hz évolue à 1 cycle d'horloge par seconde.

Problématique : comment optimiser l'exécution des instructions ?

Le travail à la chaîne

Considérons une usine qui produit des pots de confiture. L'instruction que suit l'usine est d'assembler, en boucle, des pots de confiture. Cette tâche se divise en quatre sous-tâches :

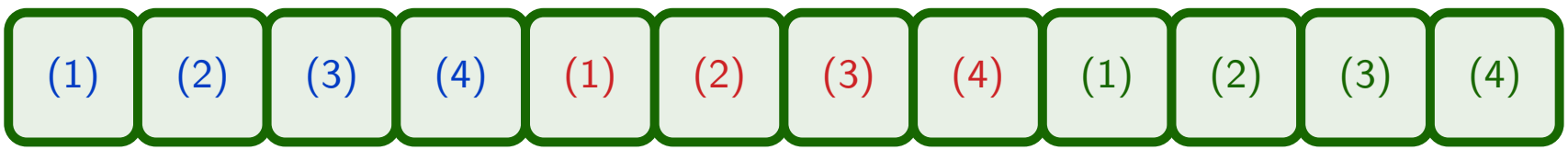
1. placer un pot vide sur le tapis roulant ;
2. remplir le pot de confiture ;
3. visser le couvercle ;
4. coller l'étiquette.

Le travail à la chaîne

Considérons une usine qui produit des pots de confiture. L'instruction que suit l'usine est d'assembler, en boucle, des pots de confiture. Cette tâche se divise en quatre sous-tâches :

1. placer un pot vide sur le tapis roulant ;
2. remplir le pot de confiture ;
3. visser le couvercle ;
4. coller l'étiquette.

La création séquentielle de trois pots de confiture s'organise de la manière suivante :



et nécessite $3 \times 4 = 12$ étapes.

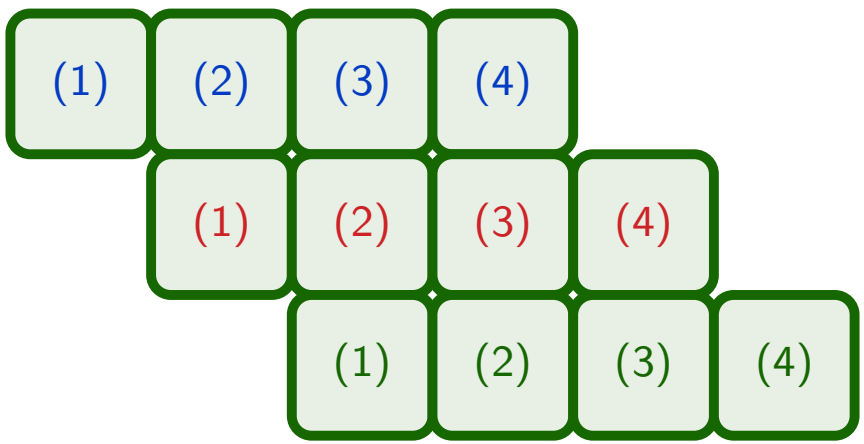
Le travail à la chaîne efficace

Observation : au lieu de réaliser ces assemblages séquentiellement, lorsqu'un pot est sujet à une sous-tâche (i), $1 \leqslant i \leqslant 4$, on peut appliquer à un autre pot une autre sous-tâche (j), $1 \leqslant j \neq i \leqslant 4$.

Le travail à la chaîne efficace

Observation : au lieu de réaliser ces assemblages séquentiellement, lorsqu'un pot est sujet à une sous-tâche (i), $1 \leqslant i \leqslant 4$, on peut appliquer à un autre pot une autre sous-tâche (j), $1 \leqslant j \neq i \leqslant 4$.

Cette organisation se schématise en



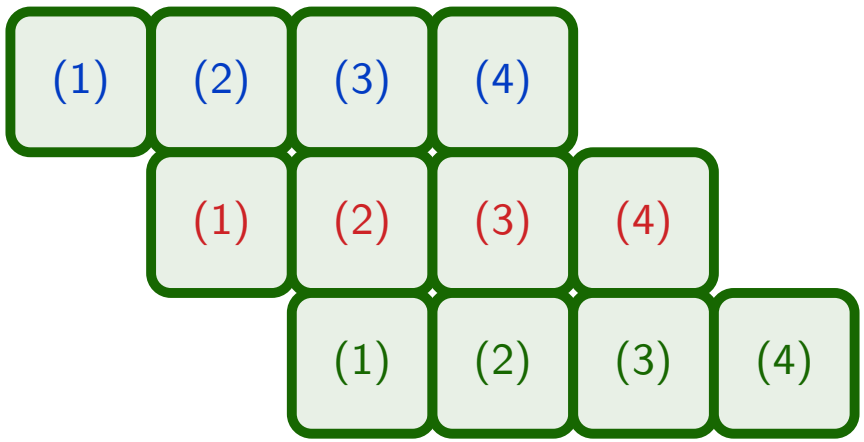
et nécessite $4 + 1 + 1 = 6$ étapes.

À chaque instant, toute sous-tâche est sollicitée au plus une fois.

Le travail à la chaîne efficace

Observation : au lieu de réaliser ces assemblages séquentiellement, lorsqu'un pot est sujet à une sous-tâche (i), $1 \leqslant i \leqslant 4$, on peut appliquer à un autre pot une autre sous-tâche (j), $1 \leqslant j \neq i \leqslant 4$.

Cette organisation se schématise en



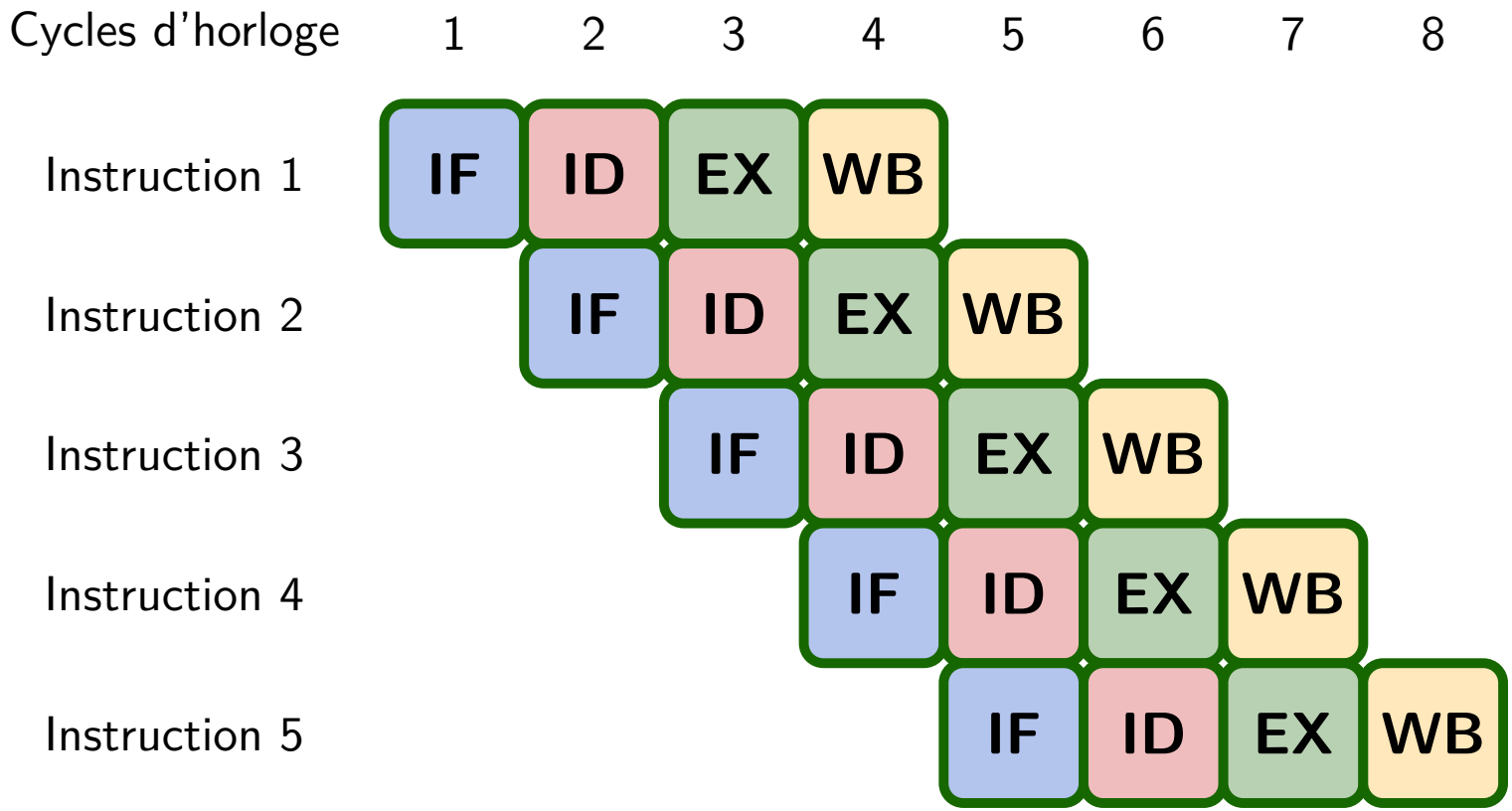
et nécessite $4 + 1 + 1 = 6$ étapes.

À chaque instant, toute sous-tâche est sollicitée au plus une fois.

Plusieurs sous-tâches sont exécutées en même temps.

Pipeline — schéma

Le pipeline (chaîne de traitement) permet l'organisation suivante :



Il permet (dans cet exemple) d'exécuter 5 instructions en 8 cycles d'horloge au lieu de 20.

Pipeline — informations

Le pipeline étudié ici est une variante simplifiée du *Classic RISC pipeline* introduit par D. Patterson.

Chaque étape est prise en charge par un étage du pipeline. Il dispose ici de quatre étages.

Pipeline — informations

Le pipeline étudié ici est une variante simplifiée du *Classic RISC pipeline* introduit par D. Patterson.

Chaque étape est prise en charge par un étage du pipeline. Il dispose ici de quatre étages.

Les processeurs modernes disposent de pipelines ayant bien plus d'étages :

Processeur	Nombre d'étages du pipeline
Intel Pentium 4 Prescott	31
Intel Pentium 4	20
Intel Core i7	14
AMD Athlon	12

Aléas

Étant donné que l'utilisation d'un pipeline induit un entrelacement dans l'exécution des instructions, des problèmes peuvent survenir. Ceux-ci sont appelés aléas et peuvent être de trois sortes :

1. aléas structurels (conflit d'utilisation d'une ressource matérielle) ;

Aléas

Étant donné que l'utilisation d'un pipeline induit un entrelacement dans l'exécution des instructions, des problèmes peuvent survenir. Ceux-ci sont appelés aléas et peuvent être de trois sortes :

1. aléas structurels (conflit d'utilisation d'une ressource matérielle) ;
2. aléas de donnée (dépendance d'un résultat d'une précédente instruction) ;

Aléas

Étant donné que l'utilisation d'un pipeline induit un entrelacement dans l'exécution des instructions, des problèmes peuvent survenir. Ceux-ci sont appelés aléas et peuvent être de trois sortes :

1. aléas structurels (conflit d'utilisation d'une ressource matérielle) ;
2. aléas de donnée (dépendance d'un résultat d'une précédente instruction) ;
3. aléas de contrôle (saut vers un autre endroit du code).

Aléas

Étant donné que l'utilisation d'un pipeline induit un entrelacement dans l'exécution des instructions, des problèmes peuvent survenir. Ceux-ci sont appelés aléas et peuvent être de trois sortes :

1. aléas structurels (conflit d'utilisation d'une ressource matérielle) ;
2. aléas de donnée (dépendance d'un résultat d'une précédente instruction) ;
3. aléas de contrôle (saut vers un autre endroit du code).

Une **solution générale** pour faire face aux aléas est de suspendre l'exécution de l'instruction qui pose problème jusqu'à ce que le problème se résolve. Cette mise en pause crée des « bulles » dans le pipeline.

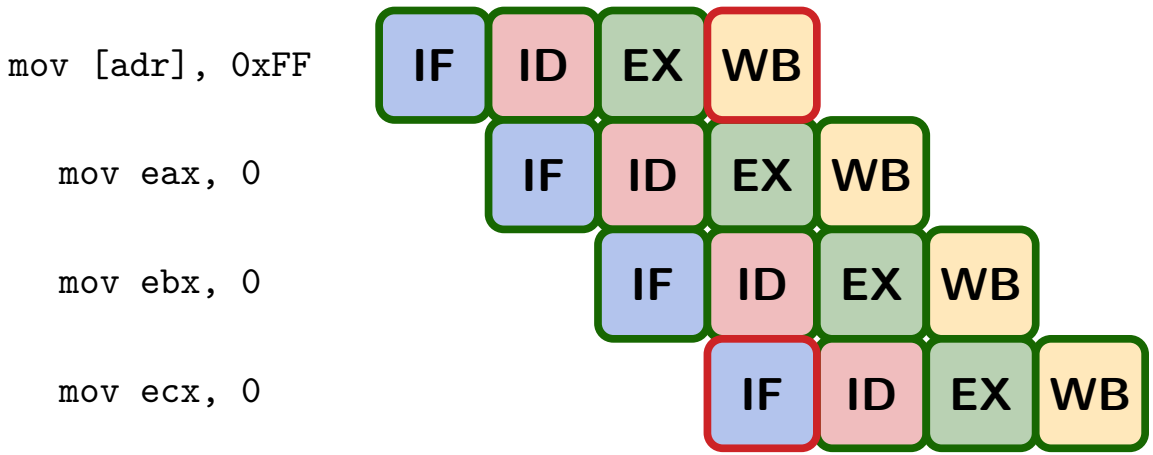
Aléas structurels

Un aléa structurel survient lorsque deux étapes d'exécution d'instructions souhaitent accéder au même moment à une même ressource.

Aléas structurels

Un aléa structurel survient lorsque deux étapes d'exécution d'instructions souhaitent accéder au même moment à une même ressource.

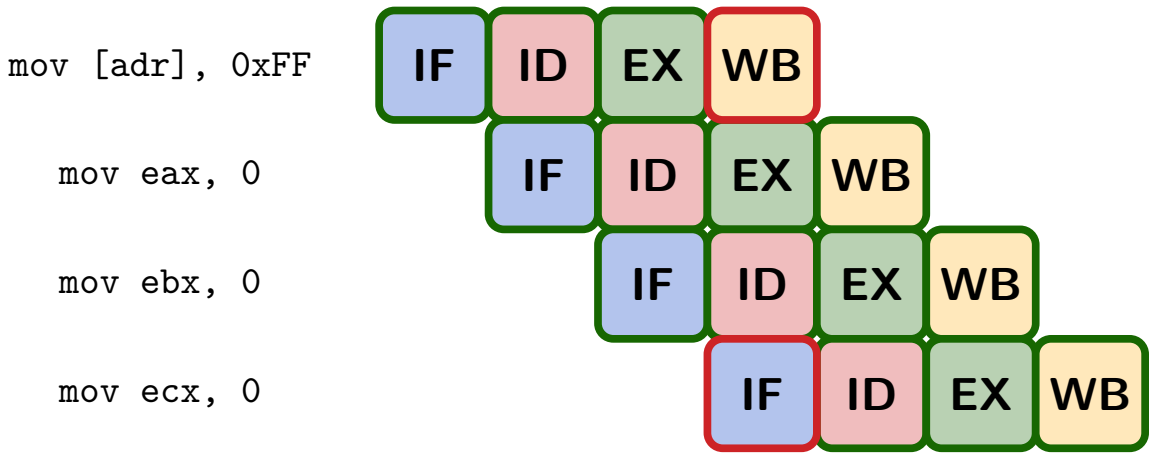
Considérons p.ex. un pipeline dans la configuration suivante :



Aléas structurels

Un aléa structurel survient lorsque deux étapes d'exécution d'instructions souhaitent accéder au même moment à une même ressource.

Considérons p.ex. un pipeline dans la configuration suivante :



Problème : le **WB** de la 1re instruction tente d'écrire en mémoire, tandis que le **IF** de la 4e instruction cherche à charger la 4e instruction. Toutes deux sollicitent au même moment le bus reliant l'unité arithmétique et logique et la mémoire.

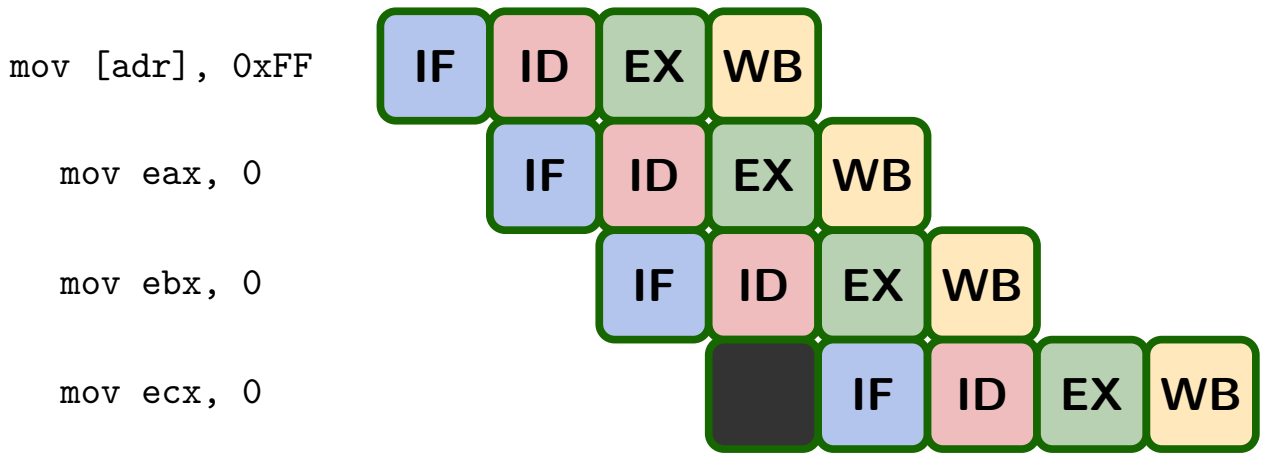
Aléas structurels

Solution 1. : temporiser la 4^{e} instruction en la précédant d'une bulle.

Aléas structurels

Solution 1. : temporiser la 4^{e} instruction en la précédant d'une bulle.

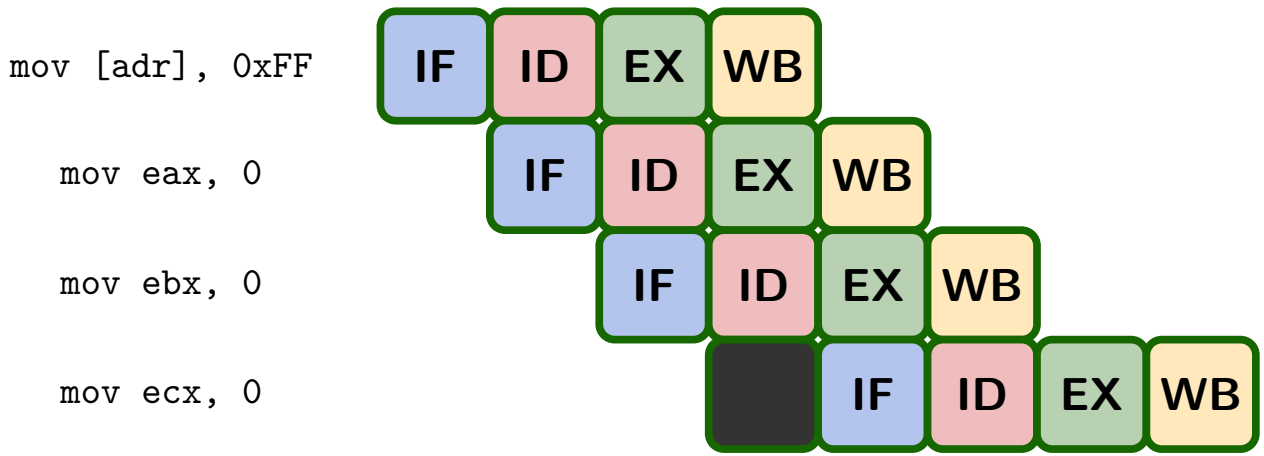
On obtient :



Aléas structurels

Solution 1. : temporiser la 4^{e} instruction en la précédant d'une bulle.

On obtient :



On continue d'appliquer ce raisonnement (en ajoutant des bulles) jusqu'à ce qu'il n'y ait plus d'aléa structurel.

Aléas structurels

Solution 2. : diviser la connection entre unité arithmétique et logique et la mémoire en deux bus.

L'un sera utilisé pour accéder aux données et l'autre, au programme.

Aléas structurels

Solution 2. : diviser la connection entre unité arithmétique et logique et la mémoire en deux bus.

L'un sera utilisé pour accéder aux données et l'autre, au programme.

Ceci ne résout tout de même pas l'aléa **ID** *vs* **WB** car

- d'une part, **ID** lit des données en mémoire (qui sont les arguments de l'instruction) ;

Aléas structurels

Solution 2. : diviser la connection entre unité arithmétique et logique et la mémoire en deux bus.

L'un sera utilisé pour accéder aux données et l'autre, au programme.

Ceci ne résout tout de même pas l'aléa **ID** *vs* **WB** car

- d'une part, **ID** lit des données en mémoire (qui sont les arguments de l'instruction) ;
- d'autre part, **WB** écrit des données en mémoire (dans le cas où l'instruction le demande).

Aléas structurels

Solution 2. : diviser la connection entre unité arithmétique et logique et la mémoire en deux bus.

L'un sera utilisé pour accéder aux données et l'autre, au programme.

Ceci ne résout tout de même pas l'aléa **ID** *vs* **WB** car

- d'une part, **ID** lit des données en mémoire (qui sont les arguments de l'instruction) ;
- d'autre part, **WB** écrit des données en mémoire (dans le cas où l'instruction le demande).

Il y a donc ici une sollicitation simultanée du bus d'accès aux données.

Aléas structurels

Solution 2. : diviser la connection entre unité arithmétique et logique et la mémoire en deux bus.

L'un sera utilisé pour accéder aux données et l'autre, au programme.

Ceci ne résout tout de même pas l'aléa **ID** *vs* **WB** car

- d'une part, **ID** lit des données en mémoire (qui sont les arguments de l'instruction) ;
- d'autre part, **WB** écrit des données en mémoire (dans le cas où l'instruction le demande).

Il y a donc ici une sollicitation simultanée du bus d'accès aux données.

On peut, pour améliorer cette solution, imaginer des architectures avec plusieurs bus connectant l'unité arithmétique et logique et les données.

Aléas de donnée

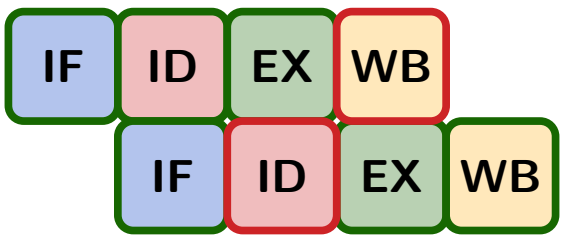
Un aléa de donnée survient lorsqu'une instruction I_1 a besoin d'un résultat calculé par une instruction I_0 précédente mais I_0 n'a pas encore produit un résultat exploitable.

Aléas de donnée

Un aléa de donnée survient lorsqu'une instruction I_1 a besoin d'un résultat calculé par une instruction I_0 précédente mais I_0 n'a pas encore produit un résultat exploitable.

Considérons un pipeline dans la configuration suivante :

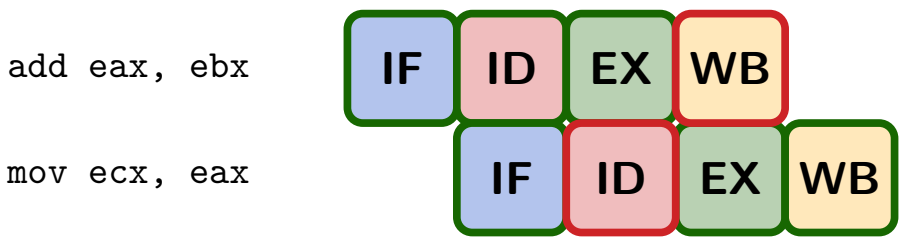
```
add eax, ebx
mov ecx, eax
```



Aléas de donnée

Un aléa de donnée survient lorsqu'une instruction I_1 a besoin d'un résultat calculé par une instruction I_0 précédente mais I_0 n'a pas encore produit un résultat exploitable.

Considérons un pipeline dans la configuration suivante :



Problème : le **ID** de la 2e instruction charge son argument `eax`. Cependant, ce chargement s'effectue avant que le **WB** de la 1re instruction n'ait été réalisé. C'est donc une mauvaise (la précédente) valeur de `eax` qui est chargée comme argument dans la 2e instruction.

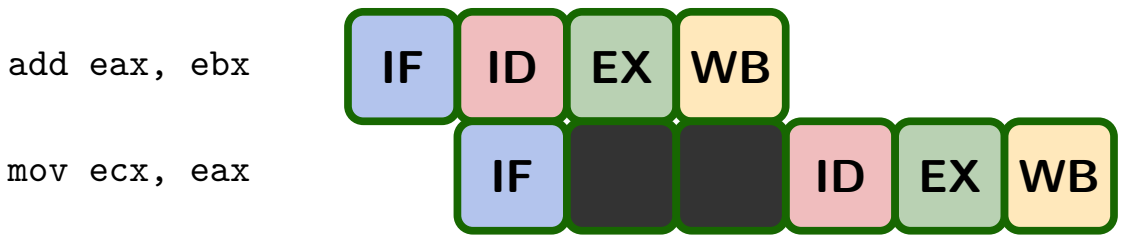
Aléas de donnée

Solution : temporiser les trois dernières étapes de la 2^{e} instruction en les précédant de deux bulles.

Aléas de donnée

Solution : temporiser les trois dernières étapes de la 2^{e} instruction en les précédant de deux bulles.

On obtient :

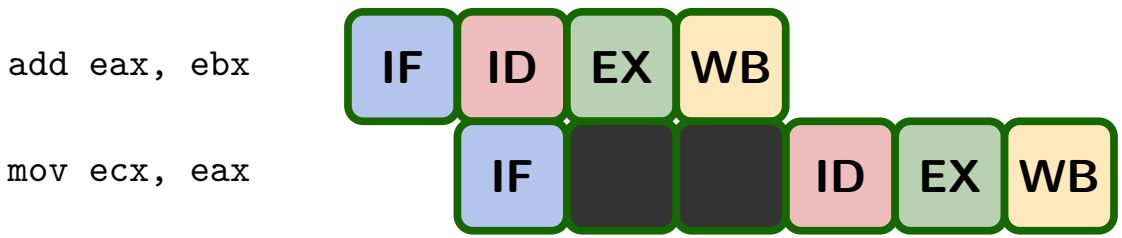


La seconde bulle est nécessaire pour éviter l'aléa structurel **ID** *vs* **WB**.

Aléas de donnée

Solution : temporiser les trois dernières étapes de la 2e instruction en les précédant de deux bulles.

On obtient :



La seconde bulle est nécessaire pour éviter l'aléa structurel **ID** *vs* **WB**.

On continue d'appliquer ce raisonnement (en ajoutant des bulles) jusqu'à ce qu'il n'y ait plus d'aléa de donnée ou structurel.

Aléas de donnée

Changer, lorsque cela est sémantiquement possible, l'ordre des instructions d'un programme peut avoir une conséquence sur le nombre de bulles créées dans le pipeline (et donc sur la vitesse d'exécution).

Aléas de donnée

Changer, lorsque cela est sémantiquement possible, l'ordre des instructions d'un programme peut avoir une conséquence sur le nombre de bulles créées dans le pipeline (et donc sur la vitesse d'exécution).

Considérons p.ex. les instructions en C suivantes :

```
a = a + b;
c = c + d;
```

Aléas de donnée

Changer, lorsque cela est sémantiquement possible, l'ordre des instructions d'un programme peut avoir une conséquence sur le nombre de bulles créées dans le pipeline (et donc sur la vitesse d'exécution).

Considérons p.ex. les instructions en C suivantes :

```
a = a + b;
c = c + d;
```

Il existe au moins deux manière de les traduire en assembleur :

```
mov eax, [adr_a]
mov ebx, [adr_b]
add eax, ebx
mov [adr_a], eax
mov ecx, [adr_c]
mov edx, [adr_d]
add ecx, edx
mov [adr_c], ecx
```

Aléas de donnée

Changer, lorsque cela est sémantiquement possible, l'ordre des instructions d'un programme peut avoir une conséquence sur le nombre de bulles créées dans le pipeline (et donc sur la vitesse d'exécution).

Considérons p.ex. les instructions en C suivantes :

```
a = a + b;
c = c + d;
```

Il existe au moins deux manière de les traduire en assembleur :

```
mov eax, [adr_a]
mov ebx, [adr_b]
add eax, ebx
mov [adr_a], eax
mov ecx, [adr_c]
mov edx, [adr_d]
add ecx, edx
mov [adr_c], ecx
```

```
mov eax, [adr_a]
mov ebx, [adr_b]
mov ecx, [adr_c]
mov edx, [adr_d]
add eax, ebx
add ecx, edx
mov [adr_a], eax
mov [adr_c], ecx
```

Aléas de donnée

La première version est une traduction directe du programme en C.

Elle provoque cependant de nombreux aléas de donnée à cause des instructions `mov` et `add` trop proches et opérant sur des mêmes données.

Aléas de donnée

La première version est une traduction directe du programme en C.

Elle provoque cependant de nombreux aléas de donnée à cause des instructions `mov` et `add` trop proches et opérant sur des mêmes données.

La seconde version utilise l'idée suivante.

On sépare deux instructions qui se partagent la même donnée par d'autres instructions qui leur sont indépendantes. Ceci permet de diminuer le nombre de bulles dans le pipeline.

Aléas de contrôle

Un aléa de contrôle survient systématiquement lorsqu'une instruction de saut est exécutée.

Aléas de contrôle

Un aléa de contrôle survient systématiquement lorsqu'une instruction de saut est exécutée.

Si I est une instruction de saut, il est possible qu'une instruction I' qui suit I dans le pipeline ne doive pas être exécutée. Il faut donc éviter l'étape **EX** de I' (parce qu'elle modifie la mémoire).

Aléas de contrôle

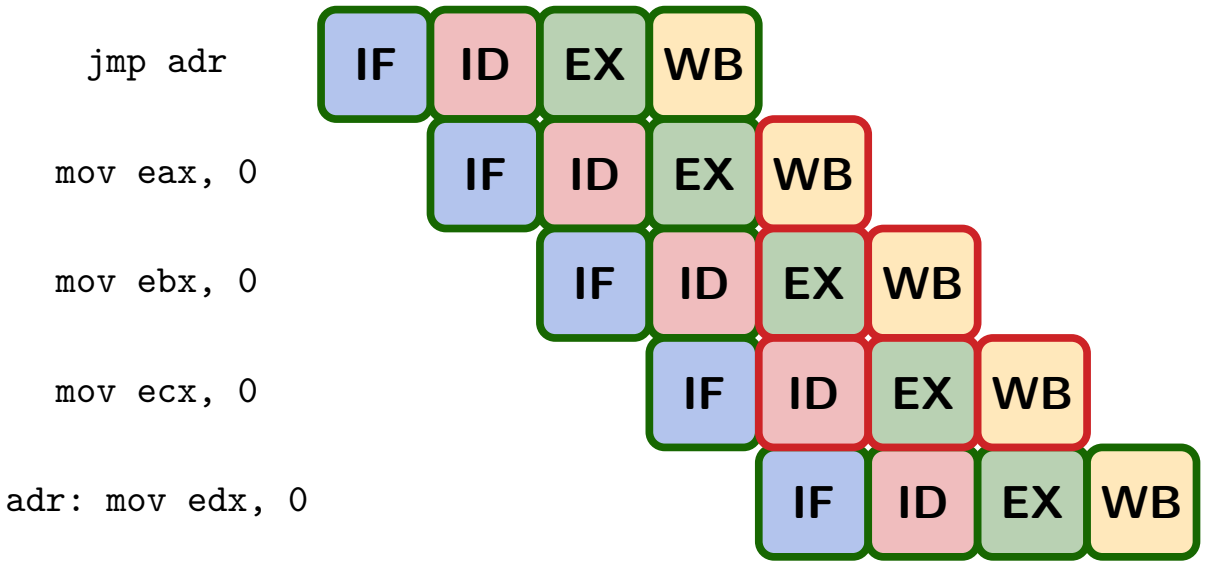
Un aléa de contrôle survient systématiquement lorsqu'une instruction de saut est exécutée.

Si I est une instruction de saut, il est possible qu'une instruction I' qui suit I dans le pipeline ne doive pas être exécutée. Il faut donc éviter l'étape **EX** de I' (parce qu'elle modifie la mémoire).

L'adresse cible d'une instruction de saut est calculée lors de l'étape **EX**. Ensuite, lors de l'étape **WB**, le registre `eip` est mis à jour.

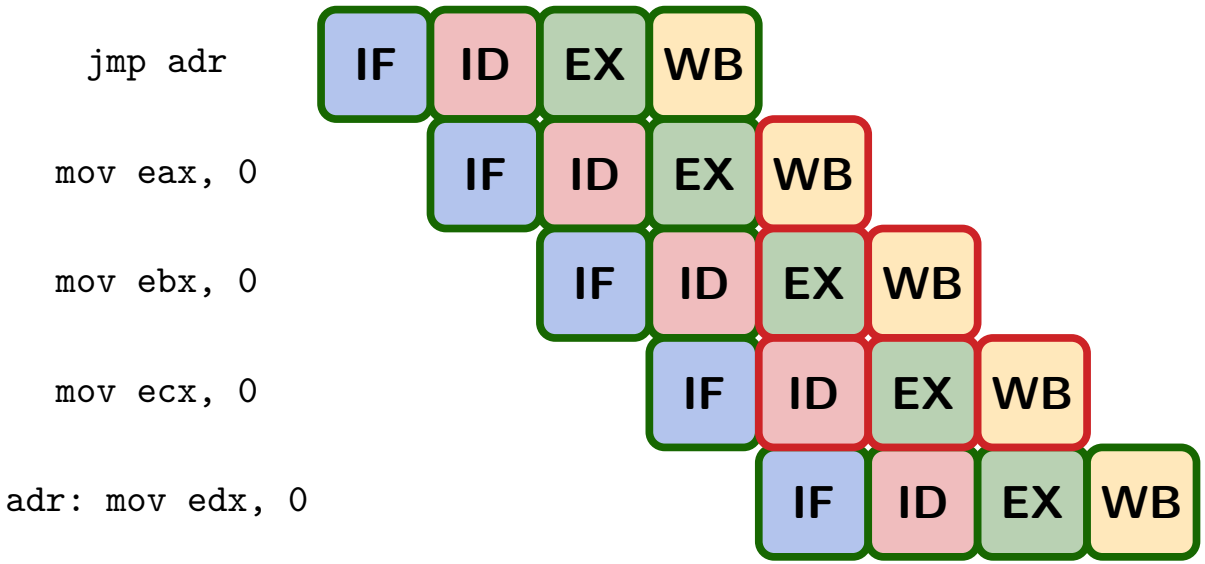
Aléas de contrôle

Considérons un pipeline dans la configuration suivante :



Aléas de contrôle

Considérons un pipeline dans la configuration suivante :



Problème : la 1re instruction, qui est un saut, ordonne le fait qu'il faut interrompre le plus tôt possible l'exécution des trois instructions suivantes (situées entre la source et la cible du saut).

Ces trois instructions sont chargées inutilement dans le pipeline.

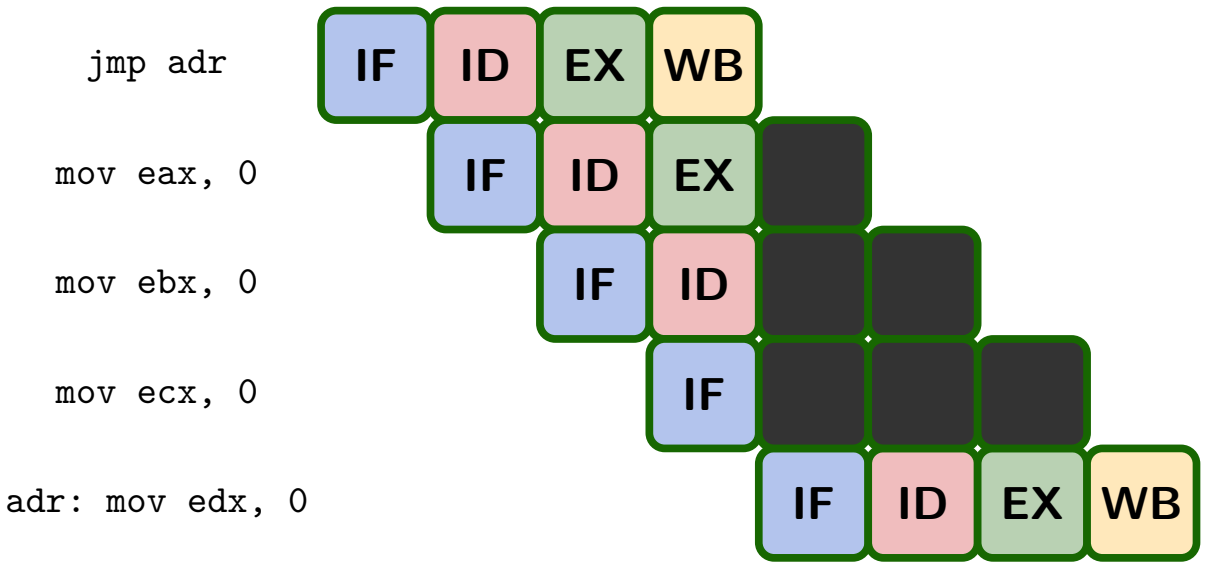
Aléas de contrôle

Solution : ne pas exécuter les étapes des instructions sautées après avoir exécuté le **WB** de l'instruction de saut.

Aléas de contrôle

Solution : ne pas exécuter les étapes des instructions sautées après avoir exécuté le **WB** de l'instruction de saut.

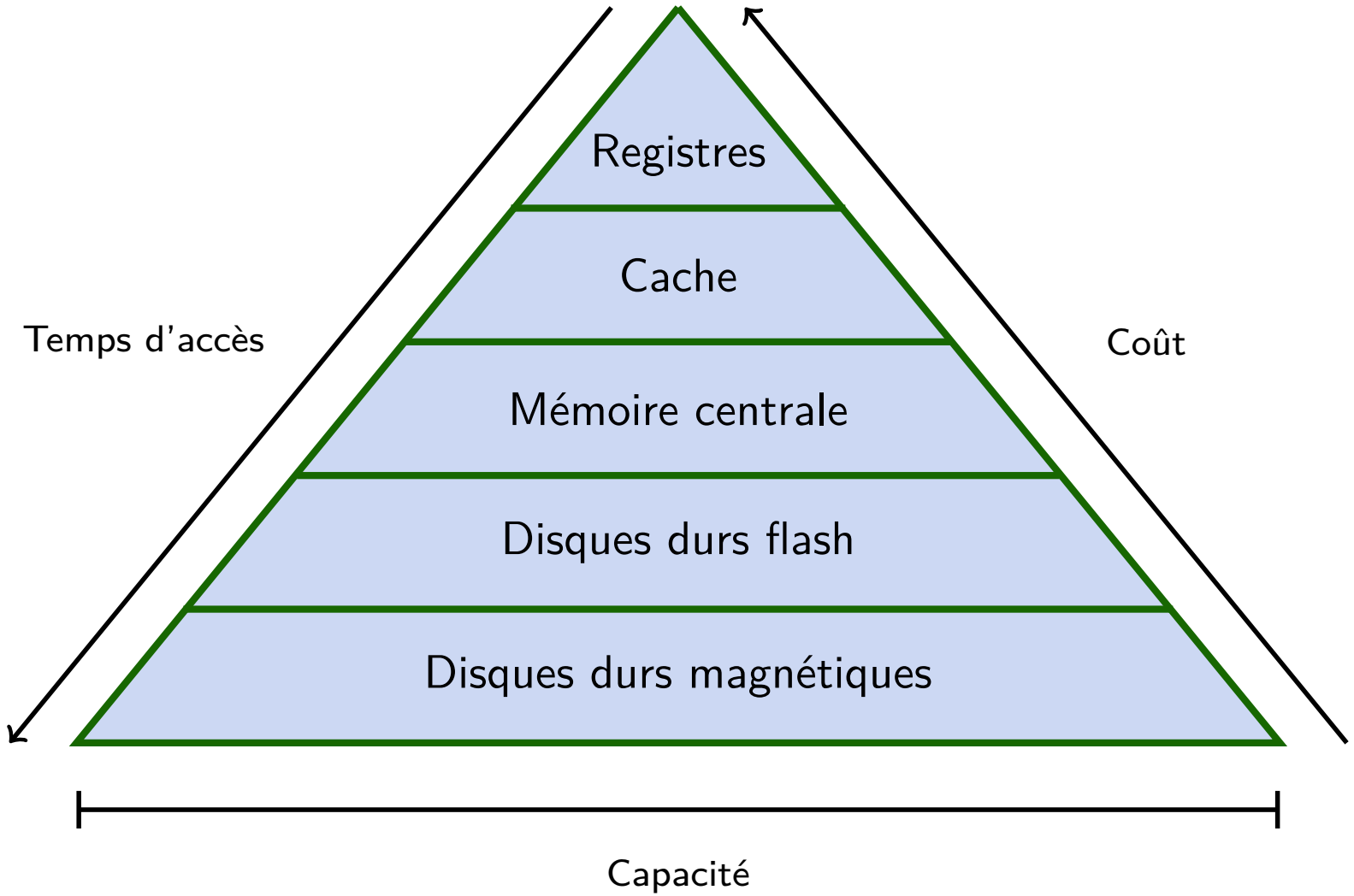
On obtient :



Ce faisant, trois cycles d'horloge ont été perdus.

Plan

Les trois dimensions de la mémoire



Mémoires

Les caractéristiques principales des mémoires — en plus des dimensions capacité, temps d'accès et coût — sont

- la volatilité (présence obligatoire ou non de courant électrique pour conserver les données mémorisées) ;

Mémoires

Les caractéristiques principales des mémoires — en plus des dimensions capacité, temps d'accès et coût — sont

- la volatilité (présence obligatoire ou non de courant électrique pour conserver les données mémorisées) ;
- le nombre de réécritures possibles ;

Mémoires

Les caractéristiques principales des mémoires — en plus des dimensions capacité, temps d'accès et coût — sont

- la volatilité (présence obligatoire ou non de courant électrique pour conserver les données mémorisées) ;
- le nombre de réécritures possibles ;
- le débit de lecture ;

Mémoires

Les caractéristiques principales des mémoires — en plus des dimensions capacité, temps d'accès et coût — sont

- la volatilité (présence obligatoire ou non de courant électrique pour conserver les données mémorisées) ;
- le nombre de réécritures possibles ;
- le débit de lecture ;
- le débit d'écriture.

Mémoires

Voici les caractéristiques de quelques mémoires :

- registre : volatile, réécriture possible, temps d'accès de l'ordre de 1 ns ;

Mémoires

Voici les caractéristiques de quelques mémoires :

- registre : volatile, réécriture possible, temps d'accès de l'ordre de 1 ns ;
- mémoire morte (ROM, **R**ead **O**nly **M**emory) : non volatile, pas de réécriture possible, temps d'accès de l'ordre de 150 ns ;

Mémoires

Voici les caractéristiques de quelques mémoires :

- registre : volatile, réécriture possible, temps d'accès de l'ordre de 1 ns ;
- mémoire morte (ROM, **R**ead **O**nly **M**emory) : non volatile, pas de réécriture possible, temps d'accès de l'ordre de 150 ns ;
- mémoire vive (RAM, **R**andom **A**ccess **M**emory) : volatile, réécriture possible, débit de lecture/écriture de l'ordre de 8 Gio/s, temps d'accès de l'ordre de de 10 ns ;

Mémoires

Voici les caractéristiques de quelques mémoires :

- registre : volatile, réécriture possible, temps d'accès de l'ordre de 1 ns ;
- mémoire morte (ROM, **R**ead **O**nly **M**emory) : non volatile, pas de réécriture possible, temps d'accès de l'ordre de 150 ns ;
- mémoire vive (RAM, **R**andom **A**ccess **M**emory) : volatile, réécriture possible, débit de lecture/écriture de l'ordre de 8 Gio/s, temps d'accès de l'ordre de de 10 ns ;
- mémoire flash : non volatile, réécriture possible (de l'ordre de 10^5 fois), débit de lecture/écriture de l'ordre de 500 Mio/s, temps d'accès de l'ordre de 0.1 ms ;

Mémoires

Voici les caractéristiques de quelques mémoires :

- registre : volatile, réécriture possible, temps d'accès de l'ordre de 1 ns ;
- mémoire morte (ROM, **R**ead **O**nly **M**emory) : non volatile, pas de réécriture possible, temps d'accès de l'ordre de 150 ns ;
- mémoire vive (RAM, **R**andom **A**ccess **M**emory) : volatile, réécriture possible, débit de lecture/écriture de l'ordre de 8 Gio/s, temps d'accès de l'ordre de de 10 ns ;
- mémoire flash : non volatile, réécriture possible (de l'ordre de 10^5 fois), débit de lecture/écriture de l'ordre de 500 Mio/s, temps d'accès de l'ordre de 0.1 ms ;
- mémoire de masse magnétique : non volatile, réécriture possible, débit de lecture/écriture de l'ordre de 100 Mio/s, temps d'accès de l'ordre de 10 ms.

Principes de localité

Problématique : comment optimiser les accès mémoire ?

Principes de localité

Problématique : comment optimiser les accès mémoire ?

On se base sur les deux principes raisonnables suivants.

Principes de localité

Problématique : comment optimiser les accès mémoire ?

On se base sur les deux principes raisonnables suivants.

Localité temporelle : si une zone de la mémoire a été considérée à un instant t donné, elle a une forte chance d'être reconsidérée à un instant t' proche de t.

Principes de localité

Problématique : comment optimiser les accès mémoire ?

On se base sur les deux principes raisonnables suivants.

Localité temporelle : si une zone de la mémoire a été considérée à un instant t donné, elle a une forte chance d'être reconsidérée à un instant t' proche de t.

La localité temporelle s'observe par exemple dans les **boucles** : la variable de contrôle de la boucle est régulièrement lue/modifiée.

Principes de localité

Problématique : comment optimiser les accès mémoire ?

On se base sur les deux principes raisonnables suivants.

Localité temporelle : si une zone de la mémoire a été considérée à un instant t donné, elle a une forte chance d'être reconsidérée à un instant t' proche de t.

La localité temporelle s'observe par exemple dans les **boucles** : la variable de contrôle de la boucle est régulièrement lue/modifiée.

Localité spatiale : si une zone de la mémoire à une adresse x donnée a été considérée, les zones de la mémoire d'adresses x' avec x' proches de x ont une forte chance d'être considérées.

Principes de localité

Problématique : comment optimiser les accès mémoire ?

On se base sur les deux principes raisonnables suivants.

Localité temporelle : si une zone de la mémoire a été considérée à un instant t donné, elle a une forte chance d'être reconsidérée à un instant t' proche de t.

La localité temporelle s'observe par exemple dans les **boucles** : la variable de contrôle de la boucle est régulièrement lue/modifiée.

Localité spatiale : si une zone de la mémoire à une adresse x donnée a été considérée, les zones de la mémoire d'adresses x' avec x' proches de x ont une forte chance d'être considérées.

La localité spatiale s'observe dans la manipulation de **tableaux** ou encore de la **pile** : les données sont organisées de manière contiguë en mémoire.

Principes de localité

Problématique : comment optimiser les accès mémoire ?

On se base sur les deux principes raisonnables suivants.

Localité temporelle : si une zone de la mémoire a été considérée à un instant t donné, elle a une forte chance d'être reconsidérée à un instant t' proche de t.

La localité temporelle s'observe par exemple dans les **boucles** : la variable de contrôle de la boucle est régulièrement lue/modifiée.

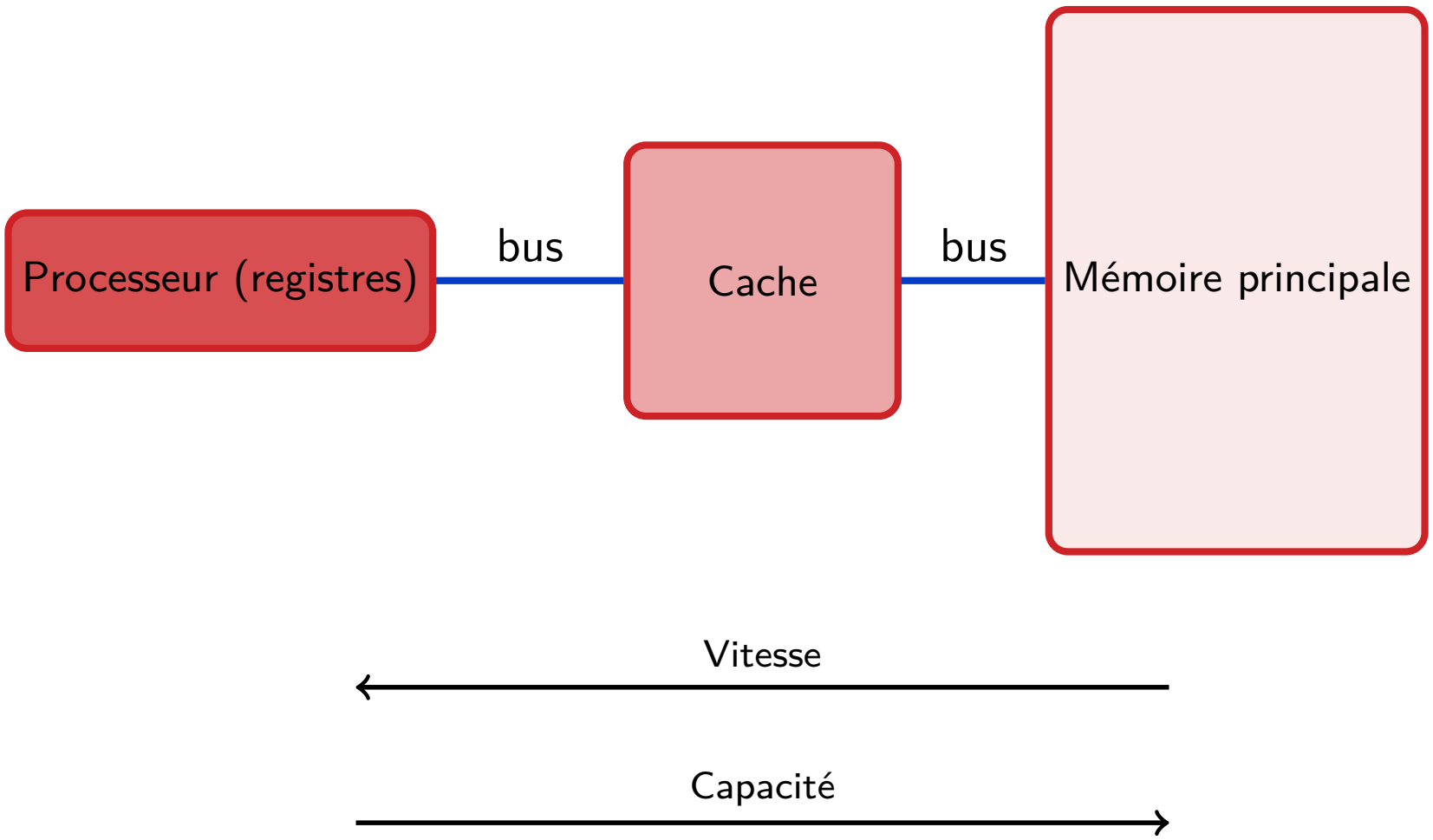
Localité spatiale : si une zone de la mémoire à une adresse x donnée a été considérée, les zones de la mémoire d'adresses x' avec x' proches de x ont une forte chance d'être considérées.

La localité spatiale s'observe dans la manipulation de **tableaux** ou encore de la **pile** : les données sont organisées de manière contiguë en mémoire.

Ces deux principes impliquent le fait qu'à un instant donné, un programme **n'accède qu'à une petite partie de son espace d'adressage**.

Organisation de la mémoire

La mémoire est organisée comme suit :



La mémoire cache dans l'organisation de la mémoire

La mémoire cache est une mémoire très rapide en lecture et en écriture (entre les registres et la mémoire centrale).

La mémoire cache dans l'organisation de la mémoire

La mémoire cache est une mémoire très rapide en lecture et en écriture (entre les registres et la mémoire centrale).

Elle est constituée de plusieurs couches : L1, L2, L3 où L1 a la plus petite capacité (de l'ordre de 128 Kio) et la plus grande vitesse, et L3 a la plus grande capacité (de l'ordre de 10 Mio) et la plus petite vitesse.

La mémoire cache dans l'organisation de la mémoire

La mémoire cache est une mémoire très rapide en lecture et en écriture (entre les registres et la mémoire centrale).

Elle est constituée de plusieurs couches : L1, L2, L3 où L1 a la plus petite capacité (de l'ordre de 128 Kio) et la plus grande vitesse, et L3 a la plus grande capacité (de l'ordre de 10 Mio) et la plus petite vitesse.

Il fonctionne de la manière suivante :

1. le processeur demande à lire une donnée en mémoire ;

La mémoire cache dans l'organisation de la mémoire

La mémoire cache est une mémoire très rapide en lecture et en écriture (entre les registres et la mémoire centrale).

Elle est constituée de plusieurs couches : L1, L2, L3 où L1 a la plus petite capacité (de l'ordre de 128 Kio) et la plus grande vitesse, et L3 a la plus grande capacité (de l'ordre de 10 Mio) et la plus petite vitesse.

Il fonctionne de la manière suivante :

1. le processeur demande à lire une donnée en mémoire ;
2. la mémoire cache, couche par couche, est interrogée :

La mémoire cache dans l'organisation de la mémoire

La mémoire cache est une mémoire très rapide en lecture et en écriture (entre les registres et la mémoire centrale).

Elle est constituée de plusieurs couches : L1, L2, L3 où L1 a la plus petite capacité (de l'ordre de 128 Kio) et la plus grande vitesse, et L3 a la plus grande capacité (de l'ordre de 10 Mio) et la plus petite vitesse.

Il fonctionne de la manière suivante :

1. le processeur demande à lire une donnée en mémoire ;
2. la mémoire cache, couche par couche, est interrogée :
    1. si elle contient la donnée, elle la communique au processeur ;
    2. sinon, la mémoire principale est interrogée. La mémoire principale envoie la donnée vers la mémoire cache qui l'enregistre (pour optimiser une utilisation ultérieure) et la transmet au processeur.

Organisation du cache

La mémoire cache est organisée en lignes. Chaque ligne est en général constituée de 32 octets.

Organisation du cache

La mémoire cache est organisée en lignes. Chaque ligne est en général constituée de 32 octets.

- `V` est un **bit de validité** : il informe si la ligne est utilisée.

Organisation du cache

La mémoire cache est organisée en lignes. Chaque ligne est en général constituée de 32 octets.

- `V` est un **bit de validité** : il informe si la ligne est utilisée.
- `Indicateur` permet de connaître l'**adresse** en mémoire principale des données représentées par la ligne.

Organisation du cache

La mémoire cache est organisée en lignes. Chaque ligne est en général constituée de 32 octets.

- `V` est un **bit de validité** : il informe si la ligne est utilisée.
- `Indicateur` permet de connaître l'**adresse** en mémoire principale des données représentées par la ligne.
- `Mot_1`, `Mot_2`, `Mot_3` et `Mot_4` contiennent des **données**.

Organisation du cache

La mémoire cache est organisée en lignes. Chaque ligne est en général constituée de 32 octets.

- `V` est un **bit de validité** : il informe si la ligne est utilisée.
- `Indicateur` permet de connaître l'**adresse** en mémoire principale des données représentées par la ligne.
- `Mot_1`, `Mot_2`, `Mot_3` et `Mot_4` contiennent des **données**.

La ligne est la plus petite donnée qui peut circuler entre la mémoire cache et la mémoire principale.

Organisation du cache

La mémoire cache est organisée en lignes. Chaque ligne est en général constituée de 32 octets.

- `V` est un **bit de validité** : il informe si la ligne est utilisée.
- `Indicateur` permet de connaître l'**adresse** en mémoire principale des données représentées par la ligne.
- `Mot_1`, `Mot_2`, `Mot_3` et `Mot_4` contiennent des **données**.

La ligne est la plus petite donnée qui peut circuler entre la mémoire cache et la mémoire principale.

Le mot est la plus petite donnée qui peut circuler entre le processeur et la mémoire cache. Celui-ci est en général composé de 4 octets.

Stratégies de gestion de la mémoire cache

Invariant important : toute donnée représentée dans la mémoire cache est également présente dans la mémoire centrale. C'est la propriété d'inclusion.

Stratégies de gestion de la mémoire cache

Invariant important : toute donnée représentée dans la mémoire cache est également présente dans la mémoire centrale. C'est la propriété d'inclusion.

Ceci implique que lorsqu'une donnée qui figure dans la mémoire cache est modifiée, il faut également modifier l'instance de la donnée située en mémoire principale.

Stratégies de gestion de la mémoire cache

Invariant important : toute donnée représentée dans la mémoire cache est également présente dans la mémoire centrale. C'est la propriété d'inclusion.

Ceci implique que lorsqu'une donnée qui figure dans la mémoire cache est modifiée, il faut également modifier l'instance de la donnée située en mémoire principale.

Il existe deux stratégies pour cela :

1. l'écriture simultanée : lorsqu'une ligne du cache est modifiée, la mémoire principale est immédiatement mise à jour. Cette méthode est lente.

Stratégies de gestion de la mémoire cache

Invariant important : toute donnée représentée dans la mémoire cache est également présente dans la mémoire centrale. C'est la propriété d'inclusion.

Ceci implique que lorsqu'une donnée qui figure dans la mémoire cache est modifiée, il faut également modifier l'instance de la donnée située en mémoire principale.

Il existe deux stratégies pour cela :

1. l'écriture simultanée : lorsqu'une ligne du cache est modifiée, la mémoire principale est immédiatement mise à jour. Cette méthode est lente.

2. La recopie : lorsqu'une ligne du cache est modifiée, on active un drapeau qui la signale comme telle et la mémoire principale n'est mise à jour que lorsque nécessaire (juste avant de modifier à nouveau la ligne du cache en question).

Stratégies d'écriture dans la mémoire cache

Il existe plusieurs stratégies d'écriture dans la mémoire cache, plus ou moins complexes et plus ou moins rapides :

Stratégies d'écriture dans la mémoire cache

Il existe plusieurs stratégies d'écriture dans la mémoire cache, plus ou moins complexes et plus ou moins rapides :

- l'organisation à correspondance directe : à toute donnée est associée une position dans la mémoire cache (par un calcul modulaire).

  Si une donnée doit être écrite dans la mémoire cache à un endroit déjà occupé, ce dernier est écrasé.

Stratégies d'écriture dans la mémoire cache

Il existe plusieurs stratégies d'écriture dans la mémoire cache, plus ou moins complexes et plus ou moins rapides :

- l'organisation à correspondance directe : à toute donnée est associée une position dans la mémoire cache (par un calcul modulaire).

  Si une donnée doit être écrite dans la mémoire cache à un endroit déjà occupé, ce dernier est écrasé.

- L'organisation totalement associative : une donnée peut se retrouver à une place quelconque dans la mémoire cache.

  Si une donnée doit être écrite dans la mémoire cache à un endroit déjà occupé, une position aléatoire est générée pour tenter de placer la nouvelle donnée.