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

– Exemple –

Observons l'effet des quatre instructions

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

avec la pile dans l'état

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

`esp` = `1008`

0x01010101
0x00000003
0xAA55AA55
0xFFFFFFFF
0x00000000

`esp` = `1004`

0x01010101
0x00000003
0xAA55AA55
0xFFFFFFFF
0x00000000

`esp` = `1008`
`eax` = `0x3`

0x01010101
0x00000003
0xAA55AA55
0xFFFFFFFF
0x00000000

`esp` = `1012`
`ebx` = `0xAA55AA55`

0x01010101
0x00000003
0x00000003
0xFFFFFFFF
0x00000000

`esp` = `1008`

Instruction `call`

On souhaite maintenant établir un mécanisme pour pouvoir **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:
...
```

L'adresse de retour `suite` est connue même dans le cas où cette adresse n'est pas positionnée par le programmeur.

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 dépile la donnée en tête de pile et saute 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
```

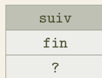
Exemple d'utilisation de `call` / `ret`

– Exemple –

Considérons la suite d'instructions suivante et observons l'état de la pile et du pointeur d'instruction au fil de l'exécution.

```
mov ecx, 8              ; Adresse a1.
call loin               ; Adresse a2.
suiv: add ecx, 24       ; Adresse a3.
loin: add ecx, 16       ; Adresse a4.
ret                     ; Adresse a5.
...                     ; Adresse a6.
fin:                    ; Adresse a7.
```

ecx = 8
eip = a2



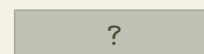
ecx = 8
eip = a4

ecx = 24
eip = a5

ecx = 24
eip = a3

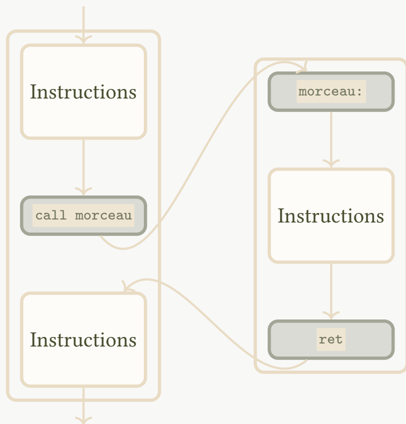
ecx = 48
eip = a4

ecx = 64
eip = a5



ecx = 64
eip = a6

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`.

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. 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 ;
4. les **valeurs** des registres `ebx`, `ecx` et `edx` doivent être dans le **même état** avant l'appel et après l'appel de la fonction.

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

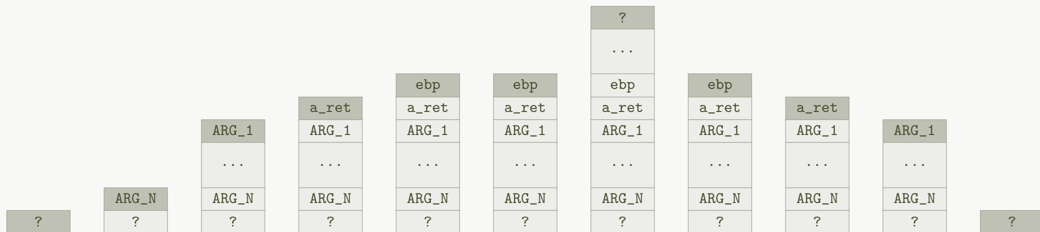
Analysons l'état de la pile, étape par étape, lors de l'appel d'une fonction.

```
; Appel de la fonction NOM_FCT.
push ARG_N
...
push ARG_1
call NOM_FCT
ADD esp, 4 * N    ; Adresse a_ret.
```

```
; Definition de la fonction NOM_FCT.
NOM_FCT:
    push ebp
    mov ebp, esp
    INSTR
    pop ebp
    ret
```

À partir du 6e dessin de pile, `ebp` pointe vers une zone fixée de la mémoire.

Il s'agit de la zone de la pile située à deux cases au dessus de là où figure le 1er argument.



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`.

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 doit 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

– Exemple –

```
; 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 procède par

```
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` à `esp` après l'appel pour dépiler d'un seul coup les deux arguments ($8 = 2 \times 4$).

Fonctions — exemple 2

– Exemple –

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

print_char:
    ; Debut.
    push ebp
    mov ebp, esp

    ; Sauv. des registres.
    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 registres.
    pop edx
    pop ecx
    pop ebx
    pop eax

    ; Fin.
    pop ebp
    ret
```

Fonctions — exemple 2 bis

– Exemple –

```
; 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 registres.
    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 registres.
    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**.

– Exemple –

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.

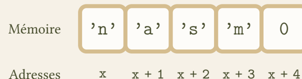
Chaînes de caractères

Une chaîne de caractères est une **suite contiguë d'octets** en mémoire se terminant par l'octet nul.

On accède à une chaîne de caractères via l'**adresse** `x` de son **1er caractère**. Par abus de langage, on parle de « la chaîne `x` ». Les adresses des autres caractères s'obtiennent par incrémentation.

– Exemple –

La chaîne `x` de valeur `"nasm"` est représentée en mémoire par



La valeur de `[x]` est `'n'`, celle de `[x + 1]` est `'a'`, . . ., celle de `[x + 4]` est `0`.

On **lit** le `i` e caractère d'une chaîne `x` en le rangeant dans `al` par `mov al, [x + i - 1]`.

On **écrase** le `i` e caractère d'une chaîne `x` en la valeur contenue dans `al` par `mov [x + i - 1], al`.

Toute fonction ayant un argument chaîne de caractère travaille à partir de l'adresse de cette chaîne. Il s'agit d'un **passage par adresse**.

Fonctions — exemple 3

– Exemple –

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

print_string:
    ; Debut.
    push ebp
    mov ebp, esp

    ; Sauv. des registres.
    push ebx

    ; Adresse du debut de la chaine.
    mov ebx, [ebp + 8]

    ; Init. compteur
    mov eax, 0

    ; Boucle d'affichage.
    boucle:
        cmp byte [ebx], 0
        je fin_boucle

        ; Appel de print_char.
        push dword [ebx]
        call print_char
        add esp, 4

        add ebx, 1
        add eax, 1
        jmp boucle
    fin_boucle:

    ; Rest. des registres.
    pop ebx

    ; Fin.
    pop ebp
    ret
```

Fonctions — exemple 4

– Exemple –

On souhaite écrire une fonction pour calculer **récursivement** le n^e nombre triangulaire triangle(n) défini par triangle$(0) := 0$ et pour tout $n \geqslant 1$, triangle$(n) := n +$ triangle$(n-1)$.

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

triangle:
    ; Debut.
    push ebp
    mov ebp, esp

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

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

```
    cmp ebx, 0
    je cas_terminal
    cas_non_terminal:

        ; Preparation appel recursif.
        mov ecx, ebx
        sub ecx, 1

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

        ; Calcul du resultat.
        add eax, ebx

        jmp fin
```

```
    cas_terminal:
        mov eax, 0

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

    ; Fin.
    pop ebp
    ret
```

Fonctions — exemple 5

– Exemple –

On souhaite écrire une fonction pour calculer **récursivement** le factorielle $\mathrm{fact}(n)$ définie par $\mathrm{fact}(0) := 1$ et pour tout $n \geqslant 1$, $\mathrm{fact}(n) := n \times \mathrm{fact}(n-1)$.

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

fact:
    ; Debut.
    push ebp
    mov ebp, esp

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

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

```
    cmp ebx, 0
    je cas_terminal
    cas_non_terminal:

        ; Preparation appel recursif.
        mov ecx, ebx
        sub ecx, 1

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

        ; Calcul du resultat.
        mul ebx

        jmp fin
```

```
    cas_terminal:
        mov eax, 1

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

    ; Fin.
    pop ebp
    ret
```

Fonctions — exemple 6

– Exemple –

On souhaite écrire une fonction pour calculer **récursivement** le n^{e} nombre de Fibonacci $\mathrm{fibo}(n)$ défini par $\mathrm{fibo}(0) := 0$, $\mathrm{fibo}(1) := 1$ et pour tout $n \geqslant 2$, $\mathrm{fibo}(n) := \mathrm{fibo}(n-1) + \mathrm{fibo}(n-2)$.

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

fibo:
    ; Debut
    push ebp
    mov ebp, esp

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

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

```
    cmp ebx, 1
    jle cas_terminaux

    cas_non_terminal :
        ; Prepa. appel 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. appel rec. 2.
        mov ecx, ebx
        sub ecx, 2

        ; Appel rec. 2.
        push ecx
        call fibo
        add esp, 4

        ; Calcul resultat.
        pop ebx
        add eax, ebx

        jmp fin
```

```
    cas_terminaux:
        mov eax, ebx

    fin:

    ; Rest. des registres.
    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'**étiquette locale**, dont la syntaxe de définition et de référence est

`.ETIQ:`

– Exemple –

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

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

Ceci 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 l'étiquette réellement souhaitée si besoin est.

Programmation modulaire — modules

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 — etiquettes visibles

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 — inclusion

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`.

– Exemple –

Voici 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
...
```