

MI01 – Automne 2011

IA32 – Sous-programmes et interfaçage avec le C

Stéphane Bonnet

Poste : 52 56

Courriel : stephane.bonnet@utc.fr

Plan du cours

- Les sous-programmes
- Passage de paramètres
- Structure d'un programme C
- La fonction C

Les sous-programmes

Sous-programmes

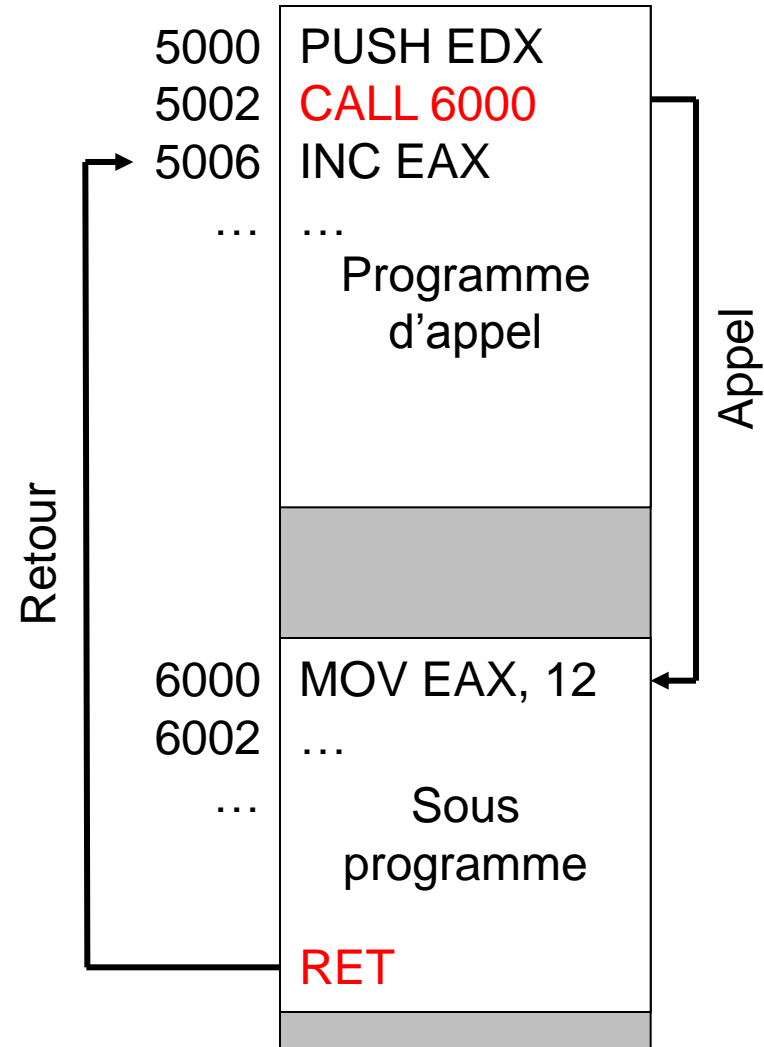
- Il s'agit de l'équivalent en assembleur des fonctions du langage C.
- Les compilateurs C transforment les fonctions en sous-programmes et utilisent les mécanismes du processeur pour y faire appel et revenir à l'instruction qui suit l'appel.

Sous-programmes

- L'utilisation de fonctions/sous-programmes permet de :
 - Structurer un programme
 - Éviter la répétition d'instructions identiques
 - Réduire le nombre d'instructions du programme
 - Réutiliser de séquences d'instructions sous la forme de bibliothèques de sous programmes

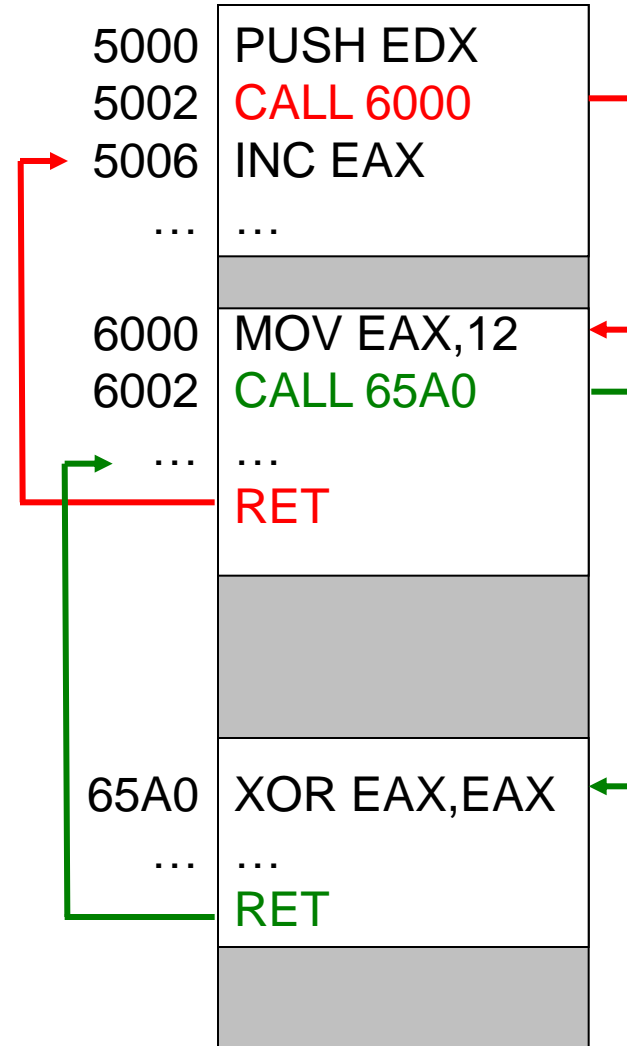
Sous-programmes

- Appelés par un programme d'appel
- A la fin du sous-programme, il faut revenir à l'instruction qui suit l'appel dans le programme d'appel.



Sous-programmes

- On peut imbriquer les appels : faire appel dans un sous-programme à un autre sous-programme.



Sous-programmes

Un sous programme peut être :

- **Réentrant :**

- Il peut être interrompu et appelé par le programme qui l'a interrompu

- **Récurif :**

- Il s'appelle lui-même. Il doit alors aussi être réentrant.

Sous-programmes

- Pour être utile, un sous-programme doit pouvoir travailler sur des informations variables en fonction du contexte
- La nature et le nombre de ces informations doivent être bien définis
- Ce sont les **arguments d'entrée et de sortie** du sous-programme.

Appels de sous-programmes

Appel de sous-programmes

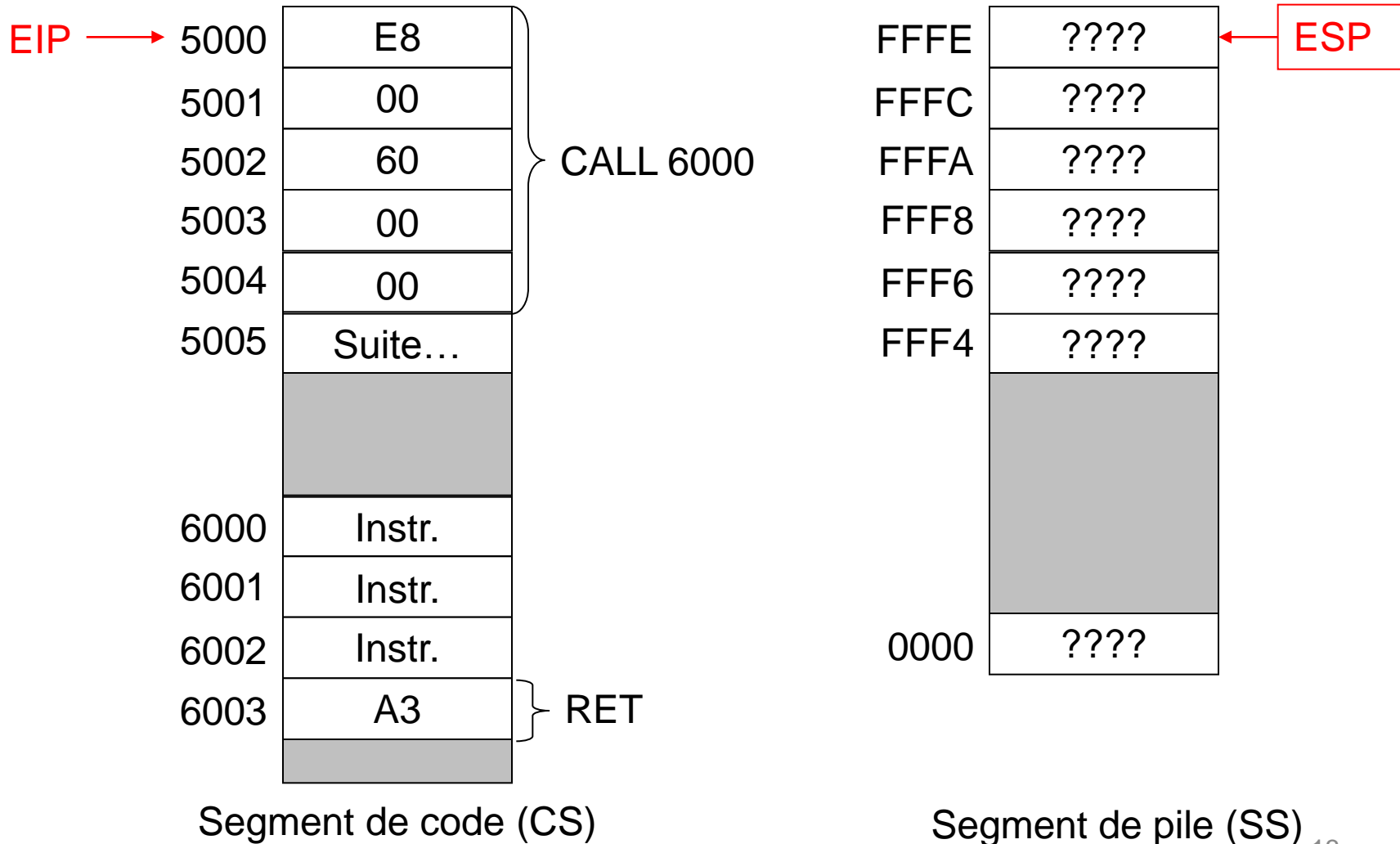
- L'appel à un sous-programme se fait par l'instruction **CALL** *<adresse>*
- Pour pouvoir revenir au programme d'appel, il faut conserver l'adresse de l'instruction qui suit le CALL. C'est **l'adresse de retour**.
- Quand le processeur a lu et décodé l'instruction CALL, le pointeur de programme (EIP) a été auto-incrémenté. Il pointe sur l'instruction suivant le CALL. L'adresse de retour est donc contenue dans EIP au moment de l'exécution du CALL.

Appel de sous-programmes

- Lors de l'exécution du CALL, le contenu de EIP est remplacé par l'adresse de la première instruction du sous-programme.
- Il faut sauvegarder cette adresse de retour. Le processeur, avant d'exécuter le CALL, va donc sauvegarder le contenu de EIP dans la pile.

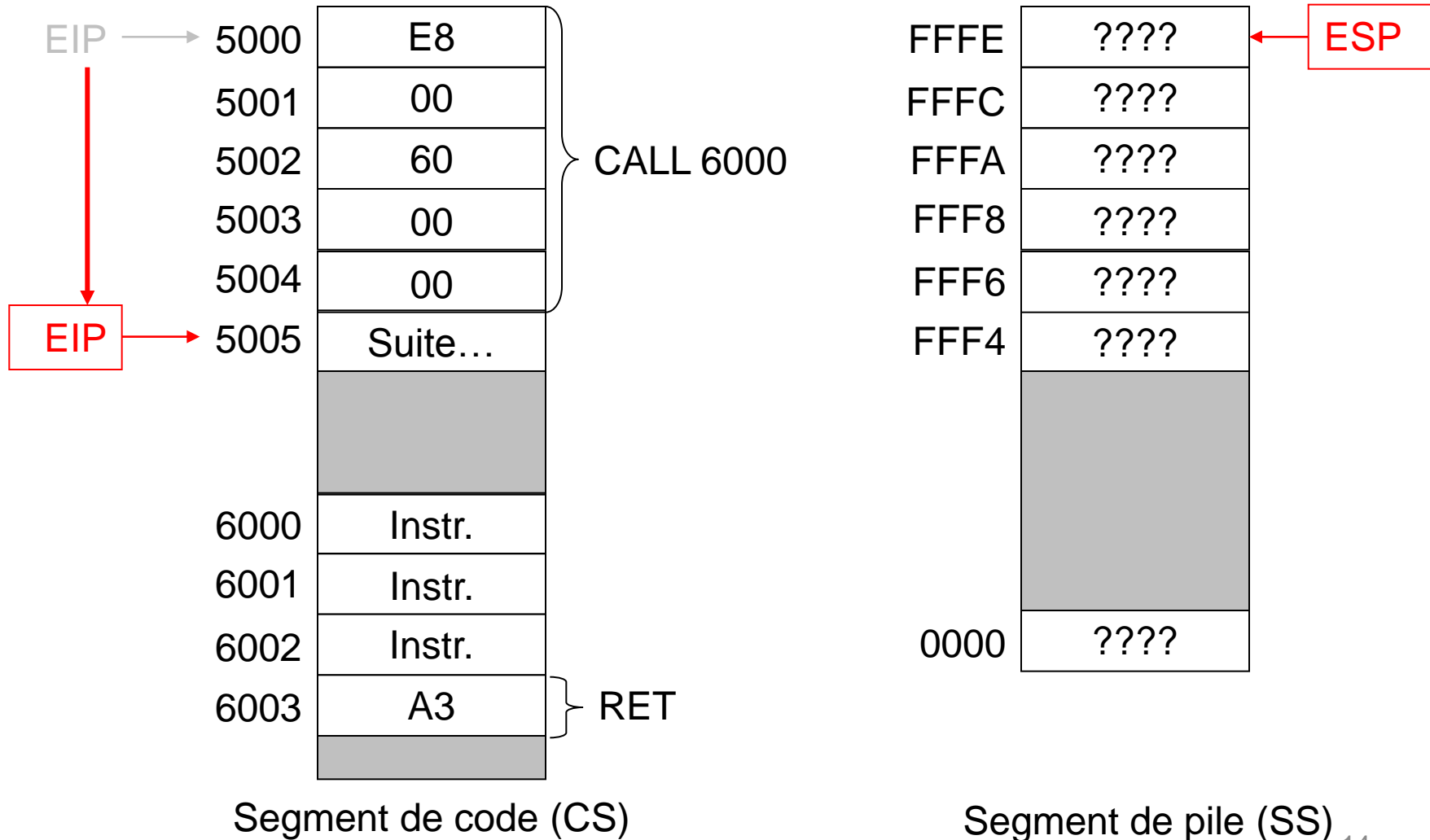
Appel de sous-programmes

- Avant recherche et décodage du CALL



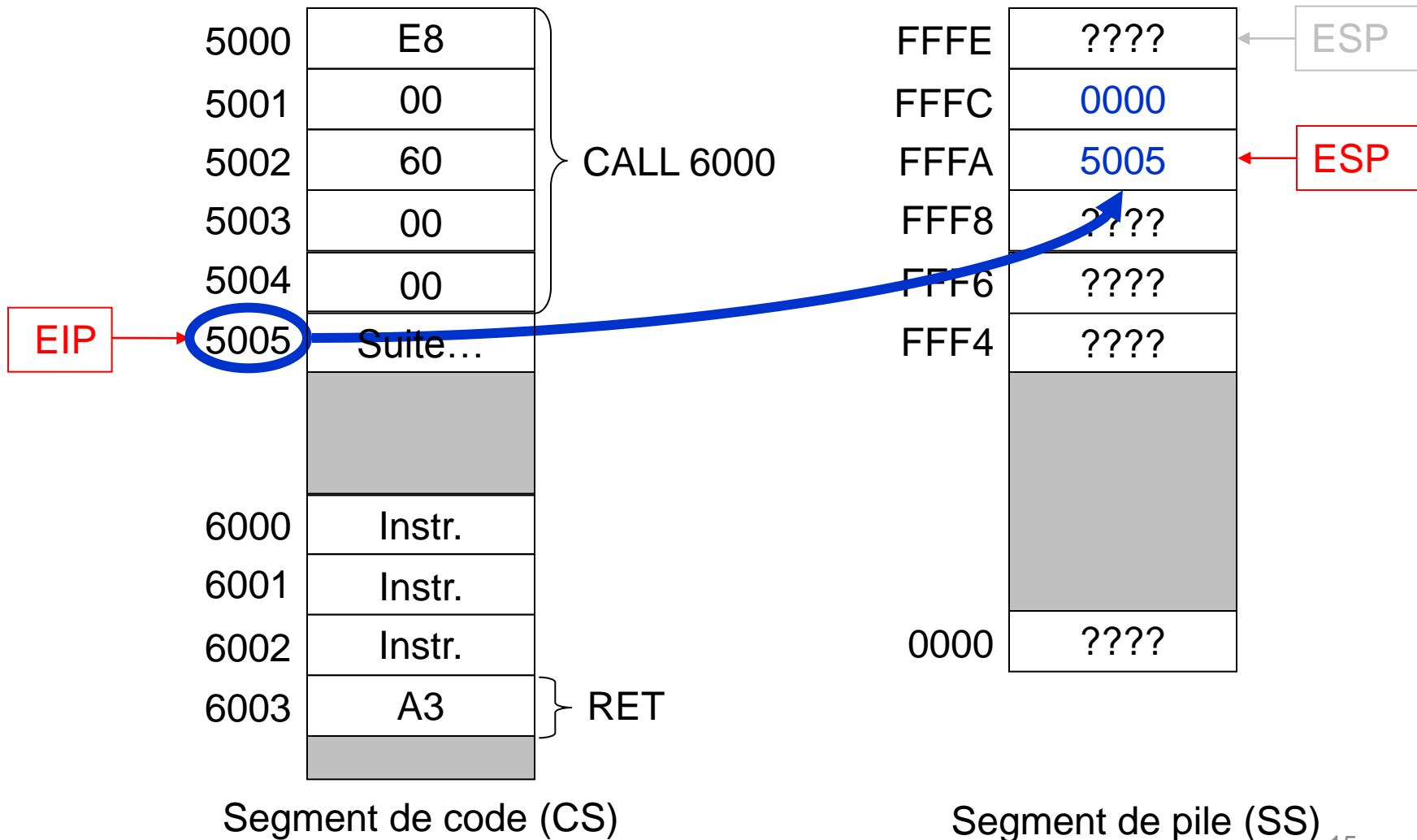
Appel de sous-programmes

- Après recherche (*fetch*) du CALL



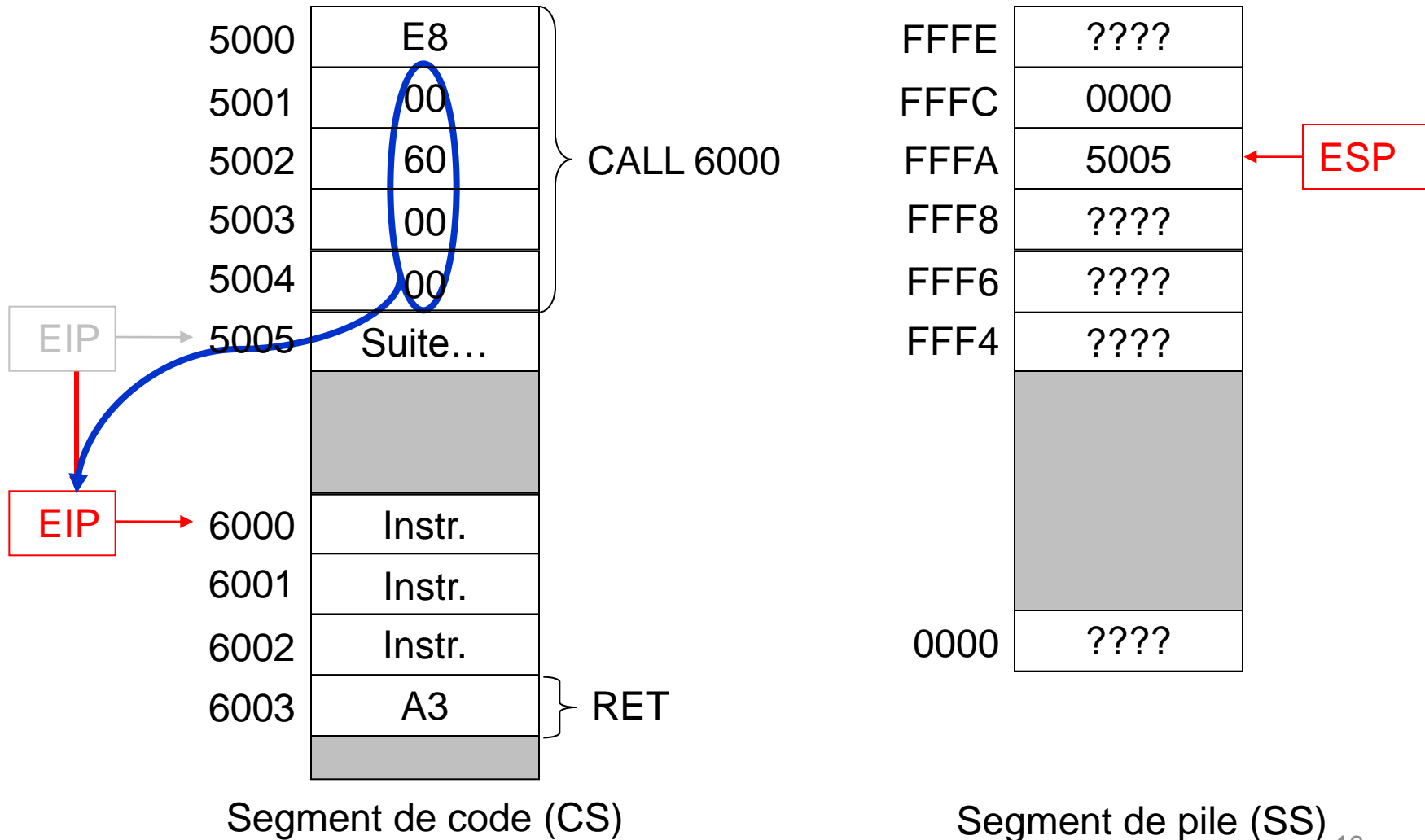
Appel de sous-programmes

- Sauvegarde de l'adresse de retour



Appel de sous-programmes

- Exécution du CALL

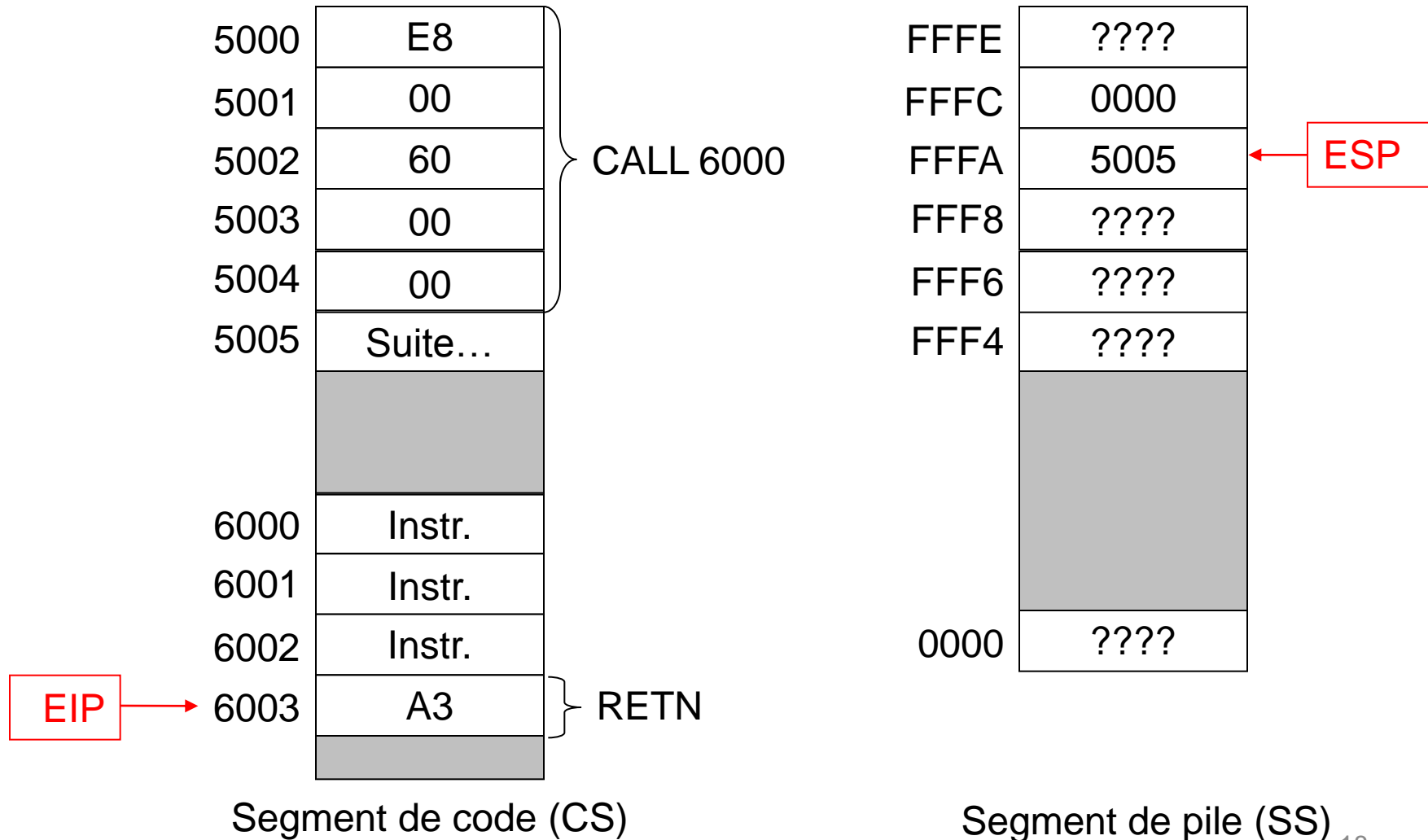


Retour de sous-programmes

- Le retour de sous-programme se fait au moyen de l'instruction RET.
- Lorsqu'il lit et décode l'instruction RET, le processeur s'attend à ce que le pointeur de pile ESP pointe sur l'adresse de retour.
- L'exécution du RET consiste donc à dépiler l'adresse de retour et à la transférer dans le pointeur d'instruction (EIP).
- **RET effectue l'opération inverse de CALL.**

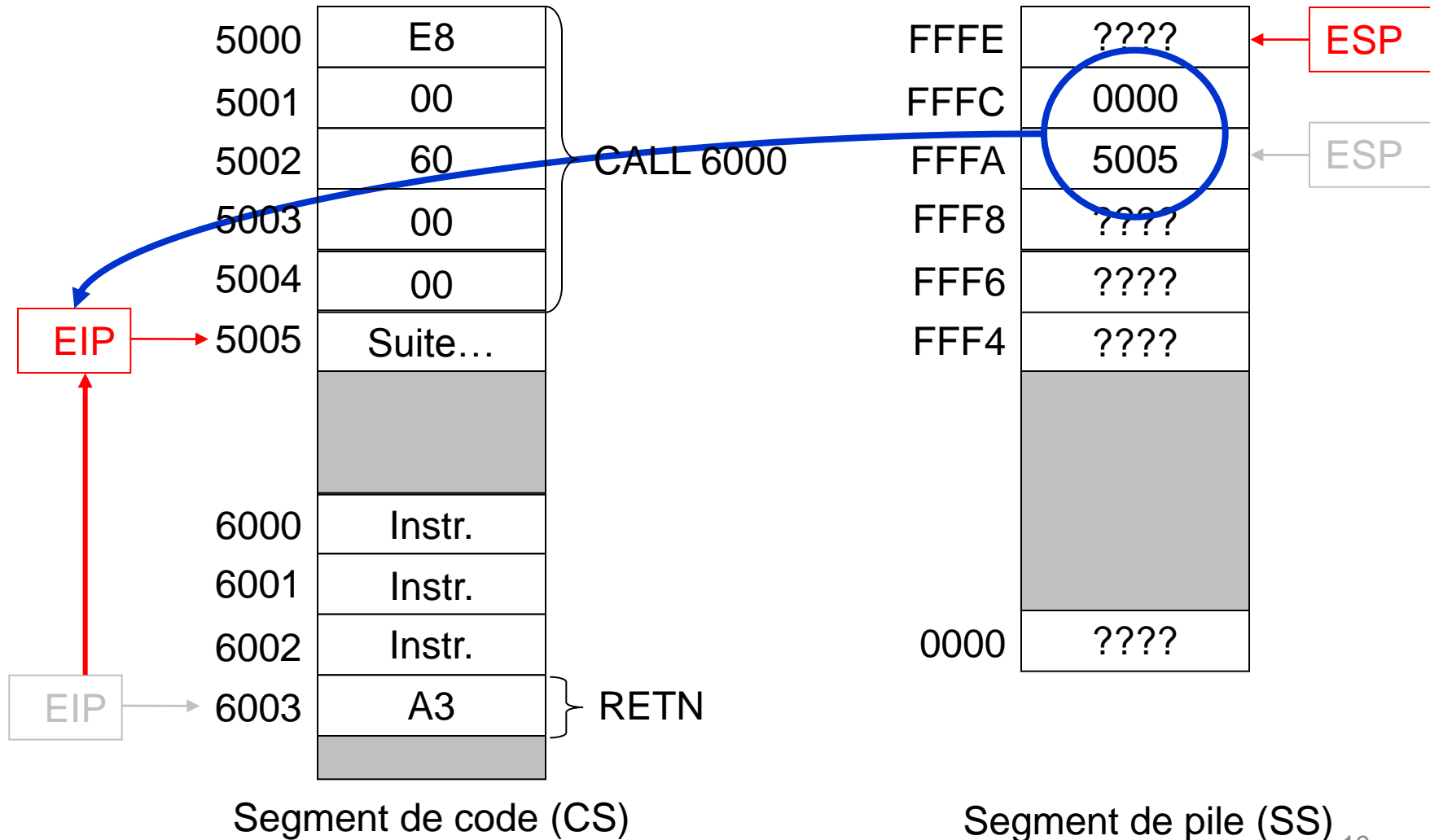
Retour de sous-programmes

- Avant le RET



Retour de sous-programmes

- Après le RET



Retour de sous-programmes

- On a alors la situation suivante :
 - Le pointeur de pile contient la même valeur qu'avant l'exécution du CALL
 - Le pointeur d'instruction pointe sur l'instruction qui suit le CALL.
- Avant l'exécution du RET, il faut s'assurer que le pointeur de pile ESP pointe bien sur l'adresse de retour, quelque soient les opérations qu'on aurait pu faire sur la pile dans le sous-programme.

Remarques

- La pile n'a pas une taille infinie : on ne peut donc pas imbriquer infiniment les sous-programmes, puisque l'adresse de retour est stockée à chaque fois.
- En particulier, lors de l'utilisation de sous-programme récur­sifs, il faut prendre garde à ne pas provoquer de débordement de pile (***stack overflow***).

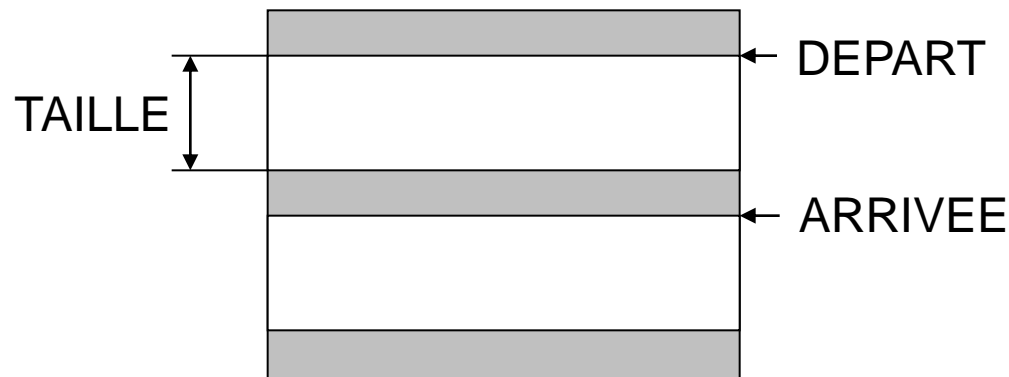
Passage d'arguments

Passage d'arguments

- Plusieurs solutions pour échanger des données entre le programme d'appel et le sous-programme :
 - Par zones de mémoire partagées
 - Par des registres
 - Par la pile

Passage d'arguments

- Exemple
 - Nom : sous-programme TRANSFERT
 - Fonction : transférer le contenu d'une zone mémoire (DEPART) vers une autre zone mémoire (ARRIVEE). Les deux zones sont dans le même segment DS.
 - Arguments d'entrée :
 - Taille de la zone à transférer (32 bits)
 - Adresse 32 bits de la zone DEPART
 - Adresse 32 bits de la zone ARRIVEE



Passage d'arguments

Algorithme (C)

```
/* On copie les blocs en
partant de la fin pour pouvoir
utiliser une comparaison avec 0
*/

char *d = depart;
char *a = arrivee;
int i = taille;

while(i-- >= 0) {
    a[i] = d[i];
}
```

Algorithme équivalent

```
/* On copie les blocs en
partant de la fin pour pouvoir
utiliser une comparaison avec 0
*/

char *d = depart;
char *a = arrivee;
int i = taille;

boucle: i = i - 1;
if (i < 0) goto fini;
a[i] = d[i];
goto boucle;

fini:
```

Passage d'arguments par la mémoire

- L'utilisateur sait que les arguments sont placés en mémoire avant l'appel et connaît les adresses correspondantes.

TAILLE		6000
DEPART		6004
ARRIVEE		6008

Passage d'arguments par la mémoire

- Sous-programme :

```
TRANSFERT    PROC
              MOV     ESI, [DEPART]      ; s
              MOV     EDI, [ARRIVEE]     ; d
              MOVZX   ECX, [TAILLE]      ; compteur i
boucle:      DEC     ECX
              JL      fini
              MOV     AL, [ESI + ECX]
              MOV     [EDI + ECX], AL
              JMP     boucle
fini:
TRANSFERT    RET
TRANSFERT    ENDP
```

- Programme principal

- Taille du bloc : 16 octets (0Fh)
- Adresse de départ : 4000h
- Adresse d'arrivée : 5000h

```
DEBUT:      MOV     [DEPART], 4000h      ; Adresse initiale
              MOV     [ARRIVEE], 5000h   ; Adresse finale
              MOV     [TAILLE], 0Fh      ; Taille du bloc
              CALL    TRANSFERT
END         DEBUT
```

Passage d'arguments par la mémoire

- Inconvénients :
 - Rend le sous-programme **non réentrant** : s'il est interrompu et relancé par un autre programme d'appel, le contenu de TAILLE, DEPART et ARRIVEE sera changé, ce qui rendra ces arguments incohérents lors du retour à son exécution.
 - On impose au sous-programme l'utilisation de zones de mémoire fixes pour ses arguments, ce qui manque de souplesse pour l'écriture du programme d'appel.

Ce mécanisme revient à utiliser des variables globales

Passage d'arguments par registres

- Dans ce cas on utilise des registres pour le passage des arguments.
- Dans l'exemple, on suppose que avant l'appel :
 - ESI contient l'adresse de départ
 - EDI contient l'adresse d'arrivée
 - ECX contient la taille du bloc.

Passage d'arguments par registres

- Sous-programme :

```
TRANSFERT    PROC
boucle:      DEC      ECX
              JL       fini
              MOV      AL, [ESI + ECX]
              MOV      [EDI + ECX], AL
              JMP      boucle

fini:
              RET
TRANSFERT    ENDP
```

- Programme principal

- Taille du bloc : 16 octets (0Fh)
- Adresse de départ : 4000h
- Adresse d'arrivée : 5000h

```
DEBUT:       MOV      ESI, 4000h      ; Adresse initiale
              MOV      EDI, 5000h     ; Adresse finale
              MOV      ECX, 0Fh       ; Taille du bloc
              CALL     TRANSFERT
              END      DEBUT
```

Passage d'arguments par registres

- Avantages :
 - Pas de transfert mémoire pour préparer les arguments → appels plus rapides
 - Pas de transfert mémoire pour récupérer les arguments dans le sous-programme → moins d'instructions et sous-programme plus rapide.
- Inconvénient :
 - Le nombre d'arguments est limité au nombre de registres généraux du processeur.

Passage d'arguments par la pile

- On positionne sur la pile les arguments dans un ordre déterminé.
- Le sous-programme y accède en les lisant directement dans la pile (sans les dépiler)
- C'est le mécanisme utilisé en général par les compilateurs de haut niveau (langage C, C++, Pascal...)
- Dans l'exemple, on suppose qu'on empile dans l'ordre : DEPART, ARRIVEE et TAILLE.

Passage d'arguments par la pile

- Début du programme principal

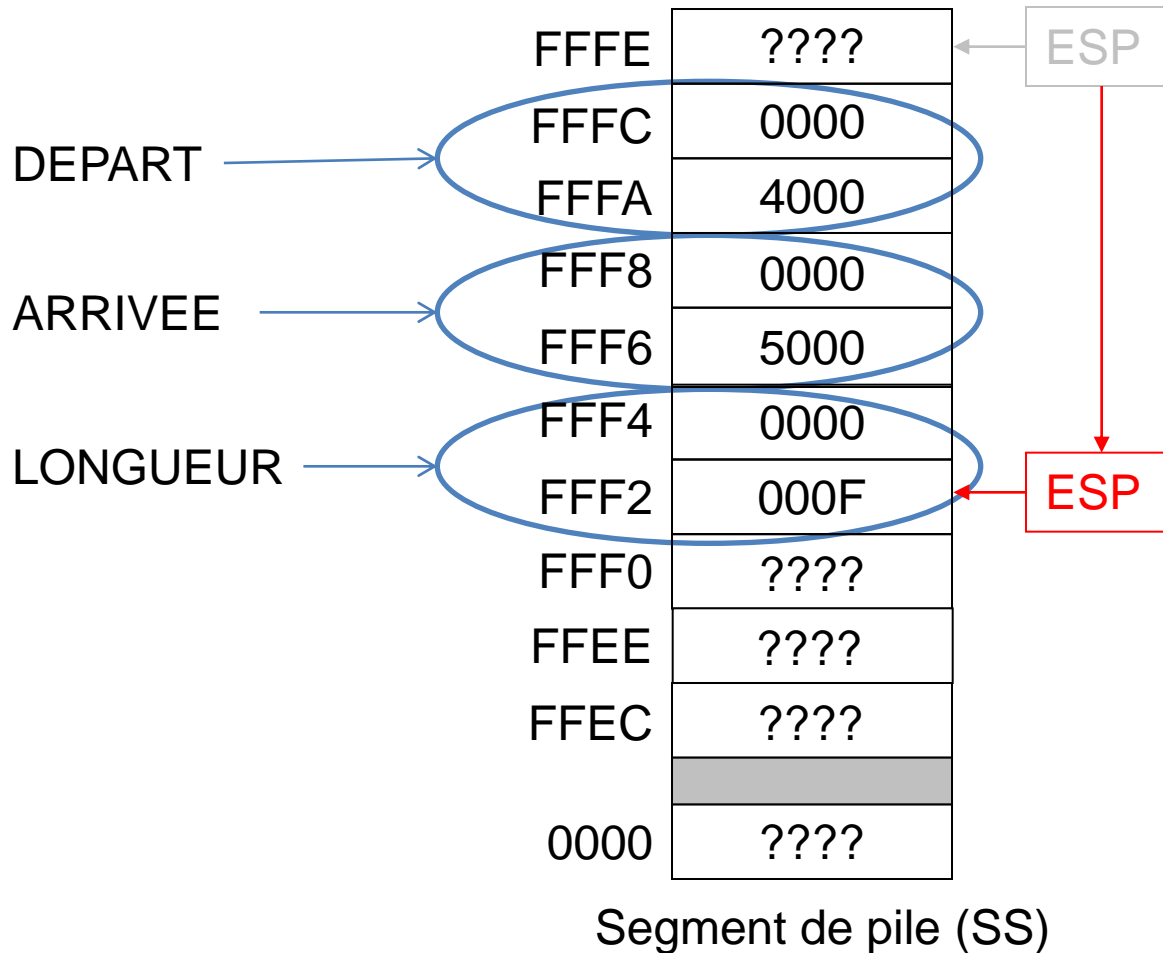
- Taille du bloc : 16 octets (0Fh)
- Adresse de départ : 4000h
- Adresse d'arrivée : 5000h

```
DEBUT:      PUSH      4000h          ; Adresse initiale
            PUSH      5000h          ; Adresse finale
            PUSH      DWORD 0Fh      ; Taille du bloc
            CALL      TRANSFERT
            ...
```

- Les différents arguments sont empilés, puis le programme appelle le sous-programme transfert.

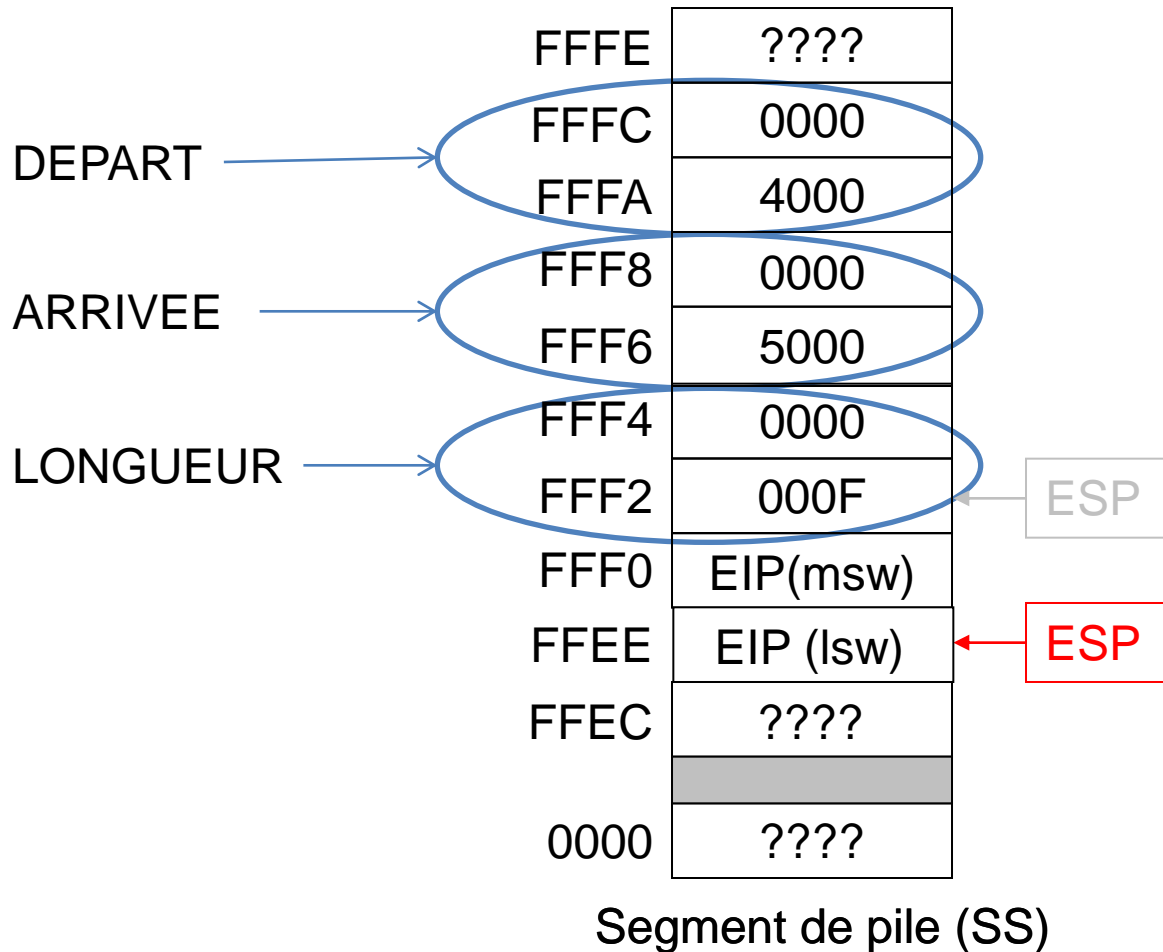
Passage d'arguments par la pile

- Structure de la pile avant le CALL



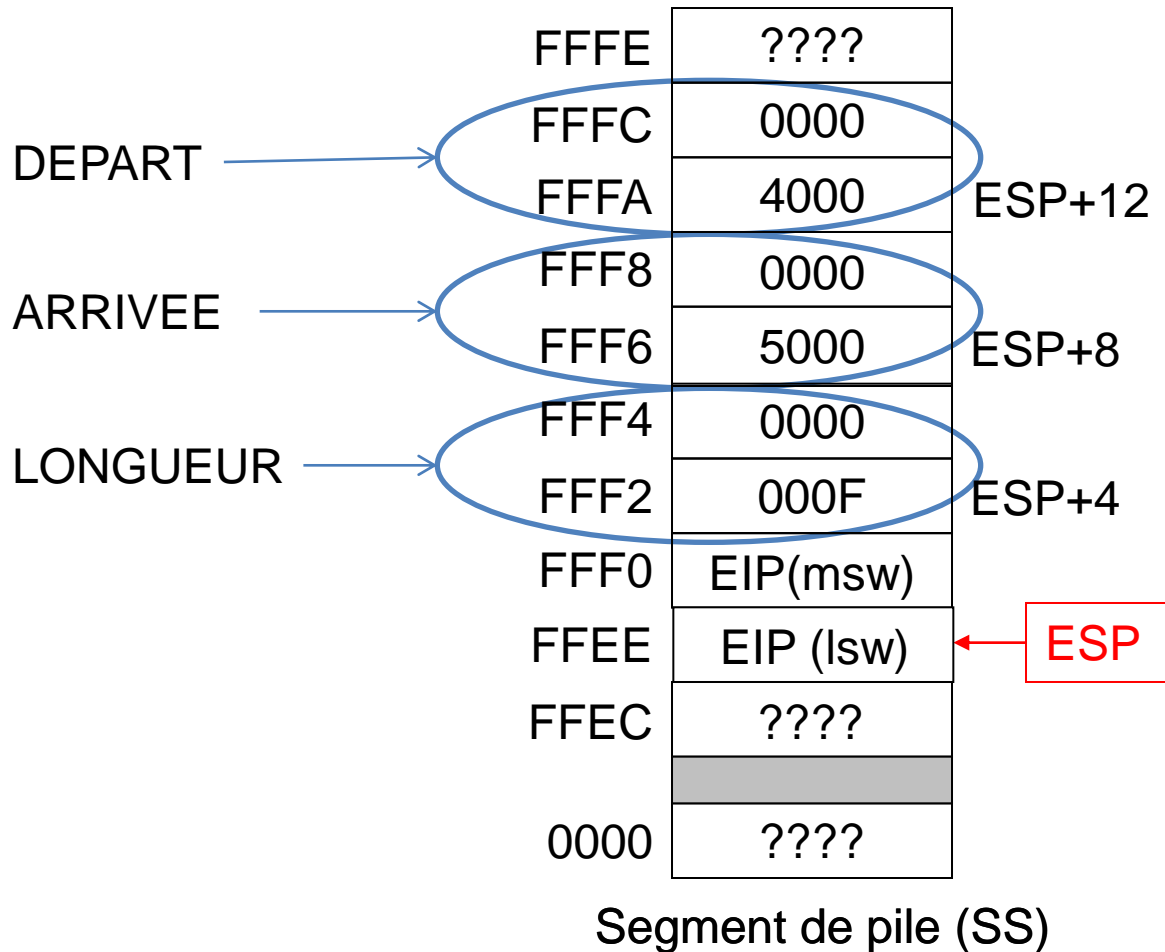
Passage d'arguments par la pile

- Structure de la pile après le CALL



Passage d'arguments par la pile

- Ou chercher les arguments ?



Passage d'arguments par la pile

- Le sous programme lit ses arguments en utilisant un déplacement par rapport au pointeur de pile. Ainsi, pour copier la taille du bloc dans ECX, on écrira :

MOV ECX, [ESP + 4] .

- MAIS : ESP peut varier pendant l'exécution du sous-programme!
 - si on empile des données, le déplacement par rapport à ESP des arguments change.
 - Par exemple, on suppose que lors de l'exécution du sous-programme, on rencontre l'instruction `PUSH AX`. (AX contient FFAAh)

Passage d'arguments par la pile

- Ou chercher les arguments ?

FFFE	????	
FFFC	0000	
FFFA	4000	ESP+14
FFF8	0000	
FFF6	5000	ESP+10
FFF4	0000	
FFF2	000F	ESP+6
FFF0	EIP(msw)	
FFEE	EIP (lsw)	
FFEC	FFAA	← ESP
0000	????	

Segment de pile (SS)

Passage d'arguments par la pile

- Après le PUSH AX, l'adresse de l'argument TAILLE devient ESP+6, alors qu'elle valait ESP+4 avant.
- Il est difficile de tenir compte de l'évolution du pointeur de pile lors de l'accès aux arguments. Afin de résoudre ce problème, on construit un **cadre de pile** (*stack frame*) qui permet d'avoir une référence fixe sur la pile.

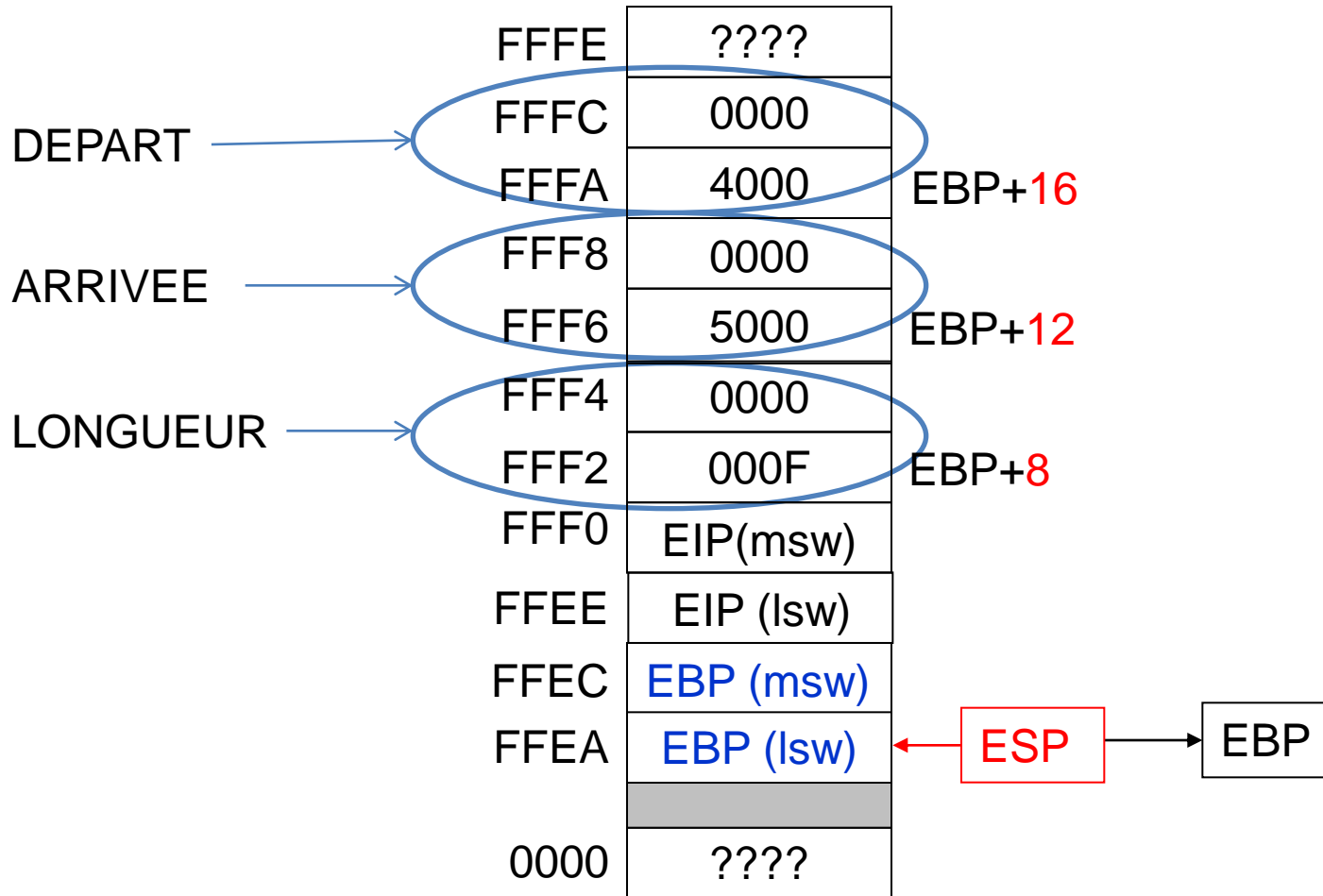
Cadre de pile

- Stocker dans un registre la valeur du pointeur de pile ESP après l'appel au sous-programme.
- On utilise en général EBP : le segment par défaut associé dans l'adressage indirect est SS (segment de pile), et non DS.
- A l'entrée du sous-programme, on ajoute :

```
PUSH    EBP                ; Sauvegarder l'ancien  
                        ; contenu de EBP  
MOV     EBP, ESP
```


Cadre de pile

- Construction du cadre de pile



Segment de pile (SS) – mots de 16 bits

Cadre de pile

- L'adresse de TAILLE est maintenant EBP+8. Elle reste invariante quelque soient les opérations subséquentes sur la pile (PUSH, POP...).
- On a donc à calculer les déplacements **qu'une fois pour toutes**

Passage d'arguments par la pile

- Sous-programme :

```
TRANSFERT    PROC    NEAR
              PUSH    EBP          ; Construction du cadre de pile
              MOV     EBP, ESP
              MOV     ESI, [EBP + 16] ; Adressage indirect avec déplacement
              MOV     EDI, [EBP + 12] ; Adresse finale
              MOV     ECX, [EBP + 8]  ; Taille du bloc à transférer
boucle:      DEC     ECX
              JL      fini
              MOV     AL, [ESI + ECX]
              MOV     [EDI + ECX], AL
              JMP     boucle
fini:        RET
TRANSFERT    ENDP
```

- Programme principal

- Taille du bloc : 16 octets (0Fh)
- Adresse de départ : 4000h
- Adresse d'arrivée : 5000h

```
DEBUT:      PUSH    4000h          ; Adresse initiale
              PUSH    5000h          ; Adresse finale
              PUSH    000Fh          ; Taille du bloc
              CALL    TRANSFERT
              ADD     ESP, 12         ; Enlever les arguments de la pile
              END     DEBUT
```

Suppression des arguments

- Lors du retour au programme d'appel, **il faut retirer les arguments devenus inutiles de la pile.**
- Deux solutions :
 - soit on demande au sous-programme de le faire,
 - soit on laisse le programme d'appel le faire.
- L'utilisation de la technique appropriée est essentielle si on veut inclure des sous-programmes assembleur dans un programme écrit en langage de haut niveau.

Suppression des arguments

- Suppression des arguments par le sous-programme
 - On utilise la forme `RET n` de l'instruction `RET`, ou *n* est le nombre **d'octets** à supprimer de la pile **après** dépileage de l'adresse de retour et retour au programme d'appel.
 - Exemple : sous-programme TRANSFERT

TRANSFERT	PROC	
	PUSH	EBP
	...	
	POP	EBP
	RET	12
TRANSFERT	ENDP	

Suppression des arguments

FFFE	????	
FFFC	0000	
FFFA	4000	
FFF8	0000	
FFF6	5000	
FFF4	0000	
FFF2	000F	
FFF0	EIP(msw)	
FFEE	EIP (lsw)	← ESP
FFEC	EBP (msw)	
FFEA	EBP (lsw)	
0000	????	

Segment de pile (SS)
avant RET 12

FFFE	????	← ESP
FFFC	0000	
FFFA	4000	
FFF8	0000	
FFF6	5000	
FFF4	0000	
FFF2	000F	
FFF0	EIP(msw)	
FFEE	EIP (lsw)	
FFEC	EBP (msw)	
FFEA	EBP (lsw)	
0000	????	

Segment de pile (SS)
après RET 12

Suppression des arguments

- Suppression des arguments par le programme d'appel
 - Dans ce cas, **après** le retour du sous-programme, il suffit d'incrémenter le pointeur de pile du nombre **d'octets** à supprimer.
 - **C'est la convention utilisé par les compilateurs C.**
 - Dans l'exemple :

```
DEBUT:      PUSH    4000h      ; Adresse initiale
            PUSH    5000h      ; Adresse finale
            PUSH    000Fh      ; Taille du bloc
            CALL    TRANSFERT
            ADD      ESP, 12      ; Enlever les arguments
                                   ; de la pile
            END      DEBUT
```

Suppression des arguments

FFFE	????	
FFFC	0000	
FFFA	4000	
FFF8	0000	
FFF6	5000	
FFF4	0000	
FFF2	000F	← ESP
FFF0	EIP(msw)	
FFEE	EIP (lsw)	
FFEC	EBP (msw)	
FFEA	EBP (lsw)	
0000	????	

Segment de pile (SS)
avant ADD ESP,12

FFFE	????	← ESP
FFFC	0000	
FFFA	4000	
FFF8	0000	
FFF6	5000	
FFF4	0000	
FFF2	000F	
FFF0	EIP(msw)	
FFEE	EIP (lsw)	
FFEC	EBP (msw)	
FFEA	EBP (lsw)	
0000	????	

Segment de pile (SS)
après ADD ESP,12

Attention aux registres !

- PROBLEME : Le programme d'appel peut utiliser des registres qui sont aussi utilisés par un sous-programme.
- CONSEQUENCE : Un sous-programme doit **sauvegarder l'ensemble des registres qu'il utilise**, afin de rendre son appel transparent pour le programme d'appel
- MAIS : Ceci ne s'applique pas pour les registres utilisés comme arguments du sous-programme.
- MAIS : On peut aussi s'en passer. Dans ce cas il faut **documenter exactement** les registres utilisés dans le ssp, pour permettre au programme d'appel de faire les sauvegardes éventuelles.

Sauvegarde / restitution du contexte

- EXEMPLE : TRANSFERT, utilise ESI, EDI et ECX, modifie DF dans les drapeaux
- Il faut sauvegarder tous ces registres. A cette fin:
 - au début du sous programme, on empile les registres utilisés,
 - avant le RET, on les dépile **dans l'ordre inverse**.
- On parle de **sauvegarde de contexte**

Sauvegarde / restitution du contexte

```
TRANSFERT      PROC
                PUSH    EBP                ; Construction du cadre de pile
                MOV     EBP, ESP

Empilage des    PUSH    ESI                ; Empiler le registre des drapeaux
registres       PUSH    EDI
utilisés       PUSH    ECX

                MOV     ESI, [EBP + 16]    ; Adressage indirect avec déplacement
                MOV     EDI, [EBP + 12]    ; Adresse finale
                MOV     ECX, [EBP + 8]     ; Taille du bloc à transférer
boucle:        DEC     ECX
                JL      fini
                MOV     AL, [ESI + ECX]
                MOV     [EDI + ECX], AL
                JMP     boucle
fini:          RET

Dépilage des    POP     ECX
registres       POP     EDI
utilisés       POP     ESI

                POP     EBP                ; Restitution de EBP sauvegardé
                RET     12
TRANSFERT      ENDP
```

Lien avec les langages de haut niveau : exemple du C

Un programme C

- Un ensemble de fonctions écrites par le programmeur
- Un code de démarrage fourni avec le système/compilateur chargé :
 - des initialisations (mise à zéro des zones de données, pointeurs de pile...)
 - d'appeler une fonction spécifique du programme (main)
- Un code de terminaison fourni avec le système/compilateur chargé de mettre fin au programme.
- La compilation et le lien de ces éléments => image exécutable sous forme d'un fichier

Types d'images exécutables

- Dépendent du système d'exploitation
- Trois types répandus pour l'IA32
 - PE (Portable Executable) => MS Windows
 - ELF32 (Executable and Linkable Format) => Linux, xBSD...
 - Mach-O (Mach Object) => Mac OS X
- Fondamentalement similaires, on y retrouve
 - Un entête (type, l'architecture cible, les différentes sections de l'image...)
 - Une suite de sections (le code exécutable, les données, les symboles non résolus, informations de débogage...)
- EIP initial peut être implicite. Toujours 08048000h sous Linux par exemple.

Un programme C : compilation

Fichier source

```
char str[13] = "Hello,world\n\0";
```

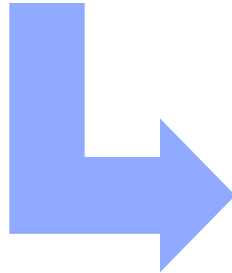
```
int main(void)
{
    printf(str);
    return(0);
}
```

Fichier objet

```
extrn    _printf:near
.data
    _str  db "Hello,world",13,0
.code
; int main(void)
public _main
_main    proc    near
    push    ebp
    mov     ebp,esp
; {
;     printf(str);

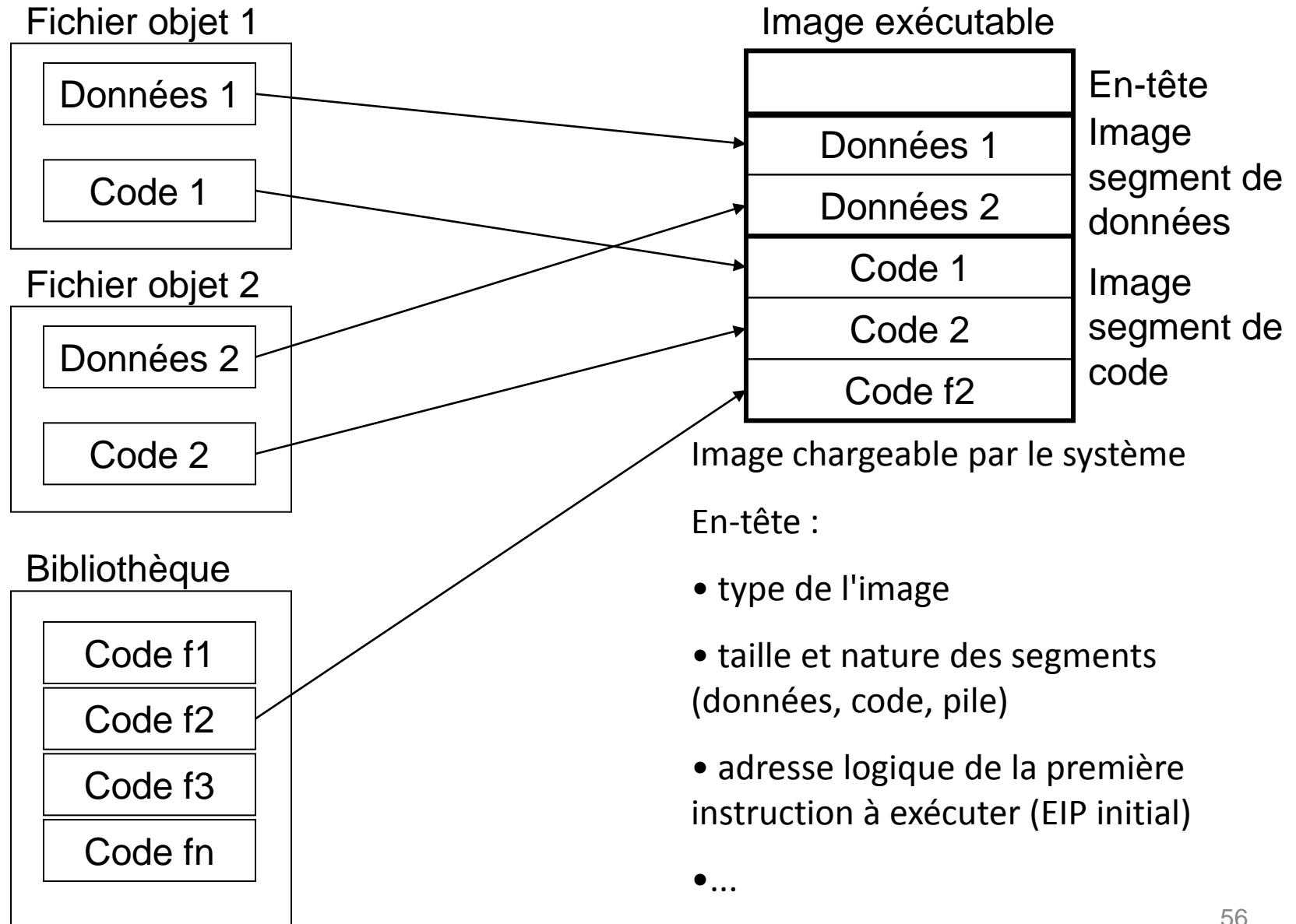
    push    offset _str
    call    _printf
    add     esp, 4
;     return(0);
    xor     eax,eax
; }
    pop     ebp
    ret
_main    endp
```

Compilation



(En réalité, un fichier objet est un fichier binaire après assemblage ; il contient les mêmes informations mais *encodées*)

Construction de l'image 1/3



Construction de l'image 2/3

Fichier objet utilisateur

```
extrn    _printf:near
.data
    _str    db
    "Hello,world",13,0
.code
public    _main
_main    proc near
    push    ebp
    mov     ebp,esp
    push    offset _str
    call    _printf
    add     esp, 4
    xor     eax,eax
    pop     ebp
    ret
_main    endp
```

Bibliothèque C

```
.data
.code
public    _printf
_printf    proc near
    push    ebp
; Code printf
    ret
_printf    endp
```

En-tête	
00000000 0000000D	Hello,world\n\0 ...
00000000 00000001 00000003 00000008 0000000D 00000010 00000012 00000013 00000014 ...	push ebp mov ebp,esp push str call _printf add esp, 4 xor eax,eax pop ebp ret push ebp ... code printf ... ret

- Regroupement des données dans le segment de données, du code dans le segment de code.
- Les adresses logiques des symboles (noms de variables, de fonction...) sont fixées.

Construction de l'image 3/3

Code d'initialisation

```
extrn      _main:near
.data
.code
; int main(void)
public _start
public _exit
_start: ; Point d'entrée
; Init du programme

...
mov esp, ffffffffh
call _main
_exit:
; Terminaison

...
end
```

```
_str { 00000000
      0000000D
```

```
_main { 00000000
        00000001
        00000003
        00000008
        0000000D
        00000010
        00000012
        00000013
```

```
_printf { 00000014
           ...
           00001A54
           00001A55
           00001FE0
           00001FE6
           00001FEC
```

EIP initial=00001A55

Hello,world\n\0
...

```
push ebp
mov  ebp,esp
push _str      ← 00000000
call _printf   ← 00000014
add  esp, 4
xor  eax,eax
pop  ebp
ret
push ebp
... code printf ...
ret
...
mov  esp, 0fffffffh
call _main     ← 00000000
...
```

- Ajout du code de démarrage fourni par le compilateur
- Résolution des références externes (édition de liens proprement dite)
- Création de l'entête (taille des segments, adresses logique de départ...)



Image exécutable complète, prête à être chargée par l'OS

Chargement et exécution

1. Lecture de l'entête
2. Allocation de la mémoire (données et code)
3. Chargement des sections de données dans le segment de données
4. Chargement des sections de code dans le segment de code
5. Initialisation de la pile et des registres
6. Branchement à l'adresse de EIP initial

Exécution

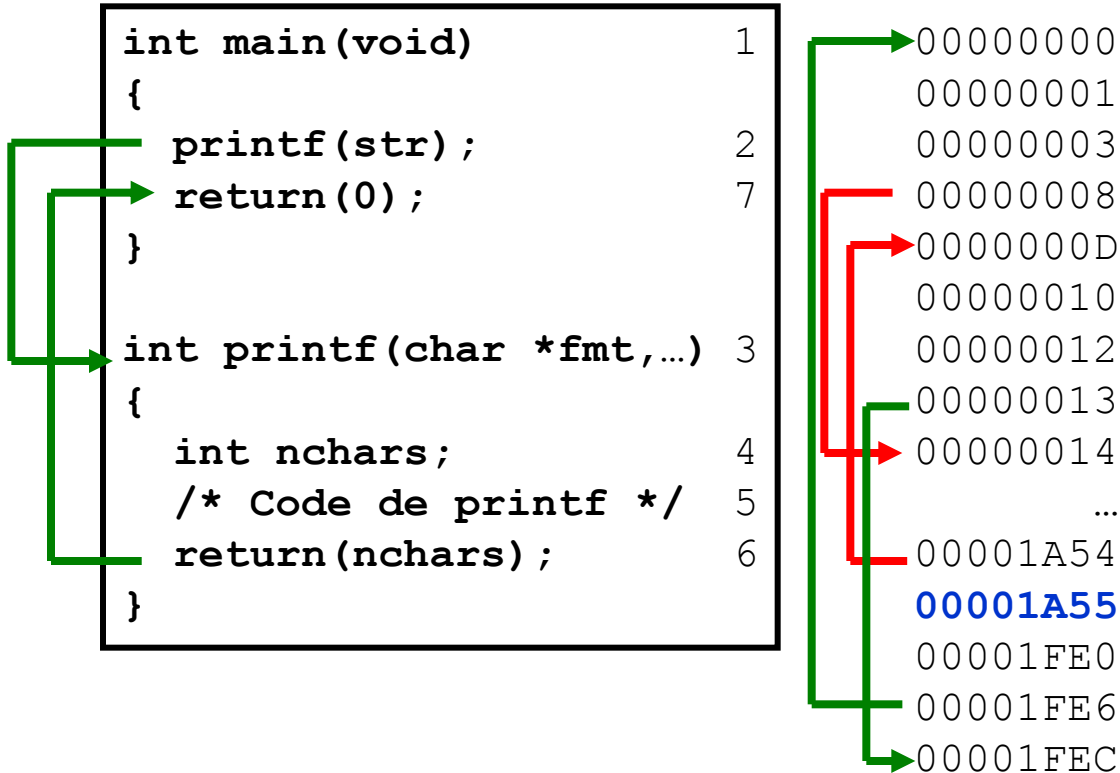
Programme C

```
int main(void)      1
{
    printf(str);    2
    return(0);      7
}

int printf(char *fmt,...) 3
{
    int nchars;     4
    /* Code de printf */ 5
    return(nchars); 6
}
```

Segment de code

```
push ebp           4
mov  ebp,esp       5
push 00000000      6
call 00000014      7
add  esp, 4        11
xor  eax, eax      12
pop  ebp           13
ret                14
push ebp           8
... code printf ... 9
ret                10
...                1
mov  esp, 0fffffffh 2
call 00000000      3
...                17
```



Exécution

- Normalement, le code s'exécute en séquence.
- Les fonctions regroupent des tâches spécifiques dans des blocs de programme réutilisables :
 - L'appel de fonction transfère le contrôle au code de la fonction
 - Le retour de fonction permet de reprendre automatiquement l'exécution à l'instruction qui suit l'appel.
- En C, tout est fonction : c'est la construction fondamentale du langage.

La fonction C

La fonction C

- On n'utilise en général plus l'assembleur pour construire des programmes complets
- Utilisations :
 - Programmation système
 - Pilotes de périphériques
 - Optimisation de procédures critiques (jeux, traitement du signal...)
 - Systèmes embarqués contraints (taille mémoire / contraintes temporelles / architecture inadaptée)
- Dans ces cas, on va devoir appeler des sous-programmes assembleur à partir de code de plus haut niveau comme le langage C.
- Il faut que le code assembleur se conforme aux conventions du langage d'appel.

La fonction C

- Une fonction C c'est :
 - un identifiant, constante symbolique dont la valeur est l'adresse de la première instruction de la fonction
 - 0, 1 ou plusieurs paramètres
 - une valeur de retour (qui peut être ignorée)
 - des instructions qui peuvent faire ou non usage de variables locales.


La fonction C

Exemple 'C' : échange de deux variables. On a deux fonctions : *main* et *echange*

```
void echange(int *a, int *b)
{
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;
}
```

```
int main(void)
{
    int a = 0;
    int b = 1;
    echange(&a, &b);
    return(0);
}
```

A diagram consisting of a horizontal line with an upward-pointing arrow at its right end, connecting the `echange(&a, &b);` line in the `main` function to the `echange` function definition box.

Comment le compilateur traduit ces fonctions ?

La fonction main

<code>_main</code>	<code>proc</code>	<code>near</code>	
<code>;</code>	<code>int</code>	<code>main(void)</code>	
	<code>push</code>	<code>ebp</code>	Construction du cadre de pile
	<code>mov</code>	<code>ebp, esp</code>	
	<code>sub</code>	<code>esp, 8</code>	Réservation d'espace sur la pile pour variables locales
<code>;</code>	<code>{</code>		
<code>;</code>	<code>int</code>	<code>a = 0;</code>	
	<code>xor</code>	<code>eax, eax</code>	Initialisation de a et b
	<code>mov</code>	<code>dword ptr [ebp-4], eax</code>	
<code>;</code>	<code>int</code>	<code>b = 1;</code>	
	<code>mov</code>	<code>dword ptr [ebp-8], 1</code>	
<code>;</code>	<code>exchange(&a, &b);</code>		
	<code>lea</code>	<code>edx, dword ptr [ebp-8]</code>	Préparation de l'appel (empilage des paramètres)
	<code>push</code>	<code>edx</code>	
	<code>lea</code>	<code>ecx, dword ptr [ebp-4]</code>	
	<code>push</code>	<code>ecx</code>	
	<code>call</code>	<code>_exchange</code>	Appel et nettoyage de la pile
	<code>add</code>	<code>esp, 8</code>	
<code>;</code>	<code>return(0);</code>		Calcul de la valeur de retour
	<code>xor</code>	<code>eax, eax</code>	
<code>;</code>	<code>}</code>		
	<code>add</code>	<code>esp, 8</code>	Destruction du cadre de pile et retour à l'appelant
	<code>pop</code>	<code>ebp</code>	
	<code>ret</code>		
<code>_main</code>	<code>endp</code>		

Variables locales

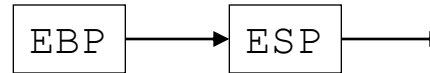
- `main` a deux variables locales (a et b)
 - Les variables locales (ou automatiques) ont la même durée de vie que la fonction.
 - Elles sont créées au moment de l'entrée dans la fonction
 - Elles sont détruites lors du retour à l'appelant
- Le moyen de plus simple est d'utiliser la pile pour les stocker.
- Elles peuvent aussi prendre place dans des registres.

Variables locales

```

_main  proc    near
;      int main(void)
      push    ebp
      mov     ebp, esp
      sub     esp, 8
;      {
;      int a = 0;
      xor     eax, eax
      mov     dword [ebp-4], eax
;      int b = 1;
      mov     dword [ebp-8], 1
      ...
      pop     ecx
      pop     ecx
      pop     ebp
      ret
_main  endp

```

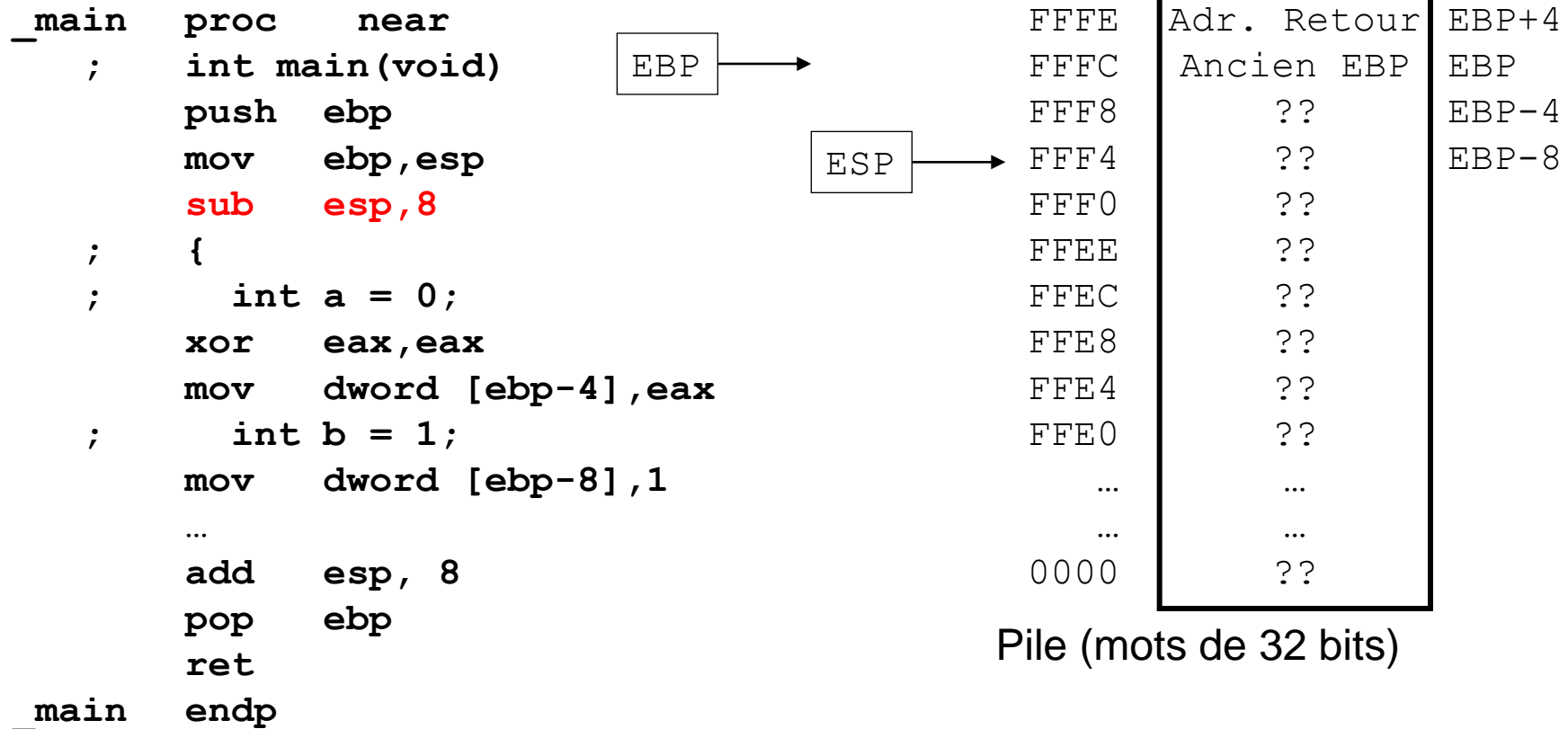


FFFFE	Adr. Retour
FFFFC	Ancien EBP
FFF8	??
FFF4	??
FFF0	??
FFEE	??
FFEC	??
FFE8	??
FFE4	??
FFE0	??
...	...
...	...
0000	??

Pile (mots de 32 bits)

On vient de créer le cadre de pile : ESP pointe sur l'emplacement de sauvegarde de EBP.

Variables locales



On réserve la place pour deux entiers de 32 bits après le cadre de pile :
c'est l'emplacement des variables locales

Variables locales

<code>_main</code>	<code>proc near</code>				FFFFE	Adr. Retour	EBP+4
<code>;</code>	<code>int main(void)</code>		EBP →		FFFFC	Ancien EBP	EBP
	<code>push ebp</code>				FFFF8	00000000	EBP-4
	<code>mov ebp, esp</code>			ESP →	FFFF4	??	EBP-8
	<code>sub esp, 8</code>				FFFF0	??	
<code>;</code>	<code>{</code>				FFEE	??	
<code>;</code>	<code>int a = 0;</code>				FFEC	??	
	<code>xor eax, eax</code>				FFE8	??	
	<code>mov dword [ebp-4], eax</code>				FFE4	??	
<code>;</code>	<code>int b = 1;</code>				FFE0	??	
	<code>mov dword [ebp-8], 1</code>				
	<code>...</code>				
	<code>add esp, 8</code>				0000	??	
	<code>pop ebp</code>						
	<code>ret</code>						
<code>_main</code>	<code>endp</code>						

Pile (mots de 32 bits)

L'adresse de la variable locale a est EBP-4, on la met à 0

Variables locales

<code>_main</code>	<code>proc near</code>				FFFFE	Adr. Retour	EBP+4
<code>;</code>	<code>int main(void)</code>		EBP →		FFFFC	Ancien EBP	EBP
	<code>push ebp</code>				FFFF8	00000000	EBP-4
	<code>mov ebp, esp</code>			ESP →	FFFF4	00000001	EBP-8
	<code>sub esp, 8</code>				FFFF0	??	
<code>;</code>	<code>{</code>				FFFE	??	
<code>;</code>	<code>int a = 0;</code>				FFEC	??	
	<code>xor eax, eax</code>				FFE8	??	
	<code>mov dword [ebp-4], eax</code>				FFE4	??	
<code>;</code>	<code>int b = 1;</code>				FFE0	??	
	<code>mov dword [ebp-8], 1</code>				
	<code>...</code>				
	<code>add esp, 8</code>				0000	??	
	<code>pop ebp</code>						
	<code>ret</code>						
<code>_main</code>	<code>endp</code>						

Pile (mots de 32 bits)

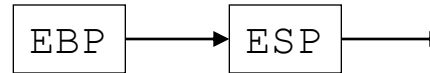
L'adresse de la variable locale b est EBP-8, on la met à 1

Variables locales

```

_main  proc    near
;      int main(void)
      push    ebp
      mov     ebp,esp
      sub     esp,8
;      {
;      int a = 0;
      xor     eax,eax
      mov     dword [ebp-4],eax
;      int b = 1;
      mov     dword [ebp-8],1
      ...
      add     esp, 8
      pop     ebp
      ret
_main  endp

```



FFFE	Adr. Retour	EBP+4
FFFC	Ancien EBP	EBP
FFF8	00000000	EBP-4
FFF4	00000001	EBP-8
FFF0	??	
FFEE	??	
FFEC	??	
FFE8	??	
FFE4	??	
FFE0	??	
...	...	
...	...	
0000	??	

Pile (mots de 32 bits)

En ajustant ESP, on revient à la situation initiale : les deux variables locales sont détruites, et on peut détruire le cadre de pile et revenir à l'appelant.

Passage de paramètres

<code>_main</code>	<code>proc</code>	<code>near</code>	
<code>;</code>	<code>int main(void)</code>		
	<code>push</code>	<code>ebp</code>	Construction du cadre de pile
	<code>mov</code>	<code>ebp, esp</code>	
	<code>sub</code>	<code>esp, 8</code>	Réservation d'espace sur la pile pour variables locales
<code>;</code>	<code>{</code>		
<code>;</code>	<code>int a = 0;</code>		
	<code>xor</code>	<code>eax, eax</code>	Initialisation de a et b
	<code>mov</code>	<code>dword ptr [ebp-4], eax</code>	
<code>;</code>	<code>int b = 1;</code>		
	<code>mov</code>	<code>dword ptr [ebp-8], 1</code>	
<code>;</code>	<code>exchange(&a, &b);</code>		
	<code>lea</code>	<code>edx, dword ptr [ebp-8]</code>	Préparation de l'appel (empilage des paramètres)
	<code>push</code>	<code>edx</code>	
	<code>lea</code>	<code>ecx, dword ptr [ebp-4]</code>	
	<code>push</code>	<code>ecx</code>	
	<code>call</code>	<code>_exchange</code>	Appel et nettoyage de la pile
	<code>add</code>	<code>esp, 8</code>	
<code>;</code>	<code>return(0);</code>		Calcul de la valeur de retour
	<code>xor</code>	<code>eax, eax</code>	
<code>;</code>	<code>}</code>		
	<code>add</code>	<code>esp, 8</code>	Destruction du cadre de pile et retour à l'appelant
	<code>pop</code>	<code>ebp</code>	
	<code>ret</code>		
<code>_main</code>	<code>endp</code>		

Passage de paramètres

- Les paramètres sont empilés dans l'ordre du dernier au premier
- Facilite la gestion de fonctions avec un nombre variable de paramètres (ex. printf)

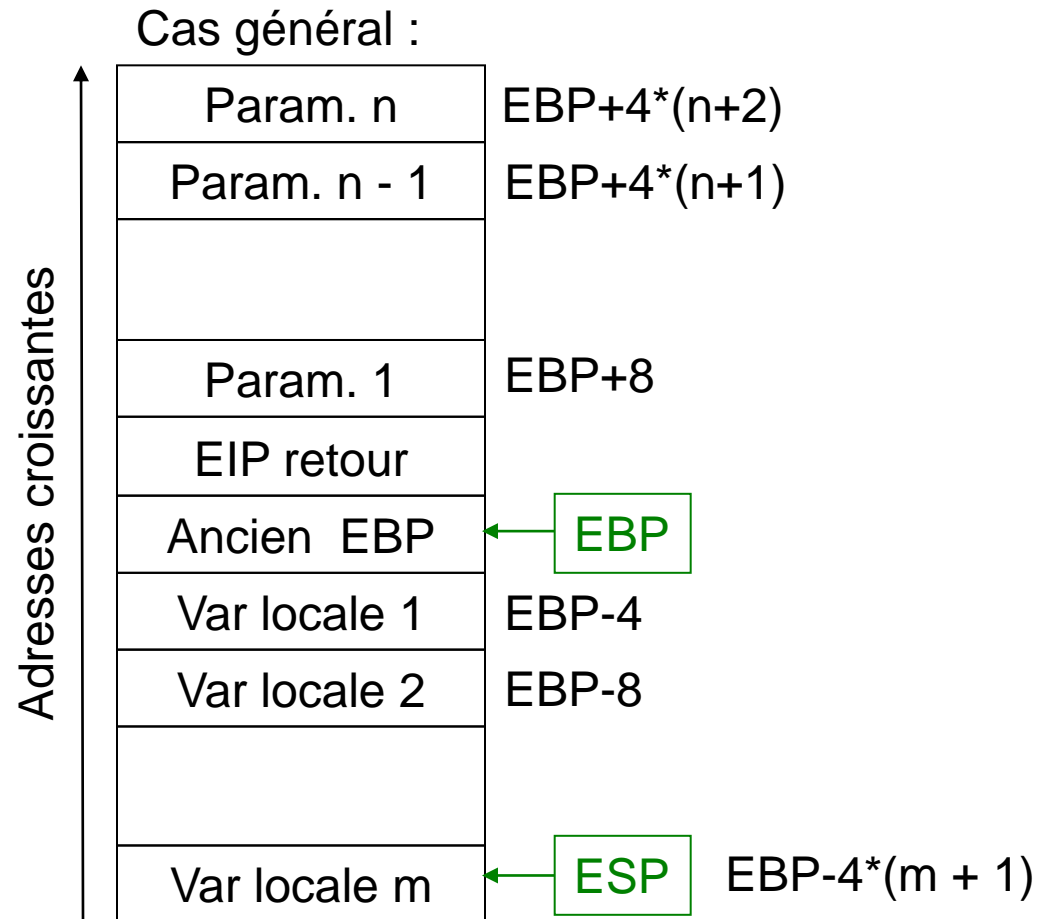
Valeur de retour

<code>_main</code>	<code>proc</code>	<code>near</code>	
<code>;</code>	<code>int main(void)</code>		
	<code>push</code>	<code>ebp</code>	Construction du cadre de pile
	<code>mov</code>	<code>ebp, esp</code>	
	<code>sub</code>	<code>esp, 8</code>	Réservation d'espace sur la pile pour variables locales
<code>;</code>	<code>{</code>		
<code>;</code>	<code>int a = 0;</code>		
	<code>xor</code>	<code>eax, eax</code>	Initialisation de a et b
	<code>mov</code>	<code>dword ptr [ebp-4], eax</code>	
<code>;</code>	<code>int b = 1;</code>		
	<code>mov</code>	<code>dword ptr [ebp-8], 1</code>	
<code>;</code>	<code>exchange(&a, &b);</code>		
	<code>lea</code>	<code>edx, dword ptr [ebp-8]</code>	Préparation de l'appel (empilage des paramètres)
	<code>push</code>	<code>edx</code>	
	<code>lea</code>	<code>ecx, dword ptr [ebp-4]</code>	
	<code>push</code>	<code>ecx</code>	
	<code>call</code>	<code>_exchange</code>	Appel et nettoyage de la pile
	<code>add</code>	<code>esp, 8</code>	
<code>;</code>	<code>return(0);</code>		Calcul de la valeur de retour
	<code>xor</code>	<code>eax, eax</code>	
<code>;</code>	<code>}</code>		
	<code>add</code>	<code>esp, 8</code>	Destruction du cadre de pile et retour à l'appelant
	<code>pop</code>	<code>ebp</code>	
	<code>ret</code>		
<code>_main</code>	<code>endp</code>		

La fonction main

- On remarque que :
 - les variables locales sont stockées sur la pile;
 - les paramètres d'une fonction sont passés par la pile;
 - les paramètres sont supprimés de la pile par le programme qui réalise l'appel;
 - la valeur de retour d'une fonction est placée dans EAX avant l'instruction RET.
- Ce sont les **conventions d'appel** utilisées par la plupart des compilateurs C (Visual C, gcc...) sur IA32
- **On doit les respecter pour intégrer des sous-programmes en assembleur à un programme C (et vice-versa)**

Structure de la pile d'une fonction C



Segment de pile 32 bits (SS)

Attention aux variables locales

<code>#include <stdio.h></code>	<code>main</code>	<code>proc</code>	<code>near</code>
	<code>;</code>		
<code>int main(void)</code>	<code>;</code>	<code>int main(void)</code>	
	<code>;</code>		
<code>{</code>		<code>push</code>	<code>ebp</code>
		<code>mov</code>	<code>ebp, esp</code>
<code>char tab[4];</code>	←	<div>add esp, 4</div>	
<code>gets(tab);</code>	←	<div>lea eax, dword ptr [ebp-4] push eax call gets</div>	
<code>return(0);</code>	←	<div>xor eax, eax sub esp, 4 pop ebp ret</div>	
<code>}</code>			
	<code>main</code>	<code>endp</code>	

Le langage C ne vérifie pas la taille des tableaux. Dans ce programme, tab est alloué sur la pile...

Débordement de buffer

Que se passe-t-il si gets lit plus de 4 caractères ?

- gets peut modifier des espaces sur la pile qui sont hors du tableau.
- On parle de débordement de buffer (*buffer overflow*). C'est une des erreurs les plus courantes en C, responsable de beaucoup de failles de sécurité dans les systèmes et réseaux.
- Si on construit « intelligemment » l'entrée, on peut obtenir l'exécution d'un code arbitraire...