Cours Assembleur 8086

Plan

Plan

Plan

Usages

Pourquoi faire de l'assembleur ?

- Ecrire un compilateur
- Environnements embarqués, micro-controlleurs
- Systèmes temps-réels durs

Avantages

- Meilleure compréhension des langages
- Meilleure compréhension des machines

Plan

La machine

Deux fonctions

- Calculer : rôle du (micro)-processeur
- Stocker : rôle de la mémoire

Langage spécifique

- Un langage par processeur, appellé jeu d'instructions
- Une référence commune : le binaire

Parler binaire, octal et hexadécimal

Syntaxe

- Binaire : 0b1010 ou 1010b
- Octal : 012 ou 12o
- Hexadécimal : 0xA ou 0Ah

Usage

- Masques de bits
- Permissions
- Adresses mémoire

Notation des entiers

Positif et négatifs

- Les entiers positifs sont stockés en binaire par conversion simple
- Les entiers négatifs sont stockés en binaire par complément à deux
- Cette méthode permet des opérations arithmétiques sans corrections

Complément à deux

- Le complément à deux se calcule en deux étapes
  - On inverse d'abord les bits (complément à un)
  - On ajoute 1 au résultat
- Par exemple
  - 13 se note 0000 1101 sur 8 bits
  - -13 se note 1111 0011 sur 8 bits
- L'opposé d'un entier est son complément à deux.
  Il se calcule avec le mnemonic neg

Plan

Registres

Mémoires de calcul

- Les calculs se décomposent en opérations élémentaires
- Ces opérations ont besoin de stocker des résultats intermédiaires
- Le processeur contient de petites mémoires appellées registres

Registres sur 8086

- 8 Registres généraux (16 bits)
  - ax, bx, cx, dx, si, di, bp et sp
- 4 Registres de segments (16 bits)
  - cs, ds, es et ss
- 2 Registres spéciaux (16 bits)
  - ip et flags

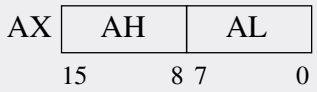
Sous-registres

Décomposition des registres

- Les registres A, B, C et D se décomposent en deux sous-registres de 8 bits chacun
  - h (partie haute) et l (partie basse).
  - Par exemple, ax se décompose en ah et al
  - En C, on aurait la définition suivante :

```
union {
  struct {
    uint8_t al;
    uint8_t ah;
  };
  uint16_t ax;
};
```

  - Et à tout instant : ax = ah * 256 + al



Plan

Plan

Jeu d'instructions

Instructions

- Les instructions permettent de spéficier les opérations à effectuer
- Elles sont données au processeur sous la forme d'une chaine binaire
- En assembleur, on les écrit sous forme de mnemonics
- On utilise un programme d'assemblage pour transformer les mnemonics en binaire
- La traduction binaire d'un mnemonic est appellée un opcode

Exemples d'instructions

Instructions "load/store"

- mov *dst*, *src* : Place la valeur de *src* dans *dst*

```
mov ax,14
```

- push *src* : Place la valeur de *src* sur la pile

```
push ax
```

- pop *dst* : Place la valeur au sommet de la pile dans *dst*

```
pop bx
```

Exemples d'instructions

Instructions de calcul

- add *dst*, *src* : Additionne *src* et *dst* et place le résultat dans *dst*

```
add ax,bx ; ax ⟵ ax + bx
```

- sub *dst*, *src* : Soustrait *src* de *dst* et place le résultat dans *dst*

```
sub cx,dx ; cx ⟵ cx - dx
```

- shl *dst*, *src* : Décale *dst* de *src* bits vers la gauche et place le résultat dans *dst*

```
shl dx,1 ; dx ⟵ dx * 2
```

Plan

Accéder à la mémoire

Structure de la mémoire

- La mémoire se comporte comme un grand tableau d'octets
- Accéder à la mémoire consiste à accéder à une case du tableau en indiquant son indice, appellé adresse
- Un accès mémoire se fait selon une certaine taille : on peut accéder à plusieurs octets contiguës en une seule fois
  - On peut lire l'octet situé à l'adresse 123h
  - On peut écrire deux octets situés à l'adresse 456h
    Le premier va dans la case 456h et le deuxième dans la case 457h

Granularité

Endianness

- Supposons qu'on veuille stocker l'entier A035h dans la mémoire, on a deux possibilités
  - Stocker d'abord l'octet A0h, puis l'octet 35h à la suite, c'est le mode big-endian
  - Stocker d'abord l'octet 35h, puis l'octet A0h à la suite, c'est le mode little-endian
- Le 8086 et ses successeurs sont little-endian
- Le Motorola 68000 et le PowerPC sont big-endian

Limitations 16 bits

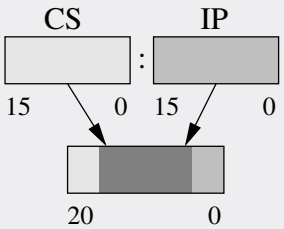
Adressabilité

- On donne l'adresse avec un registre
- Les registres font 16 bits, donc on ne peut accéder qu'à $2^{16} = 65536 = 64$ Ko de mémoire
- Pour accéder à plus de mémoire, on découpe la mémoire en segments de 64 Ko et on choisit son segment avec un registre de segment

Segmentation

Mémoire et segments

- Le 8086 accepte jusqu'à 1 Mo de mémoire. On l'adresse à l'aide d'une paire de registres :
  - Un registre de segment, dont la valeur est appellée base ou segment.
  - Un registre général ou une constante, dont la valeur est appellée déplacement ou offset.
  - L'adresse cs : ip correspond à l'octet numéro $cs * 16 + ip$



  - On a donc une adresse sur 20 bits, qui permet d'accéder à $2^{20} = 1$ Mo de mémoire, par blocs de 64 Ko
- Ce mécanisme s'appelle la segmentation en mode réel
- À tout instant, l'adresse cs : ip pointe sur la prochaine instruction à exécuter

Les modes d'adressage

Méthodes d'accès aux données

- Adressage immédiat : l'opérande est une constante

```
mov ah,10h ; => Opcode B410
```

- Adressage direct : l'opérande est une case mémoire (registre ds par défaut)

```
mov al,[10h]    ; <=> mov al,[ds:10h] => Opcode A01000
mov ax,[es:10h] ;                     => Opcode 26A11000
```

- Adressage basé : l'opérande est une case mémoire dont l'adresse est donnée par bx (avec ds par défaut) ou bp (avec ss par défaut)

```
mov ah,[bx] ; <=> mov ah,[ds:bx] => Opcode 8A27
mov al,[bp] ; <=> mov al,[ss:bp] => Opcode 8A4600
```

Les modes d'adressage

Méthodes d'accès aux données

- Adressage indexé : l'opérande est une case mémoire dont l'adresse est donnée par si ou di (avec ds par défaut, sauf mnemonic spécifique)

```
mov ah,[si] ; <=> mov ah,[ds:si] => Opcode 8A24
```

- Adressage basé et indexé

```
mov ah,[bx+di] ; <=> mov ah,[ds:bx+di] => Opcode 8A21
mov [bp+si],ah ; <=> mov [ss:bp+si],ah => Opcode 8822
```

- Adressage basé avec déplacement

```
mov ah,[bx+123h] ; <=> mov ah,[ds:bx+123h] => Opcode 8AA72301
```

Les modes d'adressage

Méthodes d'accès aux données

- Adressage indexé avec déplacement

```
mov ah,[di+123h] ; <=> mov ah,[ds:di+123h] => Opcode 8AA52301
```

- Adressage basé et indexé avec déplacement

```
mov ah,[bx+si+123h] ; <=> mov ah,[ds:bx+si+123h] => Opcode 8AA02301
```

La pile

Manipulation de la pile

- La pile est une zone de mémoire
- Son sommet est la case mémoire à l'adresse ss : sp
- On empile une valeur avec push et on dépile avec pop

```
push ax ; => Opcode 50
        ; <=> sub sp,2
        ;     mov [ss:sp],ax
```

```
pop ax ; => Opcode 58
       ; <=> mov ax,[ss:sp]
       ;     add sp,2
```

- Les données ne sont pas effacés après un pop
- Un push écrase les données précédemment dans la pile

Plan

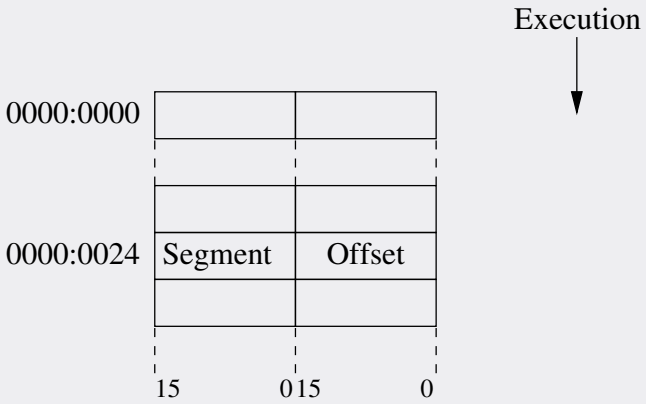
Interruptions

Fonctions

- La plupart des machines disposent d'un mécanisme pour indiquer les évènement extérieurs (clavier, disque dur, ...), appellé interruptions
- Leur fonctionnement consiste à interrompre le programme en cours d'exécution et de lancer un autre bout de code (appellé gestionnaire) pour gérer l'évènement
- Sur le 8086, les interruptions sont spécifiées par un numéro sur 8 bits : il y'a donc 256 interruptions différentes
- La programme à exécuter lors d'une interruption est donnée par une adresse segment : offset
- Une table des adresses des interruptions se situent dans la mémoire entre les adresses 0 et 1024 (inclus)
- Le numéro d'interruption sert d'indice dans cette table pour trouver l'adresse du programme à lancer

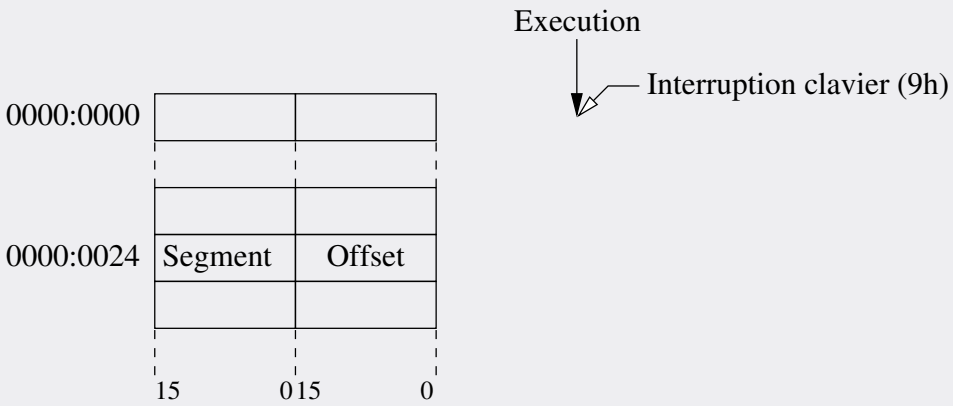
Interruptions

Schéma d'exécution



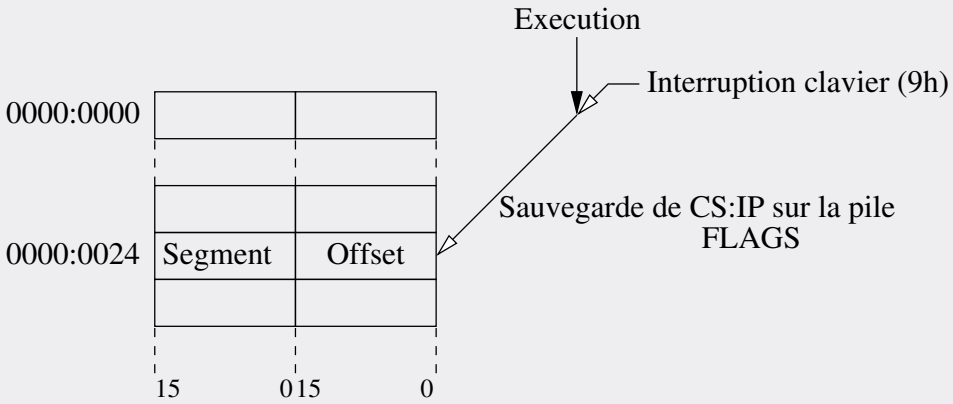
Interruptions

Schéma d'exécution



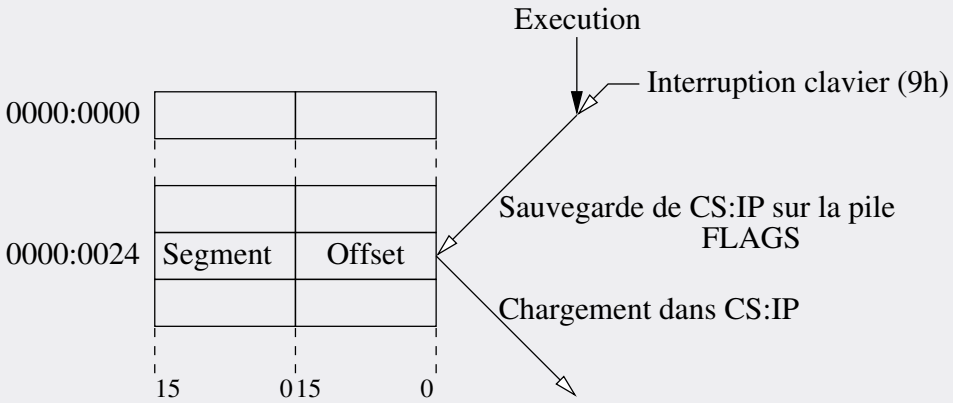
Interruptions

Schéma d'exécution



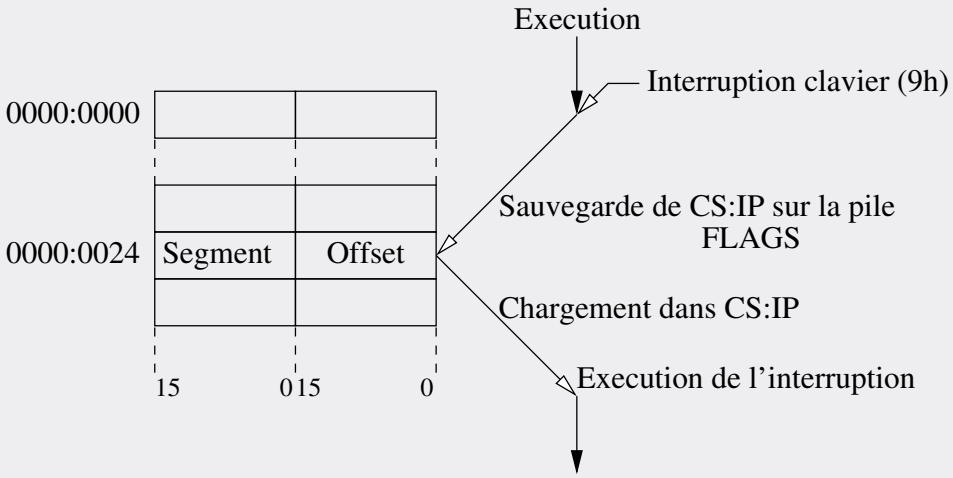
Interruptions

Schéma d'exécution



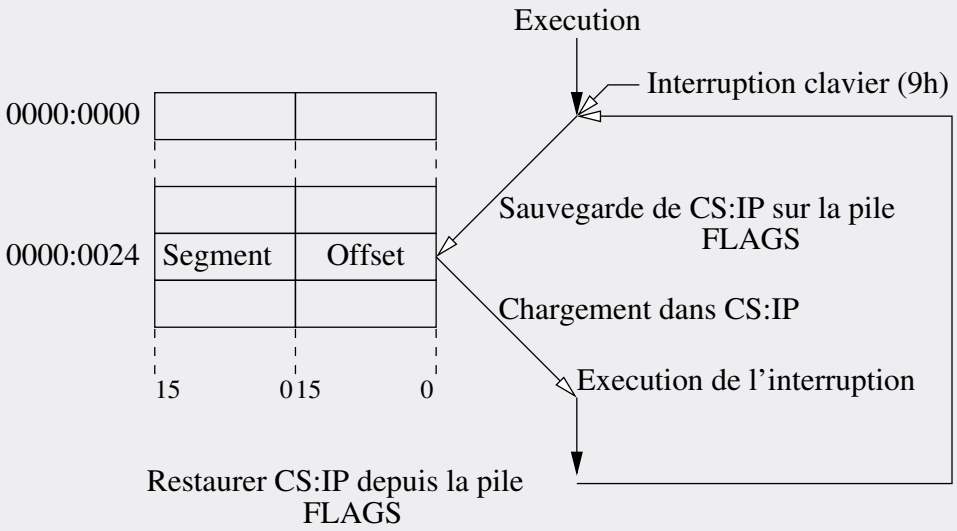
Interruptions

Schéma d'exécution



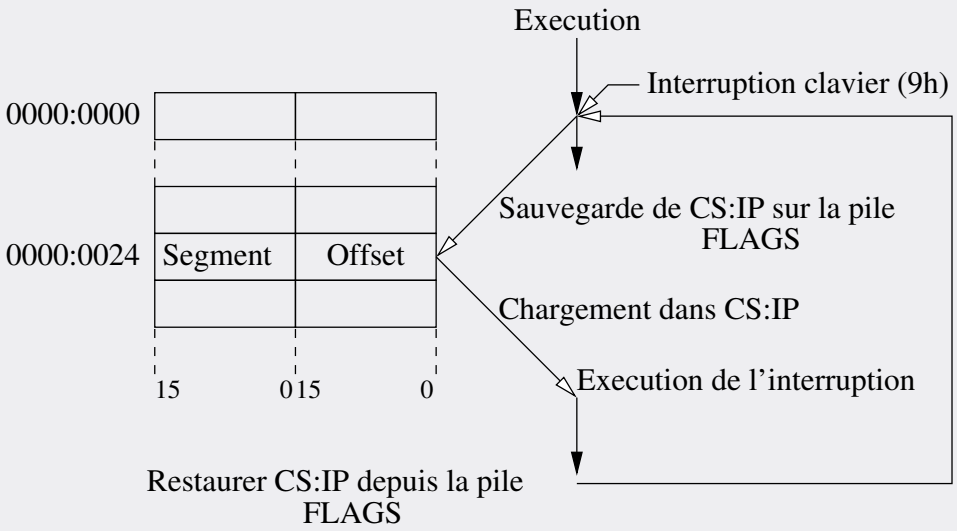
Interruptions

Schéma d'exécution



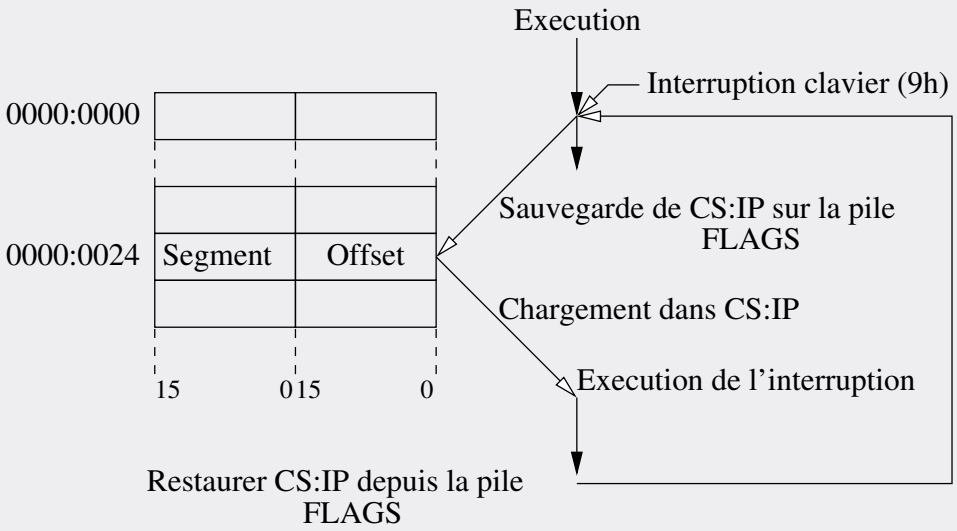
Interruptions

Schéma d'exécution



Interruptions

Schéma d'exécution



Transparence

- Les interruptions matérielles sont toujours transparentes (sauvegarde des registres)

Interruptions

Gestionnaires par défaut

- Le BIOS met en place les adresses pour les 32 premières interruptions, qui sont des interruptions générées par le matériel
- Le DOS met en place les adresses pour les 32 interruptions suivantes, qui sont des fonctions qu'il propose aux programmes
- Les interruptions au-delà de la 64ème (incluse) sont libres pour le programmeur
- Pour appeller une interruption manuellement, on utilise le mnemonic int avec comme seule opérande le numéro de l'interruption

Plan

Directive

Commandes spéciales

- Les directives sont des commandes spéciales du programme d'assemblage et ne sont pas des mnemonics.
- Pour le programme fasm, les principales directives regroupent :
  - format qui indique le format du fichier executable à générer
  - use16 qui indique que l'on souhaite compiler du code 16 bits
  - org qui indique la valeur de départ de ip

Directives communes

- db n : insère l'octet *n* à cet endroit
- dw n : insère le mot (= 2 octets) *n* à cet endroit
- rb n : réserve la place pour *n* octets
- rw n : réserve la place pour *n* mots
- Remarque : db fonctionne aussi avec une chaîne de caractères et prend les codes ASCII dans ce cas

Plan

Hello world

Code source

```
; Version fasm, syntaxe "intel"
format binary
use16
org 100h

mov ah,09h      ; Fonction 09h
mov dx,message  ; Paramètre: dx recoit l'adresse de message
int 021h        ; Appeller la fonction

int 020h        ; Appeller la fonction

message:
  db "Hello world$"
```

Plan

Registre flags

- La plupart des opérations de calculs donnent des informations sur le résultat dans le registre spécial flags
- Les informations possibles sont :
  - CF : 1 si le résultat a donné lieu à une retenue
  - PF : 1 si l'octet de poids faible du résultat à un nombre pair de 1
  - AF : 1 si le résultat a donné lieu à une retenue sur le 3^{eme} bit
  - ZF : 1 si le résultat est zéro
  - SF : 1 si le résultat est négatif
  - IF : 1 si les interruptions peuvent arriver
  - DF : 0 si la direction est incrémentée, 1 si elle est décrémentée
  - OF : 1 si le résultat ne tient pas dans la destination

Plan

Branchements

Sélectionner du code

- En C, on peut utiliser des structures de contrôle type if, while ou for pour contrôler l'exécution du code
- En assembleur, la prochaine instruction est toujours donnée par cs : ip
- Pour accéder à une autre partie du code, on utilise des branchements (ou goto)

```
mon_code:
  ; On fait des choses ici
  jmp mon_code ; On retourne à l'étiquette "mon_code"
```

Branchements conditionnels

Tester des indicateurs

- Le registre flags contient des indicateurs qui peuvent être testés et s'ils sont mis (ou éteints), on peut effecteur un saut
- C'est la méthode des branchements conditionnels

```
mon_code:
  ; On fait des choses ici
  cmp ax, 0    ; On effectue une comparaison
  jz mon_code ; On saute à mon_code si le résultat est zéro
```

Plan

Fonctions

Appel

- On découpe le code en fonctions pour simplifier la lecture et la maintenance du code
- Pour écrire une fonction en assembleur, il faut savoir comment l'appeller et comment lui passer des arguments
- L'appel de fonction se fait avec le mnemonic call suivi de l'adresse de la fonction
  - Si l'adresse est sur 16 bits, on dit que c'est un call near (i.e. dans le même segment)
  - Si l'adresse est sur 16 : 16 bits, on dit que c'est un call far (i.e. dans un autre segment)
- Le mnemonic sauvegarde ip sur la pile, c'est l'adresse de retour
- Pour un far call, l'instruction sauvegarde cs sur la pile (avant ip)

Fonctions

Retour

- À la fin d'une fonction, on utilise le mnemonic ret, qui restaure ip depuis la pile
- Pour faire un retour depuis une fonction appellée en far call, il faut utilier retf, qui restaure aussi cs depuis la pile
- Pour éviter tout problème, il est impératif que la pile soit dans le même état avant de faire un ret ou un retf qu'au début de la fonction

Fonctions

Arguments

- Il existe plusieurs méthodes pour passer des arguments à une fonction
- Le choix de la méthode dépend d'une convention entre appellant et appellé

Passage par registre

- La convention spécifie les arguments que doivent contenir les registres
- Elle est utilisée par les interruptions
- Avantages
  - Rapide à l'exécution
  - Simple à programmer
- Inconvénients
  - Il n'y a que 8 registres utilisables
  - Il faut sauvegarder les registres avant chaque appel

Fonctions

Passage par la pile

- Cette convention consiste à empiler les arguments sur la pile avant l'appel
- La fonction récupère les arguments sur la pile
- Avantages
  - On peut utiliser plus d'arguments
  - Meilleur interface avec d'autres langages
  - Plus facile d'appeller en cascade
- Inconvénients
  - Plus difficile à mettre en oeuvre
  - Moins rapide à l'exécution
- En plus de ces facteurs, la convention doit spécifier plusieurs points
  - L'ordre dans lequel empiler les arguments (en C : du dernier au premier, en Pascal : du premier au dernier)
  - Qui doit nettoyer la pile ? (en C : l'appellant, en Pascal : l'appellé)

Fonctions

Valeur de retour

- Les valeurs de retour sont aussi soumises à un choix de convention
- Les même conventions que pour le passage de paramètre s'appliquent
- Le retour sur la pile est beaucoup plus complexe que le retour par les registres

Exemple de fonction

Somme

- Comme exemple, prenons une fonction à deux arguments a et b et qui retourne leur moyenne

```
int moyenne (int a, int b) {
  return (a + b) / 2;
}
```

Exemple de fonction

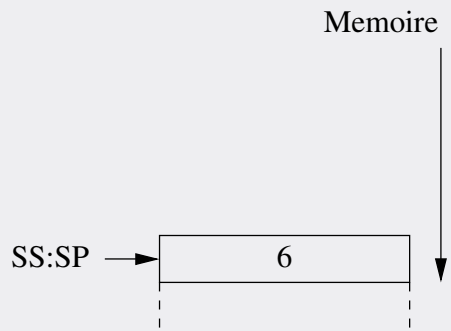
Passage et retour par registres

```
; Convention:
;   ax contient le premier entier
;   bx contient le deuxième entier
;   le résultat sera dans ax
moyenne:
  add ax, bx
  shr ax, 1
  ret

mov ax, 12
mov bx, 6
call moyenne
; ici ax contient 9
```

Exemple de fonction

Passage par la pile, retour par registre

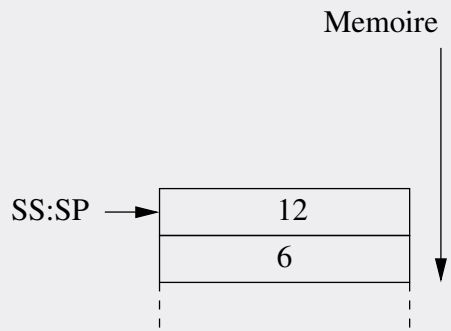
```
; Convention:
;   premier entier empilé en dernier
;   deuxième entier empilé en avant-dernier
;   le résultat sera dans ax
moyenne:
  push bp
  mov bp,sp
  push bx
  mov ax,[bp+4]
  mov bx,[bp+6]
  add ax,bx
  shr ax,1
  pop bx
  pop bp
  ret

push 6          ←
push 12
call moyenne
; ici ax contient 9
add sp,4
```



Exemple de fonction

Passage par la pile, retour par registre

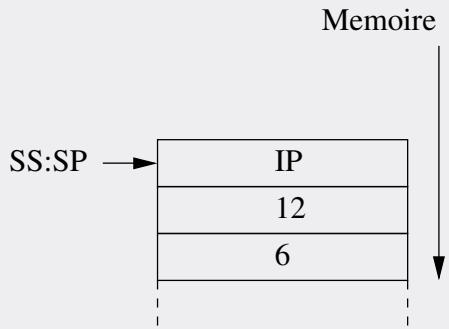
```
; Convention:
;   premier entier empilé en dernier
;   deuxième entier empilé en avant-dernier
;   le résultat sera dans ax
moyenne:
  push bp
  mov bp,sp
  push bx
  mov ax,[bp+4]
  mov bx,[bp+6]
  add ax,bx
  shr ax,1
  pop bx
  pop bp
  ret

push 6
push 12          ←
call moyenne
; ici ax contient 9
add sp,4
```



Exemple de fonction

Passage par la pile, retour par registre

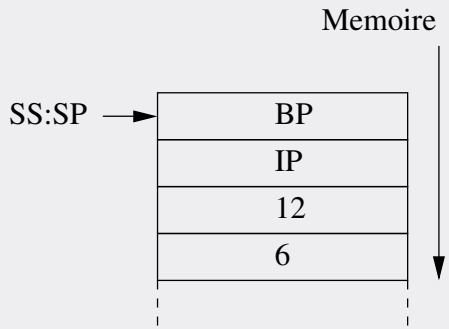
```
; Convention:
;   premier entier empilé en dernier
;   deuxième entier empilé en avant-dernier
;   le résultat sera dans ax
moyenne:
  push bp
  mov bp,sp
  push bx
  mov ax,[bp+4]
  mov bx,[bp+6]
  add ax,bx
  shr ax,1
  pop bx
  pop bp
  ret

push 6
push 12
call moyenne     ←
; ici ax contient 9
add sp,4
```



Exemple de fonction

Passage par la pile, retour par registre

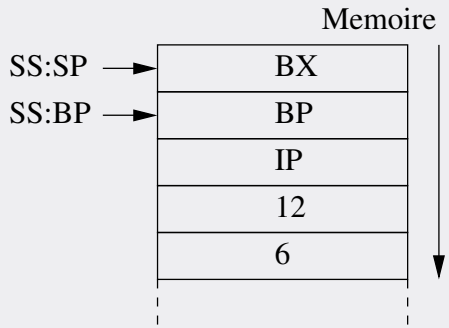
```
; Convention:
;   premier entier empilé en dernier
;   deuxième entier empilé en avant-dernier
;   le résultat sera dans ax
moyenne:
  push bp        ←
  mov bp,sp
  push bx
  mov ax,[bp+4]
  mov bx,[bp+6]
  add ax,bx
  shr ax,1
  pop bx
  pop bp
  ret

push 6
push 12
call moyenne
; ici ax contient 9
add sp,4
```



Exemple de fonction

Passage par la pile, retour par registre

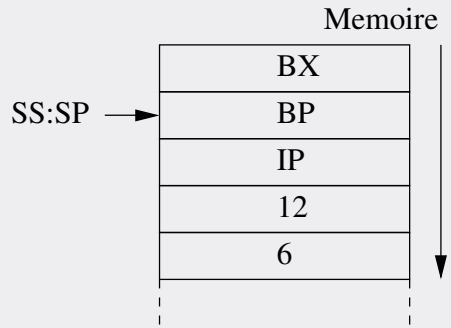
```
; Convention:
;   premier entier empilé en dernier
;   deuxième entier empilé en avant-dernier
;   le résultat sera dans ax
moyenne:
  push bp
  mov bp,sp
  push bx          ←
  mov ax,[bp+4]
  mov bx,[bp+6]
  add ax,bx
  shr ax,1
  pop bx
  pop bp
  ret

push 6
push 12
call moyenne
; ici ax contient 9
add sp,4
```



Exemple de fonction

Passage par la pile, retour par registre

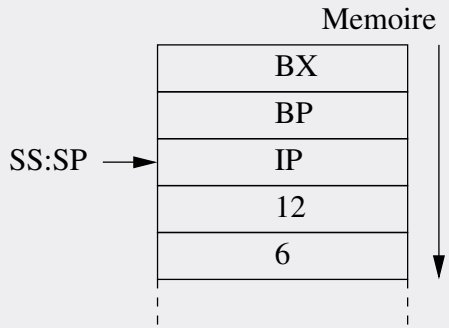
```
; Convention:
;   premier entier empilé en dernier
;   deuxième entier empilé en avant-dernier
;   le résultat sera dans ax
moyenne:
  push bp
  mov bp,sp
  push bx
  mov ax,[bp+4]
  mov bx,[bp+6]
  add ax,bx
  shr ax,1
  pop bx          ←
  pop bp
  ret

push 6
push 12
call moyenne
; ici ax contient 9
add sp,4
```



Exemple de fonction

Passage par la pile, retour par registre

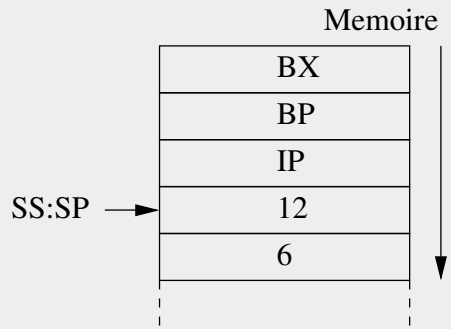
```
; Convention:
;   premier entier empilé en dernier
;   deuxième entier empilé en avant-dernier
;   le résultat sera dans ax
moyenne:
  push bp
  mov bp,sp
  push bx
  mov ax,[bp+4]
  mov bx,[bp+6]
  add ax,bx
  shr ax,1
  pop bx
  pop bp          ←
  ret

push 6
push 12
call moyenne
; ici ax contient 9
add sp,4
```



Exemple de fonction

Passage par la pile, retour par registre

```
; Convention:
;   premier entier empilé en dernier
;   deuxième entier empilé en avant-dernier
;   le résultat sera dans ax
moyenne:
  push bp
  mov bp,sp
  push bx
  mov ax,[bp+4]
  mov bx,[bp+6]
  add ax,bx
  shr ax,1
  pop bx
  pop bp
  ret                ←

push 6
push 12
call moyenne
; ici ax contient 9
add sp,4
```



Exemple de fonction

Passage par la pile, retour par registre

```
; Convention:
;   premier entier empilé en dernier
;   deuxième entier empilé en avant-dernier
;   le résultat sera dans ax
moyenne:
  push bp
  mov bp,sp
  push bx
  mov ax,[bp+4]
  mov bx,[bp+6]
  add ax,bx
  shr ax,1
  pop bx
  pop bp
  ret

push 6
push 12
call moyenne
; ici ax contient 9
add sp,4          ←
```