Agenda du cours

- Cours 1 :
  - Généralités archi/assembleur
  - Manipulation émulateur
  - **Code, UAL, registres, mémoire**
  - Exécution, visualisation registres
- Cours 2 : Hiérarchie des mémoires
  - Différents **types de mémoires**
  - Accès mémoire (code, données, E/S)
  - Manipulation structure de données en assembleur
- Cours 3 : Appel de procédures
  - **Notion de Pile**
  - Appel de procédures
- Cours 4 : **Interruptions**
  - Mécanismes internes
  - Programmation d'Interruptions
  - Application aux E/S
- Cours 5 :
  - Développement programme
  - E/S , IT,..
- Cours 6 :
  - **Examen**

Mais avant ….

- Ouvrez vos navigateurs web sur

https://b.socrative.com/student/

- RV dans la salle

406708

- Mettez votre nom et … répondez aux questions

Hiérarchie des mémoires

- Machine idéale
  - taille mémoire **illimitée**
  - **temps** d'accès aux données **nul**
- Problème
  - Coût
  - Localisation : **intégration**

=> Hiérarchie de mémoires

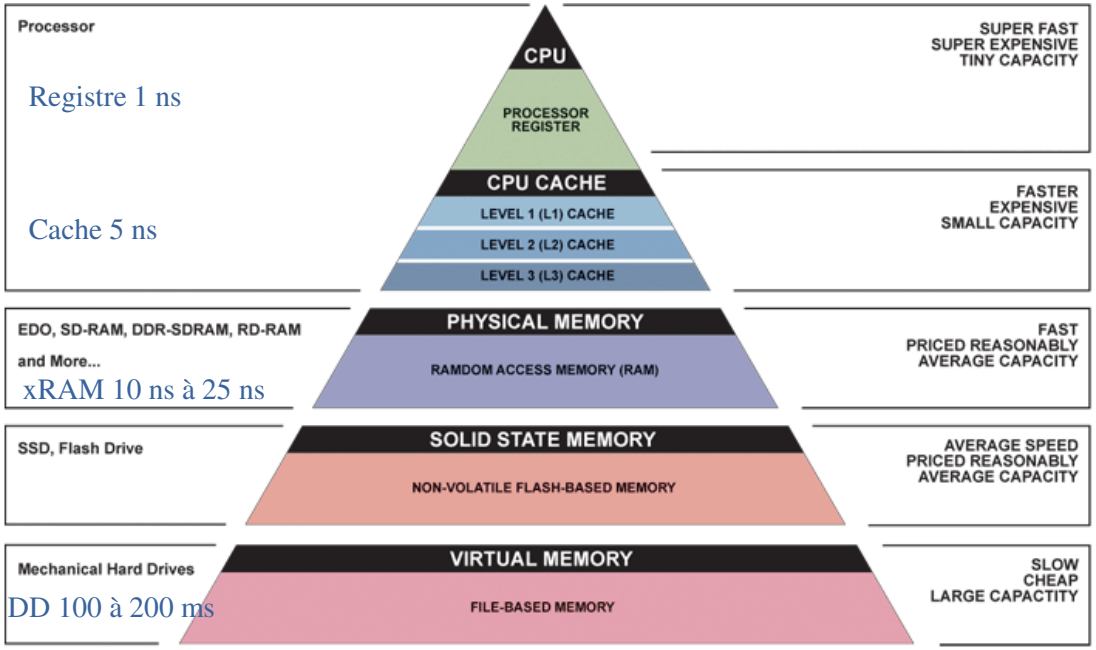
- Le processeur va chercher l'information cad données codes dans les différentes mémoires, a différents moments
- L'optimisation est faite par le gestionnaire de mémoire

Hiérarchie des mémoires

- Machine idéale
  - taille mémoire **illimitée**
  - **temps** d'accès aux données **nul**
- Caractéristiques des machines actuelles
  - **Capacité** : volume de données pouvant être stockés (bits)
  - **Temps d'accès** : intervalle entre la demande de lecture/écriture et la disponibilité de la données (secondes)
  - **Débit** : volume d'info échangé par unité de temps (bits/seconde)Coût : mémoire avec temps d'accès à 60 nano secondes
  - **Volatilité** : aptitude à garder l'info sans être alimentée électriquement.
  - Coût : mémoire les plus rapides sont les plus onéreuses

Hiérarchie des mémoires

- Organisation **hiérarchique** des mémoires
  - **Registre** : mémoire volatile, sur le sur le processeur, stocke les données, les instructions, les résultats intermédiaires…
  - **Mémoire Cache** : processeur permet l'anticipation le chargement d'info de la mémoire vive vers le processeur
  - **Mémoire Vive**: sur la carte mère, volatile, permet le stockage les programmes en cours d'exécution, leurs données
  - **Mémoire de masse** : non volatile, stockage permanent des infos



Hiérarchie des mémoires

- Adresse mémoire **physique** : une case dans la mémoire centrale.
- Adresse mémoire **logique** : adresse utilisée par un programme et calculée lors de la compilation

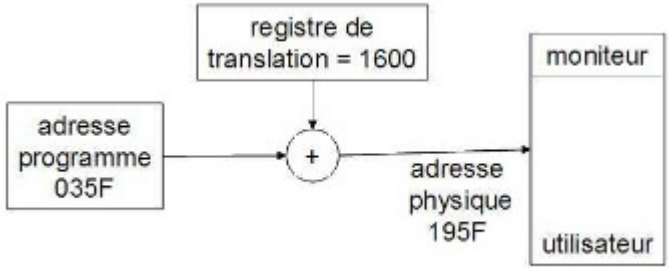
  Nécessite de faire correspondre @logique et @physique.

- **Rôle du MMU** (Memory Management Unit)
  - Allouer de la mémoire aux processus ;
  - De connaître les zones mémoire libres ou occupées ;
  - Récupérer de la mémoire en fin d'exécution ;
  - Traiter le va-et-vient (swap) entre le disque et la mémoire centrale

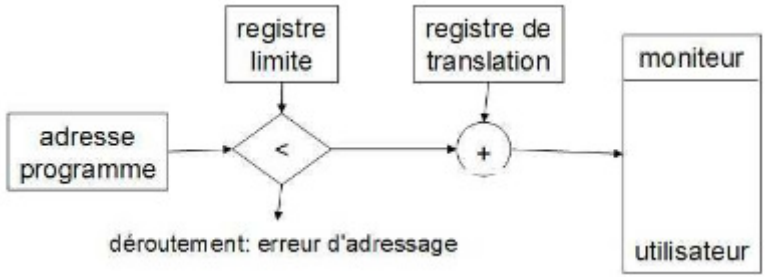
Rôle MMU : relogement du code

- Code relogeable
  - Compilateur détermine les adresses mémoires instructions et données indépendamment de l'implantation du programme en mémoire lors de son exécution.

⇒**registre de translation** contient l'adresse de départ d'implantation du code



- Protection des espaces mémoires
  - Un processus ne doit pas accéder à la zone d'un autre processus
  - Vérification de non-dépassement à l'aide du **registre de limite**

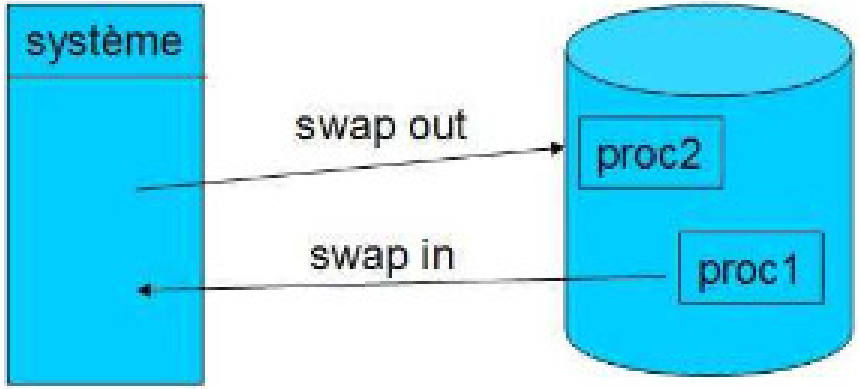


Rôle MMU : gestion du swap

- Parfois nécessaire de décharger temporairement un processus ;
- Il faut sauver l'état du processus ! => swap.
- Recopier (**swap out**) sur une "mémoire de réserve"

(un espace sur le disque) les processus non actifs ;

- Ramener (**swap in**) en mémoire centrale
- les processus réactivés



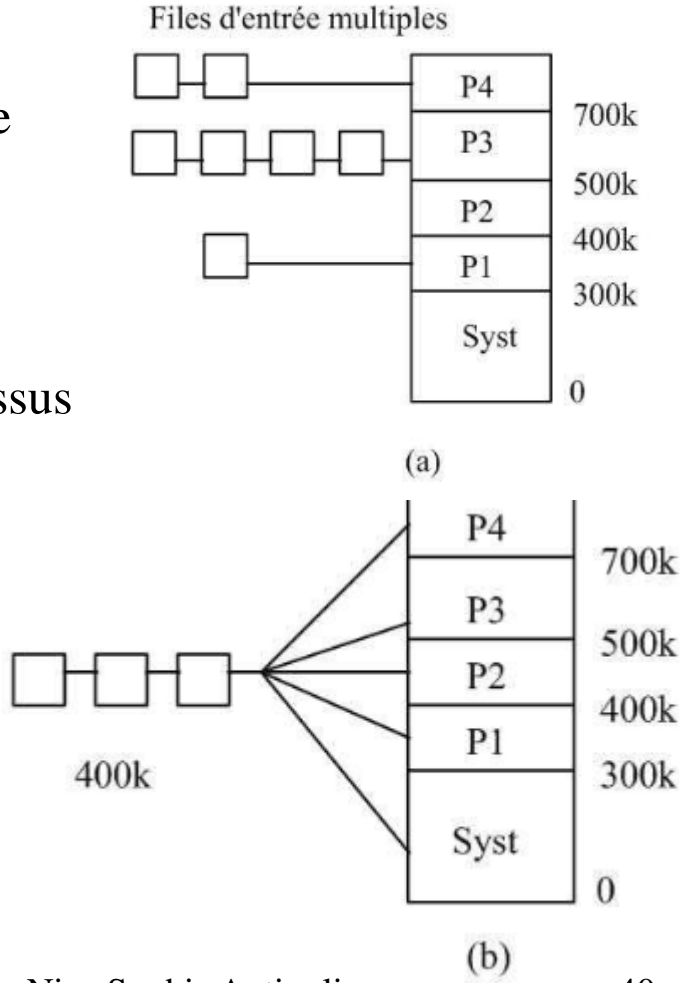
Rôle MMU : Allocation d'une zone mémoire contiguë de taille fixe

Principe : Division de la mémoire en N partitions (pas forcément de tailles égales)

a) Une nouvelle tâche est placée dans la file d'attente de la plus petite partition pouvant la contenir
   - Espace inutilisé perdu !
   - une file d'attente peut être pleine alors qu'une autre est vide.

b) Dès qu'une partition est libre, on y place le 1er processus de la file d'attente pouvant y tenir ;

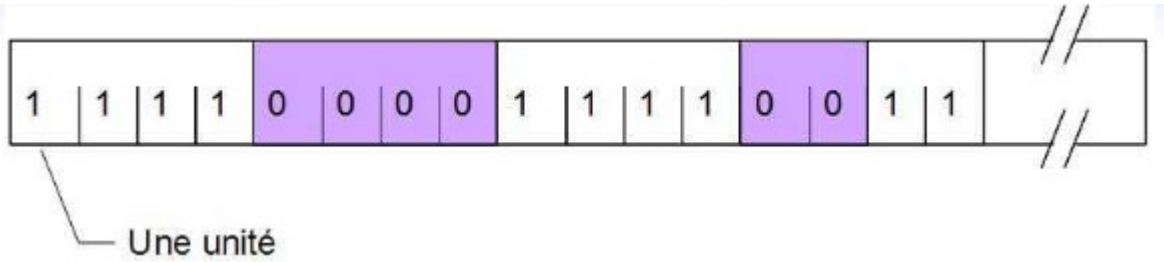
   Optimisations possibles pour (b) :
   - Parcours de la file d'attente pour trouver la plus grande tâche pouvant y tenir (pénalise les petites tâches) ;
   - Interdire la non-sélection d'une tâche prête plus de k fois.



Rôle MMU : Allocation de partitions de taille variables

Principe : Nombre et taille des partitions varient au cours du temps

- Plus adapté aux systèmes d'exploitation modernes
- Nécessite de connaître la taille mémoire requise par un processus (pb de l'allocation dynamique)
- Mémoire divisée en unités d'allocation de taille identique (k)
- Un processus de taille T nécessite T/k unités contiguës
- Gestion de l'espace dans une bitmap => une unité occupée = un bit à 1 dans la table



Rôle MMU : Allocation fragmentée
Pagination et segmentation

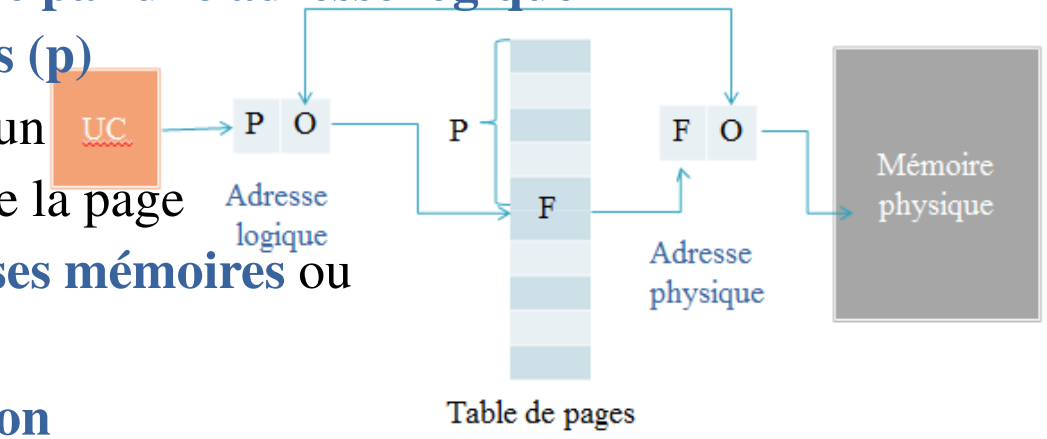
- **Principe :** Difficile de trouver une zone mémoire contiguë pour programme + données + pile
  - Diviser un programme en morceaux ;
  - Permettre l'allocation séparée de chaque morceau : **Allocation fragmentée**
- Les morceaux plus faciles à loger en mémoire (réduction des trous) ;
- On ne garde en mémoire que les parties de programme utilisées. Le reste est stocké sur disque (swap) ;
- Deux techniques de base :
  - **Pagination** : découpage arbitraire d'un programme en plusieurs pages de même taille ;
  - **Segmentation** : découpage d'un programme en fonction de sa structure en plusieurs segments de tailles différentes.

Rôle MMU : Pagination et segmentation

- **Principe** : Difficile de trouver une zone mémoire contiguë pour programme + données + pile
  - Diviser un programme en morceaux ;
  - Permettre l'allocation séparée de chaque morceau.
- Les morceaux plus faciles à loger en mémoire (réduction des trous) ;
- On ne garde en mémoire que les parties de programme utilisées. Le reste est stocké sur disque (swap) ;
- Deux techniques de base :
  - **Pagination** : découpage arbitraire d'un programme en plusieurs pages de même taille ;
  - **Segmentation** : découpage d'un programme en fonction de sa structure en plusieurs segments de tailles différentes.

Rôle MMU : Pagination

- **L'UC accède à une zone mémoire par une adresse logique**
- **Mémoire logique** divisé en **pages (p)**
- la zone mémoire est désignée par un offset (noté O) par rapport au début de la page
- **Mémoire physique** divisée en **cases mémoires** ou page frames (notées F)
- Pages et cases ont **même dimension**
- **Tables des pages** : correspondance entre pages et cases
  - Problème de l'accès rapide à la TDP
  - Améliorer cet accès



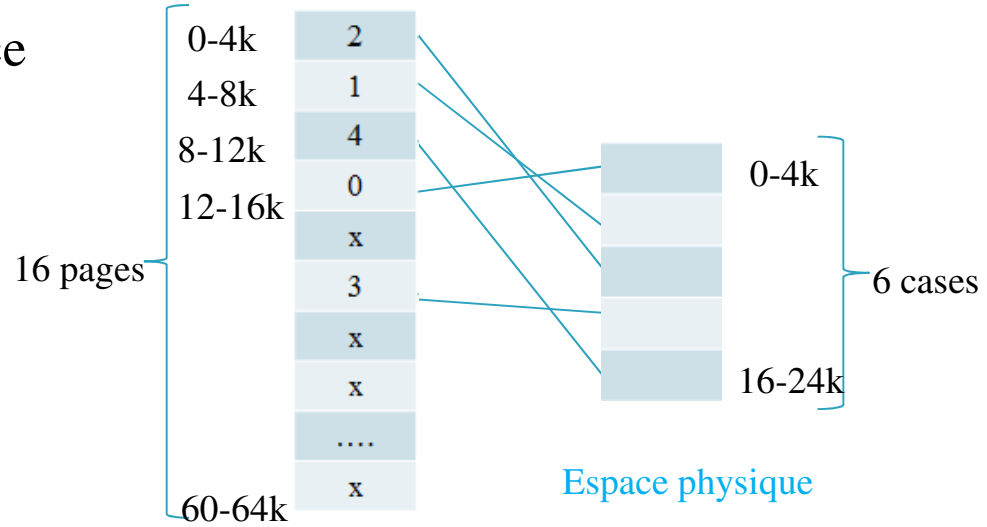
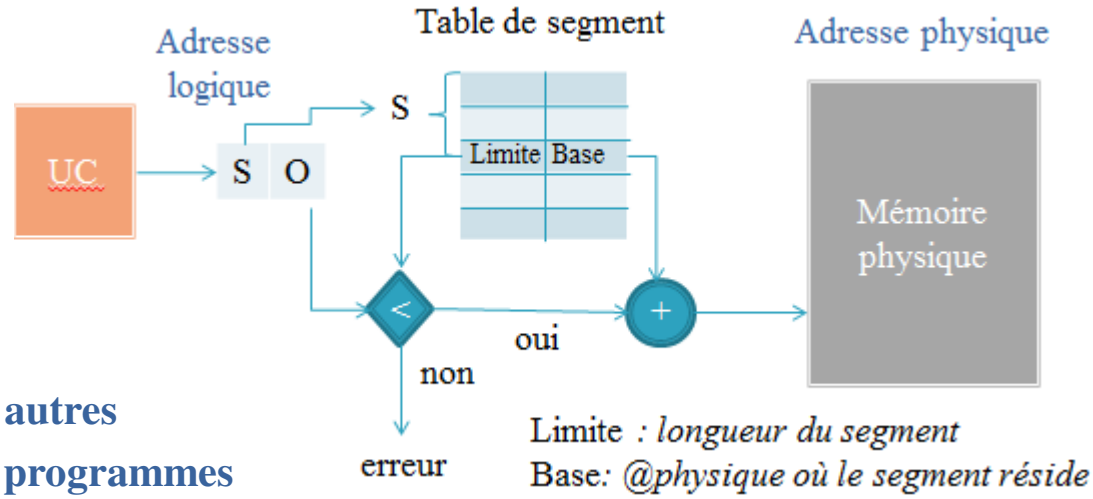


Table des pages
Espace logique

Rôle MMU : Segmentation

- Plusieurs espaces d'adresse indépendants appelés **segments**
- **Compilateur utilisera des segment différents pour**
  - **Les symboles et variables**
  - **Les constantes**
  - **Le programme**
  - **La pile**
  - Caractéristique d'un segment
    - **manipule des adresses de 0 à …**
    - **peut croître indépendamment des autres**
    - **peut être partagés entre plusieurs programmes**



- **Une adresse est spécifiée par un numéro de segment et une adresse dans le segment**

Modes d'Adressage

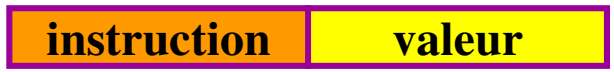
- Les opérandes peuvent être dans :
  - des registres
  - une zone mémoire: on utilise alors l'adresse de la variable qui peut être une combinaison de 3 éléments
    - la base
    - l'Index
    - le Déplacement
- Cela donne des adressages

Implicite par registre	*INC AX*
Immédiat	*MOV AL, 10*
Direct	*MOV AX [130h]*
Basé	*MOV BX offset VAR* *MOV AX[BX+4]*
Indexé	*MOV SI, 2* *MOV AX, T[SI]*

Modes d'Adressage

- Adressage **immédiat**
  - Le champs d'adresse contient la valeur de l'opérande ou son adresse physique,
  - Mode utile pour manier les constantes,
  - **Exemple :**

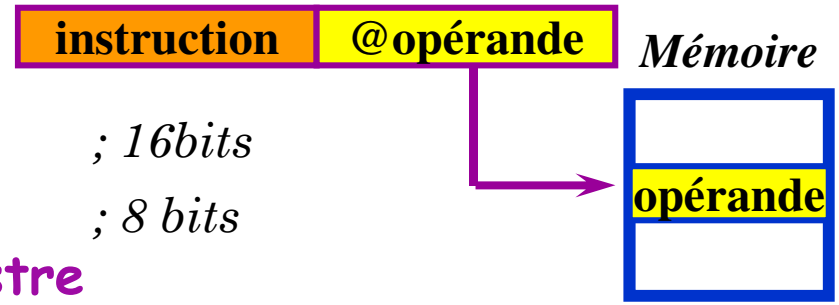
```
CMP     AX, 12H
```



Modes d'Adressage

- Adressage **direct**
  - l'adresse est l'adresse effective ou l'offset de la variable par rapport au segment de base. L'offset peut être dans
    - un registre, une variable
    - **Exemple :**

```
MOV     AX, TAUX     ; 16bits
ADD     AL, VAL      ; 8 bits
```

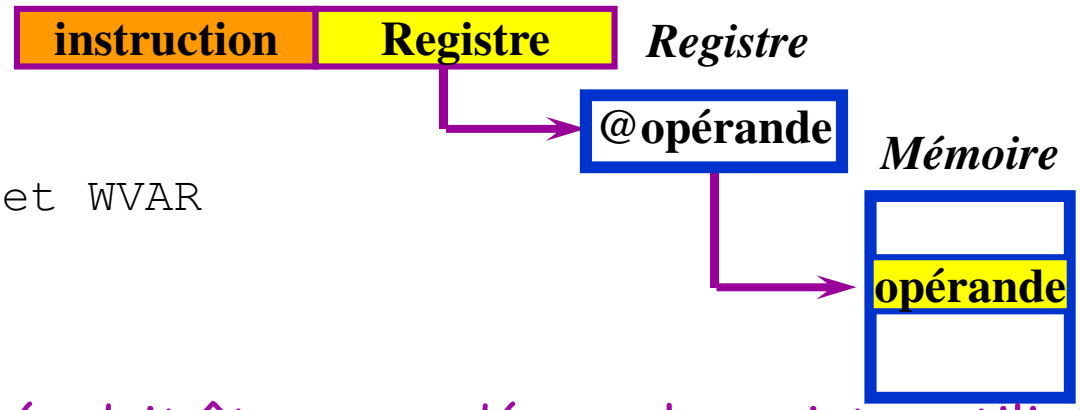


- Adressage **indirect ou par registre**

  -L'offset de la variable est contenu dans un registre de base ou d'index.

  - **Exemple :**

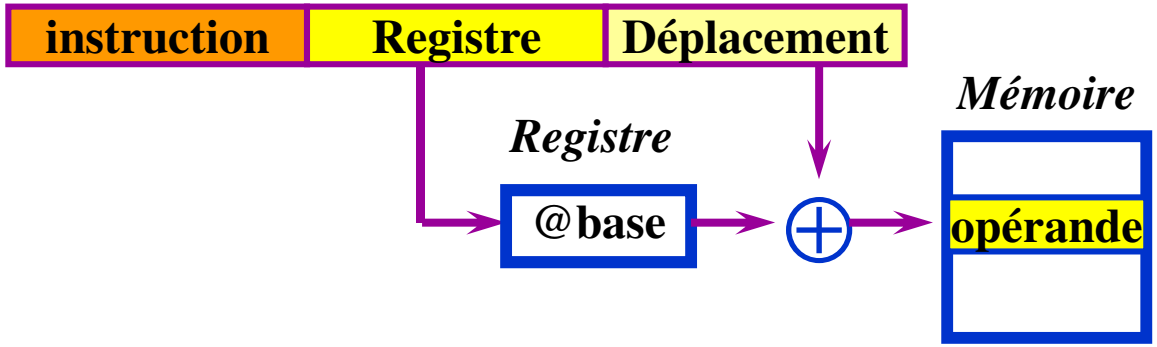
```
MOV BX, offset WVAR
MOV AX,[BX]
```



- NB : Le type de la donnée doit être accordé avec le registre utilisé

Modes d'Adressage

- Adressage **basé**
  - *utiliser pour accéder à des structures de données*
  - Similaire à l'adressage indirect par registre sauf qu'un déplacement est ajouté à la base
    adresse effective = adresse base + déplacement

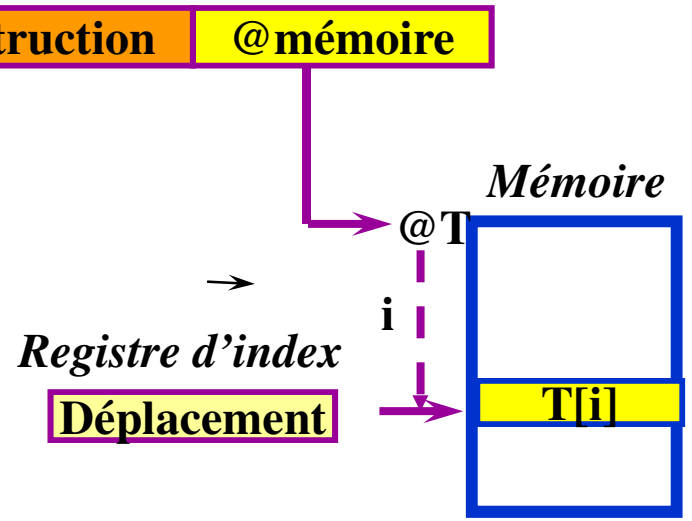


  - **Exemple** : MOV BX, OFFSET DVAR
    MOV AX, [EBX+4]

Modes d'Adressage

- Adressage **indexé**
  - *utiliser pour accéder à des tableaux*
  - On utilise un registre d'index SI ou DI plus un déplacement
  - L'adressage est noté par des crochets

  - **Exemple :** MOV SI,0
    MOV AX, T[SI]



Mode d'adressage

- Adressage **Basé indexé**
  - utilise le contenu d'un registre de base, le contenu d'un registre d'index et un déplacement optionnel.
  - Combinaison des deux modes précédents tableau de structures ou structures avec tableaux.

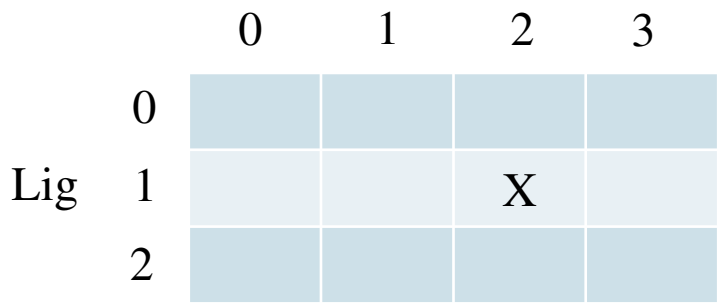
  - **Exemple :**

```
XOR AX,AX
MOV BX, OFFSET DVAR0
MOV ESI, 10
ADD AX, [BX+ESI+4]
```

Exemple d'Adressage

- Soit une matrice 4 colonnes X 3 lignes contenant des octets
- On veut accéder à une case de cette matrice

```
[case] = Offset MAT + (NBCOL * Lig +Col)
```



- `[case]=Offset MAT+4*1+2=Offset MAT + 6`
- On utilise un adressage basé indexé pour accéder aux données de cette matrice si les éléments sont sur deux octets.

0

Exemple d'adressage

```
; Déclaration des variables
        NBCOL       EQU     4
        NBLIG       EQU     3
        Lig         DW      ?               ; ligne à atteindre
        Col         DW      ?               ; colonne à atteindre
        MAT         DW      NBLIG*NBCOL     DUP(?) ; Déclaration de la matrice
; Accès à la case (Lig, Col)
        MOV         BX, Offset MAT
        MOV         AL, Lig
        MOV         CL, NBCOL
        MUL         CL                      ; AX <- Lig*NBCOL
        MOV         SI, AX
        MOV         AX, Col
        ADD         SI, AX                  ; SI <- (Lig*NBCOL +Col)
        SHL         SI, 1                   ; 2 octets par case  => multi. par 2
        MOV         AX, [BX][SI]    ; Contenu de la case
```