I. Le Matériel

a. Les Bus

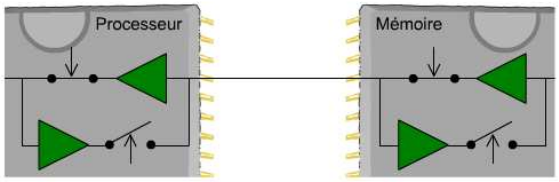
Logique 3 états : 1, 0, HZ (on le représente par un "0,5")

Porte 3 états :

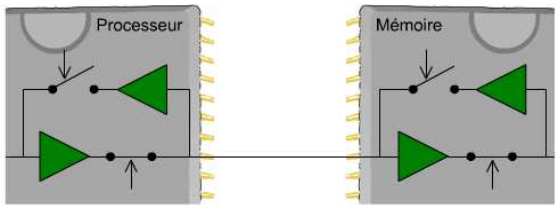
Entrée	$\overline{\text{Commande}}$	sortie
0	0	0
1	0	1
X	1	Z

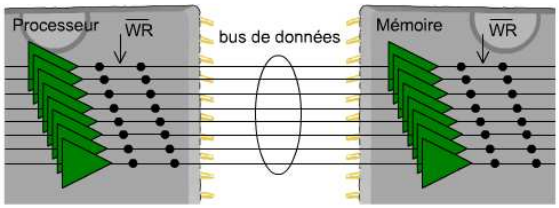
Porte bidirectionnelle : permet de gérer écriture, lecture et isolation

Lecture :



Ecriture :



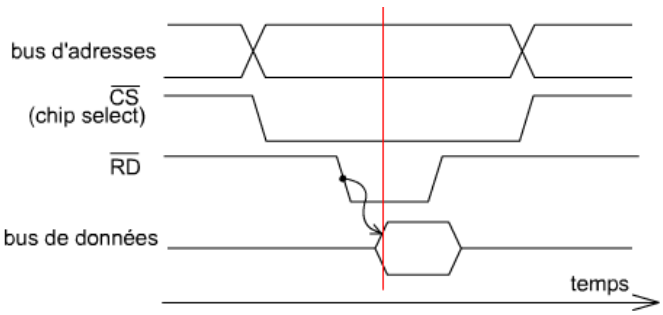


Exemple d'un bus de données en écriture

Les cycles de lecture/écriture

Le cycle de lecture

L'adresse véhiculée sur le bus d'adresses provient du processeur. Elle définit la valeur du /CS. C'est le processeur qui contrôle également le signal /RD. La donnée véhiculée sur le bus de données provient de la mémoire.

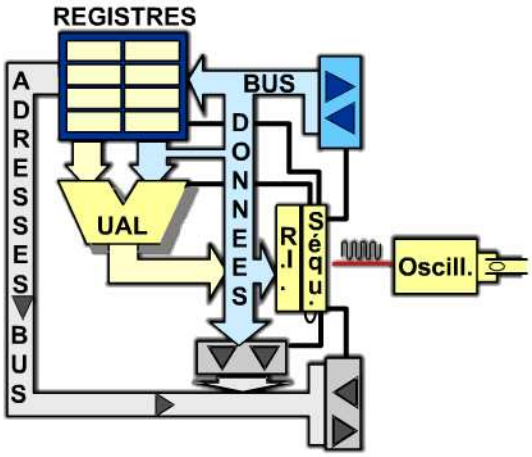


Le /CS est activé via le bus d'adresse. Dès lors, ***si /RD est activé, la donnée est envoyée*** *sur le bus de données.*

Le cycle d'écriture est quasi-identique.

La principale différence est que la donnée sur le bus de données provient du processeur. Elle peut donc apparaître (sur le chronogramme) avant l'impulsion /WR.

b. L’architecture du microprocesseur



L’UAL (Unité Arithmétique et Logique) **réalise les opérations** arithmétiques ou logiques indiquées par le séquenceur.

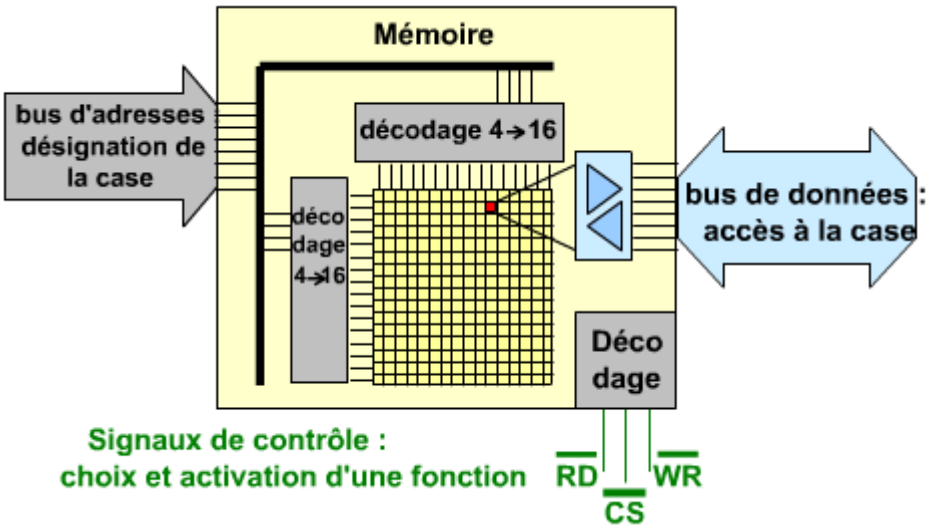
Les opérandes proviennent des **registres** du microprocesseur (Petites mémoires, mémorisation d’un seul mot à la fois) ou de la mémoire externe (via le bus de données).

Le bus d’adresses interne véhicule les adresse vers le bus d’adresse externe. Les adresse proviennent d’un registre d’adresses ou du code d’instruction présent dans le RI.

Le bus de données interne prolonge le bus de données externe assurant ainsi les échanges entre la mémoire et les différentes unités du processeur (Registre, UAL, RI -Registre Instruction-)

Le RI mémorise l’instruction lue depuis la mémoire grâce à l’horloge. **Le séquenceur la décode et organise dans le temps son exécution** en activant successivement des différentes unités du processeur. Il détermine la taille totale de l’instruction à partir du 1er octet (le code opérateur).

c. L’architecture de la mémoire



La mémoire est indispensable au microprocesseur : elle contient le programme (mémoires RAM ou ROM) et les variables (mémoire RAM).

Le bus d’adresses interne achemine les adresses auprès du décodeur lignes et colonnes. Il se sépare donc en 2.

Les décodeurs sélectionnent la ligne ou la colonne dont le numéro est codé sur la partie ligne ou colonne du bus d’adresses.

Dans **la matrice de stockage**, la case à l’intersection de la ligne et de la colonne sélectionnées par les décodeurs est mise en relation avec le buffer de données.

Le buffer de données met en relation le bus de données externe et la case mémoire sélectionnée par l’adresse. Il est orienté dans un sens ou dans l’autre par la logique de décodage interne. Quand la mémoire est inactive, il l’isole du bus de données.

La logique de décodage interne contrôle le buffer de données en fonction de la valeur des signaux de contrôle /RD et /WR et du /CS.

d. La fonction décodage

La fonction décodage consiste à sélectionner un boiter mémoire parmi plusieurs. Les boîtiers mémoire d'un système sont tous câblé(e)s de la même façon à l'exception d'une broche spécifique : le chip select. La fonction de décodage active tel ou tel "chip select" selon la valeur de l'adresse présente sur le bus d'adresses. Elle détermine ainsi la **correspondance entre une plage d'adresses et une ressource**.

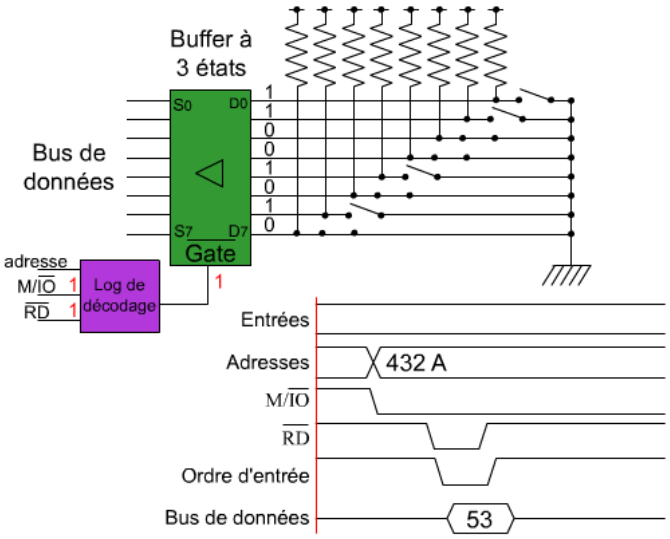
e. Les entrées-sorties

Les instructions

Activation des signaux selon l'instruction exécutée		
	M/$\overline{IO}$ = 0	M/$\overline{IO}$ = 1
$\overline{RD}$	`IN AL,adrIO`	Tout accès en lecture à la mémoire
$\overline{WR}$	`OUT adrIO,AL`	Tout accès en écriture à la mémoire

Le microprocesseur fournit un signal (M/IO#) qui indique à son environnement si l'adresse qu'il génère concerne la mémoire ou au contraire est une adresse IO, c'est-à-dire une adresse d'un registre d'entrée ou de sortie.

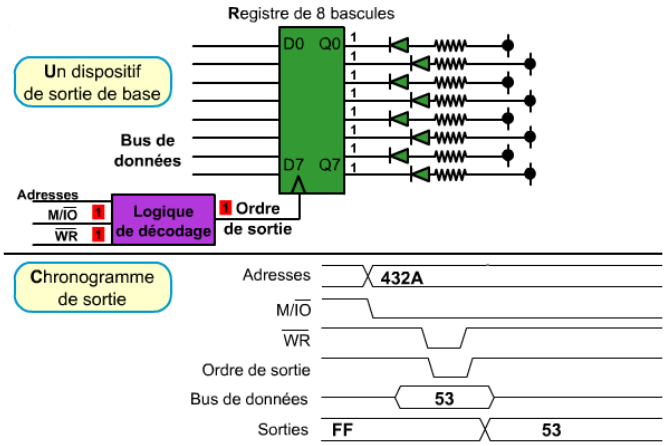
Dispositif d'entrée élémentaire



Les dispositifs d'entrée-sortie se présentent généralement sous forme de circuits intégrés appelés contrôleurs d'entrées-sorties. Ces dispositifs contiennent des registres d'entrée-sortie. Ils ne faut pas les confondre avec les registres du processeur.

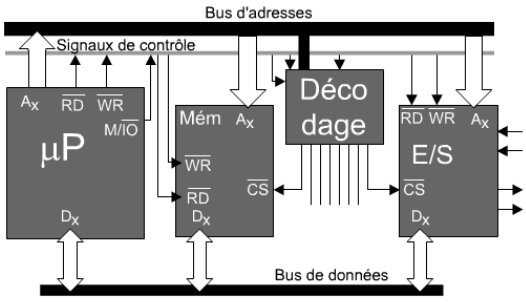
Ce dispositif élémentaire est un port parallèle très simple figé en entrée. Il donne au microprocesseur un accès à la combinaison binaire codée sur la série d'interrupteurs. Un interrupteur fermé code un 0 (masse), un interrupteur ouvert code un 1 logique (Vcc).

Dispositif de sortie élémentaire

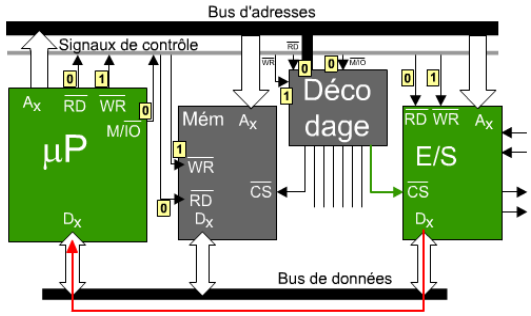


Ce dispositif élémentaire est un port parallèle très simple figé en sortie. Il permet au microprocesseur d'allumer des voyants à led selon une combinaison binaire de son choix. Un 0 (masse) sur une sortie Q du registre provoquera une différence de potentiel par rapport à Vcc, allumant ainsi le voyant. Inversement, un 1 sur une sortie Q éteint le voyant.

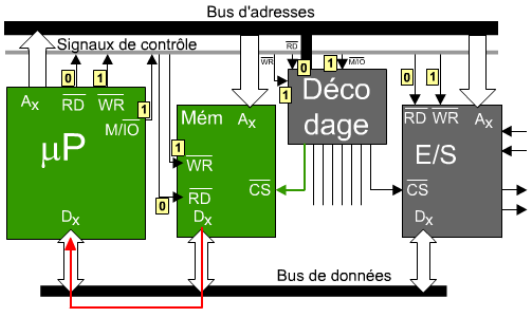
f. Synthèse



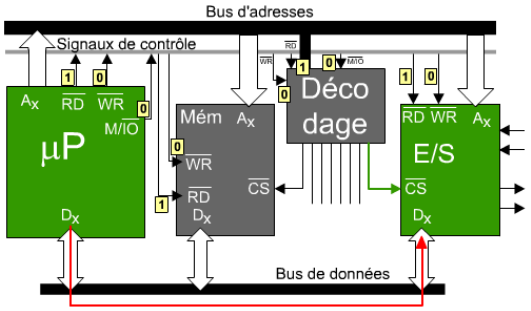
M/IO#	RD#	WR#	Instruction exécutée	Sens de fonct.du bus de données	Nom de l'opération
X	0	0	IMPOSSIBLE		
0	0	1	IN	IO→µP	entrée
1	0	1	fetch / MOV	mem→µP	lecture
0	1	0	OUT	µP→IO	sortie
1	1	0	MOVE	µP→mem	écriture
X	1	1	travail interne	Hi-Z	repos (idle)



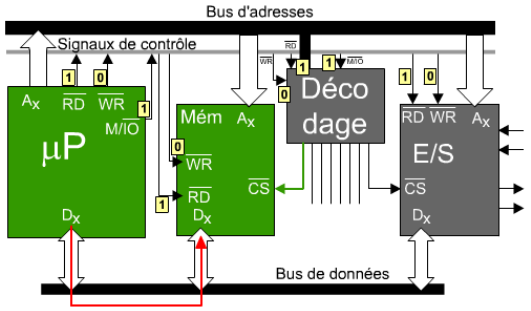
M/IO#	RD#	WR#	Instruction exécutée	Sens de fonct.du bus de données	Nom de l'opération
X	0	0	IMPOSSIBLE		
0	0	1	**IN**	IO→µP	**entrée**
1	0	1	fetch / MOV	mem→µP	lecture
0	1	0	OUT	µP→IO	sortie
1	1	0	MOVE	µP→mem	écriture
X	1	1	travail interne	Hi-Z	repos (idle)



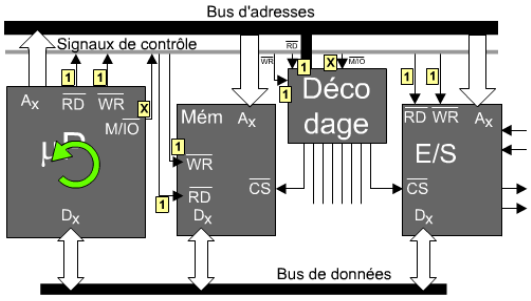
M/IO#	RD#	WR#	Instruction exécutée	Sens de fonct.du bus de données	Nom de l'opération
X	0	0	IMPOSSIBLE		
0	0	1	IN	IO→µP	entrée
1	**0**	**1**	**fetch / MOV**	**mem→µP**	**lecture**
0	1	0	OUT	µP→IO	sortie
1	1	0	MOVE	µP→mem	écriture
X	1	1	travail interne	Hi-Z	repos (idle)



M/IO#	RD#	WR#	Instruction exécutée	Sens de fonct.du bus de données	Nom de l'opération
X	0	0	IMPOSSIBLE		
0	0	1	IN	IO→µP	entrée
1	0	1	fetch / MOV	mem→µP	lecture
0	**1**	**0**	**OUT**	**µP→IO**	**sortie**
1	1	0	MOVE	µP→mem	écriture
X	1	1	travail interne	Hi-Z	repos (idle)



M/IO#	RD#	WR#	Instruction exécutée	Sens de fonct.du bus de données	Nom de l'opération
X	0	0	IMPOSSIBLE		
0	0	1	IN	IO→µP	entrée
1	0	1	fetch / MOV	mem→µP	lecture
0	1	0	OUT	µP→IO	sortie
1	**1**	**0**	**MOVE**	**µP→mem**	**écriture**
X	1	1	travail interne	Hi-Z	repos (idle)



M/IO#	RD#	WR#	Instruction exécutée	Sens de fonct.du bus de données	Nom de l'opération
X	0	0	IMPOSSIBLE		
0	0	1	IN	IO→µP	entrée
1	0	1	fetch / MOV	mem→µP	lecture
0	1	0	OUT	µP→IO	sortie
1	1	0	MOVE	µP→mem	écriture
X	**1**	**1**	**travail interne**	**Hi-Z**	**repos (idle)**

II. Le Logiciel : Modèle de programmation

a. Organisation de la mémoire

Terminologie : **octet :** 8 bits,
mot : 16 bits,
double mot : 32 bits.

Un mot = 2 octets consécutifs situés dans la mémoire à deux adresses consécutives. Deux mots consécutifs sont donc distants de 2 dans l'adressage de la mémoire

Stockage little-endian : l'octet de poids le plus faible correspond à l'adresse la plus faible n.

Ex :

Représentation par octet :

- *@n : 5F*
- *@n+1 : A3*
- *@n+2 : 3C*
- *@n+3 : 24*

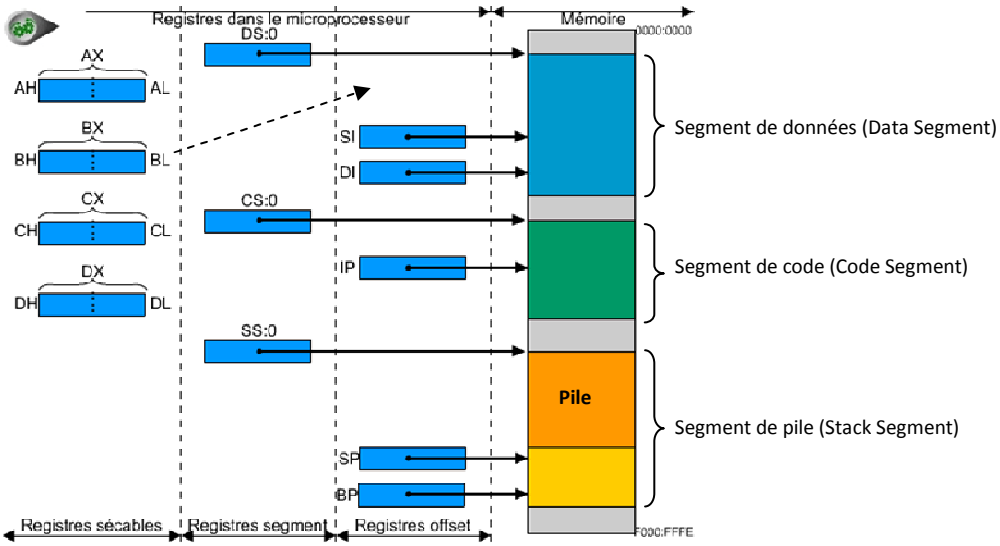
Représentation par mot :

- *@n : A35F*
- *@n+2 : 243C*

Représentation par double-mot :

- *@n : 243CA35F*

b. Les registres du x86



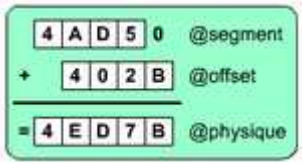
- AX-DX sont des registres de 16bits également accessibles par demi-registres de 8bits (H=High et L=Low).
- BX à un rôle double et peut servir de registre d'adresse.
- SI et DI sont des registres d'adresse. On peut les utiliser pour désigner des opérandes situés en mémoire. BX peut également servir à adresser le segment de données (comme SI et DI).
- **DS : 0 (Data Segment)** pointe le début du segment de données ou sont stockées les **variables globales** d'un programme.
- **CS : 0 (Code Segment)** pointe le début du segment de code contenant les **instructions. IP pointe le début de la prochaine instruction à exécuter.** On l'appelle aussi PC (Program Computer).
- **SS : 0 (Stack Segment)** pointe le début du **segment contenant la pile.** La pile est utilisée pour les paramètres et les variables locales des sous-programmes.
  Stratégie LIFO (Last In First Out) : on entre les infos par le sommet de la pile, et on les ressort par le sommet également.
  SP (Stack Pointer) pointe le sommet de la pile dans le segment de pile. La pile grandit dans le sens des adresses faibles. BP (Base Pointer) autorise un accès en tout point de la pile. Il sert à adresse les paramètres et les variables locales des sous-programmes.

c. Modèle de segmentation du x86

La segmentation est une façon que la microprocesseur a d'adresser et de voir la mémoire. Cela lui permet de manipuler 20 bits d'adressage (limite du microprocesseur) malgré une limite de 16 bits (limite des registres).

La segmentation fait appel à des registres de segment contenant la composante continue de l'information d'adresse et a des registres d'adresses offsets qui contiennent la partie variable. Les registres BX, SI, DI, SP, BP et IP sont des registres d'adresse offset.

Comment atteindre l'adresse 4ED7B ?



L'adresse offset 402B est la distance par rapport au début du segment (@segment).

On atteint l'adresse 4ED7B en faisant la somme des deux adresses, comme indiqué ci-dessus.

Notation en ligne : *4AD5:402B=4ED7B*

Différents segments :

- Segment de données : variables globales
- Segment de code : instructions
- Segment de pile : instruction push et pop (qui modifient le pointeur de pile)

d. Turbo-Debugger

Quelques commandes asm (assembleur) :

- *mov ax,1* : place la valeur 1 dans ax.
- *add ax,bx* : ajoute bx à ax.

Il est possible d'entrer une instruction asm dans un code C en le préfixant asm :

Asm {
Code assembleur
}

On peut manipuler les variables et les constantes déclarées en C.

III. Le Logiciel : Représentation des données (TP2)

a) Adressage immédiat

Si la donnée elle-même est connue lors de l'assemblage, alors on peut employer l'adressage immédiat. Dans un langage évolué, l'adressage immédiat correspond à l'utilisation de constantes.

Ex : mov al,5 ;

b) Adressage direct

On utilise l'adressage direct lorsqu'on connait l'adresse de la variable utilisée.

Ex :
//Déclaration de la variable
int variable ;
//On connait l'adresse de variable. On peut utiliser l'adressage direct. La valeur de variable sera chargé dans al.
mov al,[variable] ;

c) Adressage indirect

Si on ne connait pas l'adresse de la variable (lors d'un adressage dynamique par exemple), on utilise l'adressage indirect.

Ex :
*//AX=*p*
mov bx,p //Adressage direct. Tout comme p, bx pointe vers la valeur recherchée.
mov ax,[bx] //On charge la valeur de l'objet pointé par bx dans ax.

d) Adressage indirect avec déplacement

Dans ce cas là, on part d'une adresse, puis on se déplace vers l'objet recherché. On utilise le déplacement pour les tableaux.

Ex :
//AH=tab.champ2
lea bx,tab ; //lea permet de charger une adresse (ici le début du tableau) dans une variable (bx).
mov ah,[bx+2] ; //on charge dans ah la valeur de l'objet situé au début du tableau, avec un déplacement de 2 octets (pour arriver au champ2).

e) Adressage basé et indexé

Fait appel à deux registres d'adresse dont la somme donne l'adresse de l'opérande.

Ex :
//AH=tab[i]
lea bx, tab ;
mov si, [i];
mov ah,[bx+si];

f) Adressage basé et indexé avec déplacement

Permet d'accéder à un champ d'une case d'un tableau de structure.

Ex :
//AH=tab[i].champ3
lea bx,tab ; //Se place au début du tableau
mov si, [i] ; //Dans la case i
add si, si ;
add si, si ; //Dans cet exemple, les cases du tableau font 4 octets. On multiplie donc i par 4.
mov ah, [bx+si+3] ; //On se déplace sur le champ 3 de la case choisie.

g) Synthèse des modes d'adressage

mode d'adressage	l'instruction spécifie
immédiat	la donnée elle-même
direct	l'adresse de la donnée
indirect par registre	le registre contenant l'adresse de la donnée

h) Accès aux variables

Mov ax,[bx] //Charge la variable bx dans ax

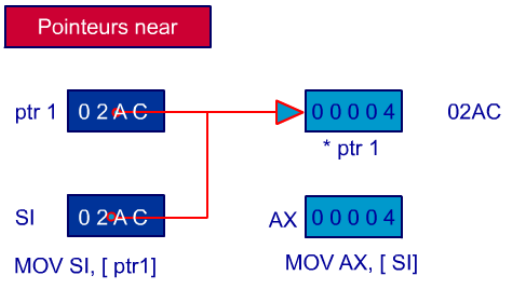
Add [si],ax //Ajoute ax à si. Ainsi si=ax+si

Lea si,[bx] //Load Effective Adress. Charge l'adresse de bx dans si

IV. Le Logiciel : Structuration des données (TD2 et TP2)

a) Les pointeurs

Pointeurs near : mémorisent la partie offset de l'adresse.

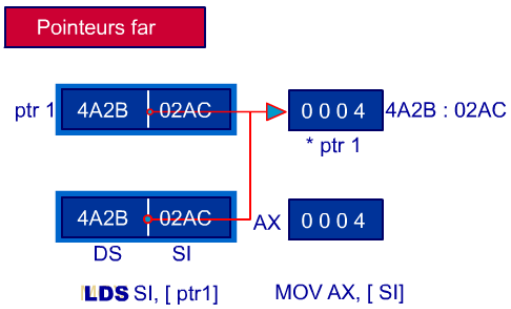


Le pointeur contient l'adresse d'un objet.
On le nomme p.
Pour récupérer la valeur de l'objet pointé, on écrit, en langage C, ***p**.
Pour récupérer l'adresse du pointeur (peu utile),on note **&p**.
En asm, pour récupérer la valeur de l'objet pointé par le pointeur, on passe par une variable intermédiaire qui va pointer également sur cet objet, SI. On charge ensuite sa valeur, [SI], dans ax.

Le pointeur est une variable, une adresse est une valeur et l'objet pointé est l'objet dont l'adresse se trouve dans le pointeur.

Attention à toujours initialiser un pointeur

Pointeurs far : mémorisent la partie segment et offset.



b) Les enregistrements

Place d'une variable :

- Char : 1 octet
- Int : 2 octets
- Float : 4 octets

Opérateur d'accès à un champ d'enregistrement : "."
Pour accéder à un champ d'enregistrement d'un pointeur, on doit mettre le pointeur entre parenthèses (priorité) : (*p).champ. On peut également le noter p->champ.

c) Les tableaux d'octets

En C, on obtient l'adresse d'un tableau en utilisant le nom du tableau, sans crochets ou en utilisant l'opérateur &.

Pour accéder à une case d'indice i, on a besoin de l'adresse du début du tableau et de la distance par rapport au début du tableau (adressage avec déplacement).

Charger l'adresse d'un tableau dans bx :

- *lea bx,[tab]* ->Adressage direct
- *mov bx,offset tab* ->Adressage immédiat

Accéder à la case i d'un tableau :

```
mov si,[i] ;
//valable pour un tableau d'octets. Pour un tableau d'entiers, on double si en faisant add si,si.
lea bx,[tab];
mov al,[si+bx];
```

Distance : d=[taille de la case en octets]*[indice]

Dans le cas d'un tableau à 2 dimensions : tab[i][j] (i lignes et j colonnes) ,
d=i*[taille de la ligne]+j*[taille de la case]

V. Le Logiciel : Structuration du code (TD3 et TP3)

a) Instructions de saut

L'instruction s'exécute normalement dans l'ordre. Le saut modifie l'ordre et fait poursuivre l'exécution à un endroit repéré par une étiquette.

Avec les sauts conditionnels, le saut est soumis à une condition exprimé par une combinaison d'indicateurs, ou flags. Si elle est vraie, on effectue ce saut.

La fonction cmp précédant le saut permet de positionner les flags. Elle permet de comparer 2 entiers.

Une instruction de base se note ainsi :

```
{
cmp ax,2 ;        //Compare ax et 2
je fin ;          //Si ax=2, on passe à l'étiquette "fin"
…                 //Instructions si ax≠2
}
fin :             //Etiquette "fin"
…                 //Instructions si ax=2
```

je	Jump if Equal
jne	Jump if Not Equal
jge	Jump if Greater or Equal
jg	Jump if Greater
jle	Jump if Lower or Equal
jl	Jump if Lower
jae	Jump if Above or Equal
ja	Jump if Above
jbe	Jump if Below or Equal
jb	Jump if Below

Un saut peut être inconditionnel : jmp (le saut s'effectue dans n'importe quelle condition)

b) SI…Alors

```
//si (conditions)
<cond (ex : cmp), indicateurs (flags)>  //positionne les flags
jxx fsi  //sors du "si" si la condition n'est pas respectée

//continue au "alors" si la condition est respectée
//…

//fin du si
} fsi:
```

```
//si a<b
cmp ax,bx ;
jge sin0

//"alors"
//…

//fin du si
} fsi :
```

c) Si…Alors…Sinon

```
//si (conditions)
<cond (ex : cmp), indicateurs (flags)>  //positionne les flags
jxx sin0  //renvoie au "sinon" si la condition n'est pas respectée

//continue au "alors" si la condition est respectée
//…
jmp fsi  //saute le "sinon"

//sinon
} sin0 :
//…

} fsi :
```

```
//si a<b
cmp ax,bx ;
jge sin0

//"alors"
//…
jmp fsi

//sinon
} sin0 :
//…

} fsi :
```

d) Switch

```
//switch (expression) {
asm {
    <évaluer expression => valeur dans un registre>   //Tout expression donne une valeur, on la place dans le registre.
//case 'valeur_possible':
    CMP  registre,'valeur_possible'                   //On compare le cas à la valeur enregistrée. Si elle est différente, on saute au cas suivant.
    JNE  dernier_cas
//
    ...                                               //Instructions si condition vraie.
//  break;
    JMP  fin_switch                                   //La dernière instruction renvoie à la fin du switch.
//case 'derniere_valeur_possible':
}
dernier_cas:  asm{                                    //cas suivant (ici, le dernier)
    CMP  registre,'derniere_valeur_possible'          //Comme précédemment, on compare le cas à la valeur. Si elle est vraie, on continue jusqu'au break. Sinon, on jump au cas suivant.
    JNE  cas_par_defaut
//
    ...
//  break;
    JMP  fin_switch                                   //Pas de condition ici puisque c'est le cas par défaut.
//default:
} cas_par_defaut: asm{
//
    ...
//}
}fin_switch: ...                                      //Fin du switch
```

e) Tant que

//tant que (cond)
tq0 : asm{ //début du tant que
<cond, indic> //évalue la condition tant que
jxx ftq0 //si la condition n'est pas respectée, on sort du tant que. Sinon on continue.
//… //instructions du tant que
//fin tant que
jmp tq0 //à la fin du tant que, on remonte au début
} ftq0 : //condition tant que terminée

f) Faire…tant que

Il suffit de placer les instructions du tant que précédent avant les conditions :

//faire
rep0 :asm{
//…
//tant que
<cond, indic>
jxx rep0 //On évalue d'abord si la condition est respectée. On remonte alors à rep0.
//… //Sinon on sort naturellement de la boucle.

g) Pour

Dans le même genre que Tant que mais avec une initialisation et une itération :

```
//pour (<init> ;<cond> :<iter>)
<init>              //initialisation (ex : initialisation du compteur)
pour0 :asm{         //début du pour
<cond, indic>       //évaluation de la condition de la boucle
jxx fpour0          //si la condition n'est pas respectée, on sort de la boucle. Sinon on continue.
//…                 //instructions du pour
//fin pour
<iter>              //instructions d'itération (ex : incrémentation du compteur)
jmp pour0           //on poursuit l'exécution vers une nouvelle évaluation
}fpour0 :           //sortie de la boucle
```

VI. Le Logiciel : Procédures et fonctions

a) Fonctionnement d'une pile

Le segment de pile est pointé par SS. La pile est située à la fin du segment de pile. SP pointe le dernier élément empilé.

Un élément empilé occupe 2 octets.

Instruction **PUSH** AX (entrée d'un élément dans la pile = **empilement**) :

- un élément empilé comporte 2 octets
- PUSH ôte donc 2 octets à SP puis copie la valeur de AX sur ce nouveau sommet de pile

Instruction **POP** BX (sortie d'un élément dans la pile, et sa récupération dans le registre = **dépilement**) :

- Le sommet de pile est copié dans BX
- Le pointeur de pile augmente de 2 (donc, la pile diminue)

NB : Dans Turbo-Debugger, la pile est représentée à l'envers : quand on empile, le pointeur de pile se déplace vers le bas.

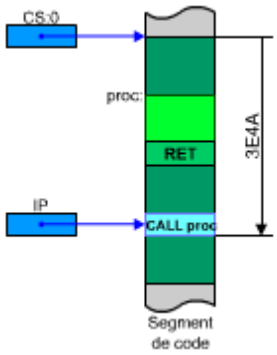
Instruction CALL : empile l'adresse de l'instruction qu'il faudra exécuter après l'appel du sous-programme. Cette adresse (adresse de retour) est normalement située dans le registre pointeur d'instructions.

L'instruction de retour de sous-programme (RET) dépile l'adresse de retour dans le registre pointeur d'instructions.

b) Appels de procédures (CALL/RET)

<u>Procédure intra-segment</u>

L'instruction CALL appelle les procédures sans paramètre, nommées proc, repérées par l'étiquette proc. Call met en jeu le segment de code mais utilise aussi la pile (plus bas).



Dans l'exemple ci-contre, 3E4A est l'adresse de l'instruction située juste après l'instruction CALL. C'est l'Instruction de retour, ou l'adresse ou l'exécution doit retourner quand la procédure proc est terminée (l'instruction suivante).

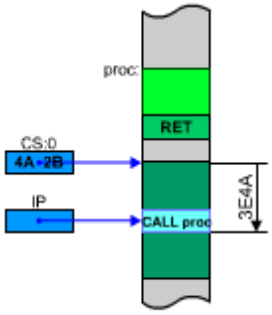
Quand les octets du code machine de call proc ont été lus, ip passe à l'instruction suivante et contient donc l'adresse de retour.

Alors, CALL commence par empiler la valeur de IP : 3E4A (rappel : empiler=entrer un élément dans la pile).

Ensuite CALL charge l'adresse offset de proc et la charge dans IP. IP remonte donc plus haut, continue l'exécution et rencontre l'instruction RET.

RET, lui, dépile dans IP. La dernière adresse entrée dans la pile étant l'adresse de retour, IP contient l'adresse de retour et poursuit l'instruction.

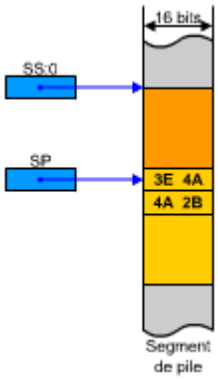
Procédure inter-segment



Un CALL inter-segment modifie CS, et donc l'emplacement du segment de code. C'est utile lorsque la procédure appelée est trop loin du point d'appel.

L'adresse de retour est composée d'une partie offset 3E4A et d'une partie segment 4A2B contenue dans CS (dessin ci-contre).

Quand les octets du code machine de CALL proc ont été lus, IP contient l'adresse offset de retour (comme indiqué plus haut).



CALL empile l'adresse de retour en commençant par la partie segment, et en terminant par la partie offset. L'adresse en bien stockée dans la pile dans l'ordre little-endian (dessin ci-contre).

CALL charge ensuite la partie segment de proc dans CS et la partie offset dans IP. Proc et CS changent alors de position.

La procédure s'exécute et rencontre l'instruction RET, également en mode inter-segment.

L'adresse de retour en dépilée dans le couple CS-IP et les deux reprennent leur place initiale.

c) Appel de fonctions

Voir partie suivante.

VII. Le Logiciel : Paramètres et variables locales

a) Programme appelant

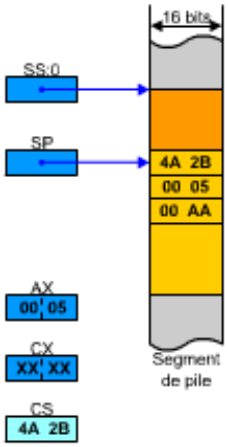
Soit un programme appelant une fonction de deux paramètres : rechercher(5,tabG).

Examinons le code assembleur correspondant à **l'appel de la fonction** recherchée.

On doit tout d'abord fournir sur la pile les arguments nécessaires à la fonction : 5 (ou 0005) et tabG (adresse : 00AA).

En assembleur, le dernier élément empilé est celui qui sera dépilé le premier. La fonction va chercher le 5 en premier, il faudra donc l'empiler en dernier.

```
mov ax,00AA          //L'adresse du tableau
push ax              //On l'empile
mov ax,0005          //La constante 5
push ax              //On l'empile
push cs              //empile la partie segment de l'adresse de retour (4A2B)
call_rechercher      //On ne détaille pas la fonction ici.
                     //Par convention, son résultat apparait dans ax.
                     //Un RET dépile l'adresse de retour dans cs.
                     //L'exécution retourne au programme appelant.
pop cx               //Les deux pop cx servent uniquement à nettoyer la pile.
pop cx
```



b) Programme appelé

- Soit un programme appelé, la fonction rechercher.

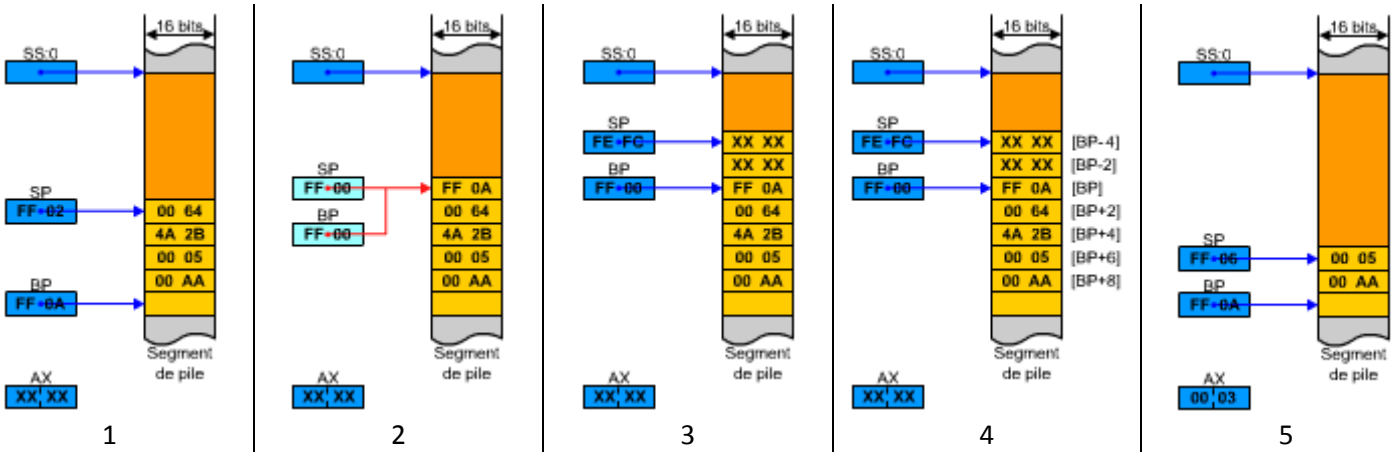
int rechercher (int e, int tab[N]) {
int i,fin;
//…
return i; }

- Examinons la façon dont elle accède à ses paramètres, e et tab.

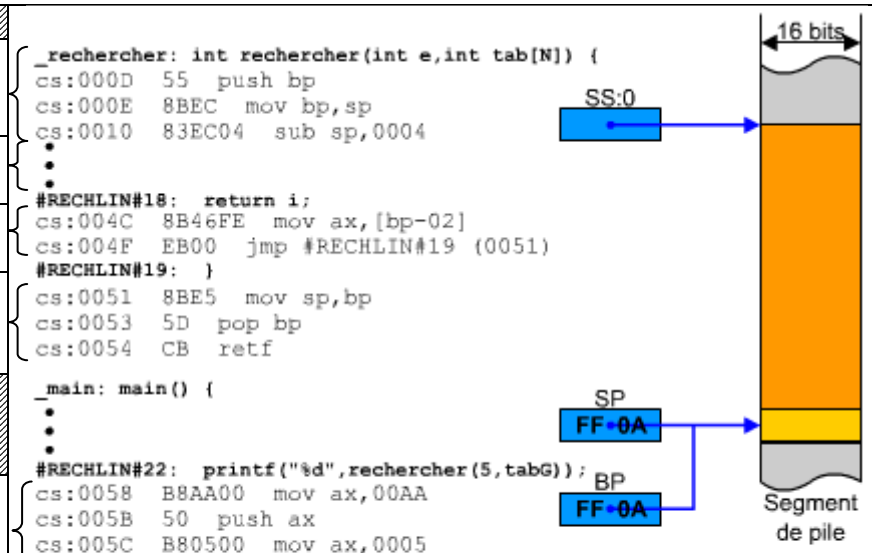
push bp //bp (base de pile) repère le contexte du programme appelant. ->1
//push bp empile sa valeur (FF0A) afin de la retrouver en fin de fonction.
mov bp,sp //fait pointer bp au même endroit que sp. ->2
sub sp,0004 //réserve 4 octets sur la pile pour le variables locales i et fin. ->3
//Le contexte est alors établi. On peut utiliser l'adressage indirect par déplacement depuis bp. ->4

- On s'intéresse maintenant au retour du résultat, et à la restitution du contexte du résultat appelant.

mov ax,[bp-2] //Le résultat est l'indice i, accessible en bp-2. On le charge dans ax.
mov sp,bp //Fait pointer sp au même endroit que bp, ce qui revient à supprimer l'espace des variables locales.
pop bp //Restitue dans bp le repère du contaxte de pile du programme appellant (FF0A).
retf //Fait revenir au programme appellant (sp pointe sur 0005). ->5



c) Chronologie



	`_rechercher: int rechercher(int e,int tab[N]) {`
	`cs:000D 55 push bp`
	`cs:000E 8BEC mov bp,sp`
	`cs:0010 83EC04 sub sp,0004`
		La fonction sauvegarde le repère du contexte de pile de l'appelant et établit son propre contexte de pile.
		L'adressage est basé sur bp :
		Accès aux paramètre : déplacement positif
		Aux variables locales : déplacement négatif.

4	• • •	L'algorithme de recherche calcule dans i (bp-2) le résultat 0003.

5	`#RECHLIN#18: return i;`
	`cs:004C 8B46FE mov ax,[bp-02]`
	`cs:004F EB00 jmp #RECHLIN#19 (0051)`
		Le résultat est placé dans ax. Le jmp est superflu ici.

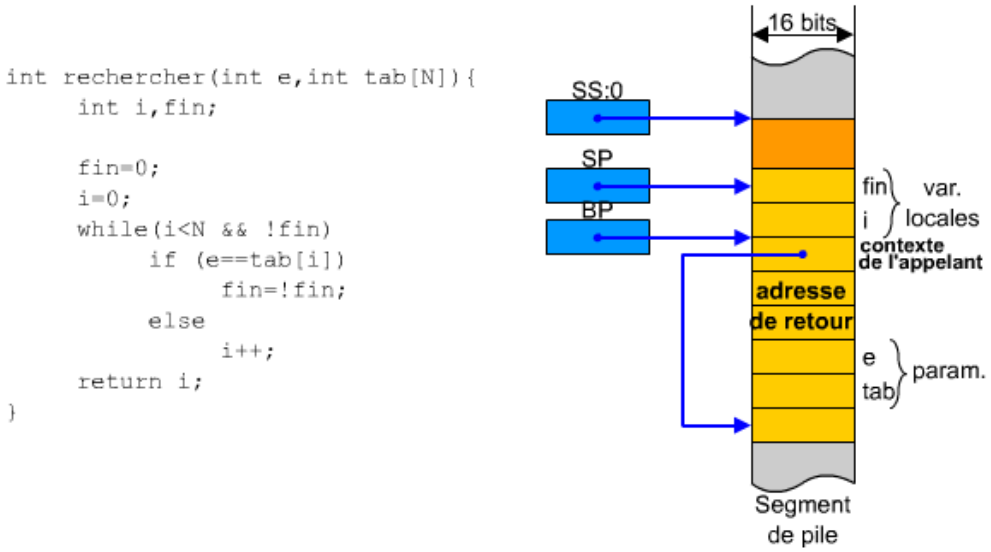
6	`#RECHLIN#19: }`
	`cs:0051 8BE5 mov sp,bp`
	`cs:0053 5D pop bp`
	`cs:0054 CB retf`
		Le contexte de rechercher est éliminé et celui de l'appelant restitué.

	`_main: main() {`
	• • •
	`#RECHLIN#22: printf("%d",rechercher(5,tabG));`	

1	`cs:0058 B8AA00 mov ax,00AA`
	`cs:005B 50 push ax`
	`cs:005C B80500 mov ax,0005`
	`cs:005F 50 push ax`
		L'appelant empile les paramètres en terminant par le premier : 00AA puis 0005.

2	`cs:0060 0E push cs` `cs:0061 E8A9FF call _rechercher`	Le call empile l'adresse de retour (4A2B 0064). La fonction continue en haut.
7	`cs:0064 59 pop cx` `cs:0065 59 pop cx`	L'appelant nettoie la pile des arguments qu'il avait empilé.
8	`printf...`	Le résultat présent dans ax peut être exploité par un printf.

d) Synthèse



Construire le contexte de pile d'une fonction :

1) Réserver de la place pour les arguments de la fonction, en commençant par le dernier
2) Réserver de l'espace pour l'adresse de retour
3) Dès qu'une fonction a un contexte (paramètre ou variable locale), on sauvegarder bp, qui repère le contexte de l'appelant.
4) Réserver de l'espace pour les variables locales (dans l'ordre où elles apparaissent).

- Connaitre la taille des paramètres ou variables utilisés
- Savoir si c'est une fonction intra ou inter-segment (pointeur near ou far)

VIII. Le matériel : gestion des entrées-sorties

IX. Les logiciels d’exploitation

X. Approfondissement du parallélisme d’exécution

XI. Performance du matériel