Corrigé du TD1

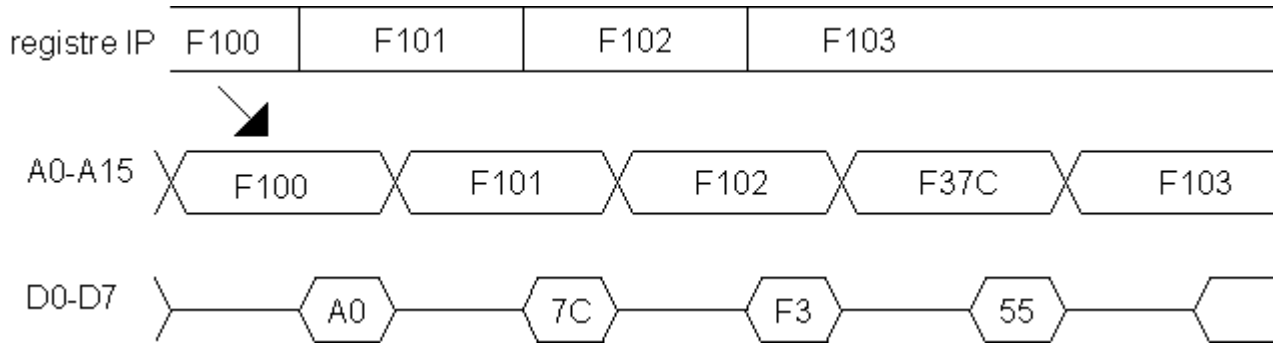
Déroulement d'une séquence d'instructions

Imaginez l'activité sur le bus d'adresses et le bus de données lorsqu'un processeur accède en lecture à une case mémoire en lecture. Pour fixer les idées, disons qu'il s'agit d'un processeur 8 bits ayant un bus d'adresses de 16 bits. L'instruction effectuée est la suivante.

```
Adresse         Code machine    Code assembleur
F100            A07CF3          mov al,[F37C]
```

On suppose également que les informations stockées en mémoire respectent l'ordre poids faible puis poids fort.

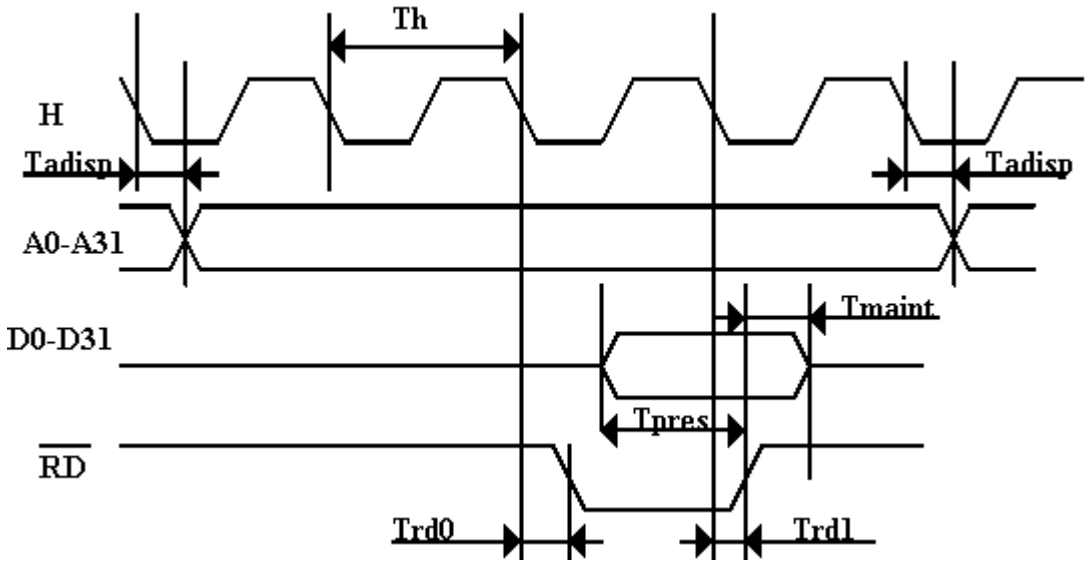
- Complétez le chronogramme suivant.



- Quelles sont les différentes informations qui circulent sur le bus de données ? ***des codes machines et les données qui font l'objet du traitement circulent sur le bus de données et sont stockées dans la mémoire***

Chronogramme de lecture

La fiche technique d'un microprocesseur comporte le chronogramme de lecture suivant. Observer :



A quoi voit-on qu'il s'agit d'un chronogramme de lecture ? ***Le signal /RD est activé.***
Combien de périodes de l'horloge H y a-t-il pour le transfert d'un mot ? ***4 périodes d'horloge H***
Quelle est la taille de ce mot ? ***4 octets car le bus de données D0-D31 occupe 32 bits.***

Le constructeur précise également que deux octets consécutifs en mémoire sont distants de 1 dans l'espace adressable.

Quelle est la taille de l'espace adressable ? ***2^{32} octets car le bus d'adresses A0-A31 comporte 32 bits et les octets sont distants de 1 dans l'espace adressable.***

Performances des processeurs

La fiche technique fournit également les durées suivantes.

Temps	Signification	Min.	Max.
Th	Période d'horloge	50 ns	
Tadisp	Adresse disponible	7 ns	10 ns
Trd0	Read à 0	8 ns	10 ns
Trd1	Read à 1	7 ns	10 ns
Tpres	Préselection	30 ns	
Tmaint	Temps de maintien	5 ns	

Questions :
Quelle est la fréquence maximale à laquelle ce processeur peut fonctionner ? ***H=1/Th soit H=10^9/ 50 Hz = 20 MHz***
Calculer en octets par seconde le débit possible sur le bus de ce microprocesseur à la fréquence maximale du microprocesseur. ***Il faut 4 périodes d'horloge pour transférer un octet mais le bus de données permet d'en transférer 4 à la fois. Le débit est donc de 20 10^6 octets par seconde.***
Sachant que le nombre moyen d'instructions exécutées par période de l'horloge H est de 0,22 calculer la puissance moyenne de ce processeur. ***0,22 instructions en 50 ns soit 0,22 * 10^9 / 50 instructions par seconde ce qui donne 4,4 Mips.***

Temps imposé à la mémoire par le microprocesseur

Des mémoires de 50 ns de temps d'accès sont reliées directement au bus de ce microprocesseur. Le processeur imposant à la mémoire le rythme des accès, au delà d'une certaine fréquence de fonctionnement, la mémoire ne répondra pas suffisamment vite.

Questions :
A quel instant les données en provenance de la mémoire doivent-elles être présentes sur le bus de données du processeur pour que la lecture s'effectue correctement ? ***à l'instant précédant de la durée Tpres (soit 30 ns) la remontée du signal /RD. Cette remontée intervient elle-même 7 à 10 ns (Trd1) après la fin de la 3ème période d'horloge. L'instant le plus tôt dans le chronogramme est donc 23 ns avant la fin de la 3ème période d'horloge.***
A quel instant la mémoire "voit-elle" démarrer le cycle de lecture ? ***quand elle "voit" changer l'adresse c'est-à-dire 7 à 10 ns (Tadisp) après le début de la 1ère période d'horloge. L'instant le plus tard dans le chronogramme est donc 10 ns après le début de la 1ère période d'horloge.***
Calculer le temps d'accès imposé par le processeur fonctionnant à sa fréquence maximale ? ***Le temps d'accès est la durée qui s'écoule entre ces deux instants. Des 3 premières périodes d'horloge (150 ns), il faut soustraire les 10 ns et 23 ns calculées précédemment. 150 ns - 33 ns = 117 ns.***
Quel temps d'accès la mémoire peut-elle tenir ? ***50 ns comme le précise l'énoncé.***

A quelle fréquence maximale peut-on faire tourner le microprocesseur pour que les mémoires puissent tenir le temps d'accès imposé par le processeur ? ***le rythme de 117 ns est donc facile à tenir pour ces mémoires qui présentent leur données 67 ns avant ce qu'exige le processeur. Le processeur peut donc fonctionner à sa fréquence maximale soit 20 MHz.***

Overhead du système d'exploitation, puissance utile

Le système d'exploitation utilise pour son propre compte une partie du temps CPU, ceci réduit d'autant la puissance disponible pour les applications. Le terme "overhead" désigne le pourcentage du temps ou de puissance consommée par le système d'exploitation. Cette charge est du reste variable au cours du temps selon le type de situation auquel il a à faire face. On parle donc d'overhead moyen ou de pointe.

Questions :
Quelle est la puissance utile minimale dont disposent les applications, si le système d'exploitation consomme en pointe 10% de la puissance CPU ? ***4,4 Mips - 10 % soit 3,96 Mips***

Entrées - sorties : ordres de grandeur

Le port série est le dispositif de communication asynchrone utilisé dans les modems. Il est chargé d'émettre sous forme série les données d'un programme et de paralléliser les données reçues sous forme série. Le processeur est ainsi libéré de cette tâche. L'horloge n'est pas émise sur un signal séparé. Le rythme de transmission doit être convenue entre l'émetteur et le récepteur et générée de chaque côté.

On parle de liaison série asynchrone. Pourquoi utilise-t-on ce terme ? la durée qui s'écoule entre deux caractères est à la discrétion de l'émetteur.
A quelles occasions y a-t-il synchronisation ? ***il y a synchronisation à chaque début d'émission de caractères (c'est le bit de start)***
Que signifie 1,5 bit d'arrêt ? Peut-on couper un bit en 2 ? ***On ne peut pas couper un bit en 2, mais on peut couper un temps bit en deux. 1,5 bit d'arrêt est donc une durée de 1,5 période bit. Pour une émission à 9600 bps, cela correspond à une durée de 1,5 / 9600 seconde, c'est-à-dire 156,25 µs.***
A quoi correspond la notion de bit d'arrêt ? ***c'est un silence minimum imposé entre la fin de l'émission d'un caractère et le début de l'émission du caractère suivant. Mais comme on est dans le cas d'une liaison asynchrone, ce silence peut être plus important.***
Quels sont les paramètres relatifs à la parité côté émetteur ? côté récepteur ? ***Côté émetteur, soit il n'émet pas de bit de parité, soit il émet une parité paire, soit il émet une parité impaire. Côté récepteur, soit il n'attend pas de parité, soit il attend une parité paire, soit il attend une parité impaire, soit il ignore le bit de parité.***
Qu'est-ce qu'une parité paire ? une parité impaire ? ***une parité est un bit émis en plus de la donnée utile. Une parité paire vaut 1 quand le nombre de 1 parmi "les bits utiles de la donnée + le bit de parité" est pair. Une parité impaire vaut 1 quand le nombre de 1 parmi "les bits utiles de la donnée + le bit de parité" est impair.***

On considère une liaison à 9600 bps, 8 bits de données utiles, 2 bits d'arrêt, parité paire.

Quel est le temps minimum qui s'écoule entre le début d'émission d'un caractère et le début d'émission du suivant ? ***dans l'ordre chronologique, on trouve 1 bit de start, 8 bits de données, 1 bit de parité et 2 bits d'arrêt soit 12 bits. La durée est de 12 / 9600 seconde soit 1,25 ms.***

On utilise un sous-programme qui prend en charge l'émission d'un tampon de mémoire de 20 octets.

Calculer le nombre d'instructions que cela représente pour un processeur 50 Mips ? ***20 octets sont émis en 25 ms au minimum (silence mimum entre les caractères). Pour un processeur exécutant 50 millions d'instructions par seconde, 25 ms représentent 50 * 0,025 millions d'instructions soit 1,25 millions***

d'instructions.

Christophe Tombelle.

Revisé: 08-10-2008.

Corrigé du TD2

On a les déclarations suivantes

```
// Constantes
#define constante 2

// Types
typedef struct {
        int champ1;
        char champ2, champ3;
} rec_t;

// Variables
int variable1, variable2, i;
rec_t  enreg, *pEnreg;
char tab_byte[9], tab_byte1[9], *pTabByte;
int tab_word[10], *pTabWord;
rec_t tab_enr[9];
```

Questions

Constante

- Quelle est la place occupée en mémoire par une constante définie par un "#define" ? ***aucune dans le segment de données***
- Où cette valeur peut-elle donc apparaître ? ***dans le code machine lui-même à chaque endroit où elle est utilisée***
- Quel mode d'adressage doit-on utiliser pour accéder à une telle constante ? ***l'adressage immédiat***

Pointeurs

Un pointeur est une variable destinée à recevoir une adresse. En langage C, un pointeur porte également en lui l'information relative au type (et donc à la taille) de l'objet pointé. Dans l'architecture segmentée du 8086 (code 16 bits) on distingue les pointeurs "near" (mémorisant la seule partie "offset" de l'adresse soit 16 bits) et les pointeurs "far" (mémorisant à la fois la partie "offset" et la partie "segment" de l'adresse soit 32 bits). Des options du compilateur ou des mots-clés spécifiques précisent de quel type de pointeur il s'agit.

- Quelle différence faites-vous entre pointeur, adresse et objet pointé ? ***un pointeur est une variable, une adresse est une valeur, l'objet pointé est l'objet dont l'adresse se trouve dans le pointeur***
- Dans les déclarations précédentes, trouvez des déclarations de pointeurs. ***`char *pTabByte; int *pTabWord;`***
- En C, quel opérateur doit-on appliquer à une variable pour obtenir son adresse ? ***l'opérateur `&`***
- En C, quel opérateur doit-on appliquer au pointeur pour obtenir l'objet pointé ? ***l'opérateur `*`***
- Comme toute variable, si un pointeur n'a pas été initialisé, son contenu n'est pas significatif. Que se passe-t-il si on applique l'opérateur `*` à un tel pointeur ? ***lors de l'exécution, le contenu du pointeur est considéré comme une adresse, un accès a lieu à "l'adresse" ainsi spécifiée. Une erreur d'exécution peut se produire. Si l'accès a lieu en écriture des parties du programme ou des données peuvent être touchées. Selon que le système d'exploitation est rigoureux ou permissif, l'erreur est plus ou moins franche. Le bug peut être détecté rapidement lors de la phase de test ; mais il peut aussi être difficile à reproduire donc à déceler. Les techniques de programmation actuelles ne permettent pas de produire des logiciels exempts d'erreur.***
- Pour signifier qu'un pointeur ne pointe sur aucun objet pointé, il est de coutume de lui affecter la valeur d'adresse 0 notée NULL. Que se passe-t-il si on applique l'opérateur * à un tel pointeur ? ***lors***

de l'exécution, un accès a lieu à l'adresse 0. Les remarques précédentes restent valables.

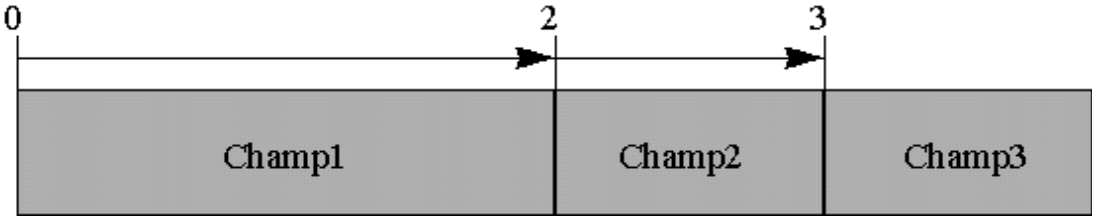
Enregistrement

On considère la représentation suivante. La surface des champs est représentative de l'espace qu'ils occupent en mémoire. Un type `char` est supposé occuper **un octet** en mémoire. Comme ceci dépend du système et du compilateur utilisés, les compilateurs C fournissent l'opérateur `sizeof` qui donne la taille en octets **d'une variable ou d'un type**. Sur la plate-forme utilisée pour les travaux pratique, `sizeof( char )` a la valeur 1.

- Trouvez parmi les déclarations précédentes le nom du type qui correspond à ce schéma. ***`rec_t`***
- En C, quel est l'opérateur d'accès à un champ d'un enregistrement ? ***l'opérateur `.` (point ou dot en anglais)***
- Quelle place un `int` occupe-t-il en mémoire ? ***2 octets pour la plate-forme utilisée en travaux pratiques***
- Quelle place une variable de ce type occupe-t-elle en mémoire ? ***4 octets ...***
- Parmi les déclarations précédentes, trouvez une variable de ce type. ***enreg***

Les champs 1, 2 et 3 sont stockés en mémoire dans l'ordre de la déclaration. Le champ1 est situé au début de l'enregistrement.

- Calculer la "distance" en octets entre le début de l'enregistrement et le champ2. ***deux octets***
- Même question pour le champ3. ***trois octets***
- Annotez le dessin ci-dessus avec les "distances" en octet, relatives au début d'enregistrement.

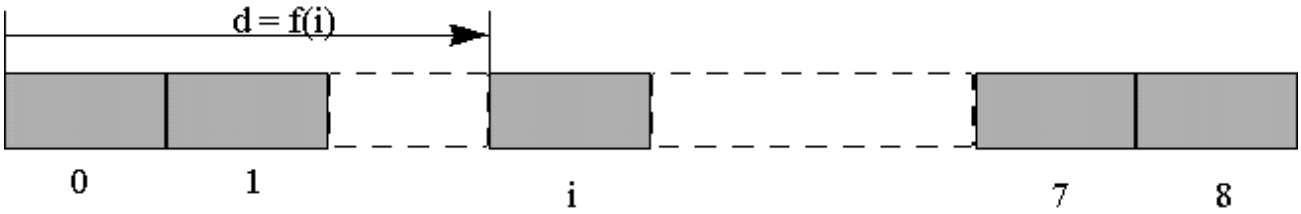


Accès via un pointeur

- Quel opérateur doit-on appliquer à un pointeur d'enregistrement pour accéder à l'un des champs ? ***l'opérateur ->***
- Recherchez une notation équivalente mettant en oeuvre deux opérateurs ? ***`(*p).champ2`***
- Pourquoi les parenthèses sont-elles obligatoires ? ***parce que l'opérateur `.` est normalement prioritaire sur l'opérateur `*`***
- Quel est l'intérêt de la notation avec un seul opérateur ? ***Elle est plus concise en évitant l'emploi des parenthèses***
- Que se passe-t-il si on applique l'opérateur -> à un pointeur non initialisé ? ***les mêmes risques que précédemment sont encourus***

Tableau d'octets

En langage C, la première "case" d'un tableau est la case d'indice 0. Un tableau de 9 octets est donc indicé de 0 à 8.

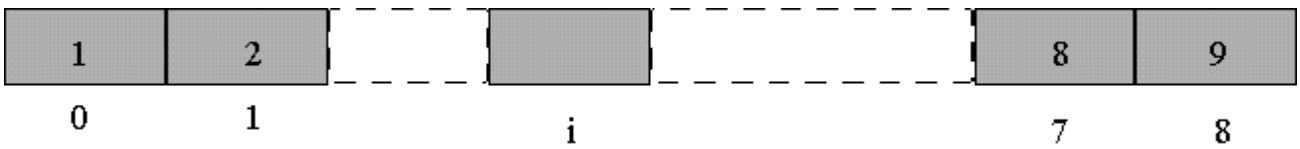


- Parmi les déclarations précédentes, trouvez un tableau d'octets. ***`char tab_byte[9]`***
- En C, comment peut-on obtenir l'adresse d'un tableau ? ***en utilisant le nom du tableau (sans crochets)***
- En utilisant l'opérateur &, comment peut-on aussi obtenir cette adresse ? ***`&tabbyte[0]`***

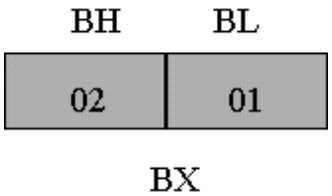
Certains langages autorise l'affectation d'un tableau dans un autre tableau. **Ce n'est pas le cas du langage C**.

- L'affectation `tab_byte = tab_byte1` a-t-elle un sens ? ***Non, car `tab_byte` a valeur d'adresse et non de pointeur ; ça n'a pas plus de sens que `5 = i`***
- L'affectation du type `pTabByte = tab_byte` a-t-elle un sens ? ***Oui, car `pTabByte` est une variable pointeur donc peut contenir une adresse***
- Pour accéder à la case d'indice `i` du tableau d'octets, qu'a-t-on besoin de connaître ? ***l'adresse de début du tableau et la distance de la case par rapport au début du tableau.***
- Calculez la distance d en fonction de `i` (`i` étant une variable d'indice). ***d* = `i`**
- Même question avec un tableau dont les cases seraient des mots de 16 bits. ***d* = `2*i`**
- L'adresse de début du tableau est-elle connue lors de la compilation ? ***oui***
- Citez deux instructions permettant de charger l'adresse du tableau dans BX. **`lea bx,tab_byte`** ***et*** **`mov bx,offset tab_byte`** ***l'opérateur offset utilisé en langage d'assemblage donne l'adresse d'une variable allouée par le compilateur***
- Quel mode d'adressage ces deux instructions mettent-elles en oeuvre ? **`lea bx,tab_byte`** ***met en oeuvre l'adressage direct et*** **`mov bx,offset tab_byte`** ***met en oeuvre l'adressage immédiat***
- Que fait l'instruction `MOV BX, tab_byte` ?

  Supposons que **`tab_byte`** ***ait le contenu suivant.***



  L'instruction **`MOV BX, tab_byte`** ***met dans*** **`BX`** ***le contenu des deux premières cases du tableau. En effet,*** **`BX`** ***étant un registre 16 bits, le transfert de données a lieu sur 16 bits. En ce qui concerne l'ordre des octets, le tableau tab_byte étant situé en mémoire, l'ordre de stockage considéré par l'instruction est l'ordre little-endian. Le poids faible de l'opérande est situé à l'adresse la plus faible et le poids fort de l'opérande est situé à l'adresse la plus élevée.*** **`BL`** ***reçoit donc le poids faible : 01,*** **`BH`** ***reçoit donc le poids fort : 02.*** **`BX`** ***reçoit donc 0201.***



- Comment accéder à la case d'indice `i` d'un tableau de caractères ?

1ère solution

```
mov     si,[i]
lea     bx,[tab_byte]
mov     al,[bx+si]
```

2ème solution

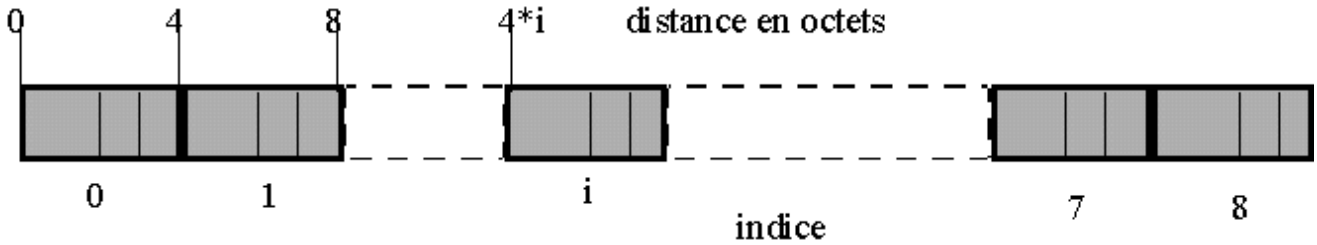
```
mov     si,[i]
mov     al,[si+tab_byte]
```

- Indiquez ce qui change avec un tableau d'entiers. ***il faut multiplier par 2 la valeur de*** `i` ***pour obtenir la distance*** `d`***. Pour cela, on peut utiliser*** `add si,si` ***qui est compacte est rapide.***

Tableau d'enregistrements

Les "cases" d'un tableau peuvent aussi être des objets plus complexes que des caractères ou des entiers, par exemple des enregistrements.

- Parmi les déclarations précédentes, trouvez un tableau d'enregistrements. `rec_t tab_enr[9];`
- Quelle est la taille occupée par ce tableau ? ***9 * 4 = 36 octets***
- Dessinez une représentation graphique d'un tableau d'enregistrements.



- Calculez la distance d en fonction de i. ***d = 4*i***
- Trouvez une formule de calcul de d en fonction de `i` convenant pour tout type de tableau. ***d = taille_case * indice***
- Dans cette formule, est-ce la valeur ou l'adresse de `i` qui intervient. ***c'est la valeur de*** `i`
- Comment accéder au champ 2 de la case d'indice `i` ?

1ère solution

```
mov     si,[i]
add     si,si
add     si,si
lea     bx,[tab_enr]
```

2ème solution

```
mov     si,[i]
add     si,si
add     si,si
mov     al,[si+tab_enr+2]
```

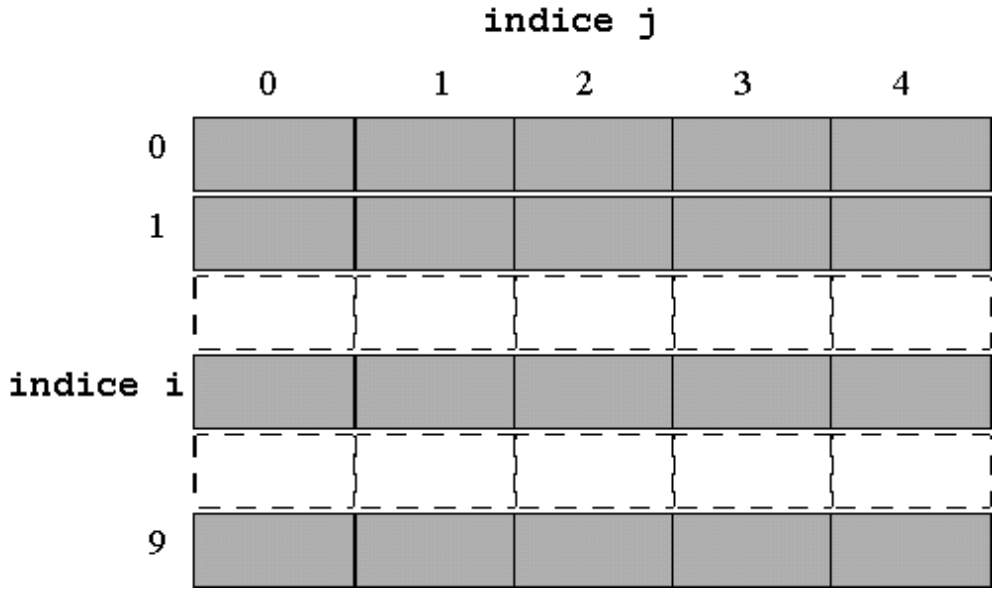
```
mov     al,[bx+si+2]
```

Tableaux à 2 dimensions

On considère les déclarations suivantes :

```
#define LIGS 10
#define COLS 5
char tab[LIGS][COLS];
```

- Dessinez une représentation graphique d'un tableau à deux dimensions.



- Quelle est la taille d'une "case" ? ***1 octet***
- Quelle est la taille d'une ligne ? ***5 octets***
- Que désigne l'expression `tab[i][j]` ? ***la case située sur la ligne*** **`i`** ***et la colonne*** **`j`**
- Cette fois-ci, d désigne la distance entre le début du tableau et la case `tab[i][j]` (`i` et `j` étant des variables). Proposez une formule de calcul de d en fonction de `i` et `j`. ***d =*** **`i`*****taille_ligne+*****`j`*****taille_case***

Tableau de pointeurs

On considère les déclarations suivantes :

```
#define LIGS 10
#define COLS 5
char *tab[LIGS];
```

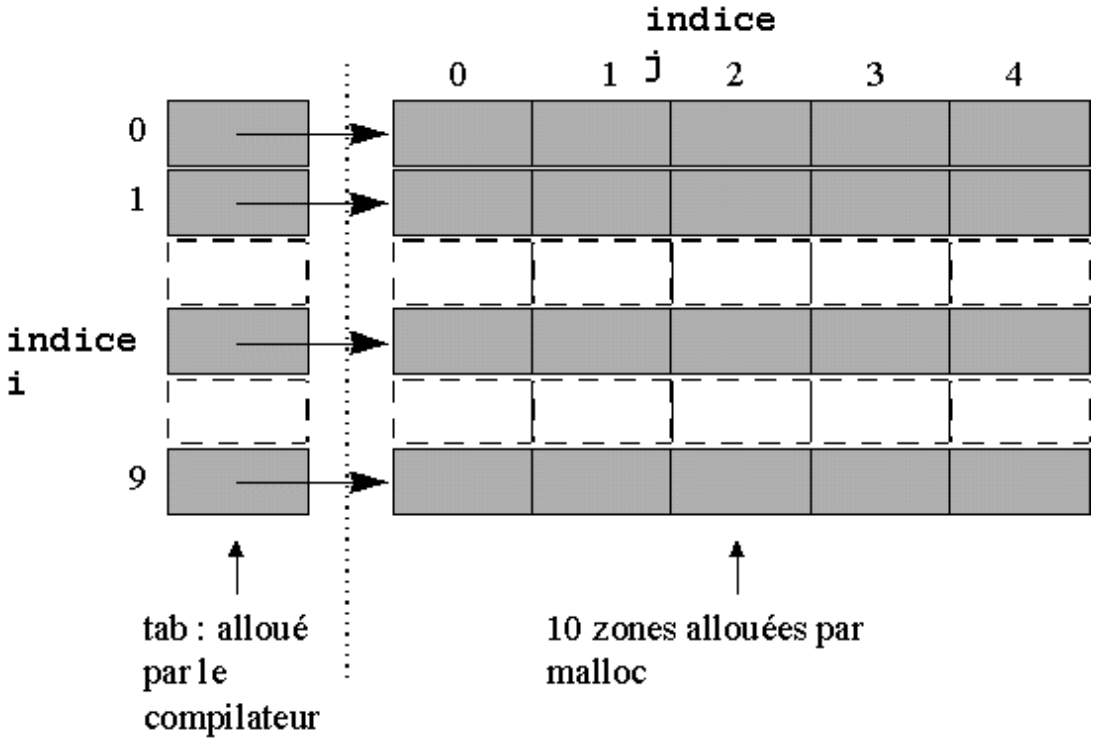
- Quelle est la taille d'une "case" ? ***2 octets s'il s'agit d'un pointeur*** **`near`****, 4 octets s'il s'agit d'un pointeur*** **`far`**

On considère le code d'initialisation suivant :

```
int i;
for (i=0; i<LIGS; i++)
        tab[i] = malloc( COLS*sizeof(char) );
```

La fonction `malloc` est chargée de réserver auprès du système d'exploitation une zone de mémoire ayant la taille spécifiée en argument (ici `COLS*sizeof( char )` c'est-à-dire 5 octets).

- Selon vous, que renvoie la fonction malloc ? ***l'adresse de la zone de mémoire allouée***
- Dessinez une représentation graphique d'un tel tableau après allocation.



- Que désigne l'expression `tab[i][j]` ? ***la case d'indice*** **`j`** ***dans la ligne pointée par*** **`tab[i]`**
- Que remarque-t-on à propos de la syntaxe ? ***c'est la même syntaxe que pour un tableau à 2 dimensions***

Revisé: 08-10-2008.

Corrigé du TD3

Structuration des traitements

Dans ce TD, on s'intéresse à la façon dont un compilateur de langage évolué génère le code correspondant aux structures de contrôle de l'exécution : choix, boucles, etc...

Le SI ALORS

Voyons d'abord ensemble le SI ALORS. L'énoncé de l'exercice correspond au point 1 de la méthode. Vous devez réaliser les points 2 et 3 de la méthode, c'est-à-dire mettre le pseudo-code en commentaire et coder en langage d'assemblage.

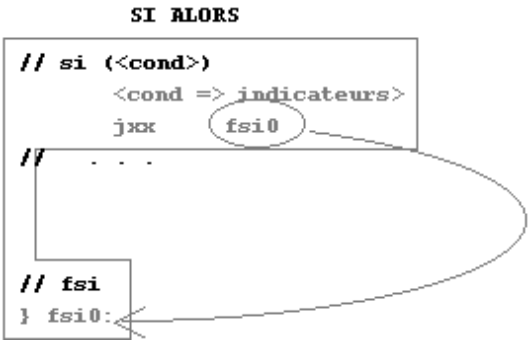
Enoncé

```
int a, min;

...

if (a < min) {
        min = a;
}
```

Codage en langage d'assemblage

Concernant l'application de la méthodologie, les éléments de l'énoncé sont en commentaires (précédés de `//`). On les met en évidence grâce aux caractères gras. Le code assembleur fait éventuellement l'objet d'un commentaire de fin de ligne. Les variables a et min sont des entiers signés 16 bits.

```
asm {
// if (a < min) {
        MOV     AX,[a]    // adressage direct pour a et min (DATA SEGMENT)
        CMP     AX,[min]  // comparer ax avec min
        JGE     fsi0      // Jump to fsi0 if ax Greater or Equal than min
//      min = a;
        MOV     [min],AX  // optimisation : AX contient la même chose que a
// }
}
fsi0:                     // ceci est l'étiquette qui sert à enjamber le corps du SI
                          // quand a >= min
```



On voit ici qu'une ligne de pseudo-langage telle que **fin si** correspond en fait à une ligne d'assembleur qui certes ne comporte pas d'instruction exécutable, mais qui comporte néanmoins une définition d'étiquette de saut.

Evaluation d'une expression booléenne

L'expression `a < min` est booléenne car elle a une valeur de vérité vrai ou faux. Deux types d'instructions

sont utiles pour l'évaluation des expressions booléennes.

- Les instructions qui positionnent les indicateurs,
- les instructions de saut conditionnel.

Pour positionner les indicateurs, on utilisera notamment l'instruction `CMP` qui compare deux opérandes et positionne les indicateurs en conséquence. Les instructions de saut conditionnel ont des mnémoniques qui ont un sens relativement aux deux opérandes de l'instruction `CMP`. Par exemple `cmp ax,min` suivi de `jge fsi0` signifie que le saut vers l'étiquette `fsi0` aura lieu si `ax` (opérande de gauche) est supérieur ou égal (`JGE` = **J**ump if **G**reater or **E**qual) à `min` (opérande de droite).

Expressions booléennes reliées par un "ou" logique

Dans l'instruction "`si a < min ou a > max alors min=a finsi`" la condition est composée de deux expressions booléennes plus simples reliées par un "`ou`" logique, c'est-à-dire que :

- il suffit que l'une des deux expressions soit vraie pour que "`min=a`" soit exécuté,
- donc il faut que les deux expressions soient fausses pour ne pas exécuter "`min=a`"

on aura donc le code machine suivant :

```
asm {
// if (a < min || a > max) {
        mov     ax,[a]
        cmp     ax,[min]
        JL      alors0  // Jump to alors0 if ax Lower than min
        cmp     ax,[max]
        JLE     fsi0    // Jump to fsi0 if ax Lower or Equal than min
// min = a
}
alors0:
asm {
        mov     [min],ax
// }
}
fsi0:
```

Expressions booléennes reliées par un "et" logique

Dans l'instruction "`si a < min et min > max alors min=a finsi`" la condition est composée de deux expressions booléennes plus simples reliées par un "`et`" logique, c'est-à-dire que :

- il faut que les deux expressions soient vraies pour que "`min=a`" soit exécuté,
- donc il suffit que l'une des deux expressions soient fausses pour ne pas exécuter "`min=a`"

on aura donc le code machine suivant :

```
asm {
// if (a < min && min > max) {
        mov     ax,[a]
        mov     bx,[min]
        mov     cx,[max]
        cmp     ax,bx
        JGE     fsi0    // Jump to fsi0 if a Greater or Equal than min
        cmp     bx,cx
        JLE     fsi0    // Jump to fsi0 if min Lower or Equal than max
// min = a
}
alors0:
```

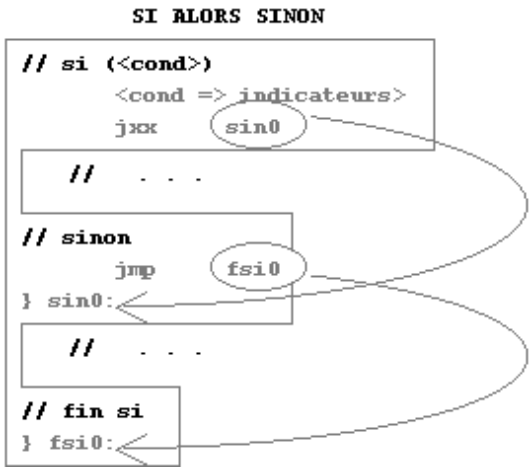
```
asm {
        mov     [min],ax
// }
}
fsi0:
```

Le SI ALORS SINON

On se propose maintenant de coder en langage d'assemblage les lignes suivantes :

Enoncé

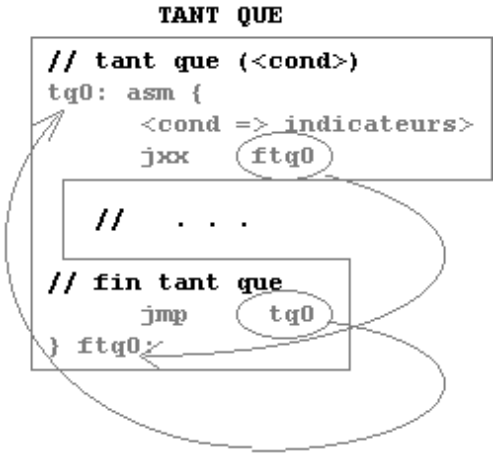
```
int a, b, min;

...

//if (a < b) {
        mov     ax,[a]
        mov     bx,[b]
        cmp     ax,bx
        jge     els0
//      min = a
        mov     [min],ax
//}
//else {
        jmp     fsi0
els0:
//      min = b
        mov     [min],bx
//}
fsi0:
```



La boucle TANT QUE

Codez en langage d'assemblage les lignes suivantes.

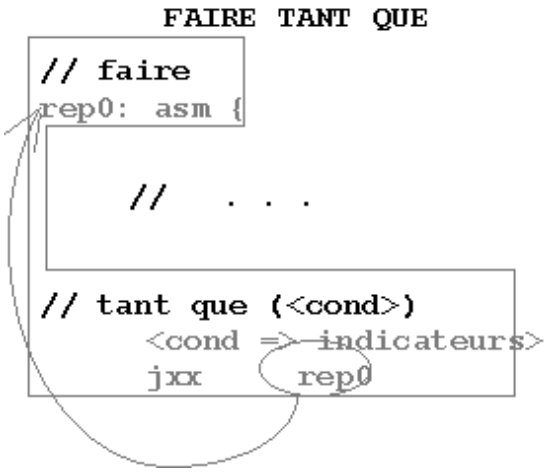
```
int i;
int tab[10];

//i = 0
        mov     [i],0
//while (i < 10) {
        lea     bx,[tab]
tq0:    mov     si,[i]
        cmp     si,10
        jge     ftq0
//      tab[i] = 0;
        add     si,si
        mov     [bx+si],0
//      i++;
        inc     [i]
//}
        jmp     tq0
ftq0:
```



La boucle FAIRE TANT QUE

Codez en langage d'assemblage les lignes suivantes.

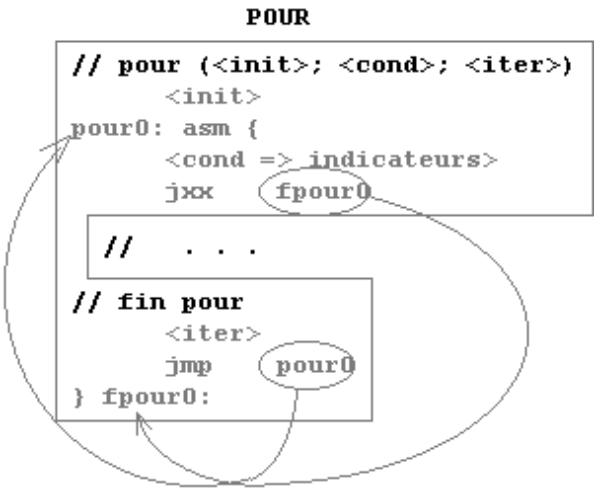
```
int a, b;

//do {
do0:
//      a++;
        inc     [a]
//      b--;
        dec     [b]
//} while (a >= b);
        mov     ax,[a]
        cmp     ax,[b]
        jge     do0
```



La boucle POUR

Codez en langage d'assemblage les lignes suivantes.

```
int i;
int tab[10];

//for (i=0; i<10; i++) {
        mov     [i],0
        lea     bx,[tab]
p0:     mov     si,[i]
        cmp     si,10
        jge     fp0
//      tab[i] = 0;
        add     si,si
        mov     [bx+si],0
//}
        inc     [i]
        jmp     p0
fp0:
```




```
//for (i=0; i<10; i++) {
        mov     [i],0
        lea     bx,[tab]
        jmp     cp0
p0:
//      tab[i] = 0;
        add     si,si
        mov     [bx+si],0
//}
        inc     [i]
cp0:    mov     si,[i]
        cmp     si,10
        jl      p0
```

Ceci est une autre implantation qui évite un saut
systématique par tour de boucle. Le jmp cp0 n'a
lieu que la première fois.

La structure de CAS

Codez en langage d'assemblage les lignes suivantes.

```
switch (etat) {
```

```
        mov     ax,[etat]
case 0 :
        cmp     ax,0
        jne     sw0c1
        etat = 2;
        mov     [etat],2
        break;
        jmp     fsw0
case 1 :
sw0c1:  cmp     ax,1
        jne     sw0c2
        etat = 3;
        mov     [etat],3
        break;
        jmp     fsw0
case 2 :
sw0c2:  cmp     ax,2
        jne     sw0df
        etat = 0;
        mov     [etat],0
        break;
        jmp     fsw0
default :
sw0df:
        etat = 3;
        mov     [etat],3
}
fsw0:
```

Imbrication des structures de contrôle

Ayant codé ces différentes structures, les possibilités d'exercices ne manquent pas dès qu'on décide de les imbriquer les unes dans les autres : deux boucles l'une dans l'autre, un SI dans un TANT QUE, etc...

Optimisation des sauts

Dans le code suivant, il est possible d'optimiser un saut.

- Lequel ? ***le jmp fsi0, qui fait sauter vers un autre saut qui saute en tq0***
- Par quoi peut-on le remplacer ? ***par jmp tq0 ; de cette façon, il n'y a plus qu'un seul saut***

```
// i <- 0
// tant que i<10 répéter
tq0:     <cond => indicateurs>
         jxx      ftq0
//   si a=tab[i] alors
         <cond => indicateurs>
         jxx      els0
//         i <- 10
         ...
//   sinon
         jmp fsi0
els0:
//         i <- i+1
         ...
//   fin si
fsi0:
// fin répéter
         jmp      tq0
ftq0:
```

Corrigé du TD4

Le passage des arguments aux sous-programmes.

Les termes *paramètres* et *arguments* ne sont pas tout à fait interchangeables. On parlera plutôt de *paramètre formel* dans le contexte de **l'appelé**. Le terme *argument* sera synonyme de *paramètre effectif* et sera employé dans le contexte de **l'appelant**.

On a le programme en C suivant :

```
#include <stdio.h>
#define N 10

int tabG[N] = { 10, 3, 7, 5, 4, 2, 7, 4, 5, 15 };

int rechercher( int e, int tab[N] ) {
        int i, fin;

        fin = 0;
        i = 0;
        while (i<N && !fin)
                if (e == tab[i])
                        fin = !fin;
                else
                        i++;

        return i;
}

main() {
        printf( "%d", rechercher( 5, tabG ) );

        return 0;
}
```

Passage par valeur, passage par adresse

Contrairement à d'autres langages, le passage des arguments aux sous-programmes s'effectue toujours **par valeur** en langage C. Cependant, par analogie avec d'autres langages, on distinguera *passage par valeur* et *passage par adresse* (ce qui n'est pas contradictoire puisqu'une adresse est une valeur un peu particulière).

- La constante 5 est passée par ***valeur*** à la fonction "rechercher".
- Le tableau tabG est passé par ***adresse*** à la fonction "rechercher".
- Citez les paramètres formels. **`e`** ***et*** **`tab`**
- Citez les paramètres effectifs (ou arguments). **`5`** ***et*** **`tabG`**

Passage d'un "tableau" en argument

Le passage d'arguments par valeur a une **sémantique de duplication**. Lorsque l'objet a une certaine taille, cela peut être pénalisant. C'est pourquoi le C **n'autorise pas** le passage des tableaux en arguments ; c'est en fait **leur adresse** qui est confiée au programme appelé.

Le programme appelé manipule alors le tableau via un pointeur et la syntaxe utilisée avec ce pointeur ressemble fort à celle utilisée pour un tableau. Si cela est commode c'est également source de confusion.

Rappelez-vous : un paramètre d'une fonction,

même s'il ressemble à une déclaration de tableau, est en fait une déclaration de pointeur.

- L'identificateur tabG est-il un tableau ou un pointeur ? `tabG` ***est un tableau***
- Quelle est la taille de tabG ? ***20 octets***
- L'identificateur tab est-il un tableau ou un pointeur ? `tab` ***est un pointeur***
- Quelle est la taille de tab ? ***2 octets pour un pointeur near ou 4 octets pour un pointeur far (cela dépend des options de compilation sélectionnées au niveau du modèle de mémoire)***

Données

Le panneau de données de Turbo-Debugger permet de voir :

```
ds:00A0 3A 79 00 00 3A 79 00 00 :y :y
ds:00A8 FF 9F 0A 00 03 00 07 00 ƒ
ds:00B0 05 00 04 00 02 00 07 00
ds:00B8 04 00 05 00 0F 00 25 64 ¤ %d
ds:00C0 00 00 00 00 CA 14 E4 67 -¶õg
ds:00C8 F5 02 E4 67 F5 02 E4 67 §õg§õg
```

On peut voir une variable dans le panneau de données du Turbo-Debugger. Attention, la représentation étant par octets, il faut penser au little-endian.

- Quelle variable voit-on dans le panneau de données ? ***le tableau tabG***
- Repérez le début et la fin de cette variable. ***voir ci-dessus les valeurs soulignées***

Le code non optimisé d'un compilateur C donne :

On cherche à analyser le code généré par le compilateur C.

```
_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
#RECHLIN#10: fin = 0;
 cs:0013 C746FC0000 mov word ptr [bp-04],0000
#RECHLIN#11: i = 0;
 cs:0018 C746FE0000 mov word ptr [bp-02],0000
 cs:001D EB21 jmp #RECHLIN#12 (0040)
#RECHLIN#13: if (e == tab[i])
 cs:001F 8B46FE mov ax,[bp-02]
 cs:0022 D1E0 shl ax,1
 cs:0024 8B5E08 mov bx,[bp+08]
 cs:0027 03D8 add bx,ax
 cs:0029 8B07 mov ax,[bx]
 cs:002B 3B4606 cmp ax,[bp+06]
 cs:002E 750D jne #RECHLIN#16 (003D)
#RECHLIN#14: fin = !fin;
 cs:0030 8B46FC mov ax,[bp-04]          // le code machine est complexe ici
 cs:0033 F7D8 neg ax                    // car le C ne possède pas de type booléen
 cs:0035 1BC0 sbb ax,ax                 // toute valeur non nulle est vraie
 cs:0037 40 inc ax                      // 0 est faux.
 cs:0038 8946FC mov [bp-04],ax
#RECHLIN#17: else
 cs:003B EB03 jmp #RECHLIN#12 (0040)
#RECHLIN#16: i++;
 cs:003D FF46FE inc word ptr [bp-02]
#RECHLIN#12: while (i<N && !fin)
 cs:0040 837EFE0A cmp word ptr [bp-02],000A
 cs:0044 7D06 jnl #RECHLIN#18 (004C)    // sortie si i>=N
```

```
 cs:0046 837EFC00 cmp word ptr [bp-04],0000
 cs:004A 74D3 je #RECHLIN#13 (001F)     // ou sortie si fin est vrai
#RECHLIN#18: return i;
 cs:004C 8B46FE mov ax,[bp-02]
 cs:004F EB00 jmp #RECHLIN#19 (0051)    // un saut bien inutile !
#RECHLIN#19: }
 cs:0051 8BE5 mov sp,bp
 cs:0053 5D pop bp
 cs:0054 CB retf

_main: main() {
 cs:0055 55 push bp
 cs:0056 8BEC mov bp,sp
#RECHLIN#22: printf( "%d", rechercher( 5, tabG ) );
 cs:0058 B8AA00 mov ax,00AA
 cs:005B 50 push ax
 cs:005C B80500 mov ax,0005
 cs:005F 50 push ax
 cs:0060 0E push cs             // nécessaire ici car le call est "near"
                                // et le retf est "far"
 cs:0061 E8A9FF call _rechercher
 cs:0064 59 pop cx
 cs:0065 59 pop cx
 cs:0066 50 push ax             // le résultat de la recherche est dans AX
 cs:0067 B8BE00 mov ax,00BE
 cs:006A 50 push ax
 cs:006B 9A8A0FE467 call far _printf
 cs:0070 59 pop cx
 cs:0071 59 pop cx
#RECHLIN#24: return 0;
 cs:0072 33C0 xor ax,ax
 cs:0074 EB00 jmp #RECHLIN#25 (0076)
#RECHLIN#25: }
 cs:0076 5D pop bp
 cs:0077 CB retf
```

Représentation de la pile

Il faut représenter la pile telle sorte que les dernières valeurs empilées soient sur le dessus. Cette représentation est commode pour l'analogie avec une pile d'assiettes. Curieusement, la représentation de la pile dans Turbo-Debugger est la représentation inverse.

Il faut représenter la pile avec des emplacements de 16 bits. En effet, la manipulation de la pile s'opère par mots de 16 bits. Ainsi, la "distance" entre deux emplacements de pile voisins est de 2 octets.

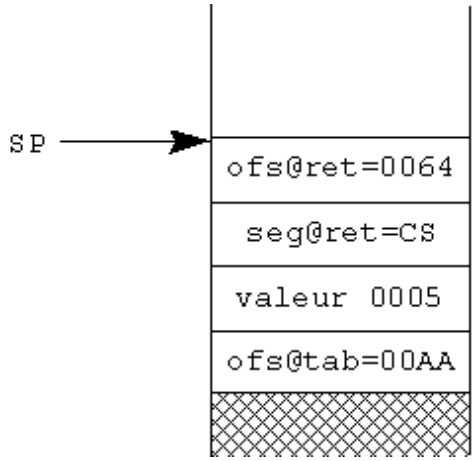
Rôle de l'appelant

On rappelle qu'en langage C, l'exécution d'un programme commence par la fonction "`main`" qui joue le rôle de programme principal qui appelle des sous-programmes. Ces derniers appellent éventuellement à leur tour d'autres sous-programmes.

- Quelle fonction joue ici le rôle de l'appelant ? ***la fonction*** **`main()`**
- Quelle fonction joue ici le rôle de l'appelé ? ***la fonction*** **`rechercher()`**
- A quoi correspond la valeur `00AA` ? ***c'est l'adresse (offset) du tableau*** **`tabG`**
- A quoi correspond la valeur `0005` ? ***c'est la valeur*** **`5`**, ***premier argument passé à la fonction*** **`rechercher`**
- En C, dans quel ordre les arguments sont-ils empilés ? ***en commençant par le dernier***
- En quoi est-ce utile pour les fonctions qui ont un nombre d'arguments variable comme `printf` ? ***le premier argument est empilé en dernier. L'appelant le trouve donc toujours au même endroit de la pile quel que soit le nombre d'arguments passé. Généralement le premier argument fournit des informations sur le nombre d'arguments présents sur la pile.***
- A quoi servent les deux `pop cx` après le `call _rechercher` ? ***à "nettoyer" la pile, la valeur dans***

`cx` *n'est pas exploitée.*

- Que peut-on dire de l'état de la pile après les deux `pop cx` ? ***la pile se trouve dans le même état que celui où elle se trouvait avant l'empilage des arguments.***
- Quel registre contient le résultat de la recherche (c'est une convention du compilateur) ? ***le registre* `ax`.**
- Dessinez l'état de la pile au moment où `IP` vaut `000D` ?



Prologue d'une fonction,

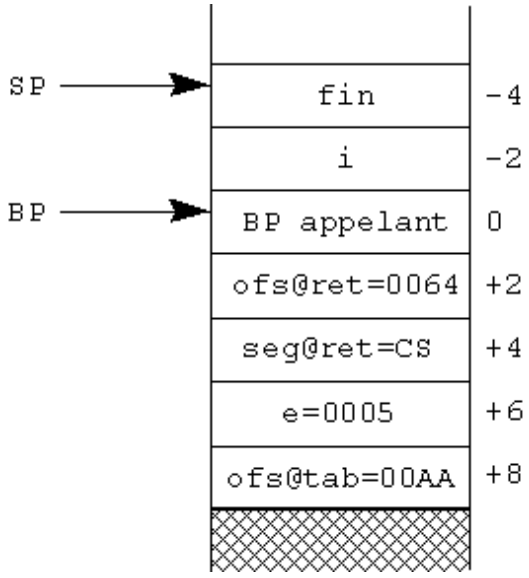
- Que fait le code machine suivant (prologue) ?

```
push bp          // sauvegarde BP positionné sur le contexte de l'appelant
mov bp,sp        // positionne BP sur le contexte de l'appelé
                 // heureusement, on l'avait sauvegardé à la ligne précédente !
```

- Pourquoi doit-on sauvegarder le `BP` ? ***pour pouvoir restituer sa valeur plus tard***
- Que fait le `sub SP,4` ? ***réserve (sans initialiser) 4 octets pour les variables locales (`i` et `fin`)***

Accès aux paramètres et aux variables locales

- Dessinez l'état de la pile une fois le prologue passé.

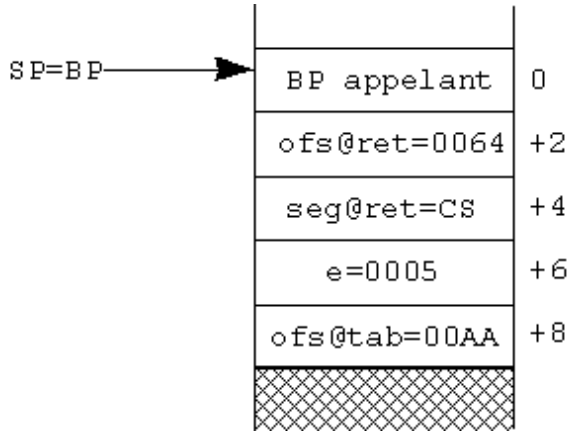


- A quoi correspond `[bp-04]` ? ***à la variable locale `fin` (voir `cs:0013`)***
- A quoi correspond `[bp-02]` ? ***à la variable locale `i` (voir `cs:0018`)***
- A quoi correspond `[bp+06]` ? ***au paramètre formel `e` (voir `cs:002B`)***
- A quoi correspond `[bp+08]` ? ***au paramètre formel `tab` (voir `cs:0024`)***

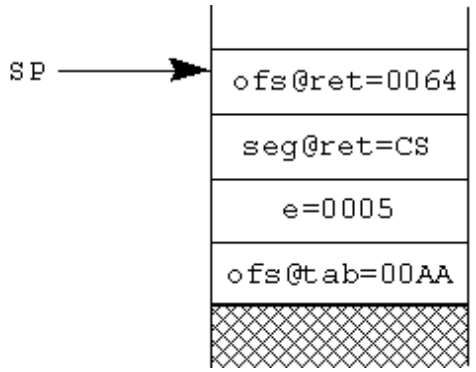
- Comment les variables locales et les paramètres sont-ils alors accessibles ? ***sur la pile*** Par quel mode d'adressage ? ***par le mode d'adressage basé sur*** **`BP`**
- Quel point de repère a-ton pour distinguer les variables locales des paramètres ? ***les paramètres sont accessibles par un déplacement positif par rapport à*** **`BP`**. ***Les variables locales sont accessibles par un déplacement négatif par rapport à*** **`BP`**.
- L'identificateur `tab` est-il un pointeur `near` ou un pointeur `far` ? ***ici, c'est un pointeur near car il occupe une seule case sur la pile.***
- Trouvez l'instruction qui accède finalement à `tab[i]` ? ***il s'agit du*** **`mov ax,[bx]`** ***à l'adresse*** **`cs:0029`**.

Epilogue d'une fonction

- Dessinez l'état de la pile après le `mov sp,bp`.
- Dessinez des flèches figurant `BP` et `SP`.



- Dessinez l'état de la pile après le `pop bp`.
- Dessinez des flèches figurant `SP`.



- Que contient `BP` ? **`BP`** ***a retrouvé sa valeur initiale sauvegardée pendant le prologue.*** **`BP`** ***pointe sur le contexte de l'appelant.***
- Que fait le code machine suivant (épilogue) ? ***la pile se retrouve dans le même état que celui où elle se trouvait avant l'épilogue en*** **`CS:000D`**.

```
mov sp,bp
pop bp
```

- Que fait le `retf` ? ***elle effectue le retour (*****`far`***) au programme appelant. Le couple de registres*** **`CS:IP`** ***se trouve donc chargé avec l'adresse segmentée complète (32 bits) présente au sommet de la pile, c'est-à-dire*** **`CS:0064`**.
- ***Chronologiquement, c'est à ce moment que les deux*** **`pop cx`** ***sont exécutés.***

Récursivité

Une fois le prologue passé, on se retrouve dans le **contexte** de l'appelé. Une procédure récursive est une procédure qui s'appelle elle-même (avec un contrôle programmé de la profondeur d'appel). Elle joue donc à la fois le rôle de l'appelant et de l'appelé.

- Quel est l'espace occupé sur la pile lors d'un appel de la procédure `rechercher` ? ***c'est l'espace occupé sur la pile lorsque celle-ci est au plus haut (paramètres+adresse de retour+`BP`+variables locales). Pour la fonction `rechercher`, cela fait 14 octets***
- Si cette procédure s'appelait elle-même, quelle serait la taille occupée par 100 appels imbriqués ? ***cela ferait 1400 octets.***

Christophe Tombelle.

Revisé: 22-11-2001.

Corrigé du TD5

Commutation de contexte

Commentez chaque instruction ci-dessous. Quel est globalement le rôle de ce code (code A) dans un environnement 16 bits ? ***Sauvegarde le contexte du programme appelant.*** Quelle est la paire de registres qui ne se retrouve pas dans la pile ? ***la paire de registres SS:SP n'est pas sauvegardée***

```
        pushf       // empiler les indicateurs
        callf   ici // empiler l'adresse de retour (segment puis offset) et sauter à
ici:

        ...

ici:    push    ax   // empiler ax
        push    bx   // empiler bx
        push    cx   // empiler cx
        push    dx   // empiler dx
        push    si   // empiler si
        push    di   // empiler di
        push    bp   // empiler bp
        push    es   // empiler es
        push    ds   // empiler ds
```

Commentez chaque instruction ci-dessous. Quel est globalement le rôle de ce code (code B) ?

```
        pop     ds   // dépiler ds
        pop     es   // dépiler es
        pop     bp   // dépiler bp
        pop     di   // dépiler di
        pop     si   // dépiler si
        pop     dx   // dépiler dx
        pop     cx   // dépiler cx
        pop     bx   // dépiler bx
        pop     ax   // dépiler ax
        iret        // dépiler l'adresse de retour dans CS:IP (offset->IP puis
segment->CS) puis les indicateurs
```

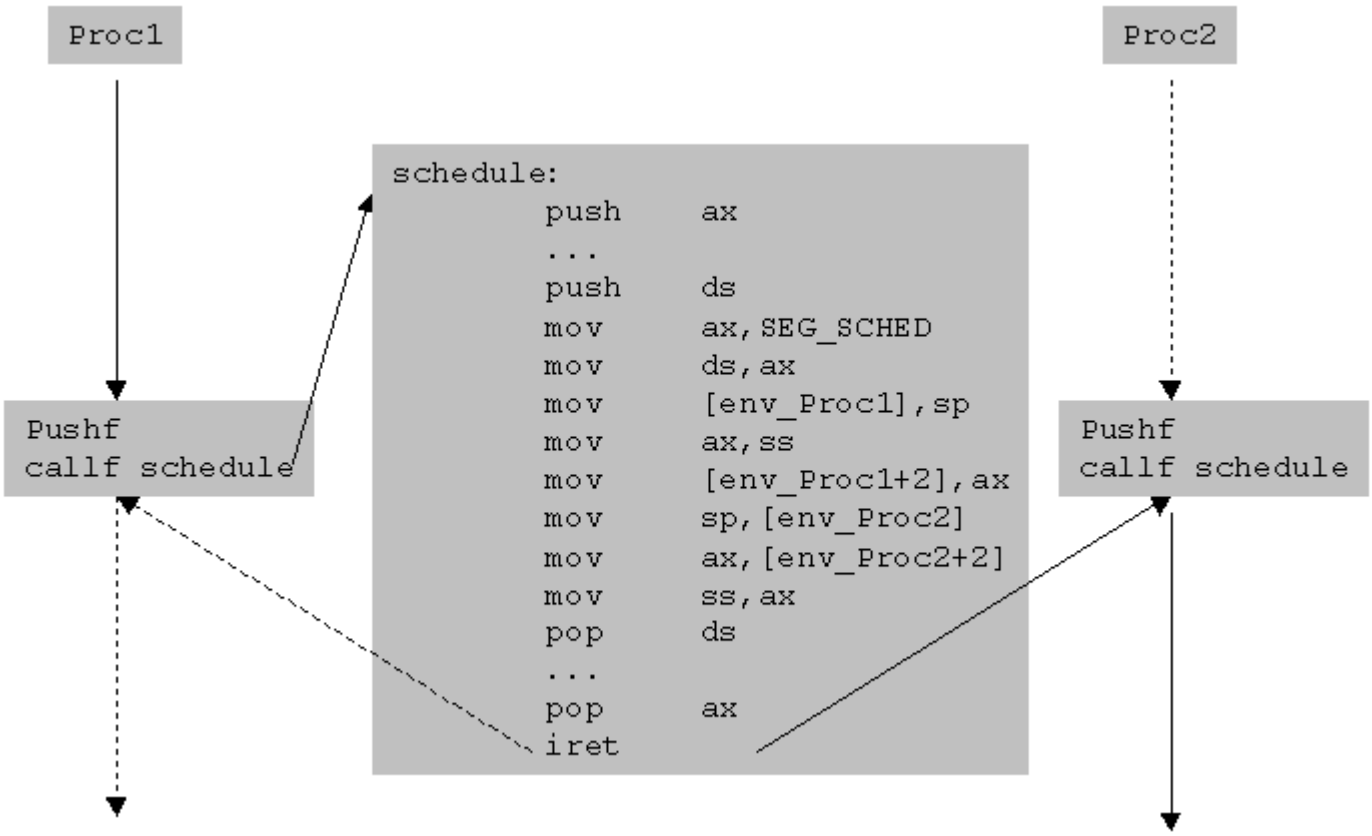
Que se passe-t-il si on enchaîne "code A" et "code B" ? ***on se retrouve dans le programme appelant (celui qui a exécuté le `callf ici`) dans une situation mais après le `callf ici`.***

Que se passe-t-il si on insère le code suivant (code C) entre "code A" et "code B" ? ***on sauvegarde le contexte du processus Proc1 puis on restitue le contexte du processus Proc2.*** Que doit-on supposer à propos de la valeur contenue dans `env_Proc2` et de la valeur `SEG_SCHED` ? ***on suppose que env_Proc2 est une variable contenant l'adresse segmentée complète du sommet de la pile du processus Proc2 et que cette pile contient le contexte de Proc2 préalablement sauvegardé de la même façon que celui de Proc1. SEG_SCHED est censée être la partie segment de l'adresse des variables env_Proc1 et env_Proc2. Ce segment est supposé a priori différent des segments de données des processus env_Proc1 et env_Proc2. C'est le segment de données du scheduler.*** Qu'a-t-on réalisé ? ***on a réalisé une commutation de contexte de Proc1 vers Proc2.***

```
        mov     ax,SEG_SCHED
        mov     ds,ax
        mov     [env_Proc1],sp
        mov     ax,ss
        mov     [env_Proc1+2],ax
        mov     sp,[env_Proc2]
        mov     ax,[env_Proc2+2]
        mov     ss,ax
```

Les variables `env_Proc1` et `env_Proc2` ayant des contenus a priori différents, que peut-on dire des processus `Proc1` et `Proc2` relativement à la notion de pile ? ***les processus `Proc1` et `Proc2` ont chacun leur***

segment de pile. Dessinez le cheminement du fil d'exécution suivi lorsqu'on enchaîne les codes A, C et B. Le cheminement indiqué en trait plein correspond à la commutation de contexte **Proc1** vers **Proc2**.



Ordonnancement fondé sur l'équité

On cherche à ce que les processus aient la main chacun leur tour de façon circulaire. Ce type d'ordonnancement est parfois qualifié de touniquet (round-robin) ou tour de table.
Proposez une structure de données en langage C et un **code C'** qui généralise le mécanisme précédent à n processus et fournit le fonctionnement en tourniquet.

```
void far * tab_Proc[MAX_PROC];    // contains far stack pointers for processes
int currentProc = 0;

void schedule() asm {
// save calling process context (flags, CS, IP already stacked)
        push    ax
        push    bx
        push    cx
        push    dx
        push    si
        push    di
        push    bp
        push    es
        push    ds
// load scheduler ds value
        mov     ax,SEG tab_Proc
        mov     ds,ax
// tab_Proc[currentProc] = calling process SS:SP value
        mov     bx,OFFSET tab_Proc
        mov     si,[currentProc]
        mov     di,si
        add     di,di
        add     di,di
        mov     [bx+di],sp
        mov     ax,ss
        mov     [bx+di+2],ax
// currentProc = (currentProc+1) % MAX_PROC
```

```
        inc     si
        cmp     si,MAX_PROC
        jne     fsi0
        xor     si,si
fis0:   mov     [currentProc],si
// SS:SP = tab_Proc[currentProc]
        add     si,si
        add     si,si
        mov     sp,[bx+si]
        mov     ax,[bx+si+2]
        mov     ss,ax
// restore new elected process context
        pop     ds
        pop     es
        pop     bp
        pop     di
        pop     si
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        iret
}
```

On appelle **schedule** le "sous-programme" commençant par l'étiquette "ici:" et constitué des codes A, C' et B. Lorsqu'un processus souhaite céder le processeur, il suffit qu'il appelle le sous-programme `schedule` avec pour seule contrainte d'utiliser `pushf` puis `callf` plutôt qu'un simple `call`.
Que pensez-vous dans ce cadre de l'instruction `int xx` ? ***Cette instruction appelle un sous-programme d'interruption via le numéro de vecteur xx (xx étant un nombre compris entre 00 et FF). Cette instruction présente l'avantage de sauvegarder les indicateurs et d'utiliser des adresses far (adresses segmentées complètes). Sur les processeurs possédant plusieurs modes d'exécution (superviseur ou privilégié et utilisateur) ce type d'instruction fait également passer en mode d'exécution privilégié qui est le mode d'exécution des systèmes d'exploitation (schedule ferait alors partie du système d'exploitation).***

Temps d'attente

Lorsqu'un processus a besoin d'effectuer une entrée ou une sortie, cela peut prendre du temps (voir TD liaison série). Une fois l'opération d'entrée-sortie lancée, il serait pénalisant que le processeur passe son temps à attendre la fin de l'opération. L'idée est de céder le processeur à un autre processus jusqu'à ce que l'opération d'entrée-sortie soit terminée.
Les opérations d'entrée-sortie étant généralement accessibles aux applications par l'intermédiaire de bibliothèques, que doit-on envisager pour l'écriture de ces bibliothèques ? ***il faut prévoir un appel à schedule une fois l'opération lancée. L'ensemble des services d'entrées-sorties offert par le système d'exploitation est donc susceptible d'appeler schedule.***
Qu'appelle-t-on processus éligible ? ***c'est un processus qui n'a pas besoin d'attendre quoi que ce soit, il ne lui manque que le processeur pour être le processus élu, à savoir le processus qui s'exécute.***
En quoi cette notion complique-t-elle le sous-programme `schedule` ?
Seuls les processus éligibles doivent participer à l'ordonnancement. Il ne s'agit plus simplement d'incrémenter `currentProc` modulo `MAX_PROC`. Il faut une structure de données qui ne stocke que les processus éligibles qui sont en nombre variable.
Comment la fin d'une opération d'entrée-sortie est-elle signalée au processeur ? ***par une interruption***
Qu'est-ce que cela implique comme traitement correspondant ? ***le traitement de l'interruption doit marquer comme éligible le processus qui était en attente de terminaison de l'opération d'entrée-sortie, puis faire appel à schedule. Une telle interruption est un événement susceptible de provoquer un ordonnancement.***

Ordonnancement préemptif

Dans les ordonnanceurs étudiés jusqu'à présent, quels sont les événements qui peuvent provoquer un ordonnancement ? (le réexamen de l'attribution du processeur)
Les interruptions signalant la fin d'une opération d'entrée-sortie.
La cession du processeur par un processus qui considère ne plus en avoir besoin pour l'instant.
Quel type d'événement permettrait-il de reprendre le processeur à un processus non discipliné ?
Une interruption se produisant quoi qu'il advienne au bout d'un certain temps. Ainsi, si un autre événement ne vient pas provoquer l'ordonnancement, il y aura au moins celui-là.
Sur une carte à microprocesseur, quelle dispositif faut-il prévoir absolument pour l'implantation d'un noyau multi-tâche préemptif ? ***un temporisateur pouvant déclencher une interruption***
A quels endroits un processus peut-il être interrompu (préempté) dans le cas d'un ordonnancement préemptif ? ***à n'importe quel endroit où l'interruption temporisateur est autorisée.***

Christophe Tombelle.

Revisé: 22-11-2001.