Langage assembleur

Version de Janvier 2004

esi

Langage assembleur

Pierre Bettens (PBT)

pbettens@heb.be

Avant-propos

Ces notes sont écrites, à la fois, pour servir de support à la présentation orale du cours et comme support pour l'étudiant.

Ce document n'est que la remise en forme, légèrement modifiée et complétée, des notes de messieurs Jean-Claude Jaumain et Jean-Marie VanLoock. Qu'ils en soient ici remerciés. J'en profite pour remercier plus particulièrement tout ceux qui aiment être remerciés.

Cette première version sera revue, augmentée ou diminuée … en tous les cas, elle sera modifiée pour les années suivantes afin de mieux répondre aux besoins de l'étudiants.

Exemples

Les exemples contenus en annexes dans ces notes sont disponibles dans mon distri et sur internet (demandez-moi où).

Langage assembleur

Jonas Beleho (BEJ)
j_beleho@msn.com
Nicolas VANSTEENKISTE (NVS)
nvansteenkiste@heb.be
Amine Hallal (HAL)
ahallal@ulb.ac.be

Copyright

E-mail : pbettens@heb.be

Table des matières

Table des matières

Table des matières

Exemple 1

Afficher un nombre donné (dans EAX) en base 10 dans une autre base ECX (comprise entre 2 et 36)

EAX=45, ECX=4	Afficher : 231_4
EAX=123, ECX=4	Afficher : 1323_4
EAX=32767, ECX=16	Afficher : $7FFF_h$

LASEx01.asm

Analyse

Pour convertir un nombre de la base 10 vers une autre base, on utilise l 'algorithme de la division entière pas la base. Les chiffres du nombre sont les restes successifs.

Logique

```
ACTION
Définition d 'un vecteur de chiffres (ici 36 symboles)
Définition d 'une zone d 'affichage (EAX<=32767 en base 2=32 chiffres)
EAX <- nombre
ECX <- base
REPETER
   EDX <- EAX mod ECX
   EAX <- EAX div ECX
   Stocker le chiffres dans la zone d 'affichage de la fin vers le début
JUSQU A CE QUE EAX=0
Afficher les chiffres mémorisés
FIN ACTION
```

Exemple 2

Trier un vecteur de bytes (des caractères) à l 'aide du tri bulle.
Afficher la chaîne de caractères avant et après le tri.

La chaîne de caractères : **a d c b**

devient la chaîne : **a b c d**

Les étapes successives sont les suivantes :

a d c b -> a c d b -> a c b d -> **a b c d**

Principe

Il s'agit d'un tri par **permutations** ayant pour but d'amener à la " surface " du vecteur (élément d'indice minimum) la valeur la plus petite du vecteur (bulle), et ainsi de suite…

En partant de l'élément d'indice maximal, on parcourt le vecteur vers le premier élément en comparant chaque couple de valeurs qui se suivent ; deux valeurs dans le désordre sont mises en ordre par permutation. Lorsqu'on arrive à l'élément d'indice 1, lors de la fin du premier parcours, il contient la valeur minimale du vecteur d'origine. On recommence ensuite le parcours en partant de la même origine que lors du premier parcours mais cette fois, on s'arrêtera au deuxième élément du vecteur; et ainsi de suite… jusqu'à ce qu'il n 'y ait plus de permutations.

Remarque

On représentera par CX l 'avant dernier élément du vecteur. Dans notre cas (voir slide suivant) il faudra calculer comme suit :

CX = longueur du vecteur - caractères non triés (10,13 et ' $ ') - dernier elt

Exemple 2 (suite)

Remarques quant-à l 'affichage

Le vecteur se termine par les bytes 10,13 et $

On définira la longueur de la chaîne comme suit :

```
.data
  tabl      DB    ' Chaîne a trier ',10,13, '$ '
  longueur  DB    $-tabl
                  Mon adresse ($) moins celle de tabl.
```

LASEx02.asm

Logique

```
ACTION
Afficher la chaîne de caractères
permutation boleen
CX <- position de l 'avant dernier élément à trier.
REPETER
   permutation <- faux
  POUR i <- 0 A CX FAIRE
    SI tabl[i]>tabl[i+1] ALORS
      inverser (tabl[i],tabl[i+1])
      permutation <- vrai
    FIN SI
  FIN POUR
  CX <- CX-1
JUSQU A CE QUE NON permutation
Afficher la chaîne de caractères
FIN ACTION
```

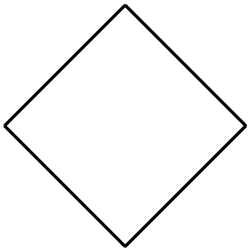
Remarque

La variable de type boleen permutation sera représentée par BL (0=fx, 1=vrai)

Exemple 3

Afficher un carré sur pointe de côté 200 pixels.

Le bord rouge et l 'intérieur vert.



Principe : Tracer les 4 côtés en rouge, ensuite tracer les horizontales intérieures en vert; du centre vers le haut et ensuite du centre vers le bas.

LASEx03.asm

Logique

```
ACTION
Selectionner le mode video VGA 640x480x16 ;  Positionner le
curseur au point 120,240
REPETER 199 fois{  x <- x+1; y<-y-1; pixel (x,y,rouge) } FIN
REPETER
REPETER 199 fois{  x <- x+1; y<-y+1; pixel (x,y,rouge) } FIN
REPETER
REPETER 199 fois{  x <- x-1; y<-y+1; pixel (x,y,rouge) } FIN
REPETER
REPETER 199 fois{  x <- x-1; y<-y-1; pixel (x,y,rouge) } FIN
REPETER
SI <- longueur de la diagonale intérieure (397) ; x <- 120 ; Y
<- 240
REPETER
  mémoriser X
  REPETER SI fois { x <- x+1; pixel (x,y,vert) }
  récupérer X
  x <- x+1; y <- y-1; SI <- SI-2;
JUSQU A CE QUE SI < 0
SI <- 395; x <- 121; y <- 241;  (en dessous de la diagonale)
REPETER
  mémoriser X
  REPETER SI fois { x <- x+1; pixel (x,y,vert) }
  récupérer X
  x <- x+1; y <- y+1; SI <- SI-2;
JUSQU A CE QUE SI < 0 … FIN ACTION
```

Procédures et programmation structurée

Pour pouvoir utiliser les procédures en assembleur, nous utiliserons

```
CALL maProcedure

maProcedure PROC
..
RET
maProcedure ENDP
```


Remarque

Il est clair que ceci est un raccourci !

Nous vous donnons ici uniquement la base afin de pouvoir utiliser les procédures dans vos projets de laboratoires. Les explications plus détaillées suivent dans le cours.

La pile du 80x86

- Une PILE est une structure de données permettant de mémoriser des informations.
- LIFO versus FIFO
  - LIFO (Last In First Out) Peut être simulée par une pile d'assiettes; on empile et on désempile une assiette.
  - FIFO (First In First Out) Peut être simulée par une file d'attente au guichet. « Premier arrivé, premier servi » !

Remarque

La notion de structure de données sera pleinement développée dans d'autres cours. Ce qu'il faut savoir c'est que, lorsque l'on veut écrire un programme, il faut s'attacher à deux choses:

- La structure des données.
- L'algorithme permettant de traiter ces données.

Tout l'art (!) de la programmation consiste à regrouper ces deux points. La structuration des données sert, quant à elle, à faire le lien entre les données abstraites d'un problème et leur représentation « binaire ».

L'objet de ce cours n'est pas de traiter des piles, files et autres types abstraits, sachez seulement que le type abstrait pile permet notamment l'écriture de programmes récurcifs.

La pile du 80x86

- Le 80x86 possède une pile LIFO
  - SS:SP sont les registres de segment (SS) et d'offset (SP) permettant d'accéder aux éléments de la pile.
  - .STACK est la directive permettant de fixer la taille de la pile.
  - *PUSH op* et *POP op* sont les instructions permettant d'empiler et désempiler un opérande.

Langage assembleur - 1ère - PBT

Directive .STACK et adresse de la pile

Cette directive précise la taille de la pile: *.STACK 100h, r*éserve une pile de 100h=256d bytes.

La pile est initialisée par le compilateur. Le système fournit donc l'adresse SS:SP qui est l'adresse du dernier élément empilé.

• SS est une valeur attribuée par le SE lorsqu'il charge le programme en mémoire.

• SP contient, lors de l'initialisation, la taille de la pile (100h par exemple). Les éléments sont donc empilés « à l'envers » sur la pile.

Lorsque j'empile un premier élément d'un word, SP=0FEh. (Je décrémente de 2 le pointeur de pile[1])

A l'inverse, lorsque je désempile un word, j'incrémente le pointeur de pile de deux.

1 Pointeur de pile = Stack Pointeur = SP

La pile du 80x86

Instruction PUSH

- Syntaxe : **PUSH op**
- But : empile *op*. Place le contenu de *op* sur la pile. *op* doit être au WORD ou DWORD
- Opérande : *op* peut-être :
  - Registre : PUSH AX
    PUSH EBX
  - Mémoire : PUSH word ptr [BX+SI]
  - Flags : PUSHF - PUSHFD (386)
  - Tout (286) : PUSHA - PUSHAD

Remarque

- L ’opérande doit être au minium un word. Je peux donc, si je défini une pile de 256 bytes empiler 128 mots … ou 54 doubles ou …
- **PUSHa**

L ’instruction PUSHa (all) empile les registres : AX, BX, CX, DX ainsi que les registres SP, BP, SI, et DI.

Dans le cas de l ’instruction PUSHad, ce sont les registres étendus correspondants qui sont empilés.

- Lorsqu’une instruction (ou une caractéristique) est disponible à partir d ’un processeur particulier, il faut le signaler au compilateur à l ’aide de la directive appropriée; **.286**, **.386** ou **.486**.

La pile du 80x86

Instruction POP

- Syntaxe : **POP op**
- But : dépile *op*. Place le contenu du sommet de la pile dans *op*.
- Opérande : *op* peut-être :
  - Registre : POP AX
    POP EBX
  - Mémoire : POP word ptr [BX+SI]
  - Flags : POPF - POPFD (386)
  - Tout (286) : POPA - POPAD

Remarque

PUSH CS est autorisé … mais pas POP CS … normal !

Exemples

- PUSH BX
- PUSH word ptr[DI]
- POP EAX
- La pile du 80x86 est de type LIFO. Quel est le résultat des opérations suivantes ?

```
PUSH AX
PUSH BX
POP  AX
POP  BX
```

Les contenus des registres AX et BX sont échangés, ceci est équivalent à :

```
MOV CX,AX
MOV AX,BX
MOV BX,CX
```

dans le premier cas, je fais l'économie d 'un registre.

Complément tableaux

Vecteur - Tableau à une dimension

- Déclaration

```
v1      DB      a,b,c,d,e,f
v2      DB      10 DUP (?)
```

- Exemples d'utilisation

```
MOV     AL,v1[0]            AL <- a
MOV     BX,offset v1
MOV     AL,[BX+2]           AL <- c
MOV     SI,2
MOV     AL,[BX+SI-1]        AL <- b
```

Langage assembleur - 1ère - PBT

Utilisation

Lorsque l 'on travaille en mode graphique, les vecteurs sont utilisés, par exemple, pour stocker des coordonnées de points. Exemple; supposons que l 'on veuille dessiner 6 carrés de coté 20 pixels à l 'écran (en mode vidéo 12h) aux positions (0,0), (0,20),(0,40),(20,0),(20,20) et (20,40). Si l 'on suppose l 'existence d 'une procédure *carré* dessinant un carré de coté 20 à l 'écran à la position DX,DL, on peut écrire le morceau de code suivant :

```
.data
  COTE      EQU         20
  NOMBRE    EQU         6
  positions DB          0,0,0,20,0,40,20,0,20,20,20,40
  …
.code
  …
  MOV       SI,0
repeter:
  MOV       DX,word ptr positions[SI]
  CALL      carre
  ADD       SI,2
frepeter:
  CMP       SI,2*NOMBRE
  JB        repeter
  …
```

Complément tableaux

Tableau à deux dimensions

- Pour représenter un tableau de taille *nxm*, je réserve n*m bytes (ou word ..).

```
N           EQU  12
M           EQU  32
tableau     DB   N*M DUP (?)
```

- J'accède à l'élément *(i,j)* :

```
MOV     SI,i
MOV     DI,j
MOV     AL,tableau[SI*M+DI]
```

Procédures

Vue minimaliste

Le 80x86 permet la gestion des procédures …

dans une vue minimaliste, cela se résume en deux instructions :

CALL *label*

RET

Langage assembleur - 1ère - PBT

Rappel

En logique de programmation, une approche top down d ’un problème demande de modulariser l ’algorithme. On aura par exemple :

```
ACTION
  instructions
  maFonction()          ; appel de la fonction
  instructions
  maFonction()          ; appel de la fonction
  instructions
FIN ACTION
; définition de la fonction maFonction
MODULE maFonction ()
   instructions
FIN MODULE
```

Procédures

L'instruction CALL *label*

Cette instruction est un branchement à l'adresse marquée par *label*.

Traitement de l'instruction :

- PUSH CS:IP, l'adresse de l'instruction suivante est placée sur la pile.
- CS:IP reçoit l'adresse marquée par *label*. L'instruction exécutée après le 'call' est celle en position *label*. (saut)

Exemple

Je place dans l'exemple des numéros de lignes pour faciliter l'explication.
Cet exemple ne contient que la partie du code qui nous intéresse.

```
(1)  ; Exemple de « procédure » servant à l 'affichage d 'une
(2)  ; chaîne de caractères
(3)  MOV  DX,offset texte1
(4)  CALL affiche
(5)  MOV  DX,offset texte2
(6)  CALL affiche
(7)  ; -------- Epilogue ----------
(8)  MOV AX,4C00h
(9)  INT 21h
(10) affiche:
(11)  MOV AH,09h
(12)  INT 21h
(13)  RET
```

Procédures

L'instruction RET

Cette instruction permet le retour à l'instruction suivant l'instruction CALL.

Traitement de l'instruction :

- POP CS:IP, l'adresse de l'instruction suivant le CALL est récupérée et placée dans CS:IP. L'instruction exécutée après le RET est donc celle qui suivait le CALL *label*.

LASEx04.asm

« Exécution » de l'exemple précédent

Ligne 3 :

Ligne 4 : CS:IP est placé sur la pile

CS:IP reçoit la valeur du label *affiche*,

il s'ensuit un saut au label

Ligne 11 : Exécution de la routine jusqu'à la ligne13

Ligne 13: RET : Récupère dans CS:IP le sommet de la pile,

il s'ensuit un saut à l'instruction suivant l'appel …

Ligne 5:

Ligne 6 : CS:IP est placé sur la pile

CS:IP reçoit la valeur du label *affiche*,

il s'ensuit un saut au label

Ligne 11 : Exécution de la routine jusqu'à la ligne13

Ligne 13: RET : Récupère dans CS:IP le sommet de la pile,

il s'ensuit un saut à l'instruction suivant l'appel …

Ligne 8 :

Ligne 9 : Fin du programme

Partage de la pile

- Le 80x86 ne possède qu'une seule pile, attention donc aux conflits entre les instructions PUSH-POP et CALL *label* - RET.
- Conséquences
  - Dans une procédure, autant de PUSH que de POP.
  - Le nombre d 'appels de procédures imbriqués est limité par la directive .STACK

Remarques

- Un « PUSH » sans « POP » dans une procédure …

```
  MOV   DX,100
  CALL  procedure
  …
procedure:
   PUSH  DX
   MOV   DX,SI
   RET
```

Lors de la rencontre avec l 'instruction RET, IP reçoit la valeur 100 (le sommet de la pile) qui n 'est pas l 'adresse de retour de la procédure …erreur.

- La directive **.stack 100h**.

S 'il y a *x* niveaux d 'appel d 'une procédure, cela entraîne *x* sauvegardes d 'adresse de retour sur la pile. Il faudra veiller à choisir une pile assez grande en cas d 'appels récurcifs d 'une procédure

Procédures

Vue plus complète …

- Tasm fournit deux pseudo-instructions, permettant de gérer proprement les procédures.

  PROC et ENDP

- Avantages
  - Ecrire proprement les procédures
  - Permet les labels locaux
  - Permet de définir un point d'entrée dans le programme, pseudo-instruction END

Exemples

• La procédure suivante sera appelée par l'instruction **CALL procedure1**

```
procedure1   PROC
  PUSHa
  … Corps de la procédure …
  POPa
  RET
procedure1   ENDP
```

• Définition d'un point d'entrée. Dans l'extrait de code suivant;

```
main PROC
  …
main ENDP
principale PROC
  …
principale ENDP
END principale
```

Ce code commencera par la procédure « principale » avant la procédure « main » grâce à la pseudo-instruction END.

Les commentaires sont absents dans cet exemple …ils viendront plus tard.

esi

Procédures

Directive LOCALS

- Remplace les labels de la forme *@@xxx* par *#nomprocedure#IP*.

- Conséquence

  Permet d'utiliser des labels de même nom dans des procédures différentes.

LASEx05.asm

Visualisation du problème

Pour implémenter un test, SI ALORS, par exemple, j'utilise, par soucis de clarté et de lisibilité du code, les labels

@@si:

@@alors:

@@sinon:

sans la directive LOCALS, l 'implémentation de deux tests dans une même procédure suscitera une erreur car le compilateur va rencontrer deux labels identiques; *@@si:* et *@@si:* .

En utilisant la directive LOCALS, le compilateur ne générera pas d 'erreur car il rencontrera des labels de la forme; *#module1#IP1:* et *#module2#IP2:*

Procédures et commentaires …

Découper un algorithme en petits blocs indépendants permet d'écrire des programmes modulaires. La modularité facilite la *mise au point*, la *lisibilité*, la *maintenance* et la *réutilisation du code*.

Pour augmenter ces avantages, on ajoute en début de procédure des **commentaires**; *auteur*, *date*, *description*, *paramètres i/o* …, on s'engage également à sauvegarder les registres et à les restaurer en fin de procédure.

Commenter un code …

En tête de programme :

```
; ================================================================
; nomPgm.asm - Titre du programme -2001-2002
;
; Auteur        : PBT
; Date          : 28 Janvier 2002
; Description   : Explication du programme
; ================================================================
```

En début de « zone » :

Les zones sont la déclaration des constantes, le code source, la définition des procédures ...

```
; ================================================================
; Procedure principale
; ================================================================
```

En début de procédure :

```
; ----------------------------------------------------------------
; nomProcedure - Titre de la procédure
;
; Auteur      : PBT
; Description  : Affiche une chaîne de caractère à l'écran.
; Use         : Utilise INT 21h,09h
; IN          : DS:DX, offset de la chaîne à afficher
; ----------------------------------------------------------------
```

Exemple

Calendrier

- Réaliser l'affichage d'un calendrier mensuel avec la date du jour.

LASCal.exe

Définition des constantes et des variables

```
; ---------------------- Constantes -----------------------------
  Bye     DB      'Examen d''ASM',10,13
          DB      'Pierre BETTENS, Septembre 1999',10,13,'$'
  ; Initialise l'extra segment pour la memoire video
  INI     DW      0B800h
  ; Coin superieur gauche du cadre
  Y       EQU     5
  X       EQU     25
  ; Attribut pour l'ecriture dans la memoire video. Deux bytes par
caractères,  l'attribut (RVBfond-RVBchar-Clignote) et le code ascii du
char
  Attribut EQU     00001111b
  Mois    DB      ' Janvier ',' Fevrier ','  Mars    ' ...
  LMOIS   EQU     9
  Jours   DB      'Dimanche','Lundi   ','Mardi   ','Mercredi'
          DB      'Vendredi','Samedi  '
  LJOUR   EQU     8
  Cadre   DB     ...
  LARGEUR EQU     28
  HAUTEUR EQU     12    ; en fait hauteur - 1
  ; ---------------------- Variables -----------------------------
  JSem    DB      ?       ; jour de la semaine courante Di=0, ..
  A       DW      ?       ; annee courante
  M       DB      ?       ; mois courant
  J       DB      ?       ; jour courant 1,2 ..
  Jour    DB      ?       ; utilisee par premierJour et dernierJour
```

Exemple

Analyse du problème

Passer dans le mode vidéo 03h (texte 80x25)

Lire la date du jour et mémoriser les valeurs - `lireDate`

Dessiner le cadre contenant Lu Ma Me … - `dessinCadre`

Compléter le calendrier - `ecrireCal`

- Ecrire le mois courant en haut - `ecrireMois`
- Ecrire la date complète JJJJJJJ JJ MM AAAA en bas - `ecrireDate`
- Remplir le calendrier avec les jours 01 02 03 … - `remplirCal`

Rendre la main

Procédure principale

```
main    PROC
            MOV     AX,@data
            MOV     DS,AX
            ; -------------------- Programme --------------
            ;Mise en place de segment et CLS
            MOV     ES,INI          ; INI vaut 0B800h
            CALL    mode03

            CALL    lireDate
            CALL    dessinCadre
            CALL    ecrireCal
            ; -------------------- Epilogue ---------------
            MOV     AH,00h
            INT     16h
            CALL    Mode03
            MOV     DX,offset Bye
            CALL    WMess
            MOV     AX,4C00h
            INT     21h
main    ENDP
```

Exemple

Points plus difficiles

- Ecrire à l'écran « en position 0B800h »
- Ecriture du mois en cours
  - Utilisation d'un vecteur mois contenant les chaînes de caractères à écrire
- Procédure `remplirCal`
  - Trouver le premier jour du mois *int 21h,2B* (set date) *int 21h,2A* (get date)
  - Ecrire les nombres 01, 02, 03 …
  - Trouver le dernier jour du mois

Langage assembleur - 1ère - PBT

Détails …

• Ecrire à l'écran

Une manière d'écrire à l'écran en mode texte (mode 03h) est d'utiliser la copie des informations écran se trouvant dans la mémoire en position 0B800h:0000h à 0B800h:0FA0h.

Pour ce faire, on positionne l'extra segment ES au « début de l'écran » (**ES <- 0B800h**). Ensuite, sachant qu'un caractère occupe **deux bytes** en mémoire, il suffit d'écrire au bon endroit !

Un caractère est composé d'un **attribut** (1 byte : IRVB(fond)-IRVB (caractère))) et de son **code ascii** (1 byte).

• Ecriture du mois en cours

Il faut définir un vecteur

mois DB ' Janvier ' , ' Février ' , ' Mars ' …

Pour obtenir les codes ascii successifs, il faut se positionner au bon endroit ! Si M=3, le calcul à faire est BX <- offset mois + (M-1)*8

• Procédure `remplirCal`

Pour trouver le premier jour du mois, il suffit de modifier la date système pour qu'elle commence au premier jour, ensuite, on lit le jour que l'interruption retourne et on restaure la date.

Procédures - Passage de paramètres

Passage de paramètres par **registres**

- Lorsque le nombre de paramètres est limité, le passage de paramètres à une procédure peut se faire à l'aide des registres
- Exemple

```
MOV     DX,offset texte
CALL    affiche
```


Exemple

L'extrait de programme suivant fait la somme des éléments d'un tableau.

```
.data
tableau       DW            10,11,12,13,14,15
lTableau      EQU           ($-tableau)/2       ;longueur tableau
resultat      DW            ?                  ;somme des elts
.code
main PROC
  …
  MOV         BX,offset tableau
  MOV         CX,lTableau
  call        somme
  MOV         resultat,AX
  …
main ENDP
; ---------- somme - Fait la somme des elts de tableau -----------------
; in   : BX : adresse du vecteur de word - CX : nbre d 'elts du vecteur
; out  : AX : somme des elts
somme PROC
  MOV         AX,0
@@repeter: ADD AX,[BX]
  ADD         BX,2
  LOOP        repeter
  RET
somme  ENDP
END  main
```

Procédures - Passage de paramètres

Passage de paramètres par **valeurs**

- Une autre solution pour passer des paramètres est d'utiliser la **pile**.
- Principe : *Placer la valeur de chacun des paramètres sur la pile, appeler la procédure. Nettoyer la pile.*
- Ce sont des **valeurs** qui sont placées sur la pile, l'original n'est pas modifié.

Conventions

Lorsque l'on dépose des valeurs sur la pile, il faut les récupérer d'une manière ou d'une autre et restaurer la pile.

Pour un compilateur C, c'est la fonction **appelante** qui nettoie le pile tandis que pour un compilateur PASCAL, c'est la procédure **appelée** qui le fait.

Comment nettoyer la pile ?

• Soit autant de POP que de PUSH pour rétablir la pile lors du retour de la procédure.

```
PUSH P1
PUSH P2
CALL fonction
POP P2
POP P1
```

• Soit SP <- SP + 2*nombre de paramètres

```
PUSH P1
PUSH P2
CALL fonction
ADD   SP,4
```

• Soit l'instruction **RET n** où n=2*nombre de paramètres

```
PUSH P1
PUSH P2
CALL fonction            la fonction ' fonction ' se termine par RET 4
```

Passage de paramètres par **valeurs** (suite)

- Utilisation du **registre BP**
  (associé à SS par option)
- Exemple

```
push    P1
push    P2
push    P3
call    somme
ADD     SP,6
```

LASEx06.asm

Procédures - Passage de paramètres

Passage de paramètres par **adresses**

- Lorsque l'on désire **modifier la valeur** d'un paramètre, la pile reçoit, non pas la valeur, mais l'adresse du paramètre.
- Un paramètre doit être une variable (un littéral n'a pas d'adresse).
- Exemple

```
LEA   AX,var1
PUSH  AX
CALL  procedure
```

Passage de paramètres par **adresses** (suite)

- Utilisation

  MOV BX,[BP+4]

  BX reçoit l 'adresse d 'une variable

  INC [BX]

  J'incrémente le contenu de la variable de 1.

LASEx07.asm

Procédures - Variables locales / globales

Variables globales

- Visibilité : programme complet
- Portée : programme complet
- Définition
  - En tête du programme
  - Par les mots-clés DB, DW ou DD
  - Elles sont associées au segment DS

Remarque

Il est possible d 'intégrer les variables au segment de code.

```
.code
JMP @@code
v1          DB          10
v2          DW          20
@@code:
```

Etant associées au segment de code, leurs adresses sont bien **CS:offset v*i*** et non DS:offset v*i*.

Procédures - Variables locales / globales

Variables locales

- Visibilité : corps de la procédure (non-accessibles par les autres procédures)
- Portée : corps de la procédure
- Utilisation : Pour mettre en place ce mécanisme, le programmeur utilise la PILE. Les variables locales sont placées sur la pile en début de procédure et libérées en fin.

Exemple d 'utilisation de variables locales.

Supposons qu'il y ait 2 paramètres par valeurs passés sur la pile et 1 variable locale.

Le code (non-commenté) de la procédure sera de la forme :

```
exemple PROC
 PUSH       BP
 MOV        BP,SP
 SUB        SP,2        ; réservation d 'une place sur la pile
 PUSHa
 ; corps de la procédure
 MOV        word ptr [BP-2],10       ; initialisation var
locale
 MOV        AX,[BP+4]                ; utilisation ds
paramètres
 MOV        BX,[BP+6]

 ; fin de procédure
 POPa
 ADD        SP,2
 POP        BP
 RET        4
exemple ENDP
```

Variables locales (suite)

Pour apporter de la lisibilité à l 'exemple qui précède, on utilise des constantes (EQU)

```
local1  EQU  -2
local2  EQU  -4
…
MOV  word ptr loacal1[BP],10
```

En effet local1[BP]=-2[BP]=[BP]-2=[BP-2]

Exemple d 'utilisation de variables locales.

Les modifications par rapport à l 'exemple précédent sont en gras.

```
exemple PROC
local1      EQU         -2
 PUSH       BP
 MOV        BP,SP
 SUB        SP,2        ; réservation d 'une place sur la pile
 PUSHa
 ; corps de la procédure
 MOV        word ptr local1[BP],10   ; initialisation var
locale
 MOV        AX,[BP+4]                ; utilisation ds
paramètres
 MOV        BX,[BP+6]
 ADD        BX,local1[BP]

 ; fin de procédure
 POPa
 ADD        SP,2
 POP        BP
 RET
exemple ENDP
```

Variables locales (suite)

- TASM fournit une directive supplémentaire, **LOCAL**, permettant d'accroître la lisibilité lors de l'utilisation de variables locales.

  LOCAL l1:BYTE,l2,l3:DWORD,l4:BYTE:51=taille

- Attention cependant, cette directive **ne modifie pas SP**. Veiller à ajouter
  - SUB SP,taille ; au début de la procédure
  - ADD SP,taille ; à la fin de la procédure

Explications

`LOCAL l1:BYTE,l2,l3:DWORD,l4:BYTE:51=taille`

l1 - Déclaration d'une variable d'un byte mais réservation d'un word car se place sur la pile.

l2 - Déclaration et réservation d'un word par défaut

l3 - Déclaration et réservation d'un double

l4 - Déclaration et réservation d'un vecteur de 51 bytes.

taille - espace occupé sur la pile

Exemple d'utilisation de variables locales.

```
exemple PROC
LOCAL l1,l2:byte=taille
 PUSH          BP
 MOV           BP,SP
 SUB           SP,taille
 PUSHa
 ; corps de la procédure
 MOV           l1,10                    ; initialisation var locale
 MOV           AX,[BP+4]                ; utilisation ds paramètres
 MOV           BX,[BP+6]
 MOV           CX,l1
 MOV           DL,l2
 ; fin de procédure
 POPa
 ADD           SP,taille
 POP           BP
 RET
exemple ENDP
```

Récursivité

- Principe
  - Il y a **récursivité** directe lorsqu'une procédure s 'appelle elle-même.
- Mise en œuvre
  - Assembleur permet les appels récursifs de procédures … en fonction de la taille de sa pile.
- Mise en garde
  - Dans cette structure, il faut veiller à l '**arrêt** des appels récursifs !

Remarques

• Un langage de programmation permet la récurcivité lorsqu 'il possède et gère une pile. En effet, les valeurs intermédiaires des différents appels récurcifs sont placées sur la pile … et récupérées au fur et à mesure.

• Le principe de récurcivité est le report d 'un problème à son cas trivial. Ce problème se pose lorsque l 'on veut calculer une valeur dépendant d 'un (ou de plusieurs paramètres). Ce qu 'il faut se dire c 'est : « Si je connais la réponse pour le cas ' n- 1 ' alors, c 'est simple ».

Observons le calcul de la factorielle.

Calculer 5! est simple si je connais 4! (et oui, c 'est 5* 4!). Je retiens 5

Calculer 4! est simple si je connais 3!. Je retiens 4

…

Calculer 0! est simple c 'est 1.

Le résultat de mon calcul est donc 1*2*3*4*5 … toutes les valeurs que j 'ai retenues … sur la pile.

• … en fonction de la taille de la pile …

Lorsque j 'écris une fonction récurcive, je ne sais pas, à priori, combien d 'appels récurcifs je vais faire. Or, à chaque appel, des paramètres sont placés sur la pile … ces paramètres seront récupérés lorsque je rencontre le point d 'arrêt de mes appels (le cas trivial).

Structure récursive

- Une structure récursive a la forme suivante :

```
SI <condition triviale> ALORS
    <traitement du cas trivial>
SINON
    <appel récursif>
FIN SI
```

- Exemple : Calcul de la factorielle.

```
SI n=0 ALORS
   factorielle = 1
SINON
   factorielle = n * factorielle(n-1)
FIN SI
```

LASEx08.asm

Exemples

- Calcul de la factorielle

n! = n * (n-1)!

0! = 1

- Somme des n premiers nombres entiers

somme (n) = n + somme (n-1)

somme (0) = 0

Traduisons en logique

```
SI n=0 ALORS
  somme=0
SINON
  somme = n + somme (n-1)
FIN SI
```

- Les nombres de Fibonacci

$$\begin{cases} F_n = F_{n-1} + F_{n-2} \\ F_1 = 1 \\ F_0 = 0 \end{cases}$$

```
SI n<2 ALORS
  SI n=0 ALORS fibo = 0
  SINON fibo = 1
  FIN SI
SINON
  fibo = fibo(n-1) + fibo (n-2)
FIN SI
```

- Les tours de Hanoi ...

Compléments sur les interruptions

- Lorsqu'une interruption est détectée, le SE exécute une routine.
- L'adresse de cette routine est stockée dans la table des interruptions.
- Cette adresse peut être redirigée …
  - int 21h,35h
  - int 21h,25h

`LASEx10.asm`

`LASEx11.asm`

Langage assembleur - 1ère - PBT

Exemple 10

L'interruption **int 23h** n'en n'est pas une ! C'est un pointeur vers une routine traitant le Ctrl-C. Cette routine est appelée par une routine gérant les séquence d'échappement au clavier.

Le but de l'exercice est de rediriger l'adresse de cette routine afin que lors de l'appui sur Ctrl-C, ce soit ma routine qui soit exécutée.

Ceci se fait simplement en changeant le vecteur d'interruption dans la table en utilisant les instructions **int 21h,35h** et **int 21h,25h**.

Exemple 11

Pour un exercice, nous devions afficher une image à l'écran 18 fois / seconde. Comment gérer cette fréquence en utilisant les timers et l'horloge interne ?

Une manière de faire est d'utiliser l'instruction **int 1Ch**. Cette interruption est appelée par le système 18 fois par seconde pour faire … qu'importe.

Le but de l'exercice est de rediriger cette interruption vers ma routine pour que ma routine soit exécutée 18 fois / seconde. Ma routine, quant-à-elle appellera (si je sauvegarde l'adresse) la routine initiale après … pour que tout se passe bien.

Segmentation mémoire

- La mémoire est découpée en segments.
- A ces segments sont associés des registres : CS, DS et SS.
- Pour définir les adresses physiques de ces segments logiques, assembleur propose deux manières de faire :
  - Les directives de segmentation **simplifiées**
  - Les directives de segmentation **standards**

Segmentation mémoire

• Les segments logiques contiennent; **le code**, **les données** et **la pile**. A chacun des ses segments est associé un registre de segment, respectivement, **CS**, **DS** et **SS**. Ils contiennent les adresses physiques des segments logiques. L 'adresse du début du segment sera représentée par CS:0000h par exemple, c 'est pourquoi l 'adresse du début d 'un segment est toujours un multiple de 16 … mais c 'est une histoire déjà racontée !

• **Les directives simplifiées** utilisent les même conventions que celles utilisées par les langages de haut niveaux.

• **Les directives standards** permettent un meilleur contrôle mais sont plus lourdes.

La directive **DOSSEG**.

Indique que les segments doivent être organisés selon l 'ordre conventionnel de MS-DOS. Les segments sont organisés dans l 'ordre :

- Le code segment

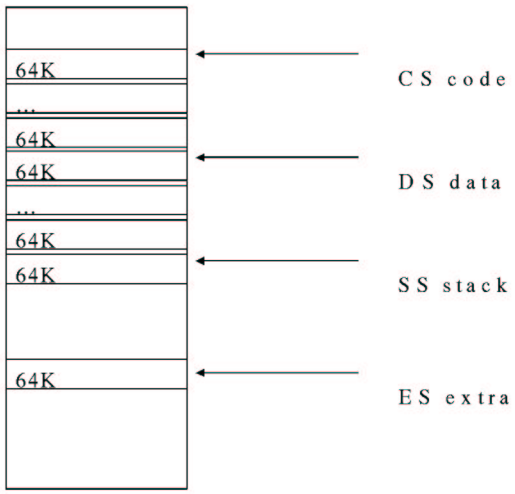
- Les DATA segment :

  . Les segments non BSS ou Stack

  . Les segments BSS (segments de données non-initialisées)

  . Les segments STACK

Segments logiques



Segmentation mémoire (3)

Les directives de segmentation simplifiées

- DOSSEG
- .MODEL *model mémoire*

  où *model mémoire* prend les valeurs :
  - tiny
  - small
  - medium
  - compact
  - large
  - huge
  - flat

La directive **MODEL**.

Définit les attributs de l 'ensemble du module : modèle mémoire, valeur par défaut des appels (near ou far) et type du stack.

Cette directive **doit** apparaître avant tout autre directive de segmentation simplifiée.

TINY : Programme et données partagent le même segment -> le programme doit être < 64K. Le code doit commencer à l 'adresse 100h (pour créer des .com). Toutes les adresse sont de type NEAR. L 'exécutable peut être transformé en .com grace à EXE2BIN ou avec /t à la compilation.

SMALL : Le segment de code et le segment de données ne doivent pas dépasser 64K. Toutes les adresses sont de type NEAR.

MEDIUM : La zone de données < 64K. Le code peut s 'étendre sur plusieurs segments physiques.

COMPACT : Le code est limité à 64K, mais les données peuvent s 'étendre sur plusieurs segments physiques.

LARGE : Le code et les données peuvent occuper plusieurs segments physiques.

HUGE : = LARGE

FLAT : Configuration non segmentée pour une architecture et un OS 32 bits (windows NT). Le code et les données sont rassemblés dans un seul segment 32 bits.

Segmentation mémoire (4)

Les directives de segmentation simplifiées (2)

- .x86
- .8087 - .287 - .387
- .DATA - .DATA?
- CONST
- .FARDATA - .FARDATA?
- .STACK
- .CODE
- END

Les directives .x86, .8087, .287 et .387

Permettent de spécifier un type de µ-processeur et/ou co-processeur.

Les directives .DATA et .DATA?

Crée un segment de données de type NEAR. Ce segment peut occuper 64K en MS-Dos et 512Mb en windows NT.

Ce segment est placé dans un groupe spécial de 64K ; DGROUP

Les directives .FARDATA et .FARDATA?

Idem que pour les directives .DATA et .DATA? Mais pour les modèles COMPACT, LARGE et HUGE. Les adresses seront de type FAR.

L'initialisation du registre de segment DS se fait comme suit :

```
MOV         AX,@fardata
MOV         DS,AX
```

La directive .STACK

Permet de réserver une zone mémoire pour la pile. Par défaut, la taille de la pile est de 1K. Pour créer une pile de taille différentes, il faut préciser la taille.

Les directives .CODE et END

.CODE signale le début du segment contenant le code, tandis que END signale la fin de ce segment.

Segmentation mémoire (5)

Les directives de segmentation standards

- Définition d ’un segment

```
nom  SEGMENT [alignement][combinaison][‘ Classe ’]
  instructions
nom  ENDS
```


Définition d ’un segment

nom définit le nom du segment. A l ’intérieur d ’un même module, tous les segments de même nom sont traité comme s ’ils n ’en faisaient qu’un.

[alignement] : **BYTE, WORD, DWORD, PARA** (adresse suivante de paragraphe, multiple de 16)**, PAGE** (page suivante, multiple de 256).

[combinaison] : Définit comment l ’éditeur de lien doit combiner les segments de même noms.

PRIVATE	: Ne combine pas, même si même nom.
PUBLIC	: Concaténation des segments de même nom.
STACK	: Désigne un segment de pile … ts ces segments sont rassembles à l ’édition de liens.

[classe] : Aide au contrôle de l ’ordre des segments. Deux segments de même noms ne seront pas combinés s ’ils sont de classes différentes. Les segments de même classe sont placés côte à côte . A l ’intérieure d ’une même classe, les segments sont arrangés suivant l ’ordre dans lequel l ’assembleur les rencontre (.ALPHA, .SEQ, .DOSSEQ).

Segmentation mémoire (6)

Les directives de segmentation standards (2)

- La directive ASSUME

```
ASSUME reg_segm : localisation [, …]
```

- Groupe de segments

```
nom GROUP segment[[,segment],…]
```

LASEx09.asm

Directive ASSUME

Beaucoup d 'instructions assembleur font référence à un segment par défaut.

```
MOV         AX,variable
```

Suppose que la donnée (variable) soit dans un segment associé à DS … et va chercher l 'information en DS:offset variable.

La directive ASSUME fait le lien entre un registre de segment et le segment associé.

```
ASSUME DS:DSEG
```

reg_segm : Nom d 'un registre de segment (DS,ES,CS,SS)

localisation : Nom du segment (ou d 'un groupe) qui doit être associé au registre de segment *reg_segm*.

ASSUME NOTHING annule toutes les options prises précédemment par les directives ASSUME.

Définition d 'un groupe de segments

Un groupe de segment est un ensemble de segments dont la taille ne dépasse pas 64K en mode 16 bits. Un programme adresse un code ou une donnée en référent le début du groupe.

Cette notion permet de développer des segments logiques différents pour différentes sortes de données et ensuite de les combiner pour y accéder grace à un seul registre de segment.

Format des données (1)

- Le co-processeur mathématique va permettre les calculs sur les réels . . . plus exactement, les nombres en **virgule flottante.**
- Format mathématique :

$$\pm \; mantisse \; 2^{exposant}$$

Représentation des nombres en virgule flottante.

Rappelons tout d 'abord, que cette représentation permet de représenter un sous-ensemble des nombres réels. Dans une représentation mathématique normalisée, un nombre en virgule flottante se représente comme suit :

`mantisse * base`$^{\texttt{exposant}}$

Quelques adaptations s 'imposent pour une représentation machine.

mantisse : Dans sa forme normalisée, elle est de la forme ±1.xxx…

Le signe sera déterminé par un bit placé à 0 ou 1.

Dans ce format, une mantisse nulle n 'est pas acceptée.

base : Mathématiquement c 'est 10 mais en machine, la base est évidemment 2.

exposant : Cet exposant sera limité dans son nombre de chiffres.

Ces considérations limitent le nombre de nombres que l 'on peut représenter. L 'erreur en base 10 avec une mantisse de 10 chiffres et un exposant de 2 chiffres serait :

Nombre > 1 : $1.000000000\ 10^{00}$

Son successeur : $1.000000001\ 10^{00}$

L 'erreur relative est de l 'ordre de 10^{-9} .

Format des données (2)

Le 80x87 utilise la représentation **80 bits**.

signe (1)	**exposant** (15)	**mantisse** (64)
0 = +	exposant de 2	< 2 normalisée
1 = -	l ' excédent de 16383	1.x…x

Exemple $4001C000000000000000_{16}$

$0100\ 0000\ 0000\ 0001\ 1100\ 0000\ \ldots\ 0000_2$

signe : 0 -> +

exposant : 100 0000 0000 0001 = 16385 -> 2

mantisse : $1.10\ldots0_2 = 1.5_{10}$

$+ 1.5 * 2^2 = \mathbf{6.0}$

Nombres représentables

Borne supérieure

$0111\ 1111\ 1111\ 1110\ 1111\ \ldots\ 1111_2 = 7FFEFFFFFFFFFFFFFFFFh$

$+ 1.11\ldots1\ 2^{1\ldots10} \approx + 2\ 2^{16383} = 1.1897\ 10^{4932}$

16383 car $11111111111110_2 = 32766_{10} = 16383+16383$

Borne inférieure

$1111\ 1111\ 1111\ 1110\ 1111\ \ldots\ 1111_2 = FFFEFFFFFFFFFFFFFFFFh$

$- 1.11\ldots1\ 2^{1\ldots10} \approx - 2\ 2^{16383} = - 1.1897\ 10^{4932}$

Plus petit nombre positif représentable

$0000\ 0000\ 0000\ 0001\ 1000\ \ldots.\ 0000_2 = 00018000000000000000h$

$+ 1.0\ldots0\ 2^{0\ldots01} \approx + 1\ 2^{-16382} = 3.362\ 10^{-4932}$

Plus petit nombre > 1

$0011\ 1111\ 1111\ 1111\ 1000\ \ldots\ 0000_2 = 3FFF8000000000000000h$

$011\ 1111\ 1111\ 1111_2 = 16383_{10}$ -> représente 0

$+ 1.0\ldots0\ 2^{01\ldots1} \approx + 1\ 2^0 = 1\ 10^0$

Son successeur …

$0011\ 1111\ 1111\ 1111\ 1000\ \ldots\ 0001_2 = 3FFF8000000000000001h$

$+ 1.0\ldots01\ 2^{01\ldots1} \approx + 1.00000000000000001084$

Remarque : Exposant = 7FFFF représente +∞ et 0000 représente -∞

Registres de données

- Le 80x87 dispose de 8 registres de 80 bits:

  St(0), St(1) … St(7)

- Ces registres contiennent des nombres en virgule flottante
- Se suivent de manière cyclique (*après St(7), vient St(0)*)
- St(i) n'est pas toujours le même registre physique -> SW.TOP

St(0)

Ce registre est le registre privilégié, beaucoup d'instructions l'utilisent.

SW.TOP

La correspondance entre les registres physiques et logiques (St(i)) se fait grâce au nombre SW.TOP.

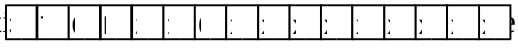
Si SW.TOP=0, ST(0) correspond au registre 0, St(1) au registre 1 …

Si SW.TOP=1, ST(0) correspond au registre 1, ST(1) au registre 2 …

…

Deux instructions permettent de modifier SW.TOP; FINCSTP et FDECFTP.

Registre spécial SW - Status Word

Ce registre cont  résultat d'une comparaison.

Les bits 3,2,1 et 0 sont des bits résumant une comparaison.

Leurs position est telle que si l'on copie le registre SW dans le registre FLAGS du 80x86, on peut les utiliser comme si on avait fait un CMP … on peut donc utiliser les jumps utilisant ces flags.

Registre spécial TW - Tag Word

Contient une description, sur **2 bits**, du contenu des 8 registres de données.

Cette description signifie :

00 : st(*i*) sontient une donnée valide

01 : st(*i*) = 0

10 : st(*i*) contient un nombre spécial (+∞ …)

11 : st(*i*) vide

Registre spécial CW - Control Word

Contient des inf[illegible]rgule flottante.

AA : Arrondi

00 : au plus proche

11 : tronqué 01/10 : non utilisé

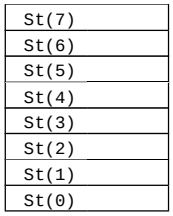
PP : Précision lors d ’un transfert

00 : 24 bits (réels courts) 01 : réservé

10 : 53 bits (réels longs) 11 : 64 réels temporaires

Pile du 80x87

- Les registres St(i) sont accessibles comme une pile.
- Le 80x87 fonctionne en notation polonaise inversée NPI (ou RPN).
  - FLD *var*
  - FSTP *var*
  - FADD



Fonctionnement RPN

FLD *var*

L 'instruction permet de charger le contenu d 'une variable (réelle) au sommet de la pile avec décalage.

St(0) prend cette valeur

St(1) <- St(0)

…

St(7) <- St(6) (et St(7) est perdu)

FSTP *var*

L 'instruction permet de décharger le contenu du registre St(0) vers une variable avec décalage.

St(0) <- St(1)

St(1) <- St(2)

…

St(6) <- St(7) (et St(7) est vide)

FADD, FMUL, FDIV, FSUB

Les opérations (par exemple l 'addition) s 'effectuent sur le sommet de la pile.

St(0) <- St(0) + St(1)

St(1) <- St(2) …. et décalage ...

Le co-processeur mathématique 80x87

Familles d'instructions

- Les instructions se décodent aisément grâce à des mnémoniques.
  - F : toutes les instructions commencent par F
  - LD : LOAD - chargement sur la pile
  - ST : STORE - déchargement de la pile
  - I : INTEGER - instruction traitant un entier
  - P : POP - shift de la pile vers le bas (décalage)
- Autres : ADD, SUB, COS, SQRT, COMP, ...

Instructions, exemples

- Instructions de transfert : FLD, FILD, FSTP, FISTP, FST, …
- Calculs simples : FADD, FMUL, FDIV, FSUB, ...
- Calculs sur les entiers : FIADD, FIMUL, FIDIV, FISUB, …
- Chargement de constantes : FLDZ, FLDPI, FLDLN2, …
- Calculs complexes : FSQRT, FSIN, FCOS, …
- Comparaisons : FCOM, FCOMP, FCOMPP, FICOM, …
- Instructions de contrôle : FINIT, FINCSTP, …

Le co-processeur mathématique 80x87

Utilisation de TD

- Comme pour tout programme, il est possible de tracer le co-processeur mathématique.

  view numeric processor

- Evaluation d'une équation du second degré : $x^2+4x-10$

  `LASEx12.asm`

Langage assembleur - 1ère - PBT

TD

Turbo Debugger permet de tracer un programme écrit pour le 80x87.

View Numeric Processeur montre :

- Les registres St(*i*), par registre,

  . Son tag (zéro, vide, NAN, correct)

  . L 'interprétation du contenu en décimal

  . Et en hexadécimal (il faut agrandir la fenêtre).

- D 'autres informations; l 'arrondi, la précision, la valeur de TOP, …

Exemples

- Illustration des méthodes d'arrondis en utilisant le calcul de $10000*\sin^2(x/10000)$ et $10000*\cos^2(x/10000)$
  - Au plus près `LASEx13.asm`
  - Tronqué `LASEx14.asm`
- Calcul d'une racine carrée par approximation successives.
  `LASEx15.asm`

Masque

Pour modifier un ou plusieurs bit(s) d'un registre, on utilise les **masques** et les opérations **AND** et **OR** sur les registres.

Dans l'exemple,

```
AND         AX,1111001111111111b
```

place les bits 10 et 11 à 0 et conserve la valeur des autres.

```
OR          AX,0000110000000000b
```

place les bits 10 et 11 à 1 et conserve la valeur des autres.

Calcul d'une racine carrée

Dans l'exemple, on utilise la formule de récurence :

$x_n=((x/x_{n-1})+x_{n-1})/2$

Pseudo-instruction DT

Pour définir une zone mémoire de 80 bits qui pourra mémoriser un nombre représenté en virgule flottante, on utilise la pseudo-instruction **DT**(define temp).

Exemples

- Conversion d'une vitesse données en km/h en m/s

  LASEx16.asm

- Comparaison de π avec 22/7

  LASEx17.asm

Comparaison de π avec 22/7

L 'instruction FCOMPP permet de comparer le contenu de ST(0) avec le contenu de son opérande (par exemple St(1)) et d 'effectuer un double POP. Cette instruction met à jour trois flags dans le Status Word (SW) dans le 80x87.

```
FCOMPP      St(1)
```

Ce registre de flags, je peux le stocker dans un registre via

```
FSTSW       AX
```

Pour pouvoir utiliser les instruction *Jf*, il faut mettre les flags du 80x86 à jour, via

```
SAHF
```

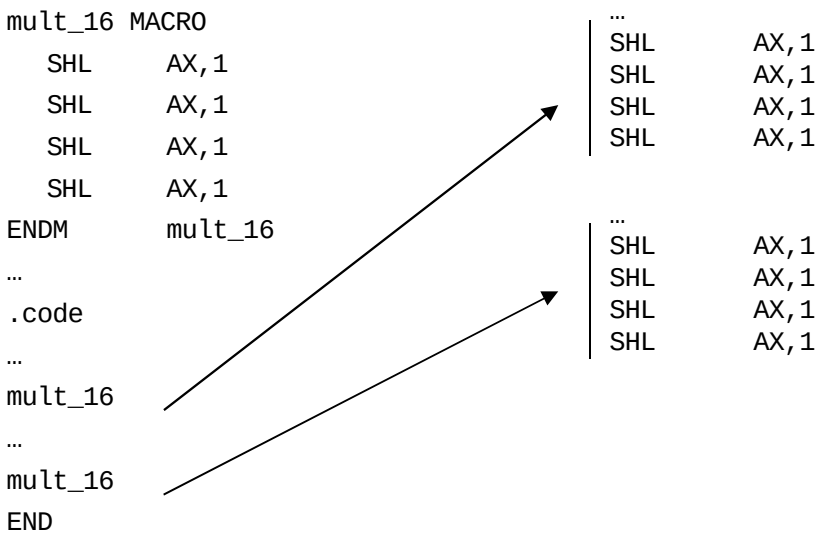
instruction qui recopie le contenu de AH dans le registre flags du 80x86.

Les macros

- Définition
  - Une *macro* est un symbole qui remplace un ensemble de lignes de code dans le programme.
- Dans le code compilé, les lignes où le nom de la macro apparaît seront remplacées par le corps de la macro.
- Différence entre une **procédure** et une **macro**.
- Exemple : *mult_16*

LASEX18.asm

Exemple



Remarques

Afin de bien visualiser l'expansion des macros, consultez TD mais aussi les fichiers .1st

Les macros

- Commentaires dans les macros.
  - **;** permet l 'insertion d 'un commentaire qui sera répété à chaque expansion de la macro.
  - **;;** permet l 'insertion d 'un commentaire qui ne sera **pas** répété lors de l expansion.
- Blocs de répétition - **rept** et **endm**

```
mult_16  MACRO
  rept 4
  shl  AX,1
  endm
ENDM     mult_16
```

Passage de paramètres

- Il est possible de passer un/des paramètre(s) lors de l'appel de la macro.
- Exemples

```
mult  MACRO  n
  rept n
  shl  AX,1
  endm
ENDM     mult
```

```
video MACRO n
  MOV AH,00h
  MOV AL,n
  INT 10h
ENDM  video
```


Remarques

En fonction du paramètre, la macro *'mult'* exécutera une multiplication par 2, 4, 8, 16, …

L'appel de la macro nécessitera l'ajout du paramètre.

Exemples:

```
mult 2
video 12h
mult 3
video 3
```

Les macros peuvent accepter un nombre illimité, la seule limitation est que les paramètres doivent tous se trouver sur **la même ligne**. Turbo assembleur limite la ligne à 255 caractères.

Assembleur ne signale aucune erreur lorsque l'on omet un paramètre lors de l'appel de la macro … mais si rept n'a pas de valeur, le compilateur signale une erreur.

Test de paramètres

- **ifb / ifnb**

  If Blank / *If Not Blank* permet de tester la présence d'un paramètre.
- **ifdif** : if different case sensitive
- **ifidn** : if identical case sensitive
- **ifdifi** : if different non case sensitive
- **ifidni** : if identical non case sensitive
- … **elseifdif, elseifidn, elseifdifi,elseifidni**

Exemple

```
mult      macro     n
   ifb    <n>
          err
          display "Nombre de répétitions absent"
   else
          rept      n
          shl       ax,1
          endm
   endif
endm      mult
```

L'opérateur %

- Opérateur d'évaluation d'expressions
- Lorsque l'on veut utiliser une macro texte définie par EQU
- En supposant que la macro soit définie

```
msg  EQU 'Hello'
.code
…
message %msg
```


Le problème peut subvenir lorsque l'on appelle la macro avec 'msg' plutôt que "Hello"

Première solution.

Ecrire le % lors de l'appel de la macro:

message %msg

Deuxième solution.

Prévoir le % dans le corps de la macro.

```
            message     macro       argument
            ifb         <argument>
                        display     "Pas de paramètres"
%           elseifidni              <argument>,<"Hello">
                        display     "Bonjour"
%           elseifdifi              <argument>,<"Bye">
                        display     "Je ne comprend pas"
            else
                        diplay      "Au revoir"
            endif
            endm        message
```

Labels locaux

- Puisqu'une macro est étendue, les labels qu'elles contient sont répétés . . . problème !
- Directive **local**

```
            maMacro             MACRO
            local  suite
            …
            jne    suite
            …
suite:
            …
            ENDM   maMacro
```


Remarque

Turbo assembleur remplace les labels par **??nnnn** où n est un chiffre. Il crée ses labels sous cette forme … évitez donc d'utiliser des labels commençant par **??** !

Les macros

- Répétitions indéfinies
- Appel d'une macro dont les instructions vont dépendre du nombre de paramètres
- Opérateur **irp**

```
pushReg     MACRO       RegList
      irp reg,<RegList>
        push      reg
      endm
ENDM        pushReg
```

- Appel : `pushReg <ax,bx,si,di,bp>`

Manipulation des chaînes de caractères

- Invariants
  - Opérande **source** pointé par **DS:SI**
  - Opérande **destination** pointé par **ES:DI**
  - Les registres d'index (SI,DI) sont modifiés *après* exécution de l'instruction
    - Incrémenté si DF = 0 où DF = Flag de Direction
    - Décrémenté si DF = 1
  - CLD positionne DF à 0
  - STD positionne DF à 1

Manipulation de chaînes de caractères

- Instructions
  - LODS : Charge AL, AX ou EAX
  - STOS : Copie AL, AX ou EAX
  - CMPS : Comparaison mem / mem
  - SCAS : Comparaison mem / regA
  - MOVS : Copie mem / mem
  - INS / OUTS : Transfert depuis/vers un port
- Préfixe de répétition. Répète CX fois
  - REP / REPE ou REPZ / REPNE ou REPNZ

Manipulation de chaînes de caractères

- Instructions LODS

  LODSB

  – Le registre AL est chargé avec DS:SI

  – Si FD = 0 -> SI+=1 sinon SI-=1

  LODSW

  – Le registre AX est chargé avec DS:SI

  – Si FD = 0 -> SI+=2 sinon SI-=2

  LODSD

  – Le registre EAX est chargé avec DS:SI

  – Si FD = 0 -> SI+=4 sinon SI-=4

Manipulation de chaînes de caractères

- Instructions STOS

  STOSB - STOSW - STOSD

  – [ES:DI] reçoit le contenu de AL - AX - EAX

  – Si FD=0 DI+=1-2-4 sinon DI-=1-2-4

- Instructions SCAS

Modifie les flags O,S,Z,A,P,C

SCASB - SCASW - SCASD

– Comparaison AL - AX - EAX avec [ES:DI]

– SI FD=0 DI+=1-2-4 sinon DI-=1-2-4

Manipulation de chaînes de caractères

- Instructions CMPS

  CMPSB - CMPSW - CMPSD

  – Comparaison [DS:SI] avec [ES:DI] *(byte, word, dword)*

  – SI FD=0 DI+=1-2-4 et SI+=1-2-4

  – sinon DI-=1-2-4 et Si-=1-2-4

- Instructions MOVS

  MOVSB - MOVSW - MOVSD

  – [DS:SI] est copié dans [ES:DI] *(byte, word, dword)*

  – SI FD=0 DI+=1-2-4 et SI+=1-2-4

  – sinon DI-=1-2-4 et Si-=1-2-4

Préfixe **REP**

- Toutes ces instructions peuvent être précédées d'un préfixe REP, REPZ, REPNZ, REPE, REPNE
  - REP : CX fois ou ECX fois
  - REPZ ou REPE : CX fois et tant que FZ = 0
  - REPNZ ou REPNE : CX fois et tt que FZ <>0

Manipulation de BITS

- Sélection d'un ensemble de bits par masque
  - AND destination,source
  - OR destination,source
  - XOR destination,source
  - NOT opérande
- L'utilisation d'un *masque* permet de positionner des bits à une valeur prédéfinie

Opérations élémentaires

Les opérations de base sur le bits sont le ET, le OU, le NON OU et le NON. Ces opérations sont directement liées à l'algèbre de Boole.

Le ET effectue un **et logique** entre chacun des bits des deux opérandes. Le résultat est placé dans l'opérande *destination*.

… idem pour les autres opérateurs.

Exemples

AND AL,0Fh - positionne les 4 bits de gauche à zéro, les 4 bits de droite restent inchangés.

OR AL,0Fh - positionne les 4 bits de droite à un, les 4 bits de gauche restent inchangés.

XOR AL,0Fh - inverse les 4 bits de droite et laisse les 4 bits de gauche inchangés.

NOT AL - inverse tous les bits de AL

Manipulation de BITS

Shift - *Décalage d'un nombre de bits*

- Opérateurs : SHR - SAR - SHL - SAL
- SHx = shift, SAx = shift arithmétique
- Sxx opérande1,opérande2
  - opérande2 = 1, CL, n
  - Multiplication ou division par puissance de 2,
  - OF n'a pas de sens, sauf pour 1, il récupère le 'bit perdu'
  - limité aux 4 bits de droite de CL ou de n

Remarque

La différence entre le **shift** et le **shift arithmétique** est que le second recopie le bit de signe tandis que le premier remplit l'opérande par un ou plusieurs zéro.

Manipulation de BITS

Rotation - *Rotation d'un nombre de bits*

- Opérateurs : RCR - RCL - ROR - ROL
- RCx = rotation avec CF, ROx = rotation
- Rxx opérande1,opérande2
  - opérande2 = 1, CL, n
  - OF n'a pas de sens, sauf pour 1, il récupère le 'bit perdu'
  - limité aux 4 bits de droite de CL ou de n

Remarque

La différence avec le *shift* est que la rotation permet de remplir l'opérande avec le bit sortant. Soit en passant par le CF soit non.

Manipulation de BTS

Test du bit n°*i* (386)

BTx opérande,i

- BT : CF = bit n°i, op est inchangé, i = val / reg
- BTC : idem BT mais le bit est complémenté
- BTR : idem BT mais le bit est mis à 0 (reset)
- BTS : idem BT mais le bit est mis à 1 (set)

(à retenir pour le cours de 2ième)

Exemple

```
            MOV         CX,16
            MOV         AX,1110111011110101b
continue:
            BT          AX,CX
            LOOP        continue
```

Le CF vaudra successivement : 10101111011110111

Si l'on remplace BT par BTR, le bit est reset, et donc à la fin AX vaut zéro.

Manipulation de BITS

Recherche du premier bit à 1 (386)

- BSF opérande1,opérande2
  - Bit Scan Forward
  - Scan de droite -> gauche
  - opérande1 (registre) <- n° du premier bit à 1
  - si opérande2 = 0 alors ZF = 0 sinon ZF = 1
- BSR opérande1,opérande2
  - idem mais de gauche -> droite

Exemple

```
MOV         BX,0000111010000000b
BSF         AX,BX       ; AX <- 7, ZF = 0
BSR         AX,BX       ; AX <- 11, ZF = 0
XOR         AX,AX
XOR         BX,BX
INC         AX
BSF         AX,BX       ; ZF = 1, AX inchangé
BSR         AX,BX       ; ZF = 1, AX inchangé
```

Manipulation de BITS

Représentation BCD

(Binary Coded Digits)

- Utilisée pour faire des opérations sur les chaînes de caractères numériques
- Représentation ASCII réduite aux chiffres
- Exemple : Représentation de la valeur 13
  - en hexadécimal, représenté par **000Dh**
  - en BCD, représenté par **0103h**

Langage assembleur - 1ère - PBT

Exemple

```
        MOV     AX,0Dh      ; AX <- 0Dh
        AAA                 ; AX <- 0103h
        XOR     AX,3030h    ; AX <- 3133h
ou alors ADD    AX,3030h
```

Manipulation de BITS

Représentation BCD

AAA - AAS - AAM - AAD

- Instructions permettant la manipulation des chaînes de caractères numériques
- *Adjust after addition / soustraction / multiplication / division*

Manipulation de BTS

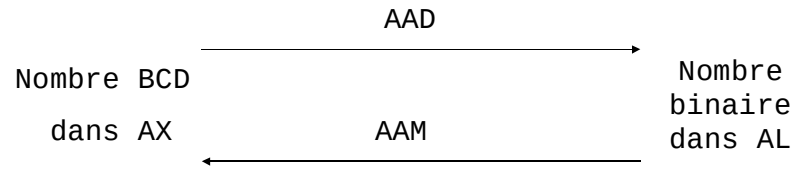
AAA - AAS

- La somme de deux nombres BCD n'est pas un nombre BCD -> il faut ajuster la représentation après l'opération.
- Exemples

```
  3          6          9
+ 4        + 7        + 9
____       ____       ____
  7          D         12
           AAD -> 0103h      AAD -> 0108h
```

Manipulation de BCTS

AAM - AAD



- AAD valable si $0 \leq AL \leq 9$ et $0 \leq AH \leq 9$
- AAM valable si $0 \leq AX \leq 99$

LASEx21.asm

Manipulation de BITS

BCD Condensé

- La représentation BCD prend beaucoup de place en mémoire
- Représentation de 1998
  - Binaire : 011111001110b = 7CEh
  - BCD : 00000001000010010000100100001000 = 01090908h
- BCD condensé, obtenu en éliminant les 4 bits de poids forts de chaque caractère

DAA - DAS

- Sur IBM 3090, la représentation **packed**
  - BCD Condensé avec les 4 derniers bits représentant le signe
  - **C** pour positif, **D** pour négatif,
  - 1998 -> 01998Ch
- DAA corrige le résultat après une addition pour le rendre compatible BCD
- DAS idem pour une soustraction

Liens entre Assembleur, C et Pascal

- L'utilisation actuelle du langage assembleur est l'inclusion de routines assembleur dans un code C, PASCAL ou autre …

- A lire :

  http://www.drpaulcarter.com

  PC Assembly Language, Paul A Carter

Liens entre Assembleur, C et Pascal

- En C
  - Le C passe les paramètres sur la pile de droite à gauche
  - La fonction appelante est responsable du nettoyage de la pile
  - Le nom de la routine assembleur doit commencer par '_'
  - C est sensible à la casse

Exemple

Source C

```
#include <stdio.h>
extern int somme (int,int);
int main (void)
{
  printf("Calcul : 2+3=%d",somme(2,3));
  return 0;
}
```

Source assembleur

```
   .model small
  .code
  public _somme
 _somme PROC near
            PUSHa
            MOV   BP,SP
            MOV             AX,[BP+4]
            ADD             AX,[BP+6]
            POP             BP
            RET
_somme ENDP

end
```

Liens entre Assembleur, C et Pascal

- En PASCAL
  - Pascal passe ses paramètres sur la pile de gauche à droite
  - La fonction appelée est responsable du nettoyage de la pile
  - Le nom de la routine assembleur doit être le même que dans le source mais en majuscules

Annexes