Méthodologie de programmation en assembleur

Philippe Preux

24 novembre 1997

Table des matières

Chapitre 1

Introduction

Dans ce document, nous indiquons une méthodologie pour concevoir et réaliser avec succès des programmes en assembleur. Tout d'abord, je tiens à porter vigoureusement un coup contre cette rumeur selon laquelle la programmation en assembleur serait difficile, voire incompréhensible. C'est on ne peut plus faux. Par contre, il est vrai que la programmation en assembleur nécessite une certaine rigueur et qu'une méthodologie doit être respectée pour mener à bien tout projet, même le plus modeste. Par ailleurs, il faut bien comprendre que l'algorithmique étant le fondement de la programmation, en assembleur, en Pascal ou dans un quelconque langage, sa maîtrise est nécessaire avant d'aborder la programmation. Afin de limiter les problèmes à ce niveau, les programmes que nous écrirons en assembleur ne demanderont généralement que la connaissance des bases de l'algorithmique :

- la notion de variable
- la notion de séquence d'instructions
- la notion de test
- la notion de boucle
- la notion de fonction et de paramètres

La section 2 indique la méthodologie proposée. La section 3 présente la résolution d'un problème en utilisant cette méthodologie. Ces deux sections doivent être lues en priorité.

Enfin, la section 4 constitue un véritable manuel de référence pour l'application de la méthode de programmation. Pour chacune des structures des langages de haut niveau (expression, affectation, séquence, tests, boucles, fonctions et procédures), on indique leur traduction en assembleur. Cette section n'est pas forcément à lire du début à la fin mais devra être systématiquement consultée lors de l'élaboration d'un programme. Outre la traduction des instructions et structures de contrôle, on y trouvera la réponse aux questions suivantes :

- comment utiliser les registres ?
- comment passer un paramètre à un sous-programme ?
- comment utiliser un tableau en assembleur ?
- comment faire les entrées-sorties de nombres et de caractères ?
- quelles instructions dois-je utiliser pour écrire une boucle `tant-que` ?

et à bien d'autres...

Chapitre 2

Méthodologie

2.1 Méthodologie

Dans la conception d'un programme en assembleur, on distinguera différentes phases :

1. la conception de l'algorithme où on exhibera les sous-programmes nécessaires en indiquant bien les paramètres et leur rôle et la structure des sous-programmes et du programme principal
2. la traduction en assembleur qui se fera en plusieurs étapes :
   (a) structure du programme : en partant d'un programme minimal, on ajoutera les débuts et fins de sous-programmes
   (b) définition du corps des fonctions **en commençant** par définir les structures de contrôle : étiquettes, instructions de saut conditionnel et inconditionnel
   (c) traduction des expressions en séquence d'instructions assembleur en utilisant des pseudo-variables
   (d) allocation des pseudo-variables : décider de l'endroit où sont stockées les données : registres, pile ou segment de données. Prendre en compte les paramètres des sous-programmes en priorité
   (e) effectuer les ajustements nécessaires en particulier dus à l'utilisation des instructions `mul/div`, `push/pop`, ...

Une *pseudo-variable* est une variable que nous utiliserons dans le but de simplifier l'écriture du programme assembleur. N'existant pas en assembleur, nous devrons les transformer en données manipulables en assembleur lors de la dernière phase de traduction en assembleur.

Sur un exemple, nous allons montrer la mise en œuvre de cette méthodologie. Dans la partie 4, on a rassemblé toutes les règles de traduction que nous allons mettre en œuvre dans l'exemple qui suit. Dans le document intitulé « Assembleur i8086 », on trouvera toutes les instructions décrites une par une. On s'y référera en cas de besoins lors des phases de traduction en assembleur.

2.2 Aperçu du langage d'assemblage

Il est bon d'avoir une vue globale des possibilités du langage d'assemblage avant de commencer quoi que ce soit. Ainsi, nous trouvons des instructions :

- arithmétiques et logiques à 2 opérandes. Aussi, toute expression ayant plus de deux opérandes devra être décomposée en opération à deux opérandes. Par exemple pour éaliser l'addition `a + b + c`, on effectuera d'abord la somme `a + b` au résultat duquel on ajoutera la valeur de `c`
- l'affectation

– de rupture de séquence. On distingue deux types : les ruptures de séquences inconditionnels qui sont impérativement effectuées ; les ruptures de séquences conditionnelles qui testent des valeurs booléennes pour décider de rompre ou non la séquence. Les tests seront réalisés à l'aide de ce type d'instructions

– appeler un sous-programme, un sous-programme étant une espèce de procédure

Par ailleurs, il existe des variables de type caractère et des variables de type entier. Les variables de type booléen sont représentées, en général, par une variable de type entier en suivant la convention suivante :

– une valeur nulle signifie faux

– une valeur non nulle signifie vrai

Enfin, on utilisera toujours des commentaires pour expliquer les programmes que nous écrivons.

Chapitre 3

Exemple

Nous appliquons la méthodologie point par point sur un exemple. C'est de cette manière que vous devez concevoir et écrire tous les programmes assembleur.

3.1 Programme à écrire

Saisir une série de nombres entiers positifs et afficher sa somme. Chaque nombre sera entré sur une ligne différente, en séparant donc deux nombres par un retour-chariot. La série de nombres sera terminée par une ligne vide (donc la frappe de deux retour-chariots successifs).

3.2 Analyse et algorithme

La première étape est celle de la rédaction d'un algorithme pour résoudre le problème posé.
Nous proposons l'algorithme suivant :

```
PROGRAMME_PRINCIPAL
somme := 0
REPETER
  lire (nombre)
  SI (nombre # 0) ALORS
    somme := somme + nombre
  FIN_SI
JUSQU'A (nombre = 0)
afficher (somme)
FIN

FONCTION lire : ENTIER
nombre := 0
REPETER
  lire_caractere (c)
  SI (est_un_chiffre (c) = vrai) ALORS
    afficher_caractere (c)
    nombre := nombre * 10
    nombre := nombre + valeur_numerique (c)
  FIN_SI
JUSQU'A (c = <retour-chariot>)
RETOURNE nombre
FIN

PROCEDURE afficher (IN nombre)
FIN

FONCTION valeur_numerique (IN caractere) : ENTIER
  RETOURNE code_ascii (caractere) - code_ascii ('0')
FIN

FONCTION est_un_chiffre (IN c) : BOOLEEN
  RETOURNE c >= '0' et c <= '9'
```

Le programme principal effectue donc une boucle de lecture et de sommation des nombres, s'arrêtant lorsqu'il n'y a plus de nombres à lire.

La fonction `LIRE` lit un nombre. Il faut savoir qu'en assembleur, il n'existe pas de fonction prédéfinie qui effectue cette action. Donc, nous devons écrire un sous-programme qui effectue cette lecture caractère par caractère et transforme la suite de caractères lus en un nombre entier.

De même, il n'existe pas de fonction permettant directement d'afficher la valeur d'un nombre à l'écran. Aussi, on doit écrire nous-même un sous-programme pourle faire. Nous n'avons pas détaillé ici le corps de cette procédure. Nous en reparlerons plus loin.

Notons que le paramètre de la fonction `valeur_numerique` est forcément le code ASCII d'un chiffre.

3.3 La réalisation du programme assembleur

Partant de l'algorithme, il s'agit maintenant de le transformer, pas à pas, en assembleur pour obtenir un programme complet.

3.3.1 Structure du programme

L'algorithme proposé se compose d'une procédure, 3 fonctions et du programme principal. C'est ainsi que nous commencons par écrire le squelette du programme assembleur :

```
          ; Ce programme lit une séquence de nombres au clavier et
          ; affiche leur somme
          .MODEL    SMALL
          .STACK    100H
          .DATA
          ;
          ; déclaration des variables
          ;
          .CODE
          ;
          ; programme principal
          ;
          mov       ax, @data
          mov       ds, ax
          ;
          ;  corps du programme principal
          ;
          mov       ah, 4ch   ; arrêt du programme
          int       21h
          ;
          ; sous-programme LIRE
          ;
LIRE      PROC
          ; lit un nombre au clavier
          ;
          ret
LIRE      ENDP
          ;
          ; sous-programme AFFICHER
          ;
AFFICHER  PROC
          ; affiche un nombre à l'écran
          ;
          ret
AFFICHER  ENDP
          ;
          ; sous programme VALEUR_NUMERIQUE
          ;
VAL_NUM   PROC
          ; renvoie la valeur numérique d'un caractère chiffre
          ;
          ret
VAL_NUM   ENDP
          ;
          ; sous-programme EST_UN_CHIFFRE
          ;
EST_UN_CHIFFRE
          PROC
          ; indique si le code ASCII passé en paramètre est
          ; celui d'un chiffre
          ;
          ret
```

```
EST_UN_CHIFFRE
          ENDP
          END
```

Squelette du programme assembleur

Dans le squelette, le programme principal comporte uniquement les instructions arrêtant son exécution. Les sous-programmes sont déclarés mais sont également vides. Notons que sur une ligne, tout ce qui suit un caractère ; est un commentaire.

3.3.2 Définition du corps des sous-programmes

On analyse maintenant les 4 sous-programmes. On commence par définir leurs paramètres et la valeur qu'ils retournent si ce sont des fonctions. Ensuite, on décrit leur structure, c'est-à-dire leur algorithme.

Les paramètres des sous-programmes

Nous devons tout d'abord bien lister les paramètres des sous-programmes et indiquer comment ils seront passés (où ils seront stockés).

1. sous-programme `LIRE` :
   - pas de paramètre en entrée
   - c'est une fonction et elle retourne le nombre lu
2. sous-programme `AFFICHER` :
   - prend un paramètre en entrée, le nombre dont la valeur doit être affichée
3. sous-programme `VAL_NUM` :
   - ce sous-programme prend en entrée le code ASCII du caractère qui a été lu et ce caractère est forcément un chiffre
   - la valeur numérique correspondant au caractère doit être renvoyée
4. sous-programme `EST_UN_CHIFFRE` :
   - ce sous-programme prend en entrée le code ASCII du caractère à tester
   - le résultat est un booléen

Structure des sous-programmes

Pour chaque sous-programme, on étudie l'implantation de l'algorithme de plus près.

Sous-programme `LIRE`

Le sous-programme `LIRE` se compose essentiellement d'une boucle `répéter`. Cette boucle se traduit en assembleur à l'aide d'une instruction de saut conditionnel (*cf.* 4.4.2). La boucle à traduire est :

```
REPETER
  lire_caractere (c)
  SI (est_un_chiffre (c) = vrai) ALORS
    afficher_caractere (c)
    nombre := nombre * 10
    nombre := nombre + valeur_numerique (c)
  FIN_SI
JUSQU'A (c = <retour-chariot>)
```

Celle-ci est transformée en :

```
debut_de_boucle:
  lire_caractere (c)
  SI (est_un_chiffre (c) = vrai) ALORS
    afficher_caractere (c)
    nombre := nombre * 10
    nombre := nombre + valeur_numerique (c)
  FIN_SI
  SI (c # <retour-chariot>) ALORS
    ALLER_EN debut_de_boucle
  FIN_SI
```

Les parties en gras résultent de l'implantation de la structure de boucle. La partie en *italique* forme le corps de la boucle. Nous laissons cette partie de côté et nous concentrons sur la traduction en assembleur de la structure de la boucle.

Si l'on poursuit la traduction, on obtient la traduction suivante :

```
debut_de_boucle:
  lire_caractere (c)
  SI (est_un_chiffre (c) = vrai) ALORS
    afficher_caractere (c)
    nombre := nombre * 10
    nombre := nombre + valeur_numerique (c)
  FIN_SI
  cmp   code_caractere, 13   ; calcul de la condition
  jne   debut_de_boucle
```

Une instruction `cmp` compare le code ASCII du caractère saisi à celui du retour-chariot. En fonction du résultat de cette instruction, l'instruction suivante `jne` déclenche une rupture de séquence en début de boucle pour effectuer une itération ou ne déclenche aucune action. Dans ce cas, l'exécution du programme se poursuit en séquence, avec l'instruction suivante du programme (non représentée ici) : on sort de la boucle.

Pour passer à une première ébauche en assembleur, nous devons encore savoir passer un paramètre à un sous-programme. Pour cela, on utilise l'instruction `push` avec, en opérande, la valeur du paramètre. Par exemple, pour passer la valeur 10 en paramètre, on utilisera l'instruction :

```
push    10
```

Dans notre cas, nous devons passer le code ASCII du caractère qui a été saisi. Si on suppose que cette valeur se trouve dans la pseudo-variable `code_caractère`, on aura donc une instruction :

```
push    code_caractère
```

L'appel d'un sous-programme se fait avec l'instruction `call` avec le nom du sous-programme a appelé en opérande.

Pour terminer, lorsqu'un appel de sous-programme est réalisé en passant des paramètres, il faut, immédiatement après l'instruction `call` mettre une instruction

`add    sp, xxx`

où `xxx` est une valeur numérique égale à deux fois le nombre de paramètres. Puisqu'il y a ici un seul paramètre, on devra utiliser l'instruction

`add    sp, 2`.

Donc, pour résumer, la traduction de l'appel de fonction `est_un_chiffre (c)` se traduira en assembleur par les trois instructions :

```
        push    code_caractère
        call    est_un_chiffre
        add     sp, 2
```

La fonction `est_un_chiffre` renvoie un résultat booléen dont il faut tester la valeur. Comme nous l'avons dit plus haut, une variable booléenne est représentée par une variable entière. Donc, nous testerons sa valeur à l'aide de l'instruction `cmp` vue plus haut et une instruction de saut conditionnel déclenchera le traitement en conséquence.

Dernier point, la saisie d'un caractère au clavier : ne cherchons pas plus loin, toutes les opérations d'entrées-sorties disponibles en assembleur sont indiquées à la section 4.11 de ce document et dans le chapitre 5 du manuel du i8086. Ainsi, nous y trouvons que pour lire un caractère au clavier, il faut utiliser les deux instructions :

```
        mov     ah, 1
        int     21h
```

L'instruction `int` déclenche l'appel d'une espèce de sous-programme prédéfini. La code ASCII du caractère saisi est ensuite accessible dans le registre `al`. Aussi, nous affectons ensuite la valeur du registre `al` à la pseudo-variable `code_caractère`.

On obtient donc la traduction en assembleur ci-dessous [1] :

```
LIRE       PROC
           ; lit un nombre au clavier
           ;
           ; en entrée : pas de paramètre
           ; en sortie, nombre contient la valeur du nombre saisi
           ;
           mov       nombre, 0
repeter:
           ; saisie d'un caractère au clavier
           mov       ah, 1
           int       21h
           mov       code_caractère, al
           ;
           ; Le code ASCII du caractère saisi est dans
           ;   la pseudo-variable code_caractère
           ;
           push      code_caractère
           call      est_un_chiffre
           add       sp, 2
           ;
           cmp       valeur_renvoyée, 0
           je        fin_de_repeter
           ;
           ; effectuer le calcul :
           ;   nombre := nombre * 10
           ;   nombre := nombre + valeur_numerique (code_caractère)
           jmp repeter
fin_de_repeter:
           ret
LIRE       ENDP
```

Sous-programme `AFFICHER`

Afficher la valeur d'un nombre n'est pas chose simple, contrairement à ce que l'on pourrait penser. Aussi, on utilisera le sous-programme `affiche_nombre` donné en 4.11.2.

Sous-programme `VAL_NUM`

En regardant une table des codes ASCII, on constate que le code ASCII d'un chiffre est égal au chiffre lui-même plus le code ASCII du caractère `'0'`. Il suffit donc de soustraire cette valeur au code ASCII pour obtenir le résultat voulu :

```
VAL_NUM    PROC
           ; renvoie la valeur numérique d'un caractère nombre
```

1. attention : ce n'est qu'un premier pas vers la traduction en assembleur ; d'autres vont suivre.

```
          ;
          sub      parametre_de_val_num, '0'
          ret
VAL_NUM   ENDP
```

Étant donnée l'extrême simplicité de ce sous-programme, on inclura directement l'instruction dans le sous-programme qui en a besoin et nous ne définirons donc pas de sous-programme `VAL_NUM`. Au contraire, on écrira tout simplement :

```
          sub      code_du_caractere, '0'
```

Sous-programme `EST_UN_CHIFFRE`

Pour déterminer si un code ASCII est celui d'un chiffre, il suffit de constater que tous les codes ASCII des chiffres se suivent de `'0'` à `'9'`. Donc, un code ASCII sera celui d'un chiffre s'il est compris entre celui de `'0'` et celui de `'9'`.

Ce sous-programme doit renvoyer un résultat booléen, donc une variable entière. Nous mettrons cette valeur dans une pseudo-variable nommée `valeur_renvoyée`.

Cela nous donne le sous-programme suivant :

```
EST_UN_CHIFFRE PROC
               ; indique si le code ASCII passe en paramètre est
               ; celui d'un chiffre
               ;
          cmp      parametre, '0'
          jl       n_est_pas_un_chiffre    ; car le code est < à celui de '0'
          cmp      parametre, '9'
          jg       n_est_pas_un_chiffre    ; car le code est > à celui de '9'
          mov      valeur_renvoyée, 1
          ret
n_est_pas_un_chiffre:
          mov      valeur_renvoyée, 0
          ret
EST_UN_CHIFFRE
          ENDP
```

On constate que l'on affecte 1 à `valeur_renvoyée` dans le cas où le code du caractère est à la fois plus grand ou égal à celui de `'0'` et plus petit ou égal à celui de `'9'`. Dans le cas contraire, on lui affecte 0.

Le programme principal

Le programme principal suit l'algorithme donné plus haut. Il consiste essentiellement en une boucle de lecture et de sommation des nombres lus.

```
    ; initialisation nécessaire dans tous les programmes assembleur
    mov      ax, @data
    mov      ds, ax
    ;
    ; programme principal
    ;
    mov      somme, 0
repeter:
    call     lire
    cmp      nombre, 0
    je       fin_de_repeter
    add      somme, nombre
    jmp      repeter
fin_de_repeter:
    ; passer le paramètre somme
    push     somme
    call     affiche_nombre
    add      sp, 2
    ;
    ; arrêt du programme
    ;
    mov      ah, 4ch
    int      21h
```

Après saisie d'un nombre, on teste s'il est nul. S'il l'est, on sort de la boucle ; sinon, on l'ajoute à la somme déjà obtenue et on itère.

On notera également l'appel du sous-programme `affiche_nombre` qui suit les règles données plus haut.

3.3.3 Traduction des expressions

Il nous faut maintenant calculer l'expression du sous-programme `LIRE` :

```
; nb := nb * 10
; nb := nb + valeur_numerique (code_caractère)
```

Les opérations en assembleur sont toujours de la forme `a := a op b` où `op` dénote une opération (addition, soustraction, ...) et se note en assembleur :

`mnenomique-pour-op a, b`

où `mnemonique-pour-op` dénote le mnémonique correspondant à l'opération que l'on veut réaliser. Donc, cela nous donne les trois instructions suivantes :

```
mul  nombre, 10                ; on fait la multiplication
sub  code_du_caractere, '0'    ; valeur numérique du caractère lu
add  nombre, code_du_caractere ; que l'on additionne au résultat courant
```

3.3.4 Allocation des pseudo-variables

Si nous résumons la situation, le programme est dans l'état indiqué ci-dessous :

```
          ; Ce programme lit une séquence de nombres au clavier et
          ; affiche leur somme
          .MODEL    SMALL
          .STACK    100H
          .DATA
          .CODE
          ;
          mov       ax, @data
          mov       ds, ax
          ;
          ; programme principal
          ;
          mov       somme, 0
repeter1: call      lire
          jnc       fin_de_repeter1
          add       somme, nombre
          jmp       repeter1
fin_de_repeter1:
          ;
          ; afficher la somme
          push      somme
          call      affiche_nombre
          add       sp, 2
          ;
          ;
          mov       ah, 4ch
          int       21h
LIRE      PROC
          ; lit un nombre au clavier
          ;
          ; en entrée : pas de paramètre
          ; en sortie, nombre contient la valeur du nombre saisi
          ;
          mov       nombre, 0
repeter2:
          ; saisie d'un caractère au clavier
          mov       ah, 1
          int       21h
          mov       code_caractère, al
          ; Le code ASCII du caractère saisi est dans
          ;   la pseudo-variable code_caractère
          ;
          push      code_caractère
          call      est_un_chiffre
          add       sp, 2
          ;
          cmp       valeur_renvoyée, 0
          je        fin_de_repeter2
          ;
          mul       nombre, 10             ; on fait la multiplication
```

```
          sub       code_caractere, '0'     ; valeur numérique du caractère lu
          add       nombre, code_caractere  ; que l'on additionne au résultat courant
          jmp       repeter2
fin_de_repeter2:
          ret
LIRE      ENDP
EST_UN_CHIFFRE PROC
          ; indique si le code ASCII passe en paramètre est
          ; celui d'un chiffre
          ;
          cmp      parametre, '0'
          jl       n_est_pas_un_chiffre
          cmp      parametre, '9'
          jg       n_est_pas_un_chiffre
          mov      valeur_renvoyée, 1
          ret
n_est_pas_un_chiffre:
          mov      valeur_renvoyée, 0
          ret
EST_UN_CHIFFRE ENDP
          END
```

Programme avant allocation des pseudo-variables

Une étiquette donnée ne pouvant référencer qu'une seule instruction, nous avons numéroté les étiquettes `repeter` et `fin_de_repeter` qui apparaissaient chacune deux fois, puisqu'il y a deux boucles, afin de les distinguer.

Les pseudo-variables utilisées sont donc :

- `somme` qui est une donnée utilisée uniquement dans le programme principal ;
- `nombre` qui est une donnée fournie par le sous-programme `LIRE` et utilisé dans le programme principal ;
- `code_caractere` qui est fourni lors de la lecture d'une touche puis utilisé dans le sous-programme `LIRE` ;
- `valeur_renvoyée` qui est renvoyé par la fonction `EST_UN_CHIFFRE` et utilisé dans le sous-programme `LIRE` ;
- `parametre` qui est passé par le sous-programme `LIRE` au sous-programme `EST_UN_CHIFFRE`.

`somme` est une variable que l'on dirait globale en Pascal. Aussi, on va l'implanter de cette manière. En assembleur, une variable globale se déclare dans la section `.data`. Sa valeur peut aisément devenir plus grande que 255. Aussi, nous la définirons comme un mot-mémoire. Soit :

```
somme       dw       0
```

Le `0` est la valeur initiale de la variable.

`nombre` étant utilisé dans la fonction `LIRE` puis retourné à son appelant, il est judicieux de l'implanter dans un registre `ax`, `bx`, `cx` ou `dx`. Choisissons `bx`.

`code_caractère` est une variable locale au sous-programme `LIRE`. On peut l'implanter dans un registre (par exemple `cl`), ce que nous faisons ici.
On peut également l'implanter autrement : le nombre de registres étant limité et le nombre de variables locales nécessaires pouvant être important, un autre mécanisme est disponible pour pouvoir utiliser autant de variables locales que l'on veut. On consultera la section 4.7.5 pour plus de détails.

`valeur_renvoyée` est une valeur renvoyée que l'on peut implanter au choix dans un registre choisi parmi `ax`, `bx`, `cx` ou `dx`. Choisissons `ax`.

`parametre` est un paramètre. Dans ce cas, comme il est indiqué dans la section 4.7.4, il faut :

1. ajouter les deux instructions :

```
push      bp
mov       bp, sp
```

en tout début de sous-programme (juste après `PROC`)

2. ajouter l'instruction :

   ```
   pop     bp
   ```

   à la fin du sous-programme, juste avant l'instruction `ret`. S'il y a plusieurs instructions `ret` dans le sous-programme, il faut mettre cette instruction `pop` à chaque fois.

3. emplacer les occurences de la pseudo-variables `parametre` par `WORD PTR [bp+4]`.

Tout cela nous donne le programme suivant (qui est presque terminé !) :

```
          ; Ce programme lit une séquence de nombres au clavier et
          ; affiche leur somme
          .MODEL    SMALL
          .STACK    100H
          .DATA
somme     dw        0
          .CODE
          mov       ax, @data
          mov       ds, ax
          ;
          ; programme principal
          ;
          ; mov       somme, 0
repeter1: call      lire
          cmp       bx, 0
          je        fin_de_repeter1
          add       somme, bx
          jmp       repeter1
fin_de_repeter1:
          ;
          ; afficher la somme
          push      somme
          call      affiche_nombre
          add       sp, 2
          ;
          ;
          mov       ah, 4ch
          int       21h
LIRE      PROC
          ; lit un nombre au clavier
          ;
          ; en entrée : pas de paramètre
          ; en sortie, bx contient la valeur du nombre saisi
          ;
          mov       bx, 0
repeter2:
          ; saisie d'un caractère au clavier
          mov       ah, 1
          int       21h
          mov       cl, al
          ; Le code ASCII du caractère saisi est dans
          ;   le registre cl
          ;
          push      cl
          call      est_un_chiffre
          add       sp, 2
          cmp       ax, 0
          je        fin_de_repeter2
          ;
          mul       bx, 10   ; on fait la multiplication
          sub       cl, '0'  ; valeur numérique du caractère lu
          add       bx, cl   ; que l'on additionne au résultat courant
          jmp       repeter2
fin_de_repeter2:
          ret
LIRE      ENDP
EST_UN_CHIFFRE PROC
          ; indique si le code ASCII passe en paramètre est
          ; celui d'un chiffre
          ;
          push     bp
          mov      bp, sp
          cmp      WORD PTR [bp+4], '0'
```

```
          jl      n_est_pas_un_chiffre
          cmp     WORD PTR [bp+4], '9'
          jg      n_est_pas_un_chiffre
          mov     ax, 1
          pop     bp
          ret
n_est_pas_un_chiffre:
          mov     ax, 0
          pop     bp
          ret
EST_UN_CHIFFRE ENDP
          END
```

Noter que la première instruction du programme principal a été mise en commentaire. L'initialisation de la variable `somme` est effectuée automatiquement.

3.3.5 Derniers ajustements

Il nous reste encore quelques petits ajustements à effectuer pour obtenir un programme assembleur complet, qui s'assemble sans erreur et s'exécute en donnant le résultat attendu.

Instructions de multiplication et division

Nous avons utilisé l'instruction `mul bx, 10` qui n'existe pas : les instructions de multiplications et de divisions (`mul`, `imul`, `div`, `idiv`) ne prennent qu'un seul opérande. La deuxième valeur prise en compte dans l'opération est toujours le registre `al` ou le registre `ax` selon que l'instruction est exécutée sur 8 ou 16 bits.

Tous les détails sur ces instructions sont données plus loin dans la section 4.7.3. Nous voulons réaliser l'opération `bx := bx * 10`. En appliquant ce qui est indiqué dans cette section, nous obtenons la transformation suivante :

```
mov    ax, 10
mul    bx
mov    bx, ax
```

qui s'explique de la manière suivante :

1. la première instruction affecte au registre `ax` la valeur 10, deuxième opérande de la multiplication
2. la deuxième instruction effectue la multiplication. Le résultat est ensuite disponible dans la paire de registre `dx` pour le poids fort, `ax`. Nous supposons que le résultat de la multiplication est inférieur à 65535, donc le registre `dx` contient ne valeur nulle
3. la dernière instruction transfert le résultat de la multiplication, qui est toujours mis dans `ax` par `mul` dans le bon registre, `bx`

Instructions `push`

Dans le sous-programme `LIRE`, nous utilisons l'instruction

```
push    cl
```

qui est incorrecte car l'instruction `push` ne prend en opérande que des données 16 bits (registre ou variable). Nous devons donc transformer cette instruction en une instruction valide. Le plus simple est de la transformer en

```
push    cx.
```

Cependant, dans ce cas, il faut s'assurer que les 8 bits de poids fort de `cx`, le registre `ch` sont à 0. Il faut donc ajouter une instruction

```
mov    ch, 0
```

auparavant. Cela nous donne donc :

```
mov    ch, 0
push   cx
```

Compatibilité entre données

L'instruction `add bx, cl` est incorrecte car elle mélange un registre 8 bits (`cl`) avec un registre 16 bits (`bx`). Il faut que les deux données soient des mots. Donc, il faut utiliser `cx` à la place de `cl`. Puisque nous venons précédemment de nous arranger pour que `cx` contienne la valeur de `cl` en mettant `ch` à zéro, nous pouvons remplacer `cl` par `cx`.

Propreté des sous-programmes

Enfin, et cette vérification doit toujours être faite en dernière étape, une règle importante demande qu'un sous-programme ne modifie aucune valeur qui ne lui appartienne pas (variable globale, registre) à moins que cela ne soit explicitement indiqué dans les spécifications du sous-programme. Donc, si nous regardons le programme que nous avons écrit, nous constatons que le sous-programme `LIRE` modifie la valeur des registres `ax` et `cx` alors que cela n'est pas demandé. Aussi, il faut modifier ce sous-programme pour que cela n'arrive pas.

Pour cela, après avoir recensé les registres en question (en général, ce sont des registres et non des variables globales), il faut ajouter des instructions aux sous-programmes incriminés pour qu'ils sauvegardent les valeurs de ces registres à leur début et les restaurent à leur sortie. Ainsi, du point de vue de l'appelant, la valeur des registres n'a pas changé lors de l'appel du sous-programme.

Sauvegarde des registres

Si le sous-programme ne prend pas de paramtètre, on place les instructions de sauvegarde après la ligne `PROC` du sous-programme.

Si le sous-programme prend des paramètres, on placera les instructions de sauvegarde juste après les deux lignes :

```
push   bp
mov    bp, sp
```

qui doivent se trouver en début de sous-programme.

La sauvegarde d'un registre est effectuée à l'aide d'instructions `push` en spécifiant le registre à sauvegarder en opérande. Il y a donc autant d'instructions à ajouter que de valeurs à sauvegarder.

Ici, on ajoutera donc les deux lignes :

```
push     ax
push     cx
```

Restauration des registres

Si le sous-programme ne prend pas de paramètre, on place les instructions de restauration avant l'instruction `ret` du sous-programme.

Si le sous-programme prend des paramètres, les instructions de restauration sont placées avant l'instruction :

`pop bp`

qui doit se trouver à la fin du sous-programme.

La trestauration de la valeur d'un registre est effectuée à l'aide d'une instruction `pop`.

Ici, nous ajouterons les deux instructions :

```
pop      cx
pop      ax
```

juste avant l'instruction `ret`.

Il faut faire très attention à observer les trois règles suivantes :

1. il doit y avoir autant d'instructions `pop` que d'instructions `push`

2. il doit y avoir un jeu d'instructions `pop` avant chaque instruction `ret` du sous-programme

3. les registres doivent apparaître dans l'ordre inverse dans les instructions `pop` par rapport aux instructions `push`. Ainsi, ici on a `push ax` puis `push cx` pour la sauvegarde, `pop cx` puis `pop ax` pour la restauration.

Le non-respect des règles 1 et 2 entraînera toujours un plantage de votre programme. Le non-respect de la règle 3 entraînera un dysfonctionnement.

3.3.6 Programme terminé

Pour résumer le résultat de tout ce qui a été dit, nous indiquons ci-dessous le programme terminé, prêt à être assemblé et exécuté. Nous n'indiquons pas ci-dessous le sous-programme `affiche_nombre` qui est donné à la section 4.11.2.

```
          ; Ce programme lit une séquence de nombres au clavier et
          ; affiche leur somme
          .MODEL    SMALL
          .STACK    100H
          .DATA
somme     dw        0
          .CODE
          mov       ax, @data
          mov       ds, ax
          ;
          ; programme principal
          ;
repeter1: call      lire
          cmp       bx, 0
          je        fin_de_repeter1
          add       somme, bx
          jmp       repeter1
fin_de_repeter1:
          ;
          ; afficher la somme
          push      somme
          call      affiche_nombre
          add       sp, 2
          ;
          ;
          mov       ah, 4ch
          int       21h
LIRE      PROC
          ; lit un nombre au clavier
          ;
          ; en entrée : pas de paramètre
          ; en sortie, bx contient la valeur du nombre saisi
          ;
          push      ax
          push      cx
          mov       bx, 0
repeter2:
          ; saisie d'un caractère au clavier
          mov       ah, 1
          int       21h
          mov       cl, al
          ; Le code ASCII du caractère saisi est dans
          ;   le registre cl
          ;
          mov       ch, 0
          push      cx
          call      est_un_chiffre
          add       sp, 2
          cmp       ax, 0
          je        fin_de_repeter2
          ;
          mov       ax, 10
          mul       bx
          mov       bx, ax
          sub       cx, '0'  ; valeur numérique du caractère lu
          add       bx, cx   ; que l'on additionne au résultat courant
          jmp       repeter2
fin_de_repeter2:
          pop      cx
```

```
          pop      ax
          ret
LIRE      ENDP
EST_UN_CHIFFRE PROC
          ; indique si le code ASCII passe en paramètre est
          ; celui d'un chiffre
          ;
          push     bp
          mov      bp, sp
          cmp      WORD PTR [bp+4], '0'
          jl       n_est_pas_un_chiffre
          cmp      WORD PTR [bp+4], '9'
          jg       n_est_pas_un_chiffre
          mov      ax, 1
          pop      bp
          ret
n_est_pas_un_chiffre:
          mov      ax, 0
          pop      bp
          ret
EST_UN_CHIFFRE ENDP
          END
```

Chapitre 4

Guide pour la traduction de structures de haut niveau en assembleur

Ce chapitre est un manuel de référence auquel on se reportera pour traduire les structures algorithmiques en assembleur. Pour chacune des structures des langages de haut niveau, nous proposons une traduction. Nous utilsons pour cela des pseudo-variables. La transformation des pseudo-variables en assembleur est vue à la fin de ce chapitre 4.7.

4.1 Expression arithmétique et logique

Nous ne nous intéressons ici qu'à la traduction d'expressions arithmétiques où n'interviennent que des valeurs de type entier codées sur 8 ou 16 bits. Les valeurs réelles ne sont en aucun cas prises en compte ici.

4.1.1 Principe

La traduction d'une expression arithmétique consiste à la décomposer en opération ayant un ou deux opérandes source et un opérande cible. Par exemple, on traduira :

Original	Traduction

`a + b - c`	`mov tmp0, a`
	`add tmp0, b`
	`mov tmp1, c`
	`sub tmp1, tmp0`
	`; tmp1 contient la valeur de l'expression`

Traduction d'une expression numérique

On prendra garde à d'éventuels parenthésages en calculant les termes dans le bon ordre. On utilisera des pseudo-variables temporaires notées `tmp...` autant qu'il est nécessaire.

4.1.2 L'affectation

L'affectation du résultat d'une expression sera réalisée en affectant le résultat de la dernière opération de l'expression à la variable à affecter :

Original

```
x := a + b - c
```

Traduction

```
mov   tmp0, a
add   tmp0, b
sub   tmp0, c
mov   x, tmp0
```

Traduction d'une affectation

4.2 Séquence d'instructions

Une séquence d'instructions se traduit instruction par instruction, l'une après l'autre. Afin d'éviter toute confusion, on utilise de nouvelles pseudo-variables pour chaque instruction. Ainsi :

Original

```
x := a - b + c
y := d - x
z := x + y
```

Traduction

```
; instruction x := a - b + c
mov  tmp0, a
sub  tmp0, b
add  tmp0, c
mov  x, tmp0
; instruction y := d - x
mov  tmp1, d
sub  tmp1, x
mov  y, tmp1
; instruction z := x * y
mov  tmp2, x
add  tmp2, y
mov  z, tmp2
```

Traduction d'une séquence d'instructions

4.3 Les tests

4.3.1 Principe

Original

```
SI (condition vraie) ALORS
  action-alors
SINON
  action-sinon
FIN_SI
suite-du-programme
```

Traduction

```
  calcul de la condition
  Jcc    etiquette_sinon
  action-alors
  ...
  JMP    etiquette_fin_si
etiquette_sinon:
  action-sinon
  ...
etiquette_fin_si:
  suite-du-programme
```

Traduction d'un test

Le calcul de la condition revient à évaluer une expression, ce que l'on décrit un peu plus bas (*cf.* 4.3.2). Ce calcul positionne certains bits du registre `FLAGS` dont l'un sera testé par une instruction de saut conditionnel (instruction `Jcc` à choisir en fonction du bit à tester) pour décider laquelle de la `partie-alors` ou de la `partie-sinon` doit être exécutée. Une fois cela traduit, il reste à traduire les deux blocs d'instructions formant les `partie-alors` et `partie-sinon` qui ne sont que des séquences d'instructions (donc, voir 4.2).

4.3.2 Conditions simples

La condition du test est une expression logique simple ou composée. Une condition simple est de l'une des 6 formes :

- `expression1 = expression2`
- `expression1 # expression2`
- `expression1 < expression2`
- `expression1 <= expression2`
- `expression1 > expression2`
- `expression1 >= expression2`

où `expression1` et `expression2` sont des expressions arithmétiques ou de simples variables.

Par exemple, on traduira:

Original

```
SI (e1 = e2) ALORS
  action-alors
SINON
  action-sinon
FIN_SI
```

Traduction

```
  cmp  e1, e2
  jne  etiquette_sinon
  ; traduction de la partie-alors
  ...
  jmp  fin_si
etiquette_sinon:
  ; traduction de la partie-sinon
  ...
fin_si:
```

Traduction d'un test

plus généralement, pour une condition donnée on utilisera une séquence de deux instructions : une instruction `cmp` suivie d'une instruction de saut conditionnel comme il est indiqué dans la table suivante :

Condition	Instruction
`e1 = e2`	`jne`
`e1 # e2`	`je`
`e1 < e2`	`jge`
`e1 <= e2`	`jg`
`e1 > e2`	`jle`
`e1 >= e2`	`jl`

Instructions pour la traduction de tests simples

4.3.3 La condition est une expression

Lorsque les valeurs à tester résultent du calcul d'une expression, on aura par exemple :

Original

```
SI (a + b = 10) ALORS
  action-alors
SINON
  action-sinon
FIN_SI
```

Traduction

```
  mov  tmp0, a
  add  tmp0, b
  cmp  tmp0, 10
  jne  etiquette_sinon
  ; traduction de la partie-alors
  ...
  jmp  fin_si
etiquette_sinon:
  ; traduction de la partie-sinon
  ...
  fin_si:
  ...
```

Traduction d'un test

4.3.4 Conditions composées

Une condition composée est constituée par des conditions simples reliées par des opérateurs `ET`, `OU` ou la négation d'une condition :

1. `condition1 ET condition2`
2. `condition1 OU condition2`
3. `NON condition`

condition1 ET condition2

Dans le premier cas, la condition est vérifiée si les deux conditions le sont. On va la traduire en :

Original

```
SI (condition1 vraie ET condition2 vraie) ALORS
  action-alors
SINON
  action-sinon
FIN_SI
```

Traduction

```
SI (condition1 vraie) ALORS
  SI (condition2 vraie) ALORS
    action-alors
  SINON ALLER_EN action_sinon
  FIN_SI
SINON
action_sinon:
  action-sinon
FIN_SI
```

Traduction d'une condition composée `ET`

où l'on s'autorise l'instruction `ALLER_EN` pour rendre compte de cette traduction, son écriture dans un langage de haut niveau étant, sinon impossible, du moins très fortement déconseillée.

condition1 OU condition2

Dans le deuxième cas, la condition est vérifiée si l'une des conditions est vraie.

Original

```
SI (condition1 vraie OU condition2 vraie) ALORS
  action-alors
SINON
  action-sinon
FIN_SI
```

Traduction

```
SI (condition1 vraie) ALORS
  ALLER_EN action-alors
SINON
  SI (condition2 vraie) ALORS
action_alors:
     action-alors
  SINON
action_sinon:
     action-sinon
  FIN_SI
FIN_SI
```

Traduction d'une condition composée `OU`

où l'on utilise à nouveau l'instruction `jmp` par « abus de langage ».

NON condition

Dans le troisième cas, la condition est vraie si sa valeur est fausse et inversement. Aussi, sa traduction s'obtient presque comme celle d'une condition simple, si ce n'est que l'on n'utilise pas la même instruction, mais celle correspondant à la condition opposée. L'instruction à utiliser est donnée dans la table suivante :

Condition	Instruction
`NON (e1 = e2)`	`je`
`NON (e1 # e2)`	`jne`
`NON (e1 < e2)`	`jl`
`NON (e1 <= e2)`	`jle`
`NON (e1 > e2)`	`jg`
`NON (e1 >= e2)`	`jge`

Instructions pour la traduction de tests simples (suite)

Bien entendu, on peut composer plus de deux conditions. Le principe de la traduction reste le même.

4.4 Les boucles

4.4.1 Boucle `tant-que`

Une boucle `tant-que` consiste à exécuter un groupe d'instructions (le « corps de boucle ») tant qu'une condition conserve la valeur vraie. Aussi, nous allons traduire ce type de boucle par un test reposant sur la valeur de la condition qui sera suivi en fonction de son résultat, de l'exécution du corps de la boucle ou d'un saut hors de la boucle, à l'instruction qui la suit. Dans le cas où le corps est exécuté, la condition doit ensuite être à nouveau calculée puis testée pour décider de poursuivre l'exécution de la boucle ou la quitter.

On obtient donc une traduction du genre :

Boucle `tant-que` originale	Boucle transformée	Traduction en assembleur de la structure de la boucle

`TANT-QUE (condition vraie) FAIRE`
`  action`
`FAIT`
	`debut_de_boucle:`
	`  SI (condition vraie) ALORS`
	`  action`
	`  ALLER_EN debut_de_boucle`
	`  FIN_SI`
		`debut_de_boucle:`
		`  calcul de la condition`
		`  jcc fin_boucle`
		`  action`
		`  jmp  debut_de_boucle`
		`fin_boucle:`

Boucle `tant-que`

Il n'y a donc rien ici de nouveau. La traduction va s'obtenir par juxtaposition des éléments vus précédemment, c'est-à-dire la traduction d'un test et d'une séquence d'instructions.

4.4.2 Boucle `répéter`

Comme une boucle `tant-que`, une boucle `répéter` doit au préalable être transformée dans une structure équivalente et directement traduisible en assembleur :

Boucle `répéter` originale	Boucle transformée	Traduction en assembleur de la structure de la boucle

`REPETER`
`  action`
`JUSQU'A (condition vraie)`
	`debut_de_boucle:`
	`  action`
	`  SI (condition fausse) ALORS`
	`  ALLER_EN debut_de_boucle`
	`  FIN_SI`
		`debut_de_boucle:`
		`  action`
		`  calcul de la condition`
		`  jcc debut_de_boucle`
		`fin-boucle:`

Boucle `répéter`

4.4.3 Boucle `pour`

Une boucle `pour` générale doit d'abord être mise sous une forme particulière pour être traduite ensuite en assembleur. En effet, en assembleur i8086, l'indice de boucle doit forcément varier d'une valeur positive à 1, l'indice étant décrémenté automatiquement à chaque itération. Par exemple, la boucle :

```
somme := 0
POUR i := 1 A 10 FAIRE
  somme := somme + i
FAIT
```

devra être transformée au préalable en :

```
somme := 0
POUR i := 10 A 1 FAIRE
  somme := somme + i
FAIT
```

Si cela ne pose aucun problème ici, cela en pose parfois. Par exemple,

```
POUR i := 1 A 10 FAIRE
  afficher (i)
FAIT
```

et

```
POUR i := 10 A 1 FAIRE
  afficher (i)
FAIT
```

ne donnent pas le même résultat. En fait, la transformation correcte de la première boucle est :

```
POUR i := 10 to 1 FAIRE
  afficher (11 - i)
FAIT
```

et la transformation correcte de la deuxière boucle est :

```
somme := 0
POUR i := 10 to 1 FAIRE
  somme := somme + 11 - i
FAIT
```

Une fois mise sous cette forme, la traduction en assembleur est obtenue de la manière suivante :

Boucle `pour` originale	Boucle transformée	Traduction en assembleur de la structure de la boucle

`POUR i := debut A fin FAIRE`
`  action`
`FAIT`
	`POUR i := n A 1 FAIRE`
	`  action transformée`
	`FAIT`
		`  mov  cx, n`
		`debut_de_boucle:`
		`  action`
		`  loop debut_de_boucle`

Boucle `pour`

4.5 Procédures et fonctions : principe des sous-programmes

En assembleur, fonctions et procédures se traduisent sous la forme de *sous-programmes*. Un sous-programme a pour objet essentiel de structurer un programme ; à un sous-programme est associée la réalisation d'un certain traitement. Une bonne règle de principe consiste à ne jamais avoir de sous-programmes dont la longueur dépasse une page de listing (environ 50 lignes). Au-delà de cette taille, on peut généralement découper très naturellement le sous-programme en plusieurs traitements, chacun faisant lui-même l'objet d'un sous-programme.

Un sous-programme se compose essentiellement d'un corps qui est une succession d'instructions. Ces instructions sont des séquences, des tests et des boucles. Aussi, leur traduction du langage de haut niveau en assembleur sera effectuée comme nous venons de le voir.

Ce qui est spécifique à l'utilisation des sous-programmes est :

- le passage de paramètre ;
- l'existence de variables locales ;

– le renvoi d'une valeur si le sous-programme traduit une fonction.

Nous allons donc nous concentrer sur ces seuls points dans ce qui suit.

Dans la suite de cette section, nous donnerons tout d'abord la forme générale d'un sous-programme et son appel. Nous étudierons ensuite le passage de paramètres, le renvoi d'une valeur puis l'utilisation de variables locales.

4.5.1 Principe

Un sous-programme s'appelant `nom` se définira à l'aide du squelette suivant :

```
nom    PROC
       ; description du sous-programme : ce qu'il fait,
       ; ses paramètres, la valeur qu'il renvoie
       ....
       ....   ; instructions du sous-programme
       ....
       ret
nom    ENDP
```

Squelette d'un sous-programme.

Nous rappelons que pour assurer le bon fonctionnement du programme, **un sous-programme ne doit en aucun cas modifier la valeur d'une donnée (registre ou autre) à moins que cela ne soit demandé explicitement dans la spécification du sous-programme**.

Aussi, **nous prendrons toujours soin de sauvegarder le contenu des registres utilisés dans le sous-programme et de restaurer leur valeur initiale à la sortie du sous-programme**. De cette manière, l'appel du sous-programme ne modifiera pas la valeur des registres affectée dans l'appelant avant l'appel.

Ces sauvegardes et restaurations concerneront en général uniquement des registres. Ceux-ci seront sauvegardés en début de sous-programme *via* des instructions `push`, restaurés en fin de sous-programme par des instructions `pop`.

4.5.2 Appel d'un sous-programme

Un sous-programme portant le nom `toto` s'appelle à l'aide de l'instruction `call` :

```
...
call   toto
....
```

Appel d'un sous-programme

4.6 Sous-programmes avec paramètres et variables locales

4.6.1 Passage de paramètre en entrée du sous-programme

Seuls des données de la taille d'un mot-mémoire peuvent être passées en paramètre. Si l'on veut passer un octet, on passera un mot et on ne considérera que la valeur de l'octet de poids faible du mot.

Les paramètres sont passés à l'aide de l'instruction `push` de la manière suivante :

```
...
push   paramètre n
...
push   paramètre 2
push   paramètre 1
call   sous_programme
add    sp, 2 * n         ; attention, lire ci-dessous
....
```

Squelette d'un appel de sous-programme avec passage de paramètres en entrée

L'expression `2 * n` doit être remplacée par sa valeur, en fonction du nombre de paramètres passés.

Si un paramètre résulte d'un calcul, des instructions effectuant ce calcul viendront s'intercaler entre les `push`. Exemple :

Appel original	Traduction en assembleur

`f (a + 3, b,  c - d)`	`mov  tmp0, a`
	`add  tmp0, 3`
	`push  tmp0`
	`push  b`
	`mov  tmp1, c`
	`sub  tmp1, d`
	`push  tmp1`
	`call  f`
	`add  sp, 6`

Appel de sous-programme avec passage de paramètres calculés

4.6.2 Réception des paramètres

Nous indiquons ci-dessous le squelette d'un sous-programme prenant des paramètres.

```
nom    PROC
       ; description du sous-programme : ce qu'il fait,
       ; ces paramètres, la valeur qu'il renvoie
       push    bp
       mov     bp, sp
       ....
       pop     bp
       ret
nom    ENDP
```

Squelette d'un sous-programme prenant des paramètres en entrée

On notera les instructions concernant le registre `bp` qui ont été ajoutées par rapport au cas du sous-programme sans paramètre. Ce registre est fondamental dans l'utilisation de la valeur des paramètres.

Dans l'appelé, on notera simplement l'accès au i^{e} paramètre à l'aide de la pseudo-variable `p`i (la lettre `p` suivie du nombre i).

Dans le morceau de programme ci-dessous, un sous-programme nommé `appelant` appelle un sous-programme nommé `appele` en lui passant trois paramètres : 56, 25 et 10. Dans l'appelant, les paramètres sont accédés. Après l'instruction d'appel `call`, on notera l'instruction `add sp, 6` qui est effectuée au retour du sous-programme.

```
appelant   PROC
           ....
           ; troisième paramètre
           push     10
           ; deuxième paramètre
           push     25
           ; premier paramètre
           push     56
           call     appele
           ; 6 = 3 x 2, où 3 est le nombre de paramètres
           add      sp, 6
           ....
           ret
appelant   ENDP
appele     PROC
           ; description du sous-programme
           push    bp
           mov     bp, sp
           ....
           mov     dx, p2
```

```
            ; dx contient la valeur du deuxième paramètre, soit 25
            .....
            cmp     ax, p1
            ; compare la valeur de ax avec celle du premier paramètre,
            ; soit 56
            ....
            mov     cx, 3
            add     cx, p3
            ; ajoute la valeur du troisième paramètre à cx, soit 10
            ; donc, le registre cx contient maintenant la valeur 13
            ....
            pop     bp
            ret
appele      ENDP
```

Passage et utilisation de paramètres en entrée

4.6.3 Variables locales

Une variable locale est spécifiée sous la forme d'une pseudo-variable.

4.6.4 Valeur de retour

Un sous-programme peut renvoyer simplement une valeur sous forme d'une pseudo-variable.

4.7 Traduction des pseudo-variables

La traduction des pseudo-variables est fortement liée à certaines contraintes dues aux instructions assembleur du type : telle instruction n'accepte que tel type d'opérande. On a *a priori* beaucoup de possibilités : pour simplifier dans un premier temps, on peut écrire son programme en utilisant de nombreuses variables globales définies dans le segment de données et n'utiliser les registres que quand on ne peut pas faire autrement. (Notons cependant que cette méthode ne fonctionne pas si on utilise des sous-programmes récursifs.) À l'autre extrême, on peut s'amuser à jongler avec les registres et la pile en n'utilisant pas, ou presque pas, de données globales. Comme toujours, la bonne solution se situe quelque part entre ces deux extrèmes : il s'agira d'optimiser l'utilisation des registres et d'éviter un trop grand nombre d'accès à la mémoire en n'allouant pas systématiquement les données en mémoire. Ainsi, si l'on suit les consignes à propos de la taille des sous-programmes, on peut généralement n'utiliser que des registres pour le stockage des variables globales et locales.

Lors de l'allocation d'une donnée en assembleur, on a le choix entre deux types de données selon la valeur que peut prendre cette donnée : octet ou mot. Dans un premier temps, on pourra suivre les règles suivantes :

1. une donnée contenant un caractère sera allouée sous forme d'un octet
2. une donnée contenant un entier sera allouée sous forme d'un mot

L'allocation des données en mémoire étant la phase la plus délicate, on tiendra toujours scrupuleusement à jour la liste des registres utilisés à toutes les instructions du programme et on saura toujours quelles sont les données qui se trouvent dans la pile et son niveau dans la pile.

4.7.1 Les variables globales du programme

Les variables globales seront alloués dans le segment de données, donc dans la section `.DATA` du source (*cf.* manuel i8086). Pour une donnée numériaue, on déclarera une donnée comme un octet ou un mot, en fonction de la valeur maximale qu'elle peut prendre dans le programme.

Si l'on utilise trois variables `a`, `b` et `c` stockables sur un octet et deux variables `d` et `e` nécessitant un mot et que l'on initialise dans le programme `a` à la valeur 23, `c` avec -10, `d` avec 10000 et les deux autres variables n'étant pas initialisées, on déclarera :

```
.DATA
```

```
a    db    23
b    db    0
c    db    -10
d    dw    10000
e    dw    0
```

On utilisera ensuite ces variables librement dans le programme en donnant leur nom dans les instructions.

4.7.2 Les variables temporaires

Les variables temporaires seront stockées dans des registres. On utilisera pour cela les registres `ax`, `bx`, `cx` et `dx` pour stocker des nombres, ou les registres `ah`, `al`, `bh`, `bl`, `ch`, `cl`, `dh` et `dl` pour stocker des caractères ou des entiers petits (dont la valeur ne dépasse pas 256). Pour ces derniers, on prendra garde lorsqu'on les utilise, qu'ils font partie des premiers ; c'est-à-dire qu'en modifiant la valeur de `al` (par exemple), on modifie en même temps la valeur de `ax`, et ainsi de suite (voir chapitre 1 du manuel i8086).

4.7.3 Les instructions de multiplication et de division

Les instructions de multiplication et de division sont très particulières en cela qu'elles n'ont pas la même forme que les instructions d'addition et de soustraction. Ainsi, lorsque nous avons écrit :

```
mul    a, b
```

indiquant de réaliser l'opération `a := a * b`, cette opération ne peut pas être traduite directement en assembleur comme une addition ou une soustraction.

Il faut procéder méthodiquement en se posant les questions suivantes :

1. les données que je multiplie sont-elles signées ou toujours positives ?
2. les données que je multiplie sont-elles des octets ou des mots ?

Si les données sont signées, nous allons utiliser l'instruction `imul` pour une multiplication, `idiv` pour une division. Sinon, nous allons utiliser l'instruction `mul` pour une multiplication, `div` pour une division.

Il faut savoir que quand on multiplie deux données codées sur chacune n bits, le résultat est codé sur $2n$ bits [1]. Ceci explique ce qui suit.

Concernant la taille des données, sachons que :

- si les données sont des octets, l'un des deux opérandes devra nécessairement se trouver dans le registre `al`. L'autre pourra être mis dans n'importe lequel des registres 8 bits restant ou dans une variable déclarée par une directive `db`. Le résultat sera placé dans le registre `ax`.
- si les données sont des mots, l'un des opérandes devra nécessairement se trouver dans le registre `ax`. L'autre pourra être mis dans n'importe lequel des registres 16 bits restant ou dans une variable déclarée par une directive `dw`. Le résultat sera placé dans les registres `dx` et `ax` (le mot de poids fort est dans `dx`, le mot de poids faible dans `ax` – voir la description des instructions de multiplication dans le manuel i8086).

Aussi, la traduction de l'instruction mentionnée plus haut `mul    a, b` se fera de la manière suivante. Supposons que les deux données soient des octets, le résultat étant mis dans ce cas dans le registre `ax`, on placera la valeur de `a` dans le registre `al`. La valeur de `b` pourra se trouver dans un registre 8 bits ou dans une variable. Supposons qu'elle soit dans une variable globale de même nom, la multiplication se traduira par les deux instructions :

```
mov   al, a
mul   b
mov   a, al
```

1. c'est la même chose en base 10 : multiplier deux nombres en base 10 qui s'écrivent sur 3 chiffres, vous pouvez avoir besoin de 6 chiffres pour exprimer le résultat : exemple $100 \times 100 = 100000$.

La démarche pour utiliser les instructions de division est très similaire à celle pour utiliser des instructions de multiplication. Aussi, vous vous reporterez au manuel i8086 pour cela.

4.7.4 Les paramètres

D'une manière générale et afin d'éviter tout ennui, nous passerons les paramètres en :

- dans l'appelant, indiquant les paramètres à passer à l'aide d'instructions `push` ; voir à ce propos la section 4.6.1
- dans l'appelé, référençant les paramètres par des pseudo-variables `p`*i* (*cf.* section 4.6.2)

Remarque sur l'instruction `push`

L'instruction `push` ne prend que deux types d'opérandes :

1. les registres 16 bits `ax`, `bx`, `cx`, `dx`, `si` ou `di`
2. des données globales déclarées dans la section `.data` avec une directive `dw`

Si l'on veut passer une valeur constante, il faudra passer par l'intermédiaire d'un registre non utilisé. Ainsi, on traduira `push 10` par une suite d'instructions :

```
mov     ax, 10
push    ax
```

en supposant que le contenu de `ax` peut être perdu. Sinon, on utilisera un autre registre dont le contenu peut l'être.

Les paramètres dans l'appelé

Les pseudo-variables notées plus haut `p`*i* seront simplement remplacée par `WORD PTR [bp+2+2i]`. Ainsi, par exemple, le paramètre `p3` sera remplacé par `WORD PTR [bp+8]`.

On prendra garde dans un sous-programme utilisant des paramètres à :

1. ajouter les deux instructions :

   ```
   push    bp
   mov     bp, sp
   ```

   en début de sous-programme, immédiatement après `PROC` ;
2. ajouter l'instruction

   ```
   pop     bp
   ```

   avant chaque instruction `ret` du sous-programme.

4.7.5 Les variables locales

Les variables locales à un sous-programme peuvent être allouées dans des registres. C'est ce que nous avons fait dans l'exemple plus haut (*cf.* 3.3.4).

Cependant cette méthode a ses limites lorsque l'on veut utiliser un nombre élevé de variables locales. Nous présentons donc une autre méthode permettant d'avoir un nombre arbitrairement grand de variables locales dans un sous-programme.

1. compter le nombre de variables locales. Nommons-le n. À chaque variable locale, lui associer un numéro compris entre 1 et n. (Ce numéro doit être différent pour chacune des variables locales du sous-programme.)

2. s'il n'y en a déjà, ajouter les deux instructions

```
push    bp
mov     bp, sp
```

   au tout début du sous-programme, juste après `PROC`

3. après ces deux instructions, mettre une instruction

```
sub     sp, 2 n
```

   où vous remplacez l'expression `2 n` par sa valeur

4. à la fin du sous-programme, juste avant l'instruction `ret`, ajouter l'instruction

```
pop     bp
```

   si elle n'est pas déjà présente

5. juste avant cette instruction, ajouter l'instruction

```
add     sp, 2 n
```

6. dans le sous-programme, remplacer chaque occurence d'une variable locale par `WORD PTR [bp-2i]`, où i est le numéro associé à la variable. Nous aurons ainsi `WORD PTR [bp-2]` pour la première variable locale, `WORD PTR [bp-4]` pour la deuxième, `WORD PTR [bp-6]` pour la troisième, ...

Si l'on reprend le sous-programme `LIRE` de l'exemple qui possède une variable locale et que nous appliquons la méthode indiquée ci-dessus, nous obtenons le source suivant :

```
LIRE       PROC
           ; lit un nombre au clavier
           ;
           ; en entrée : pas de paramètre
           ; en sortie, bx contient la valeur du nombre saisi
           ;
           push      bp
           mov       bp, sp
           sub       sp, 2
           push      ax
           mov       bx, 0
repeter2:
           ; saisie d'un caractère au clavier
           mov       ah, 1
           int       21h
           mov       ah, 0
           mov       WORD PTR [bp-2], ax
           ;
           push      WORD PTR [bp-2]
           call      est_un_chiffre
           add       sp, 2
           cmp       ax, 0
           je        fin_de_repeter2
           ;
           mov       ax, 10
           mul       bx
           mov       bx, ax
           sub       WORD PTR [bp-2], '0'
           add       bx, WORD PTR [bp-2]
           jmp       repeter2
fin_de_repeter2:
           pop      ax
           add      sp, 2
           pop      bp
           ret
LIRE       ENDP
```

4.7.6 La valeur de retour d'un sous-programme

La valeur de retour d'un sous-programme sera transmise *via* un registre. Généralement, on la placera dans le registre `ax` s'il s'agit d'un mot-mémoire, `al` s'il s'agit d'un octet.

4.8 Les tableaux en assembleur

Du fait de leur utilisation courante, de nombreuses facilités existent en assembleur pour déclarer et traiter des tableaux.

La déclaration des tableaux

Un tableau se déclare dans le segment de données. Deux types de tableaux existent :

1. tableau d'octets (ou de caractères) ;
2. tableau de mots-mémoire (ou d'entiers).

Considérons les instructions ci-dessous :

```
        .data
        .......      ; autres déclarations
t1      db    10 dup (0)              ; tableau de 10 octets initialisés
t2      db    17 dup (?)              ; tableau de 17 octets non initialisés
t3      db    5 dup ('e')             ; tableau de 5 octets initialisés
t4      dw    8 dup (10000)           ; tableau de 8 mots initialisés
t5      dw    4 dup ((43 + 56) * 25)  ; tableau de 4 mots initialisés
t6      db    'hello'                 ; chaîne de caractères
t7      dw    'hello'                 ; ceci n'est pas une chaîne de caractères
        .......      ; suite des déclarations
```

Déclaration de tableaux en assembleur

Elles déclarent les tableaux suivants :

- `t1` : tableau de 10 octets dont la valeur initiale de chacun des élélemnts est 0
- `t2` : tableau de 17 octets dont la valeur initiale des éléments n'est pas fixée
- `t3` : tableau de 5 octets dont la valeur initiale des éléments est le code ASCII de la lettre `e` minuscule
- `t4` : tableau de 8 mots dont la valeur initiale de chacun des éléments est 10000
- `t5` : tableau de 4 mots dont la valeur initiale de chacun des éléments est 2475
- `t6` : tableau de 5 octets dont le premier élément est initialisé avec le code ASCII de la lettre `h` minuscule, le deuxième élément avec le code ASCII de la lettre `e` minuscule, ... Ce tableau constitue une chaîne de 5 caractères, les chînes de caractères étant simplement des tableaux d'octets en assembleur
- `t7` : tableau de 5 mots-mémoire dont le premier élément est initialisé avec le code ASCII de la lettre `h` minuscule, le deuxième élément avec le code ASCII de la lettre `e` minuscule, ... Il ne faut en aucun cas confondre ce tableau avec le précéent (`t6`) : **le tableau `t7` n'est pas une chaîne de caractères** parce que les éléments du tableau ne sont pas des octets, mais des mots.

Utilisation d'un tableau

L'utilisation d'un tableau consiste généralement à accéder en lecture ou en écriture à un élément du tableau. Pour cela, on utilise un indice qui spécifie l'élément accédé. L'autre utilisation consiste à passer un tableau en paramètre.

Il faut toujours se rappeler qu'un tableau en assembleur a ses éléments indicés à partir de 0. Ainsi, le tableau `t1` déclaré plus haut a 10 éléments indicés de 0 à 9.

Un élément de tableau peut-être spécifié comme opérande de n'importe qu'elle instruction de traitement. Si l'on veut accèder à l'élément indicé 3 du tableau `t`, on écrira : `t + 3`. Si une pseudo-variable `i` contient la valeur de l'indice à accèder, on écrira `t + i`.

De manière générale, les indices sont placés dans les registres `si` et `di`.

Nous laissons le passage de paramètre d'un tableau de côté.

4.9 Les constantes

Les constantes que l'on déclare dans un programme Pascal peuvent se traduire en assembleur à l'aide d'une pseudo-instruction `EQU`. On trouvera un exemple de traduction de constante dans la table ci-dessous :

Original	Traduction en assembleur de la constante
`CONST N := 10`	`N EQU 10`

Traduction d'une constante.

4.10 Programme assembleur minimal

Le programme assembleur minimal qui ne fait rien, qui s'assemble et s'exécute sans provoquer d'erreur et qui est la base d'un programme réel est le suivant :

```
.MODEL    SMALL
.STACK    100H
.DATA
.CODE
mov       ax, @data
mov       ds, ax
;
mov       ah, 4ch
int       21h
END
```

4.11 Quelques fonctions utiles

Pour terminer, nous indiquons ci-dessous quelques fonctions bien utiles qui permettent d'effectuer des entrées-sorties simples avec le clavier et l'écran.

4.11.1 Afficher un caractère

L'affichage d'un caractère est réalisé en appelant une routine du BIOS via l'instruction `int`. Voir le chapitre 5 du manuel assembleur.

```
affiche_caractere  PROC
                   ; le code ASCII du caractere a afficher a ete passe
                   ; en parametre
                   push    bp
                   mov     bp, sp
                   push    dx          ; sauvegarde des registres dx et ax
                   push    ax
                   ;
                   mov     dx, [bp+4]  ; on charge le parametre dans dx.
                                       ; Seule la valeur de dl est prise
                                       ; en compte par la suite (dh vaut 0)
                   mov     ah, 2
                   int     21h         ; affichage
                   ;
                   pop     ax          ; restauration du contenu des
                   pop     dx          ; registres sauvegardes plus haut
                   pop     bp
                   ret
affiche_caractere  ENDP
```

4.11.2 Afficher un nombre positif

De nombreux algorithmes peuvent être utilisés. Nous en proposons un. Le nombre à afficher1q est stocké sur un mot mémoire (16 bits). Donc sa valeur varrient entre 0 et 65535 ce qui permet d'utiliser des algorithmes reposant sur ce fait.

L'algorithme utilisé est le suivant :

```
PROCEDURE affiche_entier (IN nombre)
diviseur := 10000
indicateur := 0
REPETER
  quotient := nombre / diviseur
  reste := nombre MOD diviseur
  SI (quotient # 0) OU (indicateur = 1) ALORS
    code_ascii := quotient + '0'
    affiche_caractere (code_ascii)
    indicateur := 1
  FIN_SI
  nombre := reste
  diviseur := diviseur / 10
  SI diviseur = 1 ALORS
    indicateur := 1
  FIN_SI
JUSQUE (diviseur = 0)
```

L'algorithme procède par divisions successives du nombre à afficher par 10000, 1000, 100, 10 puis 1 (valeurs contenus successivement dans la variable `diviseur`).

On prend garde de ne pas afficher de `0` non significatifs à gauche du nombre et d'afficher `0` si le nombre est nul. La variable `indicateur` est utilisée pour cela : elle indique si le chiffre (c'est-à-dire la valeur de la variable `quotient`) doit être affiché ou non. Il doit être affiché si un chiffre a déjà été affiché ou si l'on a atteint le chiffre des unités.

```
affiche_nombre  PROC
                ; ce sous-programme prend un paramètre : la valeur
                ; du nombre à afficher
```

```
                ; On alloue les données de la manière suivante :
                ;  ax : quotient
                ;  bx : nombre
                ;  cx : diviseur
                ;  dx : reste
                ;  si : indicateur
                push    bp
                mov     bp, sp
                push    ax          ; sauvegarde des registres utilisés
                push    bx
                push    cx
                push    dx
                push    si

                ;
                ; initialisations
                mov     bx, [bp+4]  ; bx <- nombre passé en paramètre
                mov     cx, 10000   ; initialise diviseur
                mov     si, 0       ; initialise indicateur

repeter:        ;
                ; division nombre / diviseur
                mov     dx, 0       ; préparation de la division
                mov     ax, bx
                div     cx          ; on fait la division

                ;
                ; analyse du resultat de la division
                cmp     ax, 0       ; le chiffre est-il 0 ?
                jne     affiche     ; non, donc on l'affiche
                cmp     si, 1       ; chiffre = 0 ; doit-on l'afficher ?
                jne     fin_si

affiche:        ;
                ; affichage d'un chiffre du nombre
                add     ax, '0'     ; affichage du chiffre
                push    ax
                call    affiche_caractere
                add     sp, 2
                mov     si, 1       ; indicateur positionné
fin_si:         mov     bx, dx      ; nombre := reste

                ; diviseur := diviseur / 10
                mov     dx, 0       ; prépare la division
                mov     ax, cx
                mov     cx, 10
                div     cx
                mov     cx, ax      ; affecte le résultat de la division
                                    ; au registre cx

                ;
                ; SI diviseur = 1 ALORS indicateur := 1
                cmp     cx, 1       ; si le diviseur vaut 1, on va s'occuper
                jne     suite       ; du chiffre des unites qu'il faut
                mov     si, 1       ; toujours afficher

suite:          ;
                ; SI diviseur # 0 ALORS on itère
                cmp     cx, 0
                jne     repeter

                ;
                ; restauration des registres
                pop     si
                pop     dx
                pop     cx
                pop     bx
                pop     ax
                pop     bp
                ret
affiche_nombre  ENDP
```

4.11.3 Lire un caractère

```
lit_caractere  PROC
               ; lit le code ASCII d'un caractère
               ; Ce code se trouve dans le registre al au retour de l'appel
               mov     ah, 1
```

```
                int     21h         ; lecture
                ret
lit_caractere   ENDP
```