Chapitre 8
Structure, documentation et mise au point des programmes

8.1 Objectifs de la programmation

La programmation n'est pas une discipline très ancienne, et depuis le début de l'informatique elle a beaucoup évolué. Les premiers programmes étaient écrits en langage machine (programmation numérique) et les inconvénients de cette programmation ont rapidement conduit aux langages d'assemblage qui permettent la programmation symbolique. A l'époque, l'objectif principal de la programmation en langage d'assemblage était *l'efficacité* des programmes; on entendait alors par efficacité un nombre d'instructions minimum dans un espace mémoire souvent restreint. Cet objectif unique a été conservé pendant plus d'une décennie; il a conduit à un certain nombre de problèmes causés par la compacité et l'hermétisme de programmes produits dans un style souvent très personnel, et par la mobilité importante des programmeurs laissant derrière eux des programmes souvent incompréhensibles à d'autres qu'eux-mêmes et par conséquent inutilisables. Cet état de choses a conduit progressivement à une redéfinition des objectifs de la programmation en langage d'assemblage. Les objectifs actuels de cette programmation sont ainsi la *simplicité*, la *clarté* et la *fiabilité* des programmes, l'efficacité ne venant qu'ensuite et ayant pris un sens différent.

8.1.1 Clarté des programmes

Un programme est destiné à représenter un algorithme dans une forme interprétable par l'ordinateur. Cependant si ce programme doit être utilisé régulièrement il devra vraisemblablement être corrigé, modifié et revu, c'est ce qu'on appelle la *maintenance*. L'écriture d'un programme doit se faire avec ces deux considérations en tête.

Lorsqu'on doit reprendre ou étudier un programme écrit par un autre ou par soi-même quelques mois plus tôt, il arrive souvent que l'on pose aigrement la question:

"??*!*!*!?! Mais qu'est-ce que ça peut bien faire ???"

Ceci ne devrait pas se produire et ne se produira pas si le programme est clair.

Une manière de rendre le programme clair sera de le documenter *au fur et à mesure* de sa création; l'utilisation d'identificateurs ayant un sens relié au problème permettra d'abord d'aboutir à un code plus compréhensible. De plus on y ajoutera au fur et à mesure de l'écriture les divers commentaires nécessaires (voir la section 8.3). **Cette documentation doit être faite en même temps que le codage**.

Un programme sera clair s'il est bien structuré et si sa structure correspond bien à la solution logique du problème. Le manque de structure correspond à ce qu'on appelle du "*spaghetti logique*", assez difficile à démêler. Par ailleurs l'ordre physique des différentes parties du programme contribue à en augmenter la clarté; ainsi pour un programme en assembleur on devrait voir apparaître les éléments dans l'ordre suivant:

Commentaires explicatifs généraux sur le programme
Déclarations des constantes (.EQUATE)
Code du programme principal
Réservation de mémoire et variables globales

```
        Code des sous-programmes
        .END
```

Un autre point contribuant à la clarté des programmes consiste à en faciliter la modification en groupant les opérations semblables; ainsi toutes les vérifications à faire sur les données devraient se trouver ensemble; de la même façon l'impression des messages d'erreur devrait se trouver groupée dans la même partie du programme, etc.

8.1.2 Efficacité des programmes

Une fois un programme écrit, on l'examine pour en améliorer l'efficacité. Ceci consiste surtout à déterminer quelles sont les parties du code les plus exécutées et à essayer d'en améliorer l'écriture en diminuant le nombre d'instructions.

Lorsque l'ordinateur utilisé possède plusieurs registres, l'utilisation judicieuse des registres permet souvent d'améliorer de beaucoup l'efficacité des programmes, et ceci, assez facilement. Les opérations faites sur les registres sont en effet plus rapides; il ne faut cependant pas non plus utiliser les registres à tort et à travers; le nombre maximum de registres utilisés dans un programme devrait être le nombre maximum d'accumulateurs nécessaires simultanément. Généralement, lorsqu'on programme en assembleur, on réserve un rôle spécial à certains registres; lorsqu'on doit utiliser plusieurs accumulateurs on doit les prendre contigus et à partir du début.

L'amélioration de l'efficacité des programmes suppose d'abord une bonne connaissance de l'ensemble d'instructions de l'ordinateur utilisé. À partir de là, on doit essayer de tirer le meilleur parti possible des instructions combinant plusieurs actions.

8.2 Structures de base

Un des moyens d'arriver à obtenir des programmes en langage d'assemblage clairs est d'utiliser les structures offertes par un langage évolué comme Ada 95 ou C++ et de les traduire en langage d'assemblage selon des règles précises. On présente ici un ensemble de règles illustrées par des exemples.

8.2.1 Boucles

WHILE

L'instruction WHILE comprend la vérification d'une condition à l'entrée de la boucle et, en fin de boucle, un branchement au début de la boucle. Si la vérification réussit, l'exécution des instructions de la boucle continue et lorsque la vérification échoue il y a branchement à l'instruction suivant immédiatement la boucle.

```
WHILEnn:    NOP0                    ;while
            CPA     N2,d            ;  (N1 ≥ N2){
            BRLT    ENDWHInn        ;
                    #CODE#          ;    commentaires
            BR      WHILEnn         ;
ENDWHInn:   NOP0                    ;}
```


Dans nos schémas #CODE# représentera le bloc d'instructions à répéter. Les instructions de vérification de la condition changeront évidemment avec chaque application. Notez que nous utilisons l'instruction NOP0 (un code d'instruction non réalisé dont le traitement normal ne produit rien) pour pouvoir placer des étiquettes dans le code du programme sans les attacher à des instructions spécifiques comme l'exigent les règles de PEP 8 qui veulent qu'une étiquette soit suivie d'une instruction sur la même ligne.

REPEAT

L'instruction REPEAT est l'inverse de l'instruction WHILE: la vérification de la condition est située en fin de boucle et si l'exécution doit se poursuivre il y a branchement au début de la boucle. Les instructions de la boucle sont exécutées au moins une fois, ce qui n'était pas le cas dans l'instruction WHILE.

```
REPEATnn:   NOP0                    ;do{
            #CODE#                  ;   commentaires
            ADDX    4,i             ;   augmenter indice
            CPA     Table,x         ;   vérifier élément suivant
UNTILnn:    BRLT    REPEATnn        ;}while(Table[i] > Elt);
```

La notation retenue pour les symboles de début et de fin de boucle (WHILEnn, ENDWHILEnn, REPEATnn, UNTILnn) permet de numéroter les boucles de même type, de façon à éliminer toute ambiguïté. Illustrons ceci par une validation des données lues.

```
; validation des données
REPEAT05:   NOP0                    ;do{
            DECI    donne,d         ;  Get(donnée)
            LDA     donne,d         ;
            CPA     limInf,d        ;}while(donnée < limInf
            BRLT    REPEAT05        ;    &&
            CPA     limSup,d        ;      donnée > limSup) ;
UNTIL05:    BRGT    REPEAT05        ;
```

Boucles généralisées

Dans certains cas la condition déterminant la fin de la boucle peut être détectée au milieu des instructions à répéter et les instructions WHILE et REPEAT s'avèrent malcommodes. Une boucle généralisée correspond par exemple à l'instruction LOOP d'Ada 95 qui peut avoir plusieurs sorties repérées par le mot réservé EXIT, ou à l'instruction `while(true)` de C++. Dans nos schémas ces sorties seront indiquées dans le pseudo-code des commentaires par le symbole :> aligné sur le symbole LOOP de début de boucle.

```
LOOPnn:   NOP0                      ;while(true){ [1 sortie]
          #CODE#                    ;  commentaires
          SUBX      pas,d           ;
          BRGT      ENDLPnn         ;:>if(A > B) break;
          #CODE#                    ;  commentaires
          BR        LOOPnn          ;
ENDLPnn:  NOP0                      ;}
```

Une boucle généralisée peut posséder plusieurs sorties; il est alors impératif de l'indiquer à l'entrée de la boucle dans le pseudocode: [2 sorties]. Ces boucles peuvent également être imbriquées; on doit alors

documenter les sorties de boucle en faisant précéder le test de sortie d'un nombre de signes :> indiquant le nombre de niveaux quittés.

Boucles FOR
De telles boucles se traduisent facilement par utilisation d'un compteur selon le schéma suivant.

```
FORnn:      NOP0                    ;for(Ctr = Inf; Ctr <= Sup; Ctr++){
            LDX       Ctr,d         ;  <Ctr == Sup>
FTESTnn:    NOP0                    ;
            CPX       Sup,d         ;
            BRGT      ENDFORnn      ;
            #CODE#                  ;  commentaires
            ADDX      2,i           ;  le pas égal est égal à 2
            BR        FTESTnn       ;
ENDFORnn:   NOP0                    ;}
```

Ce schéma utilise un compteur simple mais les instructions de comparaison CPX et d'addition ADDX peuvent être aisément remplacées; de même le pas peut être modifié.

8.2.2 Instructions conditionnelles

Les instructions conditionnelles vérifient une condition et selon le résultat de cette vérification exécutent ou évitent un ensemble d'instructions.

Instruction IF simple
Le schéma d'une telle instruction (à une seule branche) sera le suivant :

```
IFnn:       NOP0                    ;if
            <évaluation expression> ;  (condition)
            CPr       opérandes     ;
            BR--      ENDIFnn       ;
THENnn:     NOP0                    ;{
            #CODE#                  ;  commentaires
ENDIFnn:    NOP0                    ;}
```

Instruction IF complète.
Dans ce cas, l'instruction possède deux branches possibles; après la vérification on place un branchement à la partie ELSE qui sera activé si la condition vérifiée est fausse. On notera que la différence entre les structures présentées et une solution non structurée tient essentiellement dans l'addition d'étiquettes qui n'affectent pas le code objet. On doit être conscient du fait que ces additions sont utiles au programmeur au cours du développement et de la mise au point.

```
IFnn:       NOP0                    ;if
            <évaluer expression>    ;  (condition)
            CPr       opérandes     ;
            BR--      ELSEnn        ;
THENnn:     NOP0                    ;{
            #CODE#                  ;  commentaires
            BR        ENDIFnn       ;}
ELSEnn:     NOP0                    ;else {
            #CODE#                  ;  commentaires
```



```
ENDIFnn:    NOP0                    ;}
```

Instructions IF imbriquées

Avec la notation retenue l'imbrication d'instructions IF ne pose pas de problème particulier.

```
IF04:       NOP0                    ;if(X > Z)
            LDA       X,d           ;
            CPA       Z,d           ;
            BRLE      ELSE04        ;
THEN04:     NOP0                    ;{
            #CODE#                  ;  commentaires
            BR        ENDIF04       ;}
ELSE04:     NOP0                    ;else {
            #CODE#                  ;  commentaires
IF05:       NOP0                    ;  if(Y > X)
            LDA       Y,d           ;
            CPA       X,d           ;
            BRLE      ELSE05        ;
THEN05:     NOP0                    ;  {
            #CODE#                  ;    commentaires
            BR        ENDIF05       ;  }
ELSE05:     NOP0                    ;  else {
IF06:       NOP0                    ;    if(Z < W)
            LDA       Z,d           ;
            CPA       W,d           ;
            BRGE      ELSE06        ;
THEN06:     NOP0                    ;    {
IF07:       NOP0                    ;      if(V == W)
            LDA       V,d           ;
            CPA       W,d           ;
            BRNE      ENFIF07       ;
THEN07:     NOP0                    ;      {
            #CODE#                  ;        commentaires
ENDIF07:    NOP0                    ;      }
            BR        ENDIF06       ;    }
ELSE06:     NOP0                    ;    else {
            #CODE#                  ;      commentaires
ENDIF06:    NOP0                    ;    }
            #CODE#                  ;    commentaires
ENDIF05:    NOP0                    ;  }
ENDIF04:    NOP0                    ;}
```

8.2.3 Instruction SWITCH

Cette instruction est semblable a un ensemble d'instructions IF imbriquées; cependant elle présente une structure plus claire que celle obtenue par un grand nombre de IF imbriqués. En effet le choix à faire dépend d'une valeur entière qu'il suffit de vérifier; toutes les branches sont alors identifiées et se rejoignent à la fin de l'instruction.

```
SWITCHnn:     NOP0                  ;switch(Ind) {
              LDX       Ind,d       ;
              CPX       1,i         ;
              BRLT      OTHERnn     ;  Vérifier si Ind est
              CPX       5,i         ;  dans les limites [1..5]
              BRGT      OTHERnn     ;
              SUBX      1,i         ;  ramener à [0..4]
              ASLX                  ;  * 2 => indice réel dans
              BR        TABnn,x     ;  table (adresse = 2 octets)
TABnn:        .ADDRSS   CASnn_1     ;
              .ADDRSS   CASnn_2     ;
              .ADDRSS   CASnn_3     ;
              .ADDRSS   CASnn_4     ;
              .ADDRSS   CASnn_5     ;
CASnn_1:      NOP0                  ;  case 1:
              #CODE#                ;    commentaires
              BR        ENDCASnn    ;    break;
CASnn_2:      NOP0                  ;  case 2:
              #CODE#                ;    commentaires
              BR        ENDCASnn    ;    break;
CASnn_3:      NOP0                  ;  case 3:
              #CODE#                ;    commentaires
              BR        ENDCASnn    ;    break;
CASnn_4:      NOP0                  ;  case 4:
              #CODE#                ;    commentaires
              BR        ENDCASnn    ;    break;
CASnn_5:      NOP0                  ;  case 5:
              #CODE#                ;    commentaires
              BR        ENDCASnn    ;    break;
OTHERnn:      NOP0                  ;  default:
              #CODE#                ;    commentaires
ENDCASnn:     NOP0                  ;}
```

8.3 Normes de programmation

Les normes de conception et de documentation qui suivent sont semblables à la pratique industrielle courante bien qu'elles puissent en différer par la forme[1]. On devra prévoir un guide d'utilisation ainsi que des documentations externes et internes pour faciliter la compréhension d'un programme.

Le *guide d'utilisation* est simplement une description indiquant à un non programmeur comment utiliser le système. Il doit comprendre une description du format d'entrée des données du programme et des

[1] Voir à ce sujet : Gabrini, Ph. *Normes de programmation*, Département d'informatique, UQAM, 2005 http://www.grosmax.uqam.ca/prog/Inf3105/Inf3105_frame.htm

résultats produits. On devra, dans la mesure du possible, produire des programmes capables de traiter les erreurs et les données erronées de façon claire, utile et compréhensible; le guide d'utilisation devra décrire comment le programme traite ces conditions anormales.

La *documentation externe* d'un système comprend la description de chaque module principal ainsi qu'une explication plus générale de la façon dont ces diverses composantes communiquent pour former un système complet. Pour chaque module important, cette description doit comprendre les spécifications des appels, des paramètres, de l'utilisation des données, de l'algorithme, etc. La stratégie de conception globale du système devrait être aussi décrite à un niveau de détail dépendant de sa complexité, sous forme de diagrammes hiérarchiques et de pseudo-code proche du français.

Cependant, pour un sous-programme, le guide d'utilisation et la documentation externe se fondent en un seul document orienté vers le programmeur voulant utiliser le sous-programme. On doit inclure une description de haut niveau de l'algorithme avec les détails sur les paramètres, les codes de retour et d'autres informations de ce type. Ceci peut, en fait, faire partie de la documentation interne.

La question de savoir où placer la limite entre documentation externe et documentation interne est difficile à résoudre. Cela dépend de la taille et de l'utilisation du programme. Un gros système exige plusieurs types de documentation externe, comme un guide d'utilisation pour le non programmeur et un manuel exposant la logique du programme à des fins de maintenance. D'autre part un sous-programme nécessite un bref guide d'utilisation pour un programmeur ainsi qu'une brève description logique.

La documentation interne commence au niveau où en est resté la documentation externe. Des commentaires sont placés devant chaque module en expliquant la fonction, l'utilisation des registres, les paramètres d'appel, les résultats, etc. De courts commentaires descriptifs sont aussi utilisés avant les ensembles d'instructions particulièrement complexes. La documentation des programmes en langage d'assemblage diffère ici de celle des programmes en langages évolués et est poussée plus loin, en demandant pratiquement des commentaires pour chaque instruction. Ces commentaires sont soit des descriptions de branchement indiquant comment on utilise les branchements conditionnels et inconditionnels représentant les instructions structurées, soit des descriptions en français de ce que fait l'instruction (et les instructions suivantes). Ces commentaires en français doivent indiquer le but de l'instruction dans le contexte des instructions qui l'entourent et donner une information utile comme « Augmenter le compteur de données erronées » et non une information inutile comme « Ajouter 1 au registre A ». Ces commentaires doivent être décalés pour refléter l'imbrication des structures de contrôle, de sorte qu'un lecteur puisse suivre les instructions en regardant simplement les commentaires rapidement de haut en bas.

On peut utiliser le pseudocode écrit lors de la phase de conception des programmes comme commentaires. Le but de ces commentaires en pseudocode est de minimiser la mise au point en permettant une simulation à la main du pseudocode et une vérification de la traduction de ce pseudo-code. Dans le cas de commentaires en pseudocode il est inutile d'utiliser les étiquettes numérotées reflétant la structure (comme celles que nous avons présentées plus tôt : IFnn, THENnn, etc.) puisque celle-ci est déjà apparente dans le pseudocode. Si les commentaires ne sont pas en pseudocode, il est nécessaire d'utiliser ces étiquettes qui sont le seul reflet de la structure du programme.

Tout ceci n'empêche évidemment pas d'utiliser les espacements, les sauts de page, les en-têtes placés dans des boîtes, l'utilisation d'étiquettes et d'identificateurs ayant un sens, etc.

8.4 Exemples de programmes complets

8.4.1 Programme de multiplication rapide

Comme le programme vu au chapitre 7, qui effectuait la multiplication de deux nombres entiers, ce programme effectue, lui aussi, la multiplication de deux valeurs entières lues en entrée. La méthode utilisée est cependant plus rapide et surtout plus astucieuse, ce qui la rend, sans doute, moins facile à comprendre. La lecture des valeurs, l'affichage des valeurs lues et le traitement en cas de valeur négative sont semblables à ce qui a déjà été vu. Les valeurs à traiter sont placées dans les registres A et X, ainsi que dans les variables `mult2` et `div2`. La valeur de `div2` est continuellement divisée par deux (par un décalage à droite) et chaque fois qu'il y a un reste, on ajoute la valeur de `mult2` au résultat. La valeur de `mult2`, elle, est parallèlement multipliée par deux au moyen d'un décalage à gauche. Le traitement continue jusqu'à ce que `div2` soit nul.

Pour illustrer ce traitement, si les valeurs lues sont -9 pour `n` et 11 pour `m`, on place -9 dans `mult2`, 11 dans `div2`, et 0 dans `resultat`. La valeur de `div2` passe à 5, celle de `resultat` à -9, et celle de `mult2` à -18 ; ensuite, la valeur de `div2` passe à 2, celle de `resultat` à -27, et celle de `mult2` à -36 ; puis, la valeur de `div2` passe à 1, celle de `resultat` ne change pas, et celle de `mult2` passe à -72 ; enfin, la valeur de `div2` passe à 0, celle de `resultat` à -99, et celle de `mult2` à -144 et le traitement se termine par affichage de la valeur de `resultat` : **`-9*11=-99`**.

```
; Multiplication de deux nombres entiers par la méthode des moujiks.
;   Lorne H. Bouchard (adapté par Ph. Gabrini mars 2006)
;
Multi:    DECI     n,d        ; lire n
          DECI     m,d        ; line m
          DECO     n,d        ; 'n'
          CHARO    '*',i      ; '*'
          DECO     m,d        ; 'm'
          CHARO    '=',i      ; '='
          LDX      m,d        ; RegX = m;
          BRGE     Debut      ; if(m < 0){
          NEGX                ;
          STX      m,d        ;   RegX = m = -m;
          LDA      n,d        ;
          NEGA                ;
          STA      n,d        ;   RegA = n = -n;
Debut:    LDA      n,d        ;  }
          STA      mult2,d    ; mult2 = n;
          LDA      m,d        ;
          STA      div2,d     ; div2 = m;
          LDA      0,i        ;
          STA      resultat,d ; resultat = 0;
          LDA      mult2,d    ;
          LDX      div2,d     ;
Repete:   CPX      0,i        ; while(div2 != 0){
          BREQ     Fini       ;
          ASRX                ;   div2 /= 2;
```

```
          BRC     Ajoute    ;   if(div2 % 2 != 0)  Ajoute
Par2:     ASLA              ;   mult2 *= 2;
          BR      Repete    ; }
Ajoute:   STA     mult2,d   ;
          ADDA    resultat,d;
          STA     resultat,d;   resultat += mult2;
          LDA     mult2,d   ;
          BR      Par2      ;
Fini:     DECO    resultat,d;
          STOP              ;
n:       .WORD    0         ;
mult2:   .WORD    0         ; puissances binaires de n
m:       .WORD    0         ;
div2:    .WORD    0         ; facteurs binaires de m
resultat:.WORD    0         ;
         .END
```

8.4.2 Programme de tri

La figure 8.3 donne un exemple de programme complet. Ce programme lit les valeurs d'un vecteur de 20 entiers au terminal (grâce à l'appel de l'instruction `DECI` dans la boucle `Lire` répétée 20 fois). On notera l'emploi de l'instruction de comparaison `CPX` et du mode d'adressage indexé, « `,x` » qui permet de progresser dans le vecteur mot par mot.

On applique ensuite aux éléments du vecteur un algorithme de tri simple, qui trie les éléments du vecteur en ordre croissant. On utilise un algorithme de tri par échanges qui fait N-1 passes sur les éléments du vecteur et qui, à chaque passe, recherche le plus grand des éléments rencontrés et le range dans sa position finale dans le vecteur. Cet algorithme est réalisé au moyen des deux boucles imbriquées suivantes:

```
for(int i = 19; i >=0; i--)
     for(int j = i-1; j >= 0; j--)
          if(Vec[j] > Vec[i])
               Echanger(Vec[j],Vec[i]);
```

La progression des compteurs du programme assembleur dans ces boucles se fera avec un pas de 2 pour respecter la taille des éléments entiers du vecteur (mots de 2 octets); le contrôle de la boucle externe et de la boucle imbriquée sera fait par deux instructions `CPX`. Comme les éléments du vecteur occupent chacun deux octets on doit multiplier l'indice d'un élément par deux pour obtenir le vrai indice (en termes d'octets) dans le vecteur.

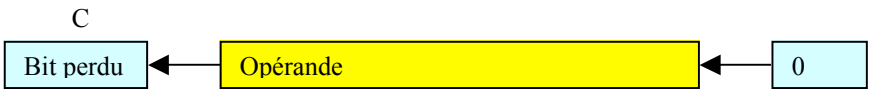

Figure 8.1 Décalage arithmétique à gauche ASLr

Pour multiplier une valeur binaire par deux il suffit de la décaler vers la gauche d'une position, exactement comme la multiplication par dix se fait dans le système décimal. Pour faire cela nous utilisons une instruction `ASLr` (Arithmetic Shift Left Register). Comme on l'a vu au chapitre précédent,

cette instruction décale le contenu du registre d'une position vers la gauche en faisant rentrer un bit zéro de la droite comme le montre la figure 8.1. Le bit perdu à gauche est placé dans le code de condition C. Les autres codes de condition sont affectés par l'opération de la façon suivante :

N prend la valeur du bit le plus significatif du résultat;
Z indique si le résultat est zéro ou non;
V prend la valeur 1 si le bit le plus significatif de l'opérande a changé.

Comme l'indice est situé dans le registre d'index X, nous utilisons l'instruction `ASLX`. Une dernière boucle fait imprimer les éléments du vecteur, du premier au dernier séparés par des espaces grâce aux instructions `DECO` et `CHARO` comme le montre la figure 8.2 qui illustre l'exécution du programme.

```
Donnez une valeur: 9
Donnez une valeur: 8
Donnez une valeur: 7
Donnez une valeur: 6
Donnez une valeur: 5
Donnez une valeur: 4
Donnez une valeur: 3
Donnez une valeur: 2
Donnez une valeur: 1
Donnez une valeur: 10
Donnez une valeur: 20
Donnez une valeur: 11
Donnez une valeur: 19
Donnez une valeur: 12
Donnez une valeur: 18
Donnez une valeur: 13
Donnez une valeur: 17
Donnez une valeur: 14
Donnez une valeur: 16
Donnez une valeur: 15

1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20
Fin du traitement
```

Figure 8.2 Exécution du programme de tri

```
;   Tri d'une table d'entiers en ordre croissant
;     Lecture des valeurs au terminal
;     Tri de la table en ordre croissant
;     Affichage des valeurs triées au terminal
;  Philippe Gabrini    Octobre 2005
TAILLE: .EQUATE 20             ;taille du vecteur 0..19
LF:     .EQUATE 0xA            ;Line Feed
                               ;int main(){
Tri:    LDX     0,i            ;  indice de boucle de lecture
        STX     indice1,d      ;
Lire:   CPX     TAILLE,i       ;  for(int indice = 0; indice <= 19; indice++){
        BRGE    Trier          ;
        STRO    msg1,d         ;    cout << "Donnez une valeur: ";
        LDX     indice1,d      ;    avance dans vecteur
        ASLX                   ;      2 octets
        DECI    vecteur,x      ;    cin >> vecteur[i];
        LDX     indice1,d      ;      avance dans vecteur
        ADDX    1,i            ;
        STX     indice1,d      ;
        BR      Lire           ;  }//for
```

Figure 8.3 Programme de tri (début)

```
Trier:  LDX     TAILLE,i       ;
        STX     indice1,d      ;
```

```
Boucle1:LDX     indice1,d     ;  for(int indice1 = 19; indice1 >=0; indice1 --){
        SUBX    1,i           ;
        STX     indice1,d     ;
        CPX     0,i           ;
        BRLT    Sortir        ;
        STX     indice2,d     ;
        ASLX                  ;    [ 2 octets par mot ]
        LDA     vecteur,x     ;    [ AC= vecteur[indice1] ]
Boucle2:LDX     indice2,d     ;    for(int indice2 = i-1; indice2 >= 0;indice2--){
        SUBX    1,i           ;      [indice2--;]
        STX     indice2,d     ;
        CPX     0,i           ;
        BRLT    FinBouc       ;
        ASLX                  ;      [ 2 octets par mot]
        CPA     vecteur,x     ;      if(vecteur[indice2]>vecteur[indice1]){
        BRGE    PasEch        ;
        STA     temp1,d       ;         Echange(vecteur[indice1],vecteur[indice2])
        LDA     vecteur,x     ;
        STA     temp2,d       ;         vecteur[indice2]
        LDX     indice1,d     ;
        ASLX                  ;         [ 2 octets par mot ]
        STA     vecteur,x     ;
        LDA     temp1,d       ;
        LDX     indice2,d     ;
        ASLX                  ;         [ 2 octets par mot ]
        STA     vecteur,x     ;
        LDA     temp2,d       ;      }//if
PasEch: BR      Boucle2       ;    }//for
FinBouc:BR      Boucle1       ;  }//for

Sortir: CHARO   LF,i          ;  cout << endl;
        LDX     0,i           ;  for(int i = 0; i <= 19; i++)
        STX     indice1,d     ;
Sortie: ASLX                  ;    [ 2 octets par mot ]
        DECO    vecteur,x     ;    cout << vecteur[indice1]
        CHARO   ' ',i         ;         << ' ';
        LDX     indice1,d     ;
        ADDX    1,i           ;    [ i++ ]
        STX     indice1,d     ;
        CPX     TAILLE,i      ;
        BRLT    Sortie        ;

        CHARO   LF,i          ;  cout << endl
        STRO    msg2,d        ;       << "Fin du traitement"
        CHARO   LF,i          ;       << endl;
        STOP                  ;  return 0;
                              ;}//main
msg1:   .ASCII  "Donnez une valeur: \x00"
msg2:   .ASCII  "Fin du traitement\x00"
indice1:.WORD   0
indice2:.WORD   0
temp1:  .WORD   0
temp2:  .WORD   0
vecteur:.BLOCK 40    ;int vecteur[TAILLE]
        .END
```

Figure 8.3 Programme de tri (suite et fin)

8.4.3 Programme de recherche binaire

Il arrive très souvent que l'on ait à fouiller une table à la recherche d'un élément donné. La méthode de fouille la plus intuitive est une fouille séquentielle du début de la table jusqu'à ce qu'on trouve l'objet cherché ou que l'on atteigne la fin de la table. Ceci est acceptable lorsque la table fouillée est courte. Il est parfois possible de conserver les éléments d'une table en ordre croissant ou décroissant. Si on dispose d'une grande table ordonnée on pourra faire une fouille logarithmique; illustrons la méthode sur un exemple de table de 512 éléments.

Pour chercher une valeur S dans la table T, on commence par comparer S à la valeur du milieu de la table T[256]; si c'est la valeur cherchée, la fouille est terminée, sinon si S est plus grand que T[256], il ne nous reste qu'à examiner la moitié supérieure de la table. Si S est plus petit que T[256], il ne nous reste qu'à examiner la moitié inférieure de la table. Dans les deux cas la moitié restante est aussi une table ordonnée à laquelle on applique la même technique: examen de la valeur du milieu de la table et si ce n'est pas la bonne, élimination d'une des deux moitiés de cette table. Le processus est répété jusqu'à ce qu'on trouve l'élément cherché ou jusqu'à ce qu'il ne reste plus qu'un élément à examiner. Notre table de 512 éléments peut ainsi être fouillée en 10 étapes ou moins.

Étape	Nombre d'éléments restants
0	512
1	256
2	128
3	64
4	32
5	16
6	8
7	4
8	2
9	1

La figure 8.5 nous donne l'exemple d'un tel programme de fouille. Le programme commence par placer les limites de la table dans les variables `bas` et `haut`.

La boucle principale commence à l'étiquette `Encore` et est répétée jusqu'à ce que les indicateurs de début et de fin de table (`haut` et `bas`) se rejoignent. Dans cette boucle, on calcule le milieu relatif de la table par addition des deux limites et division par deux de cette somme (un décalage à droite d'une position de la valeur binaire accomplit cette division). On calcule le vrai indice de l'élément milieu de la table en multipliant `milieu` par 2 pour tenir compte de la taille des éléments du vecteur (un mot, soit 2 octets). On compare alors l'élément cherché à l'élément milieu de la table. S'il y a égalité, on passe a l'instruction `Trouve`; sinon, si l'élément cherché est supérieur à l'élément milieu, on ajuste l'indicateur du bas de la table en lui donnant la valeur du milieu relatif augmentée de 1. Sinon, l'élément cherché est inférieur à l'élément milieu et on ajuste l'indicateur du haut de la table en lui donnant la valeur du milieu relatif diminuée de 1. On vérifie alors que la limite inférieure est toujours inférieure à la limite supérieure et on boucle, sinon la fouille se termine par un échec qui fera imprimer zéro. La fin du programme fait appel aux instructions `DECO` et `CHARO` pour indiquer l'indice de la valeur trouvée dans le tableau ou zéro en cas d'échec.

Pour diviser une valeur binaire par deux, il suffit de la décaler vers la droite d'une position, exactement comme la division par dix se fait dans le système décimal. Pour faire cela nous utilisons une instruction `ASRr` (Arithmetic Shift Right Register). Comme on l'a vu au chapitre précédent, cette instruction décale le contenu du registre d'une position vers la droite en faisant rentrer une copie du bit de signe de la gauche comme le montre la figure 8.4.

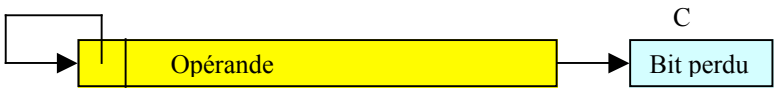

Figure 8.4 Décalage arithmétique à droite ASRr

Le bit perdu à droite est placé dans le code de condition C. Les autres codes de condition affectés par l'opération sont les suivants :

N prend la valeur du bit le plus significatif du résultat;
Z indique si le résultat est zéro ou non;

Comme l'indice est situé dans le registre d'index X, nous utilisons l'instruction `ASRX`.

```
; Fouille logarithmique d'une table d'entiers ordonnée
; On cherche un élément dans une table d'entiers, si on le
; trouve, le résultat est l'indice de cet élément sinon le
; résultat est nul
;
;     Philippe Gabrini   Octobre 2005
;
ELEMENT: .EQUATE     13    ;  valeur à trouver
N:       .EQUATE     20    ;  nombre d'éléments du vecteur
                           ;int main(){
Fouille: LDA     0,i       ;  bas = limite inférieure
         STA     bas,d     ;
         LDA     N,i       ;  haut = limite supérieure
         SUBA    1,i       ;
         STA     haut,d    ;
Encore:  LDX     bas,d     ;  while(true){
         ADDX    haut,d    ;    milieu = bas+haut
         ASRX              ;                / 2
         ASLX              ;                * taille (2);
         LDA     tableau,x ;    élément = valeur à chercher
         CPA     ELEMENT,i ;    if(élément == tableau[indice])
         BREQ    Trouve    ;      trouvé
         BRLT    Ajubas    ;    else if(élément < tableau[indice])
         SUBX    2,i       ;      haut = milieu - 1
         ASRX              ;             / 2;
         STX     haut,d    ;
         BR      Verifin   ;    else
Ajubas:  ADDX    2,i       ;      bas = milieu + 1
         ASRX              ;             / 2;
         STX     bas,d     ;
Verifin: LDA     haut,d    ;
         CPA     bas,d     ;
         BRGE    Encore    ;  if(bas > haut) break;
Echec:   LDX     -2,i      ;    pour obtenir un index qui soit zéro
Trouve:  ASRX              ;    index / 2
         ADDX    1,i       ;    index++; (indices commencent à 1)
         STX     indice,d  ;
         DECO    indice,d  ;  cout >> index
         CHARO   0xA,i     ;       >> endl;
         STOP              ;}//main

bas:     .WORD    0
haut:    .WORD    0
indice:  .WORD    0
tableau: .WORD    1
         .WORD    2
         .WORD    3
         .WORD    4
         .WORD    5
         .WORD    6
```

```
.WORD    7
.WORD    8
.WORD    9
.WORD    10
.WORD    11
.WORD    12
.WORD    13
.WORD    14
.WORD    15
.WORD    16
.WORD    17
.WORD    18
.WORD    19
.WORD    20
.END
```

Figure 8.5 Programme de fouille

8.5 Mise au point des programmes

L'objectif du programmeur est d'écrire des programmes corrects. Cependant, malgré tous les efforts déployés, un programmeur écrit rarement des programmes parfaits. Étant conscient de cela et en ayant fait l'expérience un grand nombre de fois, le programmeur sait qu'une fois un programme écrit, il reste à le vérifier, à en retirer les erreurs ou "bugs", en d'autres termes à le mettre au point.

L'objectif de la phase de mise au point est d'obtenir un programme qui produise les résultats escomptés et qui ait le comportement prévu dans des contextes divers. Avant d'être déclaré au point, un programme doit être essayé dans toutes sortes de combinaisons de circonstances pouvant conduire aux différents traitements prévus ou non par le programme. Ceci est difficile à définir de façon précise et également difficile à faire en pratique. Après des années d'utilisation sans problèmes, on découvre encore des programmes contenant des erreurs subtiles qui n'apparaissent que dans certaines circonstances inhabituelles. Même s'il est difficile d'être sûr qu'un programme est absolument sans erreur, un travail méthodique de mise au point et de vérification conduit à d'excellents résultats.

8.5.1 Principes de mise au point

Selon les auteurs des ouvrages de génie logiciel, il existe un plus ou moins grand nombre de principes permettant de mettre un programme au point de la meilleure façon. Cependant, comme l'écriture de programmes, la mise au point de programmes demeure un processus personnel demandant de la méthode, de la réflexion, de l'expérience, de la maturité et du bon sens.

Le premier principe que l'on peut indiquer est qu'avant qu'un programme ne soit soumis à sa première vérification, il doit être listé et relu attentivement. Il est en effet irréaliste de penser que l'ordinateur mettra les erreurs à jour; une relecture permettra de découvrir un certain nombre d'erreurs (ou d'oublis) que la mise au point mettrait beaucoup plus de temps à découvrir. Cette relecture permet entre autres de parfaire l'ensemble de vérifications déjà prévues.

Le second principe indique qu'un programme doit être découpé en parties qui remplissent chacune une fonction spécifique. Si ces parties sont suffisamment importantes pour former des sous-programmes, ceux-ci doivent être vérifiés séparément. Pour vérifier le fonctionnement d'une partie de programme il convient la plupart du temps d'insérer un certain nombre d'instructions de sortie supplémentaires

afin de pouvoir juger des résultats intermédiaires produits à différentes étapes de l'exécution du programme.

Le troisième principe est de déterminer exactement la raison d'une fin anormale de l'exécution du programme. Si une telle chose se produit, le système d'exploitation fournit un message explicatif parfois assez cryptique. Il est alors payant, en utilisant les manuels appropriés, de découvrir le sens du ou des messages qui permettent ainsi de partir à la recherche des erreurs sur une bonne piste et non à l'aveuglette. L'information fournie par le système sous forme de messages d'erreur doit être exploitée et non laissée pour compte.

Un quatrième principe sera d'isoler l'erreur; en effet, une fois une erreur découverte, que ce soit grâce à un message d'erreur du système ou grâce à des résultats erronés, il faut isoler la section de code qui a causé l'erreur. Dans le cas des petits programmes, ceci ne pose évidemment pas de difficultés, mais dans le cas de gros programmes ce sera moins facile; il se pourra d'ailleurs qu'il faille refaire exécuter le programme avec des instructions de vérification supplémentaires afin de pouvoir cerner la section de code responsable de l'erreur.

Un dernier principe de mise au point est celui de retracer ce qui s'est passé à partir de l'erreur. Un branchement à une mauvaise adresse n'a pu se produire que parce que l'adresse de branchement a été mal calculée; si cette adresse a été mal calculée et si le déplacement est bon, par exemple, peut-être faut-il vérifier le contenu du registre de base. Comme on peut supposer que ce dernier était bon au début du programme, il a du être modifié; une inspection des instructions précédant l'erreur devrait fournir la réponse. Avec l'expérience, ce genre de raisonnement deviendra vite naturel.

8.5.2 Outils de mise au point

Dans tous les systèmes il existe un certain nombre d'outils qui peuvent aider à la mise au point.

Messages d'erreur
Bien que les messages d'erreur fournis par le système ne soient jamais reçus avec joie, ils facilitent la tâche du programmeur en train de mettre son programme au point. L'indication fournie par un message d'erreur est en effet précise, une fois le message décodé et compris, et permet d'identifier l'erreur assez vite, alors qu'un résultat erroné, pour peu que le programme soit important, est plus difficile à détecter et à corriger.

Les erreurs signalées par le système sont nombreuses; selon les systèmes d'exploitation, les messages peuvent varier et nous ne reprendrons pas ici le libellé de tous les messages d'erreur. Les programmeurs sont enjoints de consulter les manuels de référence de leur constructeur pour leur modèle d'ordinateur et leur version du système d'exploitation utilisés.

On peut cependant mentionner ici les types d'erreur provoquant l'arrêt de l'exécution du programme. Un premier type d'erreur peut être l'essai par l'unité centrale d'exécuter une instruction dont le code opération n'existe pas; l'unité centrale peut également essayer d'exécuter une instruction privilégiée alors qu'elle ne se trouve pas en mode superviseur, mais en mode utilisateur. Certaines opérations affectant directement les données d'état du programme ou les opérations d'entrée-sortie ne sont en effet pas toujours permises aux programmes d'application et sont réservées au système. Une

instruction peut aussi faire référence à une portion de mémoire protégée: il y aura erreur ainsi d'ailleurs que si une instruction fait référence à une position mémoire qui n'existe pas. Il peut se produire des erreurs de spécification si, sur certaines machines, une instruction utilise une paire de registres comme opérandes et si le premier registre n'est pas pair; la même erreur peut se produire si une instruction de branchement conduit à une adresse impaire: en effet sur certains processeurs les instructions doivent posséder des adresse paires. Certaines autres erreurs sont reliées à la manipulation de données décimales codées binaires. Il y a enfin un certain nombre d'erreurs liées à l'arithmétique utilisée: débordement entier, division entière par zéro ou quotient trop grand, débordement décimal, division décimale par zéro, débordement de l'exposant réel (trop grand ou trop petit), division réelle par zéro, etc.

Vidanges de mémoire

Un des outils classiques du programmeur en assembleur est la vidange de mémoire; en effet, dans un passé pas si lointain, dès qu'ils détectaient une erreur, certains systèmes d'exploitation signalaient cette erreur en même temps qu'ils produisaient une vidange de la mémoire occupée par la programme.

Une vidange de mémoire est tout simplement une liste du contenu de la mémoire occupée, les contenus étant généralement donnés en hexadécimal et en caractères ASCII à raison de 8 mots (16 octets) par ligne. Cette vidange peut débuter par l'impression du contenu des registres et l'indication de l'adresse de l'instruction en cours d'exécution. On peut alors utiliser la vidange de mémoire pour vérifier le contenu des diverses variables au moment de l'arrêt de l'exécution. Ceci, ajouté au contenu des registres et à l'adresse où l'arrêt s'est produit, donne généralement suffisamment d'information pour pouvoir déterminer la raison de l'arrêt.

D'autres systèmes d'exploitation ne produisent pas de vidange de mémoire dans le cas d'erreur à l'exécution. C'est au programmeur de se familiariser avec le système qu'il utilise et de savoir quels sont les outils à sa disposition. Le contenu de la mémoire reste, en effet, la meilleure information disponible.

```
   Adresse    Hexadécimal
002A    60FF FA16 006D 004C 696D 6974 6520 3120   `ÿú..m.Limite 1
003A    532E 562E 502E 2000 4C69 6D69 7465 2032   S.V.P. .Limite 2
004A    2053 2E56 2E50 2E20 0020 2020 4164 7265    S.V.P. .   Adre
005A    7373 6520 2020 2048 6578 6164 E963 696D   sse    Hexadécim
006A    616C 0068 000C E300 0AEB 0008 5000 0A44   al.h..ã..ë..P..D
007A    0012 6800 0216 0161 C300 0060 0002 E300   ..h....aÃ..`..ã.
008A    0450 000A 4400 1068 0002 1601 61C3 0000   .P..D..h....aÃ..
009A    6000 02E3 0006 B300 0410 00AF CB00 04E3   `..ã........Ë..ã
00AA    0004 EB00 0650 000A 4400 0EC8 0000 EB00   ..ë..P..D..È..ë.
00BA    0250 000A C300 0473 0002 E3FF FE68 0002   .P..Ã..s..ãÿþh..
00CA    1601 B750 0020 5000 2050 0020 5000 20C0   ...P. P. P. P. À
00DA    0000 E300 00C7 0004 E3FF FE68 0002 1601   ..ã..Ç..ãÿþh....
00EA    B778 0002 5000 20C3 0000 7000 01E3 0000   .x..P. Ã..p..ã..
00FA    B000 0808 00DF 5000 2050 0020 CB00 02C0   .....ßP. P. Ë..À
```

Figure 8.6 Vidange du code objet du programme VidangeSP

Selon les systèmes, il existe un certains nombre de macro-instructions ou de sous-programmes pouvant être utilisés à tout endroit où l'on peut placer une instruction, qui permettent de faire un certain

nombre de choses: entrée-sortie, vidange de mémoire, etc. Ainsi, il existe des macro-instructions pour imprimer des vidanges partielles de mémoire ("*snapshots*"), selon le système.

La figure 8.6 présente un exemple de vidange de mémoire du code objet produit par l'assembleur pour un sous-programme de vidange que nous avons écrit. C'est une vidange de mémoire en cours d'exécution, et par conséquent les adresses indiquées sont absolues (colonne de gauche).

On trouve dans cette vidange le contenu des mots mémoire correspondant aux adresses spécifiées, à raison de 8 mots par ligne, dont le contenu est donné en hexadécimal, ce même contenu étant donné de plus, à droite de la ligne sous forme de caractères ASCII (un point représentant un caractère non imprimable). Essayez de vous retrouver dans cette vidange à partir des codes opération. Les instructions sont toutes là, mais il y a en plus des informations qui ne correspondent pas à des instructions: pouvez-vous les identifier et en deviner le rôle?

Traces

La trace d'un programme peut être considérée comme un cas particulier d'une vidange partielle de mémoire. Une trace complète donne des indications sur l'exécution de chaque instruction et les résultats de cette exécution. Une trace du flux du programme n'indique que les instructions de branchement et les branchements choisis. Une trace de l'exécution du programme est utile lorsque le programmeur ne peut suivre le flux des instructions exécutées à partir de la liste de son programme et de vidanges partielles de mémoire.

Une trace complète met en jeu une exécution interprétative des instructions du programme: le programme de trace traite et interprète chacune des instructions du programme, il simule l'exécution de ce programme. Un tel programme prend de la place supplémentaire en mémoire ainsi que du temps machine supplémentaire. Pour cette raison on tend à décourager l'utilisation systématique de traces complètes, car des vidanges partielles judicieusement utilisées sont moins coûteuses et aussi utiles. Il y a cependant des cas où un mécanisme de trace permet d'accélérer la mise au point; on peut d'ailleurs en réduire le coût en ne le rendant actif que pour certaines parties du programme. Selon les installations il existe divers systèmes de trace, là encore c'est au programmeur de découvrir quels sont les outils à sa disposition et, en toute connaissance de cause, d'en faire le meilleur usage possible.

Logiciels de mise au point

Mais le meilleur outil de mise au point est un programme de mise au point souvent appelé "*debugger*". Avec un "debugger" point n'est besoin de vidange de mémoire, point n'est besoin de trace. Le programme de mise au point permet de placer des *points d'arrêt* dans le code source, d'exécuter le code *pas à pas* instruction par instruction, d'examiner la valeur des registres et des variables, et d'arrêter l'exécution du programme en cas de besoin.

Un tel outil permet donc de suivre l'exécution d'un programme pas à pas et d'examiner les valeurs prises par les variables en tout point de l'exécution. Ceci et l'utilisation bien pensée des points d'arrêt permet de mettre un programme assembleur au point en bien moins de temps qu'il ne le fallait il y a dix ans! Profitez bien du progrès!

8.5.3 Exécution d'un programme source PEP 8

Le système complet PEP 8 peut être téléchargé à partir de l'URI qui suit; pour démarrer PEP 8 il suffit de cliquer deux fois sur l'exécutable. ftp://ftp.pepperdine.edu/pub/compsci/pep8

L'exécution d'un programme en langage d'assemblage est un processus en quatre étapes.

(1) Écrire le programme dans un nouveau document à l'aide de l'éditeur de texte intégré. Le nombre de caractères par ligne n'est pas vraiment limité, mais la taille des fenêtres ouvertes impose une limite pour la facilité de lecture qui dépend de la police et de la taille de caractère que vous utilisez (par exemple, 80 caractères en Courrier 10 points, 91 en Courrier New 8 points, etc.). Si vous dépassez ces limites la ligne est coupée et continue à la ligne suivante de la fenêtre ; cette coupure apparaît aussi dans la liste d'assemblage.

(2) Assembler le programme au moyen de l'option Assemble du menu *Build* (*Build -> Assemble*). Cette commande utilise le texte de la fenêtre de code source comme programme à traduire. S'il y a des erreurs dans le programme source le système y insère des messages d'erreur en rouge à l'endroit des erreurs ; notez que ces messages d'erreur ajoutés ne font pas partie de votre programme source. Vous pouvez les enlever en sélectionnant *Build -> Remove Error Messages*. Que ces messages d'erreur soient enlevés ou non, vous pouvez corriger les erreurs indiquées et assembler de nouveau. L'élimination des messages d'erreur est cependant facultative, la commande *Build -> Assemble* les enlevant automatiquement avant d'assembler.

(3) Lorsqu'il n'y a plus d'erreur, le résultat de l'assemblage sera la fenêtre code objet affichant le code objet en hexadécimal et la fenêtre de la liste d'assemblage. Vous pouvez alors sélectionner *Build -> Load* pour charger le programme dans la mémoire centrale de PEP 8.

(4) Si votre programme a besoin de données d'entrée, en mode Batch I/O vous pouvez écrire vos données dans la fenêtre d'entrée (*Input*) et sélectionner *Build -> Execute* pour lancer l'exécution de votre programme. Les résultats apparaîtront alors dans la fenêtre de sortie (*Output*). Si vous désirez que les données d'entrée viennent du clavier pendant l'exécution, avant de lancer l'exécution sélectionnez le mode *Terminal I/O* (voir figure 8.7) pour changer le mode d'entrée des données.

Vous pouvez combiner ces étapes de diverses manières au moyen des raccourcis de commandes du menu de PEP 8.

Données d'entrée

Lorsque vos données sont nombreuses il vaut mieux les lire à partir de la fenêtre d'entrée (dans laquelle vous pouvez copier, par exemple, le contenu d'un fichier de données). Si vous données sont peu nombreuses, il est plus facile d'utiliser l'entrée de données interactive. Dans ce mode sachez cependant qu'un caractère fin de ligne (LF) est ajouté à la fin de chaque ligne de la boîte de dialogue (chaque fois que vous tapez sur la touche *Return*). Si vous lisez des données qui sont des caractères il vous faudra consommer ce caractère de fin de ligne.

Si un programme demande plusieurs entiers à l'utilisateur; ces derniers peuvent être donnés tous les trois à la fois sur la même ligne séparés par une espace ou un par ligne. Le gestionnaire d'interruption qui traite l'instruction DECI consomme en effet le caractère de fin de ligne. Cependant si vous entrez des caractères au moyen de l'instruction CHARI il vous faut tenir compte du caractère de fin de ligne supplémentaire. Ces petites complications sont inévitables lorsqu'on programme à des niveaux plus proches de la machine, mais les caractéristiques des systèmes d'entrée-sortie changent aussi d'un langage évolué à un autre.

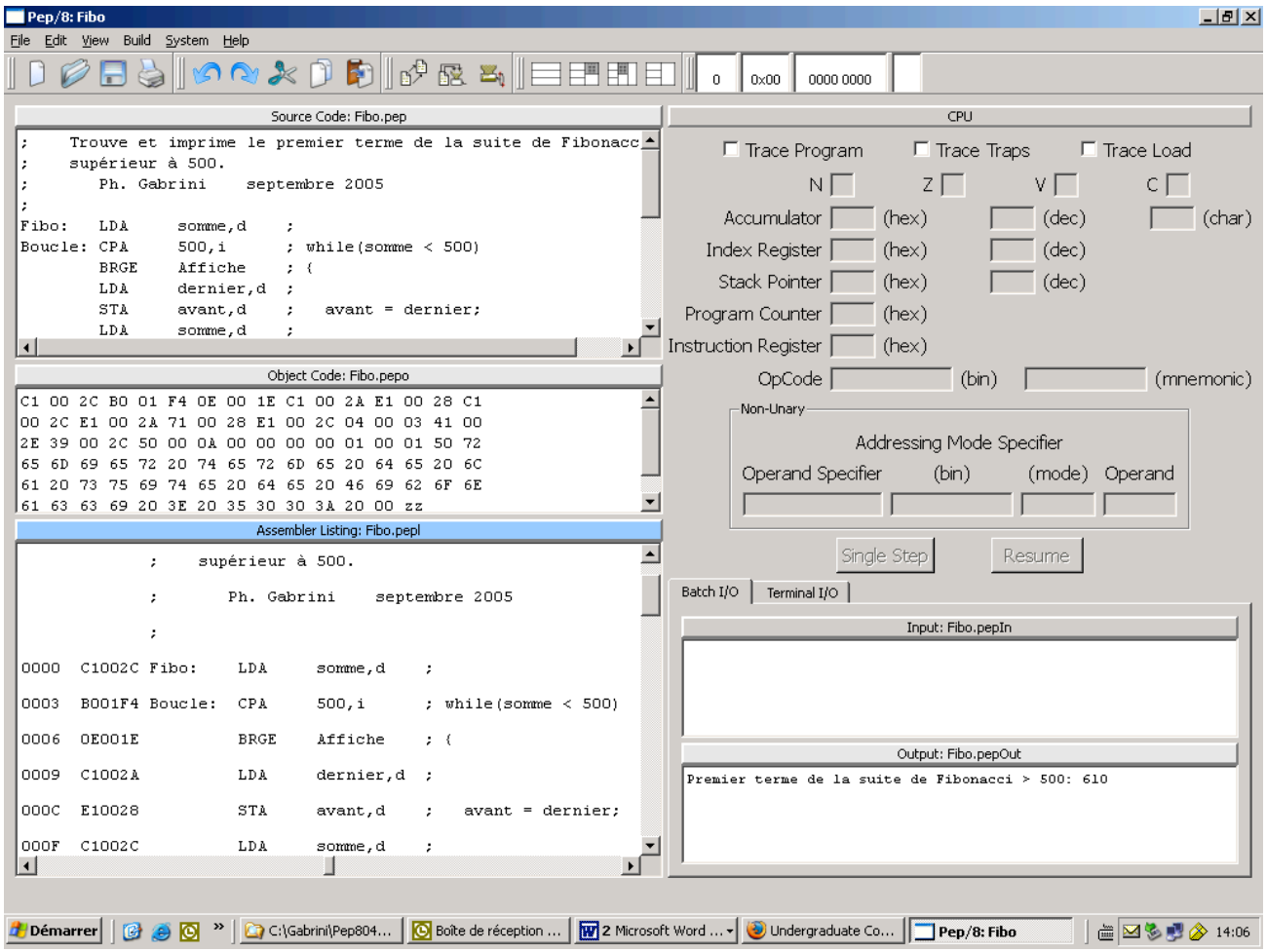

Figure 8.7 Choix de l'entrée et des options de trace du programme de Fibonacci

Trace des programmes

Le système PEP 8 permet d'exécuter les programmes tel que nous l'avons vu plus haut, mais il permet aussi d'en faire la trace en exécutant les instructions pas à pas et en plaçant des points d'arrêt à certains endroits du programme exécuté. Pour activer le système de trace cliquez la case *Trace Program* dans la fenêtre CPU de la figure 8.7. Il vous est possible de suivre l'évolution du contenu des positions de mémoire, de placer des points d'arrêt à plusieurs endroits du programme et d'exécuter vos instructions pas à pas (*Single Step*) ou de point d'arrêt en point d'arrêt (*Resume*).

La boîte de dialogue de trace donne le contenu des deux registres A et X, du compteur ordinal, du pointeur de pile et du registre d'instruction dans la fenêtre *PEP 8 CPU*, ainsi que les caractéristiques de l'opérande de l'instruction en cours d'exécution. La fenêtre *Memory*, qui n'est pas ouverte automatiquement (ouvrez-la en cliquant *View -> Show/Hide Memory Pane*), donne le contenu de la mémoire en hexadécimal et en caractère par octets. En cochant la case *Trace Program*, vous pouvez faire la trace de votre programme et de ses variables. Vous pouvez également faire la trace du chargeur en cochant la case *Trace Load*; ceci vous permet d'exécuter les instructions du chargeur pas à pas. De même en cochant la case *Trace Traps*, vous pouvez faire la trace des programmes de traitement des interruptions du système d'exploitation. En cochant plusieurs cases vous pouvez effectuer toutes les combinaisons que vous désirez.

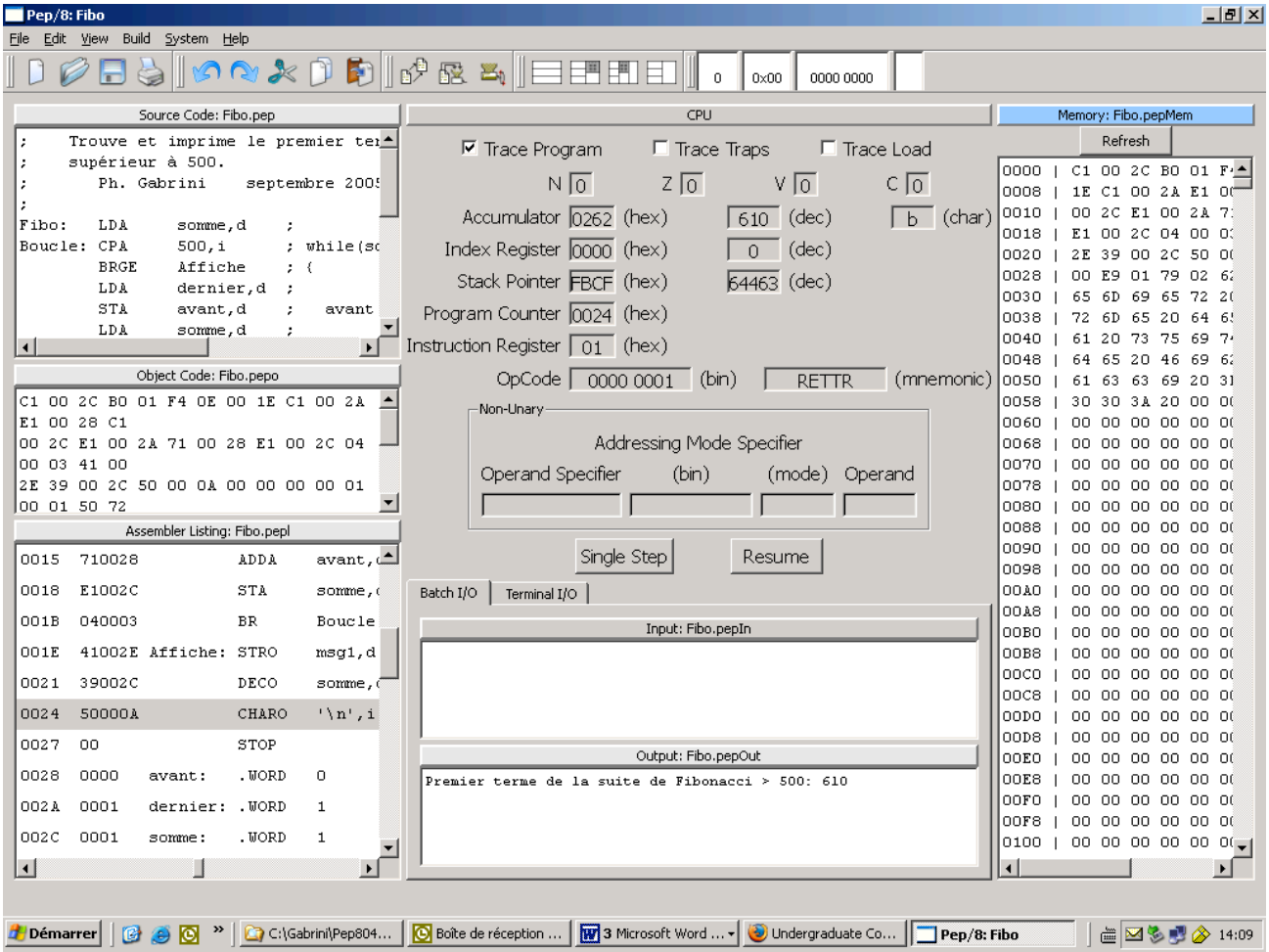

Figure 8.8 Trace de l'exécution du programme de Fibonacci en cours

Le système PEP 8 compte les instructions exécutées dans votre code et dans le code du système d'exploitation; si le nombre d'instructions exécutées dépasse une des limites du système (500 000 pour votre programme et la même chose pour le système d'exploitation) et que votre programme n'est pas terminé vous devrez augmenter cette limite et sauvegarder la nouvelle valeur avant de pouvoir continuer votre exécution. Bien que ceci puisse paraître ennuyeux, c'est fort utile pour contrôler les

boucles infinies qui se produisent bien trop souvent. De toute manière l'option *System -> Set Execution Limits...* vous permet à tout moment de fixer les limites d'exécution tant pour le nombre d'instructions du programme que pour le nombre d'instructions du système d'exploitation qui seront exécutées.

Méthode pour la mise au point

Afin de pouvoir mettre au point vos programmes de la manière la plus efficace possible, vous devriez procéder de la façon suivante. Après avoir assemblé et chargé votre programme, réduisez la taille de la fenêtre source et de la fenêtre objet. Cliquez sur l'option *Trace Program*. La liste d'assemblage vous donne les adresses des instructions et vous permet de placer des points d'arrêt dans les cases précédant chaque ligne du programme dans la fenêtre. Lancez alors l'exécution (option *Build -> Execute*) et positionnez vos fenêtres comme le montre la figure 8.8. L'exécution du programme ne démarrera que lorsque vous cliquerez sur les options *Single Step* (exécution pas à pas une instruction à la fois), ou *Resume* (exécution de toutes les instructions jusqu'à la rencontre du prochain point d'arrêt ou de l'instruction `STOP`). En cours d'exécution votre liste d'assemblage suit le déroulement de vos instructions en colorant l'instruction en cours d'exécution.

8.6 Exemple de programme : compression d'une chaîne de caractères

La figure 8.10 présente la liste d'un programme effectuant la compression d'une chaîne de caractères. La chaîne d'entrée a une longueur quelconque (inférieure à 120 caractères) et est terminée par le caractère "_"; elle comprend des mots séparés par une[2] ou plusieurs espaces. Cette chaîne d'entrée sera transformée par le programme en une chaîne de sortie dans laquelle les blocs de plus d'une espace sont remplacés par le caractère ayant pour code hexadécimal "FF", suivi du nombre d'espaces dans un octet; les espaces solitaires demeurent telles quelles. Par exemple la chaîne d'entrée:

```
....voici..un.texte...à.comprimer.....
```

où les espaces sont représentées par un point, sera transformée en:

```
•4voici•2un.texte•3à.comprimer•5_
```

où • représente le caractère ayant le code hexadécimal "FF" qui ne s'imprime pas et où une espace est encore représentée par un point.

Ces règles de transformation sont simples mais demandent de considérer un grand nombre de cas. Dans l'état initial on lit un caractère qui peut être un caractère normal à sortir, une espace ou le caractère "_". L'espace nous place dans un second état où il y a encore la possibilité de rencontrer ces trois caractères: un caractère normal nous renvoie à l'état initial (il n'y avait qu'une seule espace), un "_" nous fait terminer, mais une espace nous envoie dans un troisième état de compression. Dans ce troisième état un "_" nous fait terminer après génération de •n, un caractère normal nous fait revenir à l'état initial après génération de •n, et une espace nous laisse dans le même état à compter les espaces. Ceci correspond à un automate que nous pouvons définir par la figure 8.9.

Le programme traduira l'existence de ces trois états et le choix des actions à entreprendre par une table de branchements comportant neuf choix possibles. Après initialisation des zones d'entrée et de sortie on lit la chaîne à comprimer. On traite ensuite les caractères, les uns à la suite des autres en

[2] Rappelez-vous que l'espace typographique est féminine!

déterminant s'il s'agit d'un caractère normal, d'une espace, ou d'un "_" et en prévoyant un décalage respectif de 0, de 2 ou de 4 pour la table des branchements aux traitements appropriés. On utilise ensuite la variable `etat` pour compléter le décalage et pouvoir effectuer le branchement aux actions correspondant à l'état actuel et au caractère traité. D'après notre automate de la figure 8.9, pour chaque état il y a trois cas possibles; par conséquent, notre table de branchement comprendra 3 cas par état, et comme il y a trois états, neuf éléments. Les décalages prévus pour les trois états sont 0,6 et 12, puisqu'une adresse occupe 2 octets, et à l'intérieur de chaque état les décalages sont 0, 2 et 4.

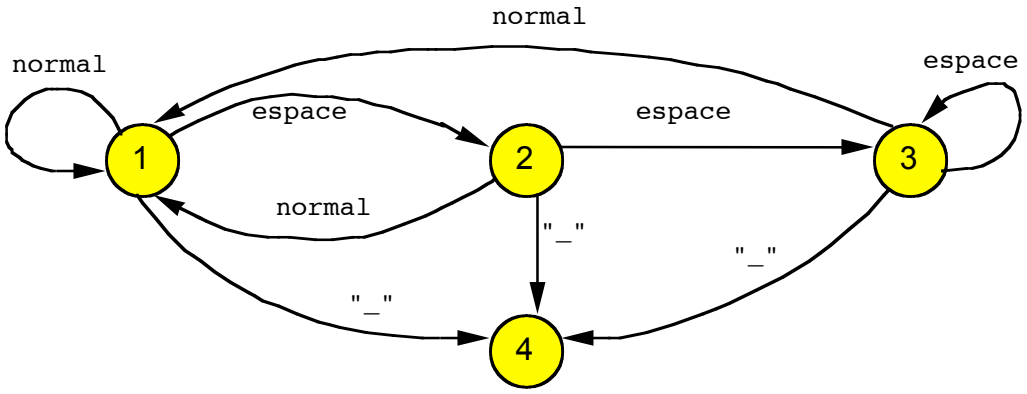

Figure 8.9 États de l'automate

Les étiquettes `Etatmn` identifient les actions à entreprendre dans les neuf cas possibles; lorsque *n* vaut 1 on traite un caractère normal, lorsque *n* vaut 2 on traite une espace et lorsque *n* vaut 3 on traite le caractère "_". La lettre *m* repère l'état dans lequel on se trouve. On remarquera l'utilisation des symboles `ETAT1`, `ETAT2`, et `ETAT3` pour modifier l'état courant. Ces valeurs sont définies par équivalence. On utilise l'instruction `STBYTEA` pour placer le caractère spécial de code hexadécimal "FF" ainsi que le nombre d'espaces dans la zone de sortie. Ce nombre est en effet suffisamment petit pour tenir dans un octet bien que le compteur soit un mot. Même si les actions de `Etat31` et `Etat33` sont semblables à trois instructions près, on a préféré les garder séparées plutôt que de les combiner, afin de conserver la clarté du programme. Ce dernier se termine de la façon conventionnelle par transfert de contrôle au programme appelant, dans ce cas-ci le superviseur, par exécution de l'instruction `STOP`.

On remarquera que, dans cette forme, le programme n'est pas très utile: la ligne comprimée qu'il affiche est peu utile car le code "FF" et les nombres d'espaces peu élevés correspondent à des caractères qui ne sont pas affichés et qui n'apparaissent donc pas dans la ligne de sortie. Pour cette raison la ligne de sortie est affichée en hexadécimal. Cependant un tel programme est très utile dans le cas du traitement d'un fichier de texte: il produit un fichier comprimé. On laisse au lecteur l'exercice d'adapter le programme pour le traitement d'un fichier, ainsi que l'écriture du programme effectuant la décompression du fichier!


```
;*******************************************************************************
;* Compression des espaces d'une ligne lue au terminal terminée
;* par le caractère "_"
;* Une espace solitaire demeure telle quelle.
;* Plusieurs espaces consécutives sont remplacées par le caractère
;* dont le code hexadécimal est 'FF' suivi du nombre d'espaces
;* dans l'octet suivant.
;* La ligne comprimée est affichée à l'écran en hexadécimal.
;*
;*    Philippe Gabrini    Novembre 2005
;*******************************************************************************
ETAT1:    .EQUATE  0
ETAT2:    .EQUATE  6
ETAT3:    .EQUATE  12
NEWLINE:  .EQUATE  0xA
MAX:      .EQUATE  120
ESPACE:   .EQUATE  0x20
FIN:      .EQUATE  0x5F
CODE:     .EQUATE  0xFF
                              ;int main(){
Compress:  LDA      ETAT1,i     ;  etat = état initial;
           STA      etat,d      ;
           STRO     msg1,d      ;  cout >> "Donner un texte"
           CHARO    NEWLINE,i   ;        >> endl;
           LDA      MAX,i       ;  [nombre maximum de caractères]
           STA      -2,s        ;
           LDA      entree,i    ;  [adresse chaîne]
           STA      -4,s        ;
           ADDSP    -4,i        ;
           CALL     LirChain    ;  cin << entree;
           LDA      0,s         ;  Nombre de caractères lus
           STA      compte,d    ;
           ADDSP    2,i         ;  désempiler résultat
           LDA      0,i         ;  [indice dans tampon de sortie]
           STA      indSort,d   ;  while(true){
Suivant:   LDX      indEnt,d    ;    pour partir à zéro
           ADDX     1,i         ;     [caractère suivant]
           STX      indEnt,d    ;    indice tampon entrée
           LDA      0,i         ;
           LDBYTEA  entree,x    ;
           CPA      ESPACE,i    ;     [espace?]
           BREQ     Espace      ;
           CPA      FIN,i       ;  if(caractère == '_') break;
           BREQ     CarFin      ;
           LDX      0,i         ;     if(caractère normal){
           BR       Cas         ;       cas normal
Espace:    LDX      2,i         ;       if(espace) décalage espace
           BR       Cas         ;
CarFin:    LDX      4,i         ;       if(caractère fin) décalage fin
Cas:       ADDX     etat,d      ;         switch(caractère){
           BR       TabCas,x    ;
TabCas:    .ADDRSS  Etat11    ;         [caractère normal]
           .ADDRSS  Etat12    ;         [espace]
           .ADDRSS  Etat13    ;         [caractère de fin]
           .ADDRSS  Etat21    ;         [caractère normal]
           .ADDRSS  Etat22    ;         [espace}
           .ADDRSS  Etat23    ;         [caractère de fin]
           .ADDRSS  Etat31    ;         [caractère normal]
           .ADDRSS  Etat32    ;         [espace]
           .ADDRSS  Etat33    ;         [caractère de fin]
```

Figure 8.10 Programme de compression de chaînes (début)

```
Etat11:    BR       Copie     ;         [copier caractère normal]
Etat12:    LDA      ETAT2,i   ;         etat = état 2;
           STA      etat,d    ;
           BR       Copie     ;         [copier caractère normal]
Etat13:    BR       Fini      ;         [arrêt]
Etat21:    LDA      ETAT1,i   ;         etat = état 1;
           STA      etat,d    ;
           BR       Copie     ;         [copier caractère normal]
Etat22:    LDA      ETAT3,i   ;         etat = état 3;
           STA      etat,d    ;
           LDA      2,i       ;         [initialiser compteur à 2 espaces]
           STA      compte,d  ;
           BR       FinCas    ;
Etat23:    BR       Fini      ;         [arrêt]
Etat31:    LDA      ETAT1,i   ;         etat = état 1;
           STA      etat,d    ;
           LDX      indSort,d ;
           SUBX     1,i       ;         effacer espace
           LDA      CODE,i    ;         [sortir code compression}
           STBYTEA  sortie,x  ;
           ADDX     1,i       ;
           LDA      compte,d  ;
           STBYTEA  sortie,x  ;         [sortir le compte d'espaces]
           ADDX     1,i       ;          prochain caractère de sortie
           STX      indSort,d ;
           BR       Copie     ;         [copier caractère normal]
Etat32:    LDA      compte,d  ;
           ADDA     1,i       ;         [augmenter le compteur d'espaces]
           STA      compte,d  ;
           BR       FinCas    ;
Etat33:    LDX      indSort,i ;
           SUBX     1,i       ;         effacer espace
           LDA      CODE,i    ;         [sortir code de compression]
           STBYTEA  sortie,x  ;
           ADDX     1,i       ;
           LDA      compte,d  ;
           STBYTEA  sortie,x  ;         [sortir le compte d'espaces]
           STX      indSort,d ;
           BR       Fini      ;
Copie:     LDX      indEnt,d  ;
           LDBYTEA  entree,x  ;         [copier caractère d'entrée dans sortie]
           LDX      indSort,d ;
           STBYTEA  sortie,x  ;
           ADDX     1,i       ;
           STX      indSort,d ;          avancer dans sortie
FinCas:    BR       Suivant   ;  }//while
Fini:      LDBYTEA  '_',i     ;   [sortir caractère de fin]
           LDX      indSort,d ;
           STBYTEA  sortie,x  ;
           ADDX     1,i       ;
           LDA      0,i       ;
           STBYTEA  sortie,x  ;    fin chaîne
           STRO     sortie,d  ;    sortie de la chaîne
           CHARO    NEWLINE,i ;  cout >> endl;
           LDX      0,i       ;
           LDA      0,i       ;
Repete:    LDBYTEA  sortie,x  ;  while(caractère non nul){
           CPA      0,i       ;
           BREQ     Final     ;
           STA      -2,s      ;
           ADDSP    -2,i      ;
           CALL     HEXO      ;     {afficher mot en hexadécimal}
           ADDX     1,i       ;     {par caractère}
           BR       Repete    ;  }//while
Final:     CHARO    NEWLINE,i ;  cout >> endl;
           STOP               ;}//main
;
```

Figure 8.10 Programme de compression de chaînes (suite)

```
;------- Lecture chaîne
;Lit une chaîne de caractères ASCII jusqu'à ce qu'il
;rencontre un caractère de fin de ligne. Deux paramètres
;qui sont l'addresse du message sur la pile et la taille maximum.
;
car:     .EQUATE 0
regX:    .EQUATE 2
regA:    .EQUATE 4
retour:  .EQUATE 6
addrBuf: .EQUATE 8          ; Addresse du message à lire
taille:  .EQUATE 10         ; taille maximum
                            ;{
LirChain:SUBSP   6,i        ;  espace local
         STA     regA,s     ;  sauvegarde A
         STX     regX,s     ;  sauvegarde X
         LDX     0,i        ;  X = 0;
         LDA     0,i        ;  A = 0;
                            ;  while)true){
EncorL:  CHARI   car,s      ;    cin << caractère;
         LDBYTEA car,s      ;  if(caractère == fin de ligne) break;
         CPA     0xA,i      ;
         BREQ    FiniL      ;
         STBYTEA addrBuf,sxf;
         ADDX    1,i        ;    indice++;
         CPX     taille,s   ;
         BRLE    EncorL     ;  }//while
FiniL:   STX     taille,s   ;  nombre de caractères lus
         LDA     retour,s   ;  adresse retour
         STA     addrBuf,s  ;  déplacée
         LDA     regA,s     ;  restaure A
         LDX     regX,s     ;  restaure X
         ADDSP   8,i        ;  nettoyer pile
         RET0               ;}//LireChaine;

;Sortie hexadécimale.
;Format sortie:  un mot en 4 caractères hexa
Hnomb:   .EQUATE 0
Hdroit:  .EQUATE 1
HregX:   .EQUATE 2
HregA:   .EQUATE 4
Hretour: .EQUATE 6
Hmot:    .EQUATE 8          ; mot à sortir
                            ;{
HEXO:    SUBSP   6,i        ;  espace local
         STA     HregA,s    ;  sauvegarde A
         STX     HregX,s    ;  sauvegarde X
         LDA     Hmot,s     ;  A = mot;
         STA     Hnomb,s    ;  opérande
         LDBYTEA Hnomb,s    ;  octet gauche dans bas de A
         ASRA               ;  Décale 4 bits
         ASRA               ;
         ASRA               ;
         ASRA               ;
         CALL    SortAC     ;  sort premier carac. hexa
         LDBYTEA Hnomb,s    ;  octet gauche dans bas de A
         CALL    SortAC     ;  sort second carac hexa
         LDBYTEA Hdroit,s   ;  octet droit dans bas de A
         ASRA               ;  Décale 4 bits
         ASRA               ;
         ASRA               ;
         ASRA               ;
         CALL    SortAC     ;  sort troisième carac. hexa
         LDBYTEA Hdroit,s   ;  octet droit dans bas de A
         CALL    SortAC     ;  sort quatrième carac. hexa
         LDA     Hretour,s  ;  adresse retour
         STA     Hmot,s     ;  déplacée
         LDA     HregA,s    ;  restaure A
         LDX     HregX,s    ;  restaure X
         ADDSP   8,i        ;  nettoyer pile
         RET0               ;}//HEXO
```

Figure 8.10 Programme de compression de chaînes (suite)

```
;
; Sous-programme interne pour sortir en hexadécimal les 4 bits de droite de A
Hcar:    .EQUATE 0          ;{
SortAC:  SUBSP   1,i        ;  caractère temporaire
         ANDA    0xF,i      ;  isoler chiffre hexa
         CPA     9,i        ;  si pas dans 0..9
         BRLE    PrefNum    ;
         SUBA    9,i        ;    convertir number en lettre ASCII
         ORA     0x40,i     ;    et préfixer code ASCII lettre
         BR      EcritHex   ;
PrefNum: ORA     0x30,i     ;  sinon préfixer code ASCII nombre
EcritHex:STBYTEA Hcar,s     ;  sortir
         CHARO   Hcar,s     ;
         RET1               ;}//SortAC

msg1:    .ASCII  "Donnez une chaîne terminée par le caractère _ : \x00"
msg2:    .ASCII  "Chaîne compressée: \x00"
entree:  .BLOCK  120
sortie:  .BLOCK  120
compte:  .WORD   1
etat:    .WORD   1
indSort: .WORD   1
indEnt:  .WORD   -1
zero:    .BYTE   0
carac:   .BYTE   0
         .END
```

Figure 8.10 Programme de compression de chaînes

Les sous-programmes `LirChain` et `HEXO` seront vus en détail dans le chapitre suivant, une fois vus les concepts de base des sous-programmes.

8.6.1 Exercices

1. L'instruction située à l'étiquette CarFin du programme de la figure 8.10 donne au registre X la valeur 4; quelle en est la raison?
2. Les symboles Etat1, Etat2 et Etat3 servent à repérer l'état de l'automate à un moment donné; ils représentent pourtant des valeurs différentes de 1, 2 et 3, en l'occurrence 0, 6 et 12. Pourquoi?

