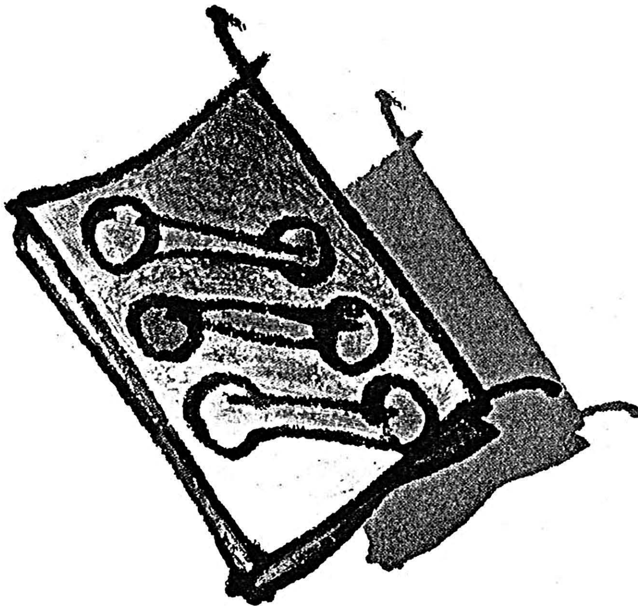
PROGRAMMATION STRUCTURÉE EN ASSEMBLEUR 6502

J-P. MALENGÉ L. ANDRÉANI P. COLLARD



MASSON

PROGRAMMATION STRUCTURÉE EN ASSEMBLEUR 6502

Jean-Pierre Malengé
Louis Andréani
Philippe Collard
enseignants au Département informatique de l'Institut Universitaire de Technologie de Nice

MASSON
Paris New York Barcelone Milan Mexico Sao Paulo
1987

TABLE DES MATIERES

Introduction 9

Chapitre 1

La programmation structurée 11

	Introduction : les objectifs et la stratégie	11
1	La documentation	11
2	La table des identificateurs	12
2.1	Les différents types de données	13
2.2	Les différents objets utilisés dans un module	13
2.3	La table des identificateurs	13
2.4	Premier exemple simple	14
2.5	Exemple plus complet	15
2.6	Création de la table des identificateurs	16
2.7	Retour sur le choix des identificateurs	16
3	La structuration des traitements	17
3.1	La séquence	17
3.2	Les structures alternatives	17
3.3	Les structures répétitives	18
3.4	Le choix d'une structure répétitive	20
4	Les automates d'états finis	21
4.1	Définition	21
4.2	L'automate d'un petit éditeur de texte	22
4.3	L'automate d'un module de transcodage	24
	Conclusion	26

Chapitre 2

Quelques aspects de la programmation en assembleur 6502 et 65C02 29

	Introduction : langage machine et assembleur	29
1	Description d'un programme assembleur	30
2	Structuration des données	30
2.1	Le contenu d'un octet	30
2.2	Les tableaux	31
2.3	La pile	32
2.4	Remarques sur les identificateurs et les types	32
3	Structuration des traitements	32
3.1	Appel d'un module et retour	33

3.2 Organisation des traitements à l'intérieur d'un module........................22
3.3 La valeur des indicateurs........................23
3.4 Les instructions qui agissent sur les indicateurs........................34
3.5 Les comparaisons numériques........................35
3.6 Le fonctionnement en mode décimal........................36
3.7 La notion de déplacement et ses conséquences........................37
3.8 Les particularités du 65C02........................37
Conclusion........................37

Chapitre 3

Les bases de la programmation structurée en assembleur 6502 et 65C02........................39
Introduction : les principes de la méthode........................39
1 La programmation des structures élémentaires........................40
1.1 L'alternative simplifiée : Si Alors Finsi........................40
1.2 L'alternative complète : Si Alors Sinon Finsi........................41
1.3 La boucle Tantque........................42
1.4 La boucle Répéter........................42
1.5 La boucle Itérer........................43
1.6 La boucle Pour........................44
2 La programmation de structures multiples........................45
3 Codage de modules en vue d'un assemblage unique........................48
4 Evaluation de la méthode........................50
4.1 Les avantages évidents........................50
4.2 Les performances........................50
4.3 Et les commentaires ?........................51
5 Détails complémentaires........................52
5.1 La structure Itérer avec plusieurs sorties........................52
5.2 Le codage des conditions composées........................53
5.3 Le codage de la structure Cas où........................56
5.4 Référence à des étiquettes lointaines........................57
Conclusion........................59

Chapitre 4

Exemples simples........................61
Introduction........................61
Exercice 1........................63
Exercice 2........................65
Exercice 3........................69
Exercice 4........................72
Exercice 5........................82
Exercice 6........................85
Exercice 7........................91
Exercice 8........................95
Exercice 9........................101
Exercice 10........................106

Chapitre 5

La programmation modulaire......111

1 Partage et localité : choix des identificateurs......111
2 Communication entre modules en assembleur......112
2.1 Communication par variable partagée......112
2.2 Communication par passage de paramètre......113
2.2.1 Mode de passage......113
2.2.2 Techniques de passage......114
2.2.2.1 Passage par les registres......114
2.2.2.2 Passage par la pile......114
2.2.2.3 Regroupement des arguments dans une table......115
2.3 Conclusion......115
3 Réalisation d'un éditeur de texte......115
3.1 Cahier des charges......116
3.2 Structure des données......116
3.3 Automate de l'éditeur......116
3.4 Décomposition modulaire......118
3.5 Réalisation de chaque module......119
3.5.1 Traiter un texte (module TT)......119
3.5.2 Saisir un texte (module SA)......123
3.5.3 Afficher le texte (module AF)......128
3.5.4 Supprimer une ligne (module SU)......131
3.5.5 Insérer une ligne (module IN)......135
3.5.6 Calculer l'indice (module CI)......141
3.5.7 Lire un nombre décimal (module LD)......143
3.5.8 Afficher en décimal (module AD)......143
3.5.9 Afficher la ligne de commande (module AC)......146
3.5.10 Variables partagées (VP)......148

Liste des abréviations pour les mots-clés......149

Index......151

INTRODUCTION

Il y a une vingtaine d'années, la réalisation d'applications complexes a conduit la communauté informatique à rechercher... et à trouver des méthodes permettant de produire de manière rentable du logiciel fiable et maintenable. On a l'habitude de regrouper sous les termes "analyse et programmation structurées", puis plus récemment "génie logiciel" et "génie informatique" tout ce qui permet de fabriquer des logiciels de qualité.

L'apparition de langages structurés et de compilateurs efficaces a fait diminuer la proportion d'applications développées en assembleur, mais en même temps, on a vu se développer l'informatique "industrielle" dans laquelle le matériel comprend beaucoup de numérique et de moins en moins d'analogique, et le logiciel beaucoup de modules en langage évolué et quelques-uns en assembleur.

Sur les machines les plus répandues, des préprocesseurs permettent d'ailleurs d'utiliser en assembleur les structures alternatives et répétitives classiques des langages évolués, mais très souvent, l'écriture de programmes en assembleur continue de se faire de manière plus ou moins anarchique, avec comme conséquence des produits difficiles à mettre au point et encore plus difficiles à maintenir.

Le but de ce livre est de montrer qu'il est parfaitement possible de normaliser la programmation en assembleur, de façon à obtenir un code dépersonnalisé, c'est-à-dire susceptible d'être mis au point et maintenu par n'importe quel membre d'une équipe et non plus uniquement par celui qui l'a écrit. Le chapitre 3 propose donc essentiellement une norme. Il est précédé de deux chapitres dans lesquels sont exposées les quelques notions de programmation structurée et d'assembleur nécessaires pour une bonne compréhension de la suite, mais ces chapitres ne sont absolument pas des introductions à ces deux domaines ; le lecteur idéal est celui qui programme déjà en assembleur et qui a quelques connaissances des techniques de programmation structurée. Les chapitres 4 et 5 contiennent essentiellement des exemples permettant de voir la mise en oeuvre des règles exposées au chapitre 3. Pour pouvoir aller jusqu'à un code réel, il fallait faire des choix : dans ce livre nous avons choisi le 6502 comme microprocesseur (une version 68000 est sous presse), l'Apple II comme micro-ordinateur, et le logiciel Lisa de Randall Hyde comme assembleur. Cela entraînera parfois des surprises pour le lecteur non familier de l'un d'eux : par exemple le code ASCII de l'Apple est un peu particulier, de même que certaines directives de Lisa ; lorsque le cas se présentera, des explications seront données au fil du texte, mais en règle générale, le contexte et l'expérience du lecteur seront suffisants pour la compréhension.

Ce livre est en fait le fruit de l'évolution de l'enseignement de l'assembleur au département informatique de l'Institut Universitaire de Technologie de Nice ; il a été rédigé par ceux qui en ont actuellement la charge, mais beaucoup d'autres enseignants ont participé à la mise au point de cette norme, en particulier Michel Koenig.

LA PROGRAMMATION STRUCTUREE

1

Il n'est pas question dans cet ouvrage de présenter en détail la programmation structurée, mais d'en rappeler ce qu'il est nécessaire de savoir pour l'appliquer à l'assembleur. Le but de ce chapitre est donc de résumer les objectifs et de présenter quelques outils de la programmation structurée. On verra successivement les problèmes de documentation, de structuration des données et de structuration des traitements.

Introduction : les objectifs et la stratégie.

Le principal objectif de la programmation structurée est l'amélioration de la productivité des activités de production de logiciel. Cet objectif final se traduit par un objectif pratique : développer des logiciels dont la maintenabilité soit bonne ; et cette **maintenabilité** est conditionnée par la **modularité**, la **fiabilité** et la **lisibilité**.

Après avoir déterminé les qualités que doit avoir un logiciel, on a pu trouver des outils permettant de produire de tels logiciels. On s'est aperçu qu'une bonne stratégie de développement comportait nécessairement trois volets complémentaires :
- documenter le travail,
- structurer les données,
- structurer les traitements.

L'ordre de l'énumération ci-dessus n'est pas anodin : la structuration des traitements ne fait pas la programmation structurée ; ce n'en est qu'un des aspects, probablement le plus connu car c'est à partir de la bataille de la suppression des GOTO que s'est développé l'ensemble des techniques de programmation structurée.

La suite de ce chapitre est consacrée à la présentation de ces trois aspects de la programmation structurée.

1 La documentation.

Le problème de la documentation d'une application est à l'heure actuelle bien cerné : chaque étape de la vie d'un logiciel doit se terminer par un document écrit, normalisé et soumis à une réception. Il ne suffit pas de documenter l'application, mais il faut aussi documenter son développement. Il est en effet fondamental, lorsqu'on modifie un logiciel, de bien comprendre quelles ont été les étapes de sa gestation : toute modification du code d'un programme doit commencer par une étude au niveau le plus

abstrait : fonction du module, description de la méthode utilisée pour atteindre cette fonction, etc...

La normalisation des divers documents et les conditions de réception sont souvent précisées par le service des méthodes s'il s'agit d'une grande société, ou par les outils de l'atelier logiciel utilisé si c'est le cas. Dans le cadre de cet ouvrage, nous nous intéressons uniquement aux documents relatifs aux étapes de la programmation et nous proposons essentiellement trois outils : le pseudo-code, la table des identificateurs et l'automate d'états finis.

Le pseudo-code a pour objectif d'aider le concepteur au moment où il développe le code d'un module et de lui permettre de formuler ses idées de manière précise et compréhensible. Ceci est obtenu par un compromis entre la rigueur d'un langage artificiel et la souplesse d'un langage naturel. On sait que dans un programme il y a des instructions ayant pour but d'effectuer un traitement et d'autres dont le but est d'organiser les traitements. Le pseudo-code combine le français pour décrire les traitements, et des structures codifiées pour décrire l'organisation des traitements.

Par ailleurs, il faut souligner dès maintenant que la description d'un module en pseudo-code passe par plusieurs versions de plus en plus détaillées et qu'au début le français sert à décrire le but des différentes parties du traitement. La façon détaillée qui est utilisée pour parvenir à ce but est décrite ensuite dans les dernières versions. Cette technique est connue sous le nom d'analyse descendante. Une analyse descendante se termine lorsque chacune des actions décrites dans le pseudo-code répond à l'un des critères suivants (on donne à chaque fois un exemple) :
- il s'agit d'une opération élémentaire (RAZ de l'accumulateur),
- son codage est évident (permutation de deux variables),
- c'est un autre module (lecture d'un caractère au clavier).

Nous reviendrons sur le pseudo-code qui permet donc de décrire la structuration des traitements dans la troisième partie de ce chapitre, mais nous allons d'abord voir comment décrire la structuration des données grâce à la table des identificateurs.

2 La table des identificateurs.

Les objets manipulés, que l'on appelle encore les données, doivent être décrits sous leurs deux aspects : dynamique et statique. La description dynamique des données permet de préciser comment les données se propagent d'un module à un autre et elle doit être étudiée et précisée dans une phase précoce de la conception de l'application. On dispose pour cela de différents outils : le diagramme des flux de données étant un des plus anciens et le précurseur d'outils plus modernes comme par exemple les méthodes MERISE ou AXIAL. L'étude de ces outils sort du cadre de cet ouvrage et nous nous limiterons dans ce chapitre à la description statique de la structure des données ; le problème de l'échange d'informations entre les modules (variables partagées et paramètres) est traité dans le chapitre 5.

2.1 Les différents types de données.

Le type d'une donnée est parfaitement défini lorsqu'on connait les valeurs que peuvent prendre les objets de ce type et les opérations que l'on peut effectuer sur ces objets. Par exemple dans les langages courants, on a l'habitude d'utiliser les objets de type entier : ce sont les objets dont les valeurs sont limitées aux entiers compris entre -32768 et 32767 (16 bits) et sur lesquels on a défini l'addition, la soustraction, la multiplication et la division entière. En assembleur, les objets sont le plus souvent de type octet, entier, caractère ASCII, réel. Nous y reviendrons dans le chapitre 2 consacré à la programmation en assembleur.

On peut également regrouper plusieurs données dans une structure dont on précise le type (valeurs et opérations permises) : on parle par exemple de tableaux de 10 chaînes de caractères. Le moyen d'accéder aux données élémentaires de la structure est une des opérations qu'il faut préciser.

Un véritable langage structuré doit mettre à la disposition de l'utilisateur le plus grand nombre possible de types prédéfinis, mais il doit également permettre de créer de nouveaux types selon les besoins de l'utilisateur. En Pascal par exemple, on peut définir un fichier de tableaux d'articles comprenant chacun deux chaînes de caractères, un entier et un réel... Le choix de la façon dont on structure les données est évidemment crucial car la rapidité d'un traitement dépend souvent de ce choix.

Une fois encore, la manière d'effectuer ce choix est hors des limites de cet ouvrage. Mais de plus, en assembleur, la structure "naturelle" est le tableau (ensemble d'objets de même type avec accès à partir de l'indice de l'objet visé) car il est facile d'accéder aux éléments d'un tableau grâce à l'adressage indexé. Le problème du choix des structures est donc reporté en amont : si on doit utiliser une structure différente (file d'attente par exemple), il faudra commencer par traduire cette structure jusqu'à parvenir à des objets élémentaires ou à des tableaux et modifier en conséquence les traitements.

2.2 Les différents objets utilisés dans un module.

Dans un module on va être amené à utiliser un certain nombre d'objets de nature différente. Nous reviendrons en détail sur ce problème dans le chapitre 5 consacré à la programmation modulaire, mais on peut dire dès à présent que l'utilisation de plusieurs modules va conduire à la création d'objets locaux à un module et d'objets partagés entre plusieurs modules, ainsi qu'à l'échange d'informations au moyen de paramètres qui seront simplement importés dans le module, ou qui servent à l'importation et à l'exportation, ou qui sont uniquement exportés. Lors du développement d'un module, il faut bien définir le type et la nature de chacun des objets utilisés ; ces renseignements sont rassemblés dans un document appelé la table des identificateurs.

2.3 La table des identificateurs.

Un échange d'informations ne peut être efficace que si les interlocuteurs sont parfaitement informés de la signification des mots employés. En informatique, les mots décrivent des traitements ou les objets sur lesquels portent les traitements. On identifie

chaque objet par un identificateur et l'ensemble des informations sur les objets doit être précisé dans un document appelé table des identificateurs.

Dans le cas le plus simple, la table des identificateurs se réduit en fait à une table des variables ; par contre, dans les cas plus complets, cette table décrit tous les objets utilisés dans un module, quelle qu'en soit la nature. Voyons successivement les deux cas.

2.4 Premier exemple simple.

Pour commencer, la figure 1.1 montre la table des identificateurs d'un des exemples du chapitre 4 : pour chaque variable du module on trouve d'abord son identificateur, sa signification, son type et la place qu'elle occupe en mémoire, puis trois colonnes dans lesquelles on place la valeur initiale, la valeur finale et les vérifications éventuelles. Expliquons-nous sur ces trois derniers points.

Id.	Signification	Type	Nb oct.	VI	VF	Vérif
NR	nombre relatif à convertir	relatif	1	$00	$FF	
VA	initialement : la valeur absolue de NR finalement : le nombre d'unités de NR	naturel	1	?	?	
ND	nombre de dizaines de NR	naturel	1	0	?	I U C

Fig. 1.1 Une table des identificateurs simple.

Il y a peu de choses à dire en ce qui concerne la valeur initiale et la valeur finale : lorsqu'elles sont connues, on les indique, et dans le cas contraire, on met un point d'interrogation ; bien entendu, si on constate que la valeur finale est égale à la valeur initiale, il faut se demander si cette variable ne serait pas en fait une constante. On se sert également de ces valeurs pour faire un certain nombre de vérifications ; dès que le pseudo-code est écrit, on doit vérifier que ces valeurs sont respectées ; si une valeur initiale est connue, on doit trouver une instruction d'affectation correspondante, et si une valeur finale est connue, cela doit se traduire quelque part : le plus souvent au niveau d'une condition, parfois simplement dans une affectation ou une écriture. Plus tard, on pourra également tester ces valeurs, soit à titre préventif, soit à titre curatif s'il y a un problème pour la mise au point.

La dernière colonne concerne les vérifications que l'on peut faire sur certaines variables qui jouent un rôle particulier : les compteurs et les indicateurs. Un compteur, comme son nom l'indique est une variable dont on se sert pour compter ou décompter des événements ; l'utilisation logique d'un compteur comporte trois étapes : son initialisation, son incrémentation ou sa décrémentation, et enfin son utilisation. Si une variable est utilisée comme compteur, il faut, lorsque le pseudo-code est écrit, vérifier que des actions correspondant à ces trois étapes ont bien été placées quelque part, et si possible au bon endroit. Pour ne pas oublier cette vérification, il faut, lorsqu'on décrit cette variable, placer dans la colonne "vérification" de la table des identificateurs trois

symboles correspondant à ces trois étapes (I pour initialiser, C pour compter, et U pour utiliser) et plus tard, lorsque le pseudo-code est écrit, biffer chacune de ces initiales après avoir vérifié l'existence et la validité des actions correspondantes. Il en sera de même pour les indicateurs (appelés parfois sémaphores ou drapeaux) qui sont des variables booléennes : il faudra vérifier leur positionnement initial, leur basculement, et leur utilisation par l'intermédiaire de trois lettres (I, B, U) dans la colonne "vérification".

2.5 Exemple plus complet.

L'exemple de la figure 1.2 reprend la table des identificateurs d'un des modules développés dans le chapitre 5. On y trouve une description des différents objets utilisés dans le module. D'abord les paramètres (importés, importés-exportés et exportés) ensuite les objets locaux (constantes et variables) et enfin les objets partagés (constantes et variables). La description des différents objets est identique à celle des variables dans le cas simple, à une différence importante près concernant le choix des identificateurs qui doit répondre à deux impératifs :
- plusieurs personnes codant indépendamment des modules qui seront assemblés ensemble doivent pouvoir choisir le même identificateur sans qu'il y ait de conflit,
- on doit pouvoir retrouver facilement le module dans lequel a été créé un objet partagé.

Pour répondre à ces deux impératifs, nous proposons de caractériser chaque module par deux lettres (par exemple LC pour un module de lecture, MN pour un module d'affichage d'un menu, etc...) et de placer ces deux lettres en tête de tous les identificateurs des objets créés dans ce module. On verra au chapitre 5 les règles précises concernant les déclarations, et en particulier on verra que tous les objets partagés sont déclarés dans un module spécialement créé dans ce but.

Id.	Signification	Type	Nb oct.	VI	VF	Vérif.
objets partagés						
VPTEXT	texte traité par l'éditeur	tableau de caractères	256			
VPNBLN	nombre de lignes du texte	naturel	1			
objets locaux						
variables						
LCNULI	numéro de la ligne à afficher	naturel	1	1	NBLN+1	
LCNUCA	numéro dans la ligne du caractère à afficher	naturel	1	1		
X	indice dans le texte du caractère à afficher	index	1			

Fig. 1.2 Une table des identificateurs volumineuse.

2.6 Création de la table des identificateurs.

Il faut créer la table des identificateurs au fur et à mesure des besoins, parallèlement à la création du pseudo-code. On a vu plus haut que le pseudo-code final s'obtient petit à petit par affinements successifs. Lors de l'écriture des différentes versions du pseudo-code, on est amené à introduire des objets : c'est au moment où on utilise pour la première fois un objet qu'il faut le décrire très précisément dans la table des identificateurs.

Autrement dit, la table des identificateurs, qui permet de décrire les objets, et le pseudo-code, qui permet de décrire les traitements sont deux documents qui doivent être créés simultanément. Bien entendu, la gomme et le crayon (ou l'éditeur de texte, qui en est une version moderne) sont des outils essentiels lors du développement d'un module. Il y a une différence entre ces deux documents : dans la documentation finale qui est mise au propre avant de passer au codage, figure une seule table des identificateurs (éventuellement classée par ordre alphabétique si elle est longue) alors qu'on peut conserver plusieurs versions du pseudo-code correspondant à différents niveaux de détail. Enfin, il faut vérifier la cohérence de ces deux documents : tout objet utilisé dans le pseudo-code doit être décrit dans la table des identificateurs, et on a vu plus haut que certaines variables entraînaient la présence de certaines actions.

2.7 Retour sur le choix des identificateurs.

Personne n'utilise l'identificateur O (la lettre) : la raison évidente est le risque de confusion avec le chiffre 0. Curieusement, la plupart des programmeurs ne craignent pas d'utiliser l'identificateur I (la lettre), malgré les risques identiques de confusion avec le chiffre 1. En fait, il existe une explication : l'habitude, venant du fond des âges de l'informatique, prise par les mathématiciens qui devaient traduire une "variable" telle que x_i en Fortran.

Cette habitude a ensuite été propagée par ces mêmes mathématiciens lorsqu'ils enseignaient l'informatique, et la pollution s'est amplifiée au cours du temps. Notre expérience personnelle montre qu'il est possible de se débarasser de cette habitude d'utilisation de I, mais qu'il est encore plus simple de ne pas faire prendre cette habitude : si vous devez initier quelqu'un à l'informatique, n'utilisez pas I ; il existe d'autres lettres : nous avons choisi K. Plus généralement, il est important de bien choisir les identificateurs, à partir de deux critères :
- la transparence : l'identificateur doit suggérer l'objet,
- la différence : les identificateurs doivent être visuellement éloignés les uns des autres, de façon à être reconnus sans confusion possible : à titre de contre-exemple, VITISAT (pour "vitesse initiale du satellite") et VITSITA (pour "vitesse au-dessus du site A").

Une autre question souvent posée au sujet des identificateurs concerne leur longueur. Des identificateurs longs sont plus parlants, mais ils sont fastidieux à écrire et à taper, sans compter les risques de fautes de frappe difficiles à détecter. Un bon compromis consiste à utiliser des identificateurs courts lors de la phase de développement et de mise au point puis, d'un coup de traitement de texte, de les remplacer par des identificateurs plus longs lorsque le produit est fini.

Maintenant que nous savons comment décrire les objets utilisés, nous pouvons passer à la structuration des traitements.

3 La structuration des traitements.

Plusieurs remarques avant d'entrer dans le détail. D'abord, rappelons que le découpage en modules de l'ensemble du traitement est une étape essentielle de la structuration des traitements, mais qu'elle est préalable à la phase de programmation qui est la seule qui nous préoccupe dans cet ouvrage ; nous supposerons donc ce découpage effectué, et nous nous intéresserons surtout à la structuration des traitements à l'intérieur d'un module. Deuxième remarque, il existe de nombreux ouvrages consacrés à la description des différentes structures de base utilisées couramment en programmation structurée ; nous nous contenterons donc de les rappeller et de préciser les conventions que nous avons choisies. Nous terminerons par une discussion au sujet du choix que l'on peut faire parmi les structures lorsqu'il s'agit d'un pseudo-code destiné au codage en assembleur.

3.1 La séquence.

La seule chose à préciser est une convention d'écriture : toute action est repérée par un caractère "." ; les différentes actions d'une séquence sont écrites sur des lignes séparées, les "." étant alignés sur une même verticale. Exemple :

```
. afficher un message de bienvenue
. lire la date
. appeler le module DEPART
```

3.2 Les structures alternatives.

Il existe quatre structures alternatives différentes :

L'alternative simple :

```
. Si condition
  Alors  . traitement
  Finsi
```

L'alternative complète :

```
. Si condition
  Alors  . traitement 1
  Sinon  . traitement 2
  Finsi
```

L'alternative généralisée avec sélecteur :

```
. Cas où le sélecteur vaut
    v1 : . traitement 1
    v2 : . traitement 2
    ...
    ...
    vn : . traitement n
    Autres cas : . traitement par défaut
  Fincas
```

L'alternative généralisée sans sélecteur :

```
. Cas où
    condition 1 : . traitement 1
    condition 2 : . traitement 2
    ...
    ...
    condition n : . traitement n
    Autres cas : . traitement par défaut
  Fincas
```

La traduction en assembleur de ces différentes formes d'alternatives est simple ; nous verrons comment au chapitre 3. On peut donc utiliser indifféremment l'une ou l'autre lorsqu'on développe le pseudo-code. Le seul conseil que l'on peut donner en vue d'améliorer la maintenabilité est d'utiliser de préférence une structure Cas où dès que le nombre d'options dépasse deux, ou si l'on suppose que des modifications ultérieures risquent de conduire à plus de deux options : il est en effet plus facile de modifier une structure Cas où que des Si imbriqués.

3.3 Les structures répétitives.

Il existe également quatre structures répétitives différentes :

La boucle avec compteur :

```
. Pour K variant de VI à VF avec un pas P
  Faire
    . traitement
  Finfaire
```

La boucle 0,n fois :

```
. Tantque condition
  Faire
    . traitement
  Finfaire
```

La boucle 1,n fois :

```
. Répéter
    . traitement
  Jusqu'à ce que condition
```

La boucle généralisée :

. Itérer
. traitement 1
Sortirsi condition
. traitement 2
Finitérer

La boucle Pour est la plus utilisée car elle existe dans les langages les plus anciens et les plus répandus. Paradoxalement, ce n'est pas la plus connue en ce qui concerne les détails pratiques de son fonctionnement. En gros, tout le monde sait qu'il y a une phase d'initialisation de la variable de contrôle, et une phase de test. Mais tout le monde ne connaît pas forcément l'endroit où se trouve le test : avant le traitement ou après ? Cette question est importante car selon la réponse à cette question, dans le cas où la condition de sortie est satisfaite dès l'entrée dans la boucle (lorsque par exemple A vaut 12 au moment d'exécuter la boucle : Pour K variant de A à 10) le traitement sera exécuté zéro ou une fois ; malheureusement la réponse à cette question est : cela dépend ! En Pascal, en Fortran 77 et dans certains Basic récents le test est placé avant le traitement, alors que en Fortran IV et dans la majorité des Basic le test est placé après. Il faut donc faire extrêmement attention lorsqu'on utilise la boucle Pour, et en particulier lorsqu'on transporte une application d'un langage à un autre, ou même d'une version à une autre, ce qui, dans les cas les plus défavorables peut se faire à l'insu de l'utilisateur. Une autre difficulté de la boucle Pour vient du fait que dans la plupart des cas, le calcul de la valeur de la borne finale est effectué une fois pour toutes, en même temps que l'initialisation de la variable de contrôle et que cette valeur finale est placée dans une variable du système inconnue du programmeur. En conséquence, si l'expression de la borne finale n'est pas une constante, mais fait intervenir une variable (par exemple Pour K variant de 1 à B) et si le programmeur place une instruction de modification de la valeur de cette variable B à l'intérieur de la boucle, l'exécution de cette instruction a pour effet de modifier effectivement la valeur de cette variable B, mais n'a aucune influence sur le test de sortie de boucle. Plus d'un programmeur en est tout surpris !

Les boucles Tantque et Répéter deviennent assez connues car elles sont présentes en Pascal et dans les versions structurées de Basic et Fortran. Elles sont en général bien comprises, même si leur utilisation peut soulever quelques problèmes sur lesquels nous reviendrons d'ici peu.

La boucle Itérer est encore peu connue : elle existe bien dans les langages les plus récents comme Ada ou Modula, mais elle n'existe pas en Pascal et de plus, elle est ignorée de beaucoup de manuels de programmation structurée ; nous allons voir dans le paragraphe suivant qu'elle présente de nombreux avantages. Puisqu'elle est peu connue, il est peut-être utile au préalable de préciser le schéma de son fonctionnement, ne fût-ce que pour confirmer les idées des lecteurs qui l'avaient deviné ; ce schéma est représenté dans la figure 1.3.

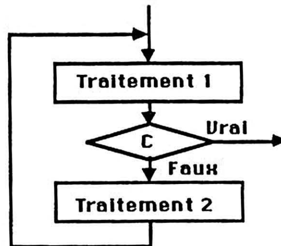


Fig. 1.3 Schéma de la boucle Itérer

3.4 Le choix d'une structure répétitive:

Il est généralement conseillé dans les manuels de programmation structurée de choisir la boucle Pour lorsque le nombre de fois où l'on doit exécuter le traitement est connu à l'entrée dans la boucle. Dans les autres cas, il est conseillé de choisir entre Répéter et Tantque selon que le traitement doit être fait au moins une fois ou pas.

Lorsqu'il s'agit d'un pseudo-code destiné à être codé en assembleur, nous conseillons d'utiliser uniquement la boucle Itérer. Voici les raisons de ce choix.

La boucle Pour est beaucoup trop ambiguë (nous l'avons vu ci-dessus). Même si le développeur sait parfaitement ce qu'il entend par une boucle Pour, il n'est pas certain qu'il en sera de même pour les différentes personnes qui seront appelées à utiliser ou à modifier le produit. Quant au choix entre les trois autres types de boucle, il suffit de constater que les boucles Répéter et Tantque sont en quelque sorte des cas de dégénérescence de la boucle Itérer pour que l'affaire soit entendue. A titre d'exercice sur la mise en oeuvre de ces différentes boucles, et pour emporter la conviction des hésitants, on peut écrire le pseudo-code du traitement suivant : lire une suite de caractères au clavier et décompter les 1 et les 2 ; s'arrêter lorsqu'on lit le chiffre 0.

Examinons successivement les trois versions :

Utilisation d'un Tantque :

```
. initialiser les compteurs
. lire un caractère
. Tantque le caractère lu est différent de 0
  Faire
     . incrémenter le bon compteur
     . lire un caractère
  Finfaire
. afficher les compteurs
```

Utilisation d'un Répéter :

```
. initialiser les compteurs
. Répéter
     . lire un caractère
     . incrémenter le bon compteur
  Jusqu'à ce que le caractère lu soit un 0
. afficher les compteurs
```

Utilisation d'un Itérer :

. initialiser les compteurs
. Itérer
    . lire un caractère
  Sortirsi le caractère lu est un 0
    . incrémenter le bon compteur
  Finitérer
. afficher les compteurs

La dernière version est plus logique, plus lisible et légèrement plus efficace. Ce dernier argument est peu important à nos yeux, mais on peut effectivement remarquer une petite perte de place dans le cas du Tantque (duplication de l'instruction de lecture) et une perte de temps dans le cas du Répéter (exécution de la recherche d'un compteur à incrémenter alors qu'on vient de lire un 0) ; cette perte de temps est faible si on lit une séquence assez longue de caractères, mais elle peut devenir importante si on lit plusieurs séquences courtes.

Il serait facile de multiplier les exemples de ce genre alors qu'il est difficile de trouver des contre-exemples. C'est pourquoi nous montrons au chapitre 3 comment coder les trois structures, mais dans les exemples des chapitres 4 et 5, nous utilisons uniquement la structure Itérer qui reste à nos yeux la seule à utiliser dans la pratique.

Après avoir discuté des principales structures permettant de décrire un traitement, nous allons exposer brièvement un dernier outil également intéressant.

4 Les automates d'états finis.

Les informaticiens connaissent moins bien que les automaticiens cet outil qui est cependant utilisé en informatique lors de différentes étapes du développement d'un logiciel. Nous allons d'abord définir rapidement ce qu'est un automate d'états finis, puis en donner deux exemples d'application.

4.1 Définition.

Un automate d'états finis est une description normalisée d'un processus qui répond à un certain nombre de conditions :
- le nombre d'états différents dans lequel peut se trouver le processus est limité (fini),
- le processus passe d'un état à un autre à des instants quelconques, en réponse à des événements extérieurs ; cette transition d'un état à un autre peut s'accompagner de l'exécution d'une action bien définie de la part du processus,
- quelle que soit l'histoire de l'évolution antérieure du processus, seul compte son état à un instant donné, et son évolution future ne dépend pas de son passé, mais uniquement de l'état dans lequel il se trouve à cet instant et des événements extérieurs à venir.

La description d'un processus sous forme d'un automate d'états finis est simple : un état est symbolisé par un cercle, et une transition par une flèche reliant deux cercles. L'automate d'états finis est ensuite décrit en pseudo-code très facilement grâce aux structures classiques. Nous allons voir tout ceci dans deux exemples.

4.2 L'automate d'un petit éditeur de texte.

Le chapitre 5 est consacré à la réalisation d'un petit éditeur de texte dont voici la description des spécifications de fonctionnement. Les opérations suivantes doivent être prises en compte : saisie du texte, affichage du texte, suppression d'une ligne, insertion d'une ligne. L'utilisateur commence par saisir complètement le texte qui est ensuite affiché. Il peut alors (et autant de fois qu'il le désire) afficher le texte, supprimer ou insérer une ligne ; une commande lui permet de quitter l'éditeur.

Ces spécifications peuvent se traduire sous forme de l'automate d'états finis de la figure 1.4. Par rapport à la définition d'un automate exposée au paragraphe 4.1, il y a une légère modification de conventions : ici, lorsque le processus arrive dans un état, il exécute un certain traitement ; lorsque ce traitement est terminé, ou bien il passe automatiquement à un autre état, ou bien il sollicite de l'extérieur un événement qui le fera transiter vers tel ou tel état ; dans l'exemple ci-dessus, c'est la frappe au clavier de l'initiale de la commande choisie par l'utilisateur qui est cet événement extérieur. On a l'habitude d'indiquer la condition d'une transition à proximité de l'origine de la flèche qui représente cette transition. Par ailleurs, on a ajouté deux états particuliers : Début et Fin.

Pour écrire le pseudo-code d'un automate, il faut d'abord numéroter les états (la coutume veut que l'on utilise la variable S, de l'anglais "state") et considérer le fonctionnement de l'automate comme une répétitive dont on sortira lorsqu'on aura atteint l'état final ; à l'intérieur de cette répétitive, on examine l'état dans lequel on se trouve et, selon le "cas", on exécute le traitement correspondant et/ou on teste l'événement extérieur qui déclenche la transition. La transition elle-même étant décrite par le changement de la valeur de la variable S. Dans notre exemple, on obtient le pseudo-code ci-après :

```
. saisir le texte initial et affecter à S la valeur 2
. Itérer
    . Cas où S vaut
        2 :  . afficher le texte et affecter à S la valeur 3
        3 :  . lire la commande C
             . Cas où C vaut
                 "A" :   . affecter à S la valeur 2
                 "I " :  . affecter à S la valeur 4
                 "S" :   . affecter à S la valeur 5
                 "Q" :   . affecter à S la valeur 6
               Fincas
        4 :  . insérer une ligne et affecter à S la valeur 3
        5 :  . supprimer une ligne et affecter à S la valeur 3
      Fincas
  Sortirsi on est à l'état final (S=6)
  Finitérer
```

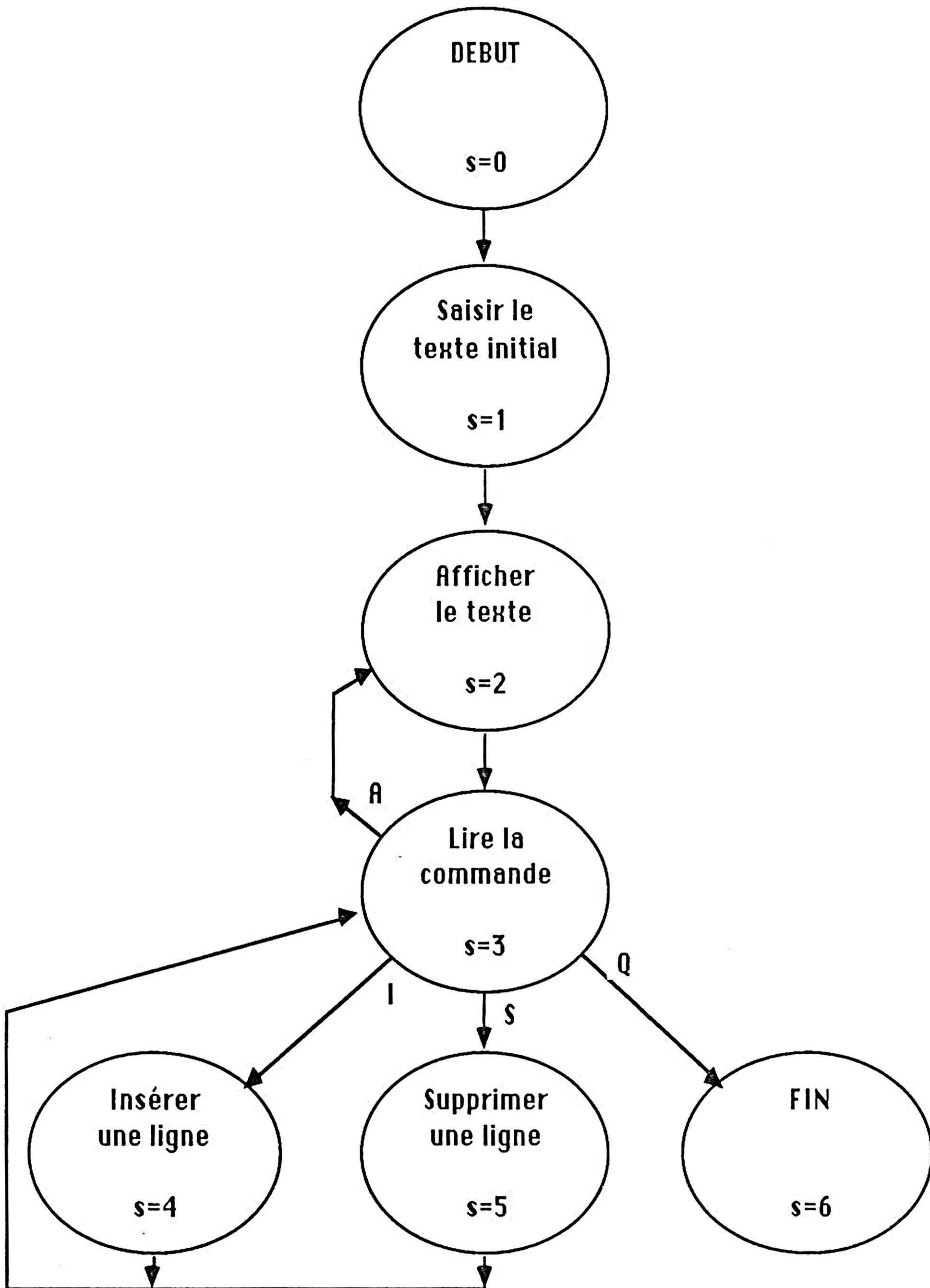


Fig.1.4 Automate d'un mini-éditeur de texte.

Après avoir montré l'utilisation d'un automate pour la décomposition modulaire, nous allons voir maintenant une autre application dans le cadre d'un module de traitement.

4.3 L'automate d'un module de transcodage.

Voici l'énoncé d'un exercice qui sera complètement développé au chapitre 4 : écrire un module permettant de traduire en binaire un nombre décimal inférieur ou égal à 99 entré au clavier et de placer le résultat dans la variable NB (un octet) ; la difficulté vient du fait qu'on suppose que l'utilisateur peut entrer ce nombre sous différentes formes (toutes terminées par un retour chariot RC) : un chiffre , deux chiffres, plus de deux chiffres ; dans ce dernier cas, il faut ne tenir compte que des deux premiers chiffres entrés ; de plus, si l'opérateur entre des caractères autres que des chiffres, il faudra les ignorer. On peut illustrer ces règles à l'aide des exemples ci-après dans lesquels le RC a été représenté par le symbole ®.

Suite de caractères saisis	Nombre retenu	NB
2®	2	0000 0010
43®	43	0010 1011
5678®	56	0011 1000
A3BC2U®	32	0010 0000
D®	aucun, erreur	code erreur

On peut traduire ces règles sous la forme de l'automate de la figure 1.5 et cet automate est ensuite traduit facilement en pseudo-code ;

```
. affecter à S la valeur 1
. Itérer
    . Cas où S vaut
        1 :  . lire un caractère et l'affecter à C1
             . Si C1 est un RC
               Alors  . affecter à S la valeur 4
               Sinon  . Si C1 est un chiffre
                             Alors  . affecter à S la valeur 2
                             Sinon  . affecter à S la valeur 1
                             Finsi
               Finsi
        2 :  . lire un caractère et l'affecter à C2
             . Si C2 est un RC
               Alors  . affecter à S la valeur 5
               Sinon  . Si C2 est un chiffre
                             Alors  . affecter à S la valeur 3
                             Sinon  . affecter à S la valeur 2
                             Finsi
               Finsi
        3 :  . lire un caractère et l'affecter à C3
             . Si C3 est un RC
               Alors  . affecter à S la valeur 6
               Sinon  . affecter à S la valeur 3
               Finsi
```

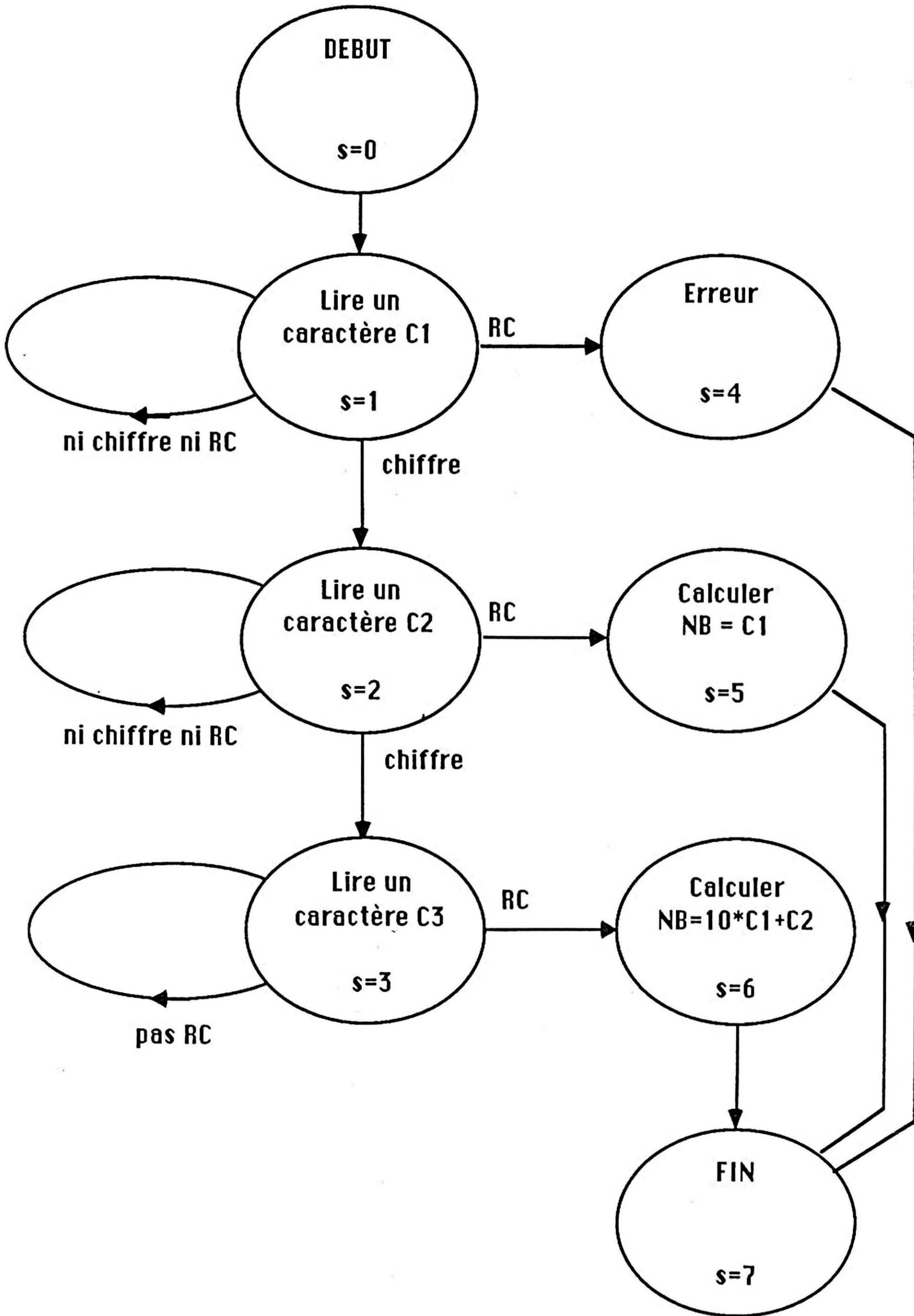


Fig 1.5 Automate d'un transcodeur (version 1).

4 : . placer la valeur symbolisant l'erreur dans NB et affecter à S la valeur 7
5 : . calculer NB (transformer C1 en binaire) et affecter à S la valeur 7
6 : . calculer en binaire NB = 10*C1+C2 et affecter à S la valeur 7
Fincas
Sortirsi on est à l'état final (S=7)
Finitérer

On verra au chapitre 4 comment coder ceci en assembleur. Pour l'instant, on peut illustrer l'efficacité de cet outil au niveau de la maintenabilité en proposant une modification aux spécifications du module : supposons que dans le cas où l'utilisateur entre plus de deux chiffres, il faille conserver les deux derniers et non plus les deux premiers. Le nouvel automate d'états finis est représenté sur la figure de la page suivante. La traduction de cette modification en pseudo-code est immédiate.

Conclusion

Il existe un certain nombre d'outils permettant de supporter la pensée dans les phases de développement d'un module : parmi ceux-ci, le pseudo-code, la table des identificateurs et l'automate d'états finis. Ces outils permettent de décrire les traitements et les données de façon précise et lisible. Ils améliorent donc la maintenabilité du logiciel. Les documents produits sont une partie de la documentation qui doit accompagner un listage de programme. Il est important de noter que ces documents sont produits avant d'entamer la phase de codage qui ne peut donc commencer que lorsque ces documents sont réceptionnés par une procédure quelconque. Certains emploient pour cette réception la méthode des revues et inspections dont le nom anglais "structured walkthrough" indique bien qu'il s'agit d'une des techniques de la programmation structurée. Cette méthode est basée sur une inspection méthodique et organisée (structurée) des documents produits à la fin de chaque phase du développement d'un produit. Elle a fait ses preuves, mais lors de son utilisation dans certaines entreprises on a constaté des réticences psychologiques qui freinent son expansion ; il est cependant bon d'en connaître l'existence. Nous ne pouvons en dire plus ici, mais il existe une bibliographie abondante sur ce sujet.

Après ce bref exposé sur la programmation structurée, nous pouvons passer à la programmation en assembleur... qui sera traitée dans le même esprit.

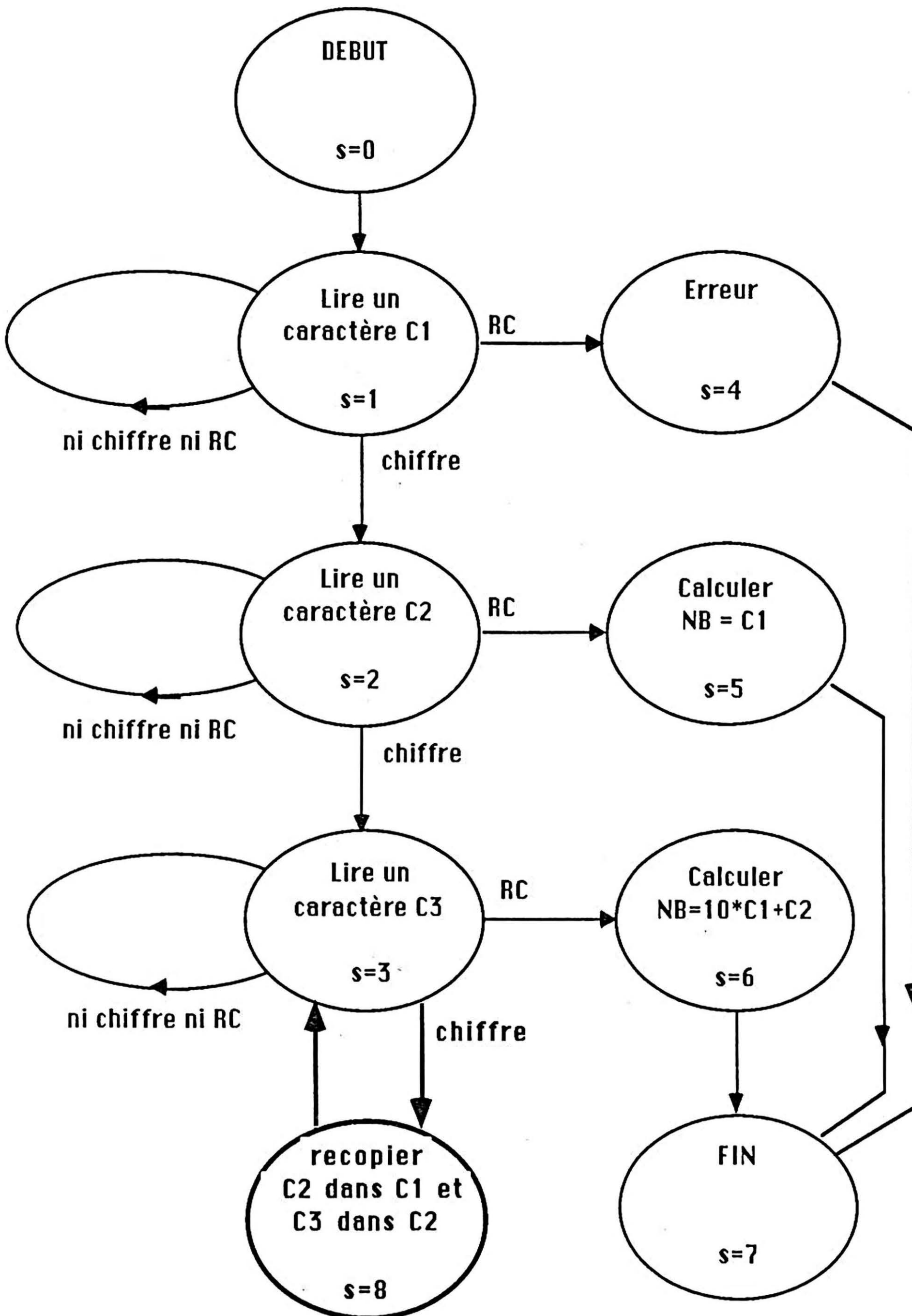


Fig 1.6 Automate d'un transcodeur (version 2).

QUELQUES ASPECTS DE LA PROGRAMMATION EN ASSEMBLEUR 6502 ET 65C02

Comme l'indique son titre, le but de ce chapitre n'est pas d'exposer la programmation en assembleur dans sa totalité, mais uniquement d'en présenter les aspects particulièrement importants pour la programmation structurée, le lecteur étant supposé connaître les bases de la programmation en assembleur et en avoir une certaine pratique. On verra successivement la structuration des données et le problème des branchements ; le problème de l'échange d'information entre les modules est exposé dans le chapitre 5.

Introduction : langage machine et assembleur.

Chaque micro-processeur a un jeu d'instructions bien défini ; chaque micro-ordinateur possède une mémoire dans laquelle on place les programmes et les données. La programmation en langage machine consiste à décrire un traitement par une suite d'instructions en binaire ou en hexadécimal, les objets à traiter étant précisés par leur adresse en mémoire. Un programme en langage machine est donc particulièrement peu lisible, difficile à mettre au point, et quasiment impossible à modifier.

Un assembleur est un langage de programmation peu évolué, dans lequel les traitements sont décrits par une suite d'instructions élémentaires du micro-processeur, mais qui permet quand même un certain nombre de choses qui facilitent la vie : en particulier les objets peuvent être identifiés par des identificateurs, et les instructions peuvent être repérées par des étiquettes. La traduction du code d'un programme écrit en assembleur en un code exécutable par la machine, donc en un programme en langage machine, se fait lors de l'assemblage.

Pour un même micro-ordinateur, le nombre d'assembleurs différents n'est pas limité. Dans le cas de l'Apple II, le constructeur livre la machine avec un assembleur rudimentaire, et on peut acheter séparément des assembleurs plus puissants ; un des plus répandus est l'assembleur LISA de Randall HYDE ; c'est celui que nous avons utilisé dans cet ouvrage pour présenter les exercices, mais la méthode que nous présentons est évidemment indépendante de l'assembleur. De la même façon, le contenu de ce chapitre concerne le micro-processeur 6502 indépendamment de l'assembleur utilisé.

Quelques systèmes possèdent des assembleurs très puissants qui mettent à la disposition des utilisateurs les structures alternatives et répétitives de la programmation structurée. Dans ce cas, notre méthode n'a plus beaucoup d'utilité.

1 Description d'un programme assembleur.

Un programme assembleur est composé d'instructions (instruction du micro-processeur ou directive de l'assembleur) ; une instruction comporte trois champs. L'étiquette permet d'identifier l'instruction, le code opératoire précise ce que fait l'instruction, l'opérande ce sur quoi porte l'opération.

Etiquette Code opératoire Opérande

Les instructions peuvent être réparties en plusieurs classes selon qu'elles décrivent un traitement, qu'elles précisent l'organisation du traitement, ou qu'elles décrivent une donnée.

```
         LDA   $900
         JMP   FIN
TABLEAU  DFS   $10
```

Le problème de la structuration en assembleur ne concerne que les deux dernières classes d'instructions : celles qui permettent de décrire la structuration des traitements et la structuration des données.

2 La structuration des données.

L'octet est la structure fondamentale sur laquelle opèrent la plupart des instructions. Par ailleurs, on peut accéder aux éléments d'un tableau grâce à l'adressage indexé. Enfin, le 6502 permet d'utiliser une pile. Nous allons voir successivement ces trois points.

2.1 Le contenu d'un octet.

Le contenu d'un octet sera interprété différemment par le 6502 selon les cas ; il est donc important que l'utilisateur comprenne bien ces différents cas.

a) Un octet = un ensemble de 8 bits indépendants. On peut effectuer des opérations logiques (AND...) ou de décalage (LSR...) ou de transfert (TAX...), etc...

b) Un octet = un caractère ASCII. Dans les opérations d'entrée-sortie, le 6502 manipule des caractères selon le code ASCII, et il faut en particulier savoir qu'après une lecture du caractère 3 au clavier, l'accumulateur contient 1011 0011 et non pas la valeur 3 en binaire. (Attention : dans l'Apple II le bit 7 est à 1)

c) Un octet = deux chiffres hexadécimaux. L'Apple II dispose d'un sous-programme PRBYTE qui affiche le contenu de l'accumulateur vu comme une paire de deux chiffres hexadécimaux : la valeur 1011 0011 sera donc affichée B3.

d) Un octet = un nombre binaire entier positif ou nul (naturel). Ce nombre peut prendre une valeur comprise entre 0 et 255. De nombreuses opérations modulo 256 sont possibles : addition, soustraction, incrémentation...

e) Un octet = un nombre binaire entier signé (relatif). Ce nombre peut prendre une valeur comprise entre -128 et +127. Après une opération arithmétique sur ce type de données, il faut tester si l'opération s'est correctement exécutée.

f) Un octet = un nombre décimal entier positif ou nul (<=99). Ce nombre est représenté en "décimal codé binaire condensé", c'est-à-dire que chacun des deux chiffres du nombre décimal est transformé en binaire sur quatre bits : par exemple 47 est codé 0100 0111. Ce type de données est utilisé par les instructions d'addition et de soustraction lorsqu'on demande au 6502 de travailler en décimal (SED).

Il est fondamental de ne pas oublier que le micro-processeur n'a aucune idée de la nature des données qu'il manipule ; c'est donc au programmeur de faire très attention à ce qu'il fait dans les opérations de traitement. Le programmeur doit donc être très précis sur le type des objets manipulés. La table des identificateurs permet de préciser ce type, et un octet pourra être de type caractère, ou naturel, ou relatif... De la même façon, si une paire d'octets consécutifs représente une adresse en mémoire, il faut le préciser dans la table des identificateurs.

2.2 Les tableaux.

Le tableau est une structure de donnée composée très facile à utiliser en assembleur. Lors de l'assemblage, il faut réserver de la place pour le tableau ; en assembleur LISA, on dispose de l'instruction DFS (Define Storage) : par exemple TAB DFS $A permet de réserver dix octets à partir de l'adresse qui sera attribuée à TAB. On accède aux éléments d'un tableau grâce à l'adressage indexé ; le 6502 dispose de deux registres d'index X et Y ; si le registre X contient la valeur 4, l'exécution de l'instruction LDA TAB,X aura pour effet de recopier dans l'accumulateur l'octet qui se trouve à l'adresse TAB + 4. C'est simple, mais il faut quand même faire attention au fait que le premier élément du tableau étant à l'adresse TAB, on aura recopié le cinquième élément de ce tableau. Sur ce principe, il est facile d'utiliser des tableaux dont les éléments occupent plus d'un octet.

2.3 La pile.

Les 256 octets de la page 1 peuvent être considérés comme les éléments d'une pile : il existe en effet des instructions permettant d'empiler la valeur de l'accumulateur (PHA) ou du registre d'état (PHP), et inversement de dépiler la valeur du sommet dans l'accumulateur (PLA) ou dans le registre d'état (PLP). Il faut cependant être très minutieux dans l'utilisation de cette pile car sa capacité est relativement faible, et elle est déjà utilisée au moment d'un appel de sous-programme : implicitement par le système pour la sauvegarde de l'adresse de retour, explicitement par le programmeur pour la sauvegarde du contexte. Il faudra donc faire très attention si on veut utiliser également cette pile pour y traiter des données structurées en pile, mais cette possibilité existe.

2.4 Remarques sur les identificateurs et les types.

Dans le chapitre précédent, au moment du choix des identificateurs de variables, nous avons souligné le danger de l'identificateur I, et nous avons indiqué notre préférence pour l'identificateur K ; nous venons de voir que l'accès aux éléments d'un tableau se faisait à l'aide des registres X ou Y. Il est donc évident que l'on choisira ces identificateurs lors de la rédaction du pseudo-code pour les "variables" servant d'indices aux éléments d'un tableau. Dans le cas d'un programme comportant plusieurs modules cela soulève un problème important : ces registres X et Y sont des ressources communes aux différents modules et il faut prendre un certain nombre de précautions en cas d'utilisation partagée de ces ressources communes ; nous détaillerons tout cela au chapitre 5 consacré à la programmation modulaire. Pour l'instant, on peut donner le conseil suivant : il faut absolument réserver l'utilisation des registres d'index à l'adressage indexé ; si on a besoin d'une variable pour compter ou décompter des événements, il ne faut pas utiliser ces registres, mais une variable ordinaire (K par exemple...).

Nous avons vu au chapitre précédent qu'un type de données est défini dès qu'on connait les valeurs que peuvent prendre les objets de ce type et les opérations que l'on peut effectuer sur ces objets. Puisque les opérations que l'on peut faire sur les registres d'index sont différentes de celles que l'on peut faire sur un emplacement en mémoire, un registre d'index n'a pas le même type qu'un octet. Dans la table des identificateurs, il faut donc bien différencier le type index et le type naturel, même si les objets de ces types prennent leurs valeurs dans le même domaine.

Après avoir décrit les différents objets manipulés par un programme, on peut passer aux possibilités concernant la structuration des traitements.

3 Structuration des traitements.

Il y a deux sortes d'instructions permettant de structurer un traitement : celles qui permettent d'organiser les traitements à l'intérieur d'un module, et celles qui permettent de déclencher l'exécution d'un module puis de revenir au module appelant. C'est ce dernier point qui pose le moins de problème et c'est par lui que nous allons commencer.

3.1 Appel d'un module et retour.

On verra dans le chapitre 3 que tout module doit commencer par une instruction vide dont l'étiquette est normalisée : XXDEB où XX sont deux lettres au choix du programmeur, mais différentes pour chaque module. De même, on verra que tout module doit se terminer par une instruction vide XXFIN. Un programme comporte un module maître dont la dernière instruction exécutable est une instruction BRK et d'autres modules dont la dernière instruction exécutable est une instruction RTS.

Exécuter un programme consiste donc à exécuter le module maître. L'exécution d'un module autre que le module maître est déclenchée par l'instruction JSR XXDEB et elle se termine lors de l'exécution de l'instruction RTS de ce module. Le 6502 mémorise, lors de l'exécution de l'instruction JSR l'adresse à laquelle il devra revenir lors de l'exécution ultérieure de l'instruction RTS, mais c'est au programmeur de sauvegarder les renseignements sur l'état de la machine dont il aura besoin à son retour. Ce problème de la sauvegarde du contexte (on dit aussi parfois du paysage) sera évoqué au chapitre 5.

3.2 Organisation des traitements à l'intérieur d'un module.

Pour décrire l'enchaînement des traitements à l'intérieur d'un module, il faut disposer de deux possibilités : le branchement inconditionnel et le branchement conditionnel. Le premier ne pose pas de difficulté : il est réalisé par l'instruction JMP dont l'opérande précise l'étiquette de l'instruction où l'on veut se brancher.

Le branchement conditionnnel pose un certain nombre de problèmes, dont certains sont spécifiques au 6502, et que nous allons exposer en détail.

Le 6502 comporte un registre d'état P composé de six bits qui indiquent l'état de la machine et un bit de contrôle des interruptions ; plus précisément :

- le bit 0 : C carry ou retenue,
- le bit 1 : Z zéro,
- le bit 2 : I interruption,
- le bit 3 : D mode décimal,
- le bit 4 : B break ou pause,
- le bit 5 : inutilisé,
- le bit 6 : V overflow ou dépassement,
- le bit 7 : N négatif ou nul.

Parallèlement, le 6502 dispose d'instructions de comparaison (CMP, CPX, CPY) qui positionnent certains de ces indicateurs et de huit instructions de branchement (BCC, BCS, BEQ, BNE, BMI, BPL, BVC, BVS). Lors de l'exécution de l'instruction B.. SUITE, le 6502 décide de poursuivre le traitement en séquence ou de se brancher à l'instruction SUITE selon l'état d'un de ses indicateurs. Dans le cas le plus général, un branchement conditionnel nécessitera donc trois instructions : chargement, comparaison, branchement. Par exemple :

```
LDA   QTTE
CMP   #$A2
BEQ   SUITE
```

Le principe des instructions de branchement conditionnel est donc simple, mais son utilisation est compliquée par la présence d'un certain nombre de pièges dont certains sont peu connus. Nous allons les passer en revue.

3.3 La valeur des indicateurs.

La règle générale qu'il faut retenir sur la valeur des indicateurs, c'est que la valeur 1 indique le "vrai" : le programme doit mettre l'indicateur D à 1 si l'on veut que le 6502 calcule en Décimal, de même l'indicateur B est mis à 1 par le 6502 après l'exécution de l'instruction BRK. Il découle de cette règle que l'indicateur Z est mis à 1 après une instruction dont l'exécution donne la valeur 0, et qu'il est mis à 0 dans le cas contraire. C'est logique, mais cela peut prêter à confusion dans certains cas. Par exemple, supposons qu'on veuille se brancher à l'adresse SUITE si le bit 5 de la variable K est à 1, il faut écrire :

```
LDA   K
AND   #%00100000
BNE   SUITE
```

3.4 Les instructions qui agissent sur les indicateurs.

Les débutants en assembleur ont tendance à penser que seules les instructions de comparaison positionnent les indicateurs Z, N et C. Il n'en est rien. Au contraire, de nombreuses instructions agissent sur ces bits. Pour en citer deux parmi les plus courantes, les instructions LDA et DEC agissent sur Z et N. On peut tirer parti de cela lorsqu'il s'agit par exemple de comparer à 0 la variable K ; plutôt que d'écrire :

```
LDA   K
CMP   #0
BEQ   SUITE
```

on peut gagner une instruction en écrivant :

```
LDA   K
BEQ   SUITE
```

ce qui revient strictement au même. On verra au chapitre 4 ce qu'il faut penser de ce style de programmation qui permet de gagner un petit quelquechose en encombrement mémoire et en rapidité d'exécution, mais qui fait perdre un petit quelquechose en lisibilité et en maintenabilité.

Dans un même souci de diminuer l'encombrement mémoire et le temps d'exécution, certains ont tendance à ne pas charger l'accumulateur avec l'un des membres de la comparaison si, quelques instructions auparavant ce chargement a déjà eu lieu :

```
LDA   K
...   ...
...   ...
CMP   #$B2
BEQ   SUITE
```

Il est évident que cette pratique est très dangereuse, car elle risque d'entraîner de graves difficultés de maintenance : si on est amené à toucher à la partie de code comprise entre le LDA et le CMP, il y a une probabilité non nulle pour que l'on modifie la valeur de l'accumulateur, et que l'on induise une erreur qui ne sera peut-être pas détectée par les quelques petits essais que l'on fera pour tester la modification.

3.5 Les comparaisons numériques.

La comparaison de valeurs numériques est certainement une des difficultés les plus méconnues de l'utilisation du 6502. Le piège vient des instructions BPL et BMI. On est tenté d'écrire par exemple :

```
LDA   K
CMP   LIMITE
BPL   SUITE
```

et d'espérer se brancher à SUITE si K>=LIMITE. Cela est faux ; ce qui est même plus grave, c'est que dans certains cas cela marche, et pas dans d'autres ! Nous allons voir successivement les problèmes de la comparaison de nombres naturels (positifs ou nuls) et de nombres relatifs (négatifs, positifs ou nuls) exprimés en complément à 2. Plutôt que de faire de l'arithmétique théorique, nous allons faire une démonstration expérimentale à l'aide de quelques exemples. Pour chaque exemple, nous allons noter la valeur de l'indicateur N, mais aussi de l'indicateur C après l'exécution de l'instruction CMP M en fonction des valeurs de l'accumulateur et de M.

a) Nombres naturels.

Nous allons voir deux exemples pour lesquels A<M ; et cependant, l'instruction de comparaison va positionner différemment l'indicateur N :

Exemple 1 : supposons A=48=$30 et M=200=$C8 donc A-M<0 ; lors de l'exécution de l'instruction de comparaison, l'opération A-M a pour résultat $68, et cela positionne C à 0 et N à 0;

Exemple 2 : supposons A=48=$30 et M=100=$64 donc A-M<0 ; lors de l'exécution de l'instruction de comparaison, l'opération A-M a pour résultat $CC, et cela positionne C à 0 et N à 1.

Conclusion : après une comparaison de nombres naturels, il ne faut pas utiliser BPL ou BMI car l'indicateur N n'est pas significatif (il vaut 1 si la différence entre les valeurs comparées est supérieure ou égale à $80 et 0 sinon) ; par contre, le bit C est significatif (même si les exemples ci-dessus ne suffisent pas pour le prouver) et il faut donc utiliser BCC pour un branchement lorsque A-M<0 et BCS pour un branchement lorque A-M>=0. A la réflexion, cela n'a rien d'étrange : le bit N est lié au bit 7 de l'octet considéré et le bit 7 est un bit de poids en arithmétique positive ; c'est donc logique que le bit N ne donne pas le résultat d'une comparaison de deux nombres positifs.

b) Nombres relatifs.

Nous allons voir trois exemples dans lesquels A-M>0 et nous allons vérifier que le bit N ne marche toujours pas, mais malheureusement, ici, le bit C ne marche pas non plus.
Exemple 1 : supposons A=48=$30 et M=-32=$E0 donc A-M>0 ; lors de l'exécution de l'instruction de comparaison, l'opération A-M a pour résultat $50, et cela positionne C à 0 et N à 0.
Exemple 2 : supposons A=100=$64 et M=-48=$D0 donc A-M>0 ; lors de l'exécution de l'instruction de comparaison, l'opération A-M a pour résultat $94, et cela positionne C à 0 et N à 1.
Exemple 3 : supposons A=-48=$D0 et M=-100=$9C donc A-M>0 ; lors de l'exécution de l'instruction de comparaison, l'opération A-M a pour résultat $134, et cela positionne C à 1 et N à 0.

Conclusion : le bit N ne permet pas de conclure une comparaison, le bit C non plus. En fait, la seule façon correcte de comparer deux nombres relatifs passe par une soustraction suivie de tests sur V et N. On trouve le détail dans les quelques bons manuels d'assembleur 6502.

3.6 Le fonctionnement en mode décimal.

L'utilisateur du 6502 peut décider de considérer le contenu d'un octet comme un nombre décimal positif ou nul de deux chiffres codés chacun en binaire sur 4 bits : les instructions SED et CLD permettent de passer du mode binaire au mode "décimal" et inversement. Lorsque le 6502 est en mode décimal, les addition, soustraction et comparaison se font évidemment en décimal codé binaire. Malheureusement les indicateurs Z et N ne sont plus utilisables. Il s'agit probablement d'une erreur de conception, mais il est facile de constater qu'après une opération en mode décimal, l'accumulateur contient bien le résultat en décimal, alors que les indicateurs Z et N sont positionnés en fonction du résultat de l'opération en binaire ; voici deux exemples :

- soit à soustraire 80 de 50 en mode décimal ; on retrouve bien dans l'accumulateur 70, c'est-à-dire 0111 0000 mais l'indicateur N est positionné à 1 car le résultat de l'opération binaire 0101 0000 (50) moins 1000 0000 (80) donne 1101 0000 ; cela n'est pas encore trop grave car il est peu probable que quelqu'un ait envie de tester l'indicateur N en mode décimal, mais par contre...
- soit à additionner 50 et 50 en mode décimal ; on retrouve bien dans l'accumulateur 00, c'est-à-dire 0000 0000 mais l'indicateur Z est positionné à 0 car le résultat de l'opération binaire 0101 0000 (50) plus 0101 0000 (50) donne 1010 0000 qui est différent de zéro.

Ce dernier cas est particulièrement vicieux pour celui qui commencerait par vérifier le fonctionnement de l'indicateur Z sur des calculs en décimal ! Il serait logiquement amené à penser que l'indicateur Z est positionné à 0 en cas de résultat nul, ce qui est faux.

On peut donc conclure qu'il ne faut jamais utiliser l'indicateur N, et qu'en ce qui concerne l'indicateur Z, on peut l'utiliser, mais uniquement après une soustraction ou une comparaison : le résultat étant le même en décimal et en binaire, cet indicateur sera correctement positionné. Pour finir sur ce sujet, précisons que l'indicateur C, quant à lui, fonctionne correctement et peut donc toujours être utilisé.

3.7 La notion de déplacement et ses conséquences.

Une instruction LDA K occupe 3 octets : un pour LDA et deux pour l'adresse de K (que l'on suppose être ailleurs qu'en page zéro). Il n'en va pas de même pour une instruction B.. SUITE qui n'occupe que deux octets : un pour B.. et un seul pour l'adresse SUITE : en effet, lors de l'assemblage, l'assembleur détermine la position de l'adresse SUITE par rapport à l'adresse où sera mémorisée cette instruction B.. SUITE et c'est la différence entre ces deux adresses, couramment appelée déplacement, qui est mémorisée dans un octet et un seul. Cette façon de faire a pour conséquence agréable de produire un code indépendant de l'adresse où est implanté le programme, mais elle a pour inconvénient de limiter l'éloignement de l'instruction à laquelle on veut éventuellement se brancher. On verra au chapitre 4 comment prendre en compte cette limitation.

3.8 Les particularités du 65C02.

Pour en terminer avec les instructions de branchement, il suffit d'ajouter quelques mots sur les instructions particulières au 65C02 : la plus notable différence en ce qui concerne la structuration, c'est l'apparition de l'instruction BRA qui réalise un branchement inconditionnel (branch always) analogue au JMP du 6502, mais avec deux améliorations. BRA est plus rapide que JMP ; BRA tient moins de place que JMP car c'est un déplacement qui est mémorisé, comme dans les autres instructions B... Par contre, il est évident que BRA ne permet pas de se brancher partout, puisque l'expression du déplacement est codée dans un octet.

Le 65C02 comporte également deux nouvelles instructions de branchement conditionnel BBS et BBR ("branch on bit set" et "branch on bit reset") qui permettent de tester n'importe quel bit d'un octet de la page zéro ; cela permet d'économiser une instruction dans la programmation de certains tests (par exemple celui donné au paragraphe 3.3 ci-dessus).

Conclusion.

Nous avons vu de l'assembleur les quelques points particuliers concernant la structuration des traitements et des données. Nous avons maintenant rassemblé les connaissances nécessaires pour entrer dans le vif du sujet : la programmation structurée en assembleur. C'est l'objet du chapitre suivant.

LES BASES DE LA PROGRAMMATION STRUCTUREE EN ASSEMBLEUR 6502 ET 65C02

Le but de ce chapitre est d'exposer une méthode simple et efficace pour coder en assembleur un traitement décrit en pseudo-code. Dans une première partie on montre comment coder chacune des structures de base. Dans la deuxième partie on montre comment coder un module faisant intervenir plusieurs structures de même type. Dans la troisième on montre comment coder plusieurs modules destinés à être assemblés ensemble. Après avoir évalué les avantages de cette méthode, on expose pour finir quelques détails d'utilisation.

Introduction : les principes de la méthode.

La méthode repose sur l'hypothèse de base suivante : on dispose d'une description en pseudo-code du module que l'on doit coder en assembleur. L'objectif de la méthode est de conserver tous les avantages de la structuration :

lisibilité
modularité
fiabilité

qui sont les préalables à une bonne **maintenabilité.**

Pour garder ces avantages, il faut parvenir à un code dépersonnalisé, c'est-à-dire indépendant de la personne qui a effectué le passage du pseudo-code à l'assembleur : ceci est obtenu essentiellement par la normalisation des étiquettes.

Les six règles ci-dessous résument les principes de la méthode ; les trois premières sont vraiment la base de la méthode : elles se comprennent immédiatement ; les trois dernières sont moins importantes, et moins évidentes, mais elles se justifient facilement dans la pratique.

R 1 : Tout mot réservé du pseudo-code (Si, Itérer,...) doit obligatoirement apparaître comme étiquette. Seuls ces mots réservés sont autorisés comme étiquettes.

R 2 : L'ordre d'apparition des mots réservés doit être conservé : le Si avant le Alors, puis le Sinon et enfin le Finsi.

R 3 : Il faut respecter scrupuleusement le pseudo-code lors du codage.

R 4 : Il faut prévoir les modifications ultérieures qui apparaîtront lors de la mise au point ou plus tard en maintenance, donc ne pas escamoter les parties vides.

R 5 : Les étiquettes de début et de fin d'une structure (Si, Finsi, Itérer, Finitérer,...) seront portées par des instructions vides.

R 6 : Il faut remettre à plus tard l'optimisation.

Avant de poursuivre, il faut bien remarquer que notre méthode est complètement différente de celle qui consiste à insérer dans le code, sous forme de commentaires, tout ou partie du pseudo-code. Dans notre cas, les mots-clés sont utilisés comme étiquettes et jouent donc un rôle lors de l'assemblage et de l'exécution du programme.

Plutôt que d'expliquer théoriquement la méthode, il est beaucoup plus simple de la mettre en application successivement sur les différentes structures élémentaires utilisées dans le pseudo-code. Dans ce début de chapitre, on ne tient pas encore compte des contraintes liées à tel ou tel assembleur, comme par exemple le nombre maximum de caractères d'une étiquette. On pourrait presque dire que l'on commence par traduire le pseudo-code en pseudo-assembleur. Ce n'est qu'au paragraphe 3 que l'on verra les détails de mise en oeuvre, et c'est seulement dans les chapitres 4 et 5 que l'on verra des exemples de programmes complets, c'est-à-dire incluant les déclarations de données.

1 La programmation des structures élémentaires.

Nous allons voir successivement comment coder les structures élémentaires : alternatives d'abord, répétitives ensuite.

1.1 L'alternative simplifiée : Si Alors Finsi.

Soit à coder le petit bout de pseudo-code suivant :

```
Si l'accumulateur vaut $12
Alors . le recopier à l'adresse $900
      . le remettre à zéro
Finsi
```

Dans un premier temps, l'application des trois premières règles de la méthode conduit à le coder de la manière suivante :

```
SI        :
          CMP   #$12
          BEQ   ALORS
          JMP   FINSI
ALORS     STA   $900
          LDA   #0
FINSI     :
```

Il est évident que la suite des deux instructions BEQ ALORS et JMP FINSI peut être condensée en une seule :

```
SI          :
            CMP     #$12
            BNE     FINSI
ALORS       STA     $900
            LDA     #0
FINSI       :
```

On remarque dans ce deuxième codage que l'étiquette ALORS a été conservée, bien qu'elle n'apparaisse dans la partie opérande d'aucune instruction : ceci est conforme à la règle 1 de la méthode, avec l'intérêt de bien visualiser le parallélisme entre le pseudo-code et le code. On remarque également que le branchement BEQ a été remplacé par un BNE.

L'instruction vide a été représentée par le caractère ":", qui est proche de celle de l'assembleur LISA que nous utiliserons pour programmer les exemples.

1.2 L'alternative complète : Si Alors Sinon Finsi.

Reprenons la structure précédente en complétant l'alternative par un Sinon :

Si l'accumulateur vaut $12
Alors . le recopier à l'adresse $900
. le remettre à zéro
Sinon . lui ajouter le contenu de $910
Finsi

Voici le code correspondant :

```
SI          :
            CMP     #$12
            BNE     SINON
ALORS       STA     $900
            LDA     #0
            JMP     FINSI
SINON       CLC
            ADC     $910
FINSI       :
```

L'instruction étiquetée SINON doit être précédée d'une instruction JMP FINSI faute de quoi le traitement correspondant au Sinon serait effectué dans tous les cas. Ceci doit faire l'objet d'une vérification systématique avant assemblage : on dispose ainsi d'un moyen simple de détecter un certain nombre d'étourderies ! Par ailleurs, on constate encore que l'étiquette ALORS n'est toujours pas utilisée comme opérande.

L'alternative généralisée Cas où se code également facilement ; on verra comment en fin de ce chapitre.

1.3 La boucle Tantque.

Soit le pseudo-code suivant :

> Tantque le registre X est strictement inférieur à $12
> Faire . additionner le contenu de $900 à l'accumulateur
>   . ajouter 2 au registre X
> Finfaire

Voici le codage correspondant :

```
TANTQUE     :
            CPX     #$12
            BCS     FINFAIRE
FAIRE       CLC
            ADC     $900
            INX
            INX
            JMP     TANTQUE
FINFAIRE    :
```

Toute instruction étiquetée FINFAIRE doit être précédée d'une instruction JMP TANTQUE (faute de quoi on aurait codé une alternative et pas une répétitive) ; toute instruction FAIRE doit être précédée d'une instruction B.. FINFAIRE : voilà encore deux points de vérification systématique à effectuer avant assemblage.

On peut discuter du choix de l'étiquette FINFAIRE : pour l'instant, nous avons conservé le mot utilisé dans le pseudo-code ; cependant, dans la mesure où cette étiquette correspond au point de sortie de la boucle et non pas à la dernière instruction du traitement à répéter, cela peut entraîner une confusion dans l'esprit de certains et il aurait peut-être été plus judicieux de choisir l'étiquette FINTANTQUE.

On rappelle par ailleurs que l'instruction BCS provoque un branchement à l'adresse indiquée si le bit C du registre P est à 1, elle correspond donc bien à une condition "supérieur ou égal".

1.4 La boucle Répéter.

Cette boucle est tout aussi simple à coder que la précédente ; soit à coder par exemple le pseudo-code suivant :

> Répéter . ajouter 2 au registre X
>   . ajouter le contenu de $900 à l'accumulateur
> Jusqu'à ce que le registre X soit supérieur ou égal à $12

On obtient immédiatement le code :

```
REPETER     :
            INX
            INX
            CLC
            ADC     $900
JUSQUA      CPX     #$12
            BCC     REPETER
FINREPETER  :
```

Ici, la vérification systématique porte sur la présence d'une instruction B.. REPETER derrière l'instruction JUSQUA.

On a donné ici les règles de codification des boucles Tantque et Répéter, mais c'est uniquement pour montrer qu'on peut les coder facilement, et non pas en vue d'une utilisation réelle.

1.5 La boucle Itérer.

On vient de voir les boucles les plus utilisées en pseudo-code lorsqu'il s'agit de coder en Pascal, mais nous vous rappelons qu'il est conseillé d'utiliser la boucle Itérer, plus générale, lorsqu'il s'agit de développer une application en assembleur.

Le principe de la méthode est suffisamment clair pour qu'il soit inutile d'expliciter un traitement précis avant et après la sortie de la boucle ; soit donc à coder le pseudo-code suivant :

```
Itérer
   . traitement 1
Sortirsi l'accumulateur vaut $12
   . traitement 2
Finitérer
```

On obtient le code suivant, dans lequel il y a deux vérifications à faire systématiquement avant de passer à l'assemblage.

```
ITERER      :
            traitement 1
SORTIRSI    CMP     #$12
            BEQ     FINITERER
            traitement 2
            JMP     ITERER
FINITERER   :
```

Lorsque le "traitement 2" est absent, il est tentant d'éviter la succession des deux instructions BEQ FINITERER et JMP ITERER pour aboutir au codage suivant :

```
ITERER      :
            traitement 1
SORTIRSI    CMP   #$12
            BNE   ITERER
FINITERER   :
```

Ceci est fortement déconseillé : en effet, en cas de modification ultérieure nécessitant l'introduction d'un "traitement 2", cela risque de mal se passer : imaginons par exemple que ce "traitement 2" consiste simplement à remettre à zéro l'accumulateur ; où faut-il placer l'instruction LDA #0 ? Après l'instruction BNE n'est pas le bon endroit puisqu'alors cette remise à zéro ne serait pas effectuée avant de recommencer le "traitement 1". Avant l'instruction BNE est encore pire (...façon de parler) puisqu'alors ce "traitement 2" sera exécuté avant de sortir... sans compter que son exécution risque de modifier la valeur du registre P et donc de rendre inopérant le branchement (ce serait le cas ici avec la remise à zéro de l'accumulateur). La seule façon correcte d'injecter ce "traitement 2" est de décomposer le BNE en un BEQ et un JMP.

Il est évident qu'en faisant très attention à ce que l'on fait, il ne devrait pas y avoir d'erreur lors d'une telle modification, mais c'est justement le but de la programmation défensive que de se protéger contre les erreurs éventuelles, en particulier lors de la mise au point ou en cours de maintenance ; c'est là l'origine de la règle 4 : "il faut prévoir les modifications ultérieures qui apparaîtront lors de la mise au point ou plus tard en maintenance, donc ne pas escamoter les parties vides."

1.6 La boucle Pour.

Une boucle Pour contient une initialisation et une incrémentation, sans compter un traitement et un test de sortie. Nous déconseillons l'utilisation de la boucle Pour dans un pseudo-code destiné à être codé en assembleur : il y a trop de façons différentes de coder cette boucle : certains mettent le test de sortie après le traitement (à la façon de Basic ou de Fortran IV) alors que d'autres mettent le test avant le traitement (à la façon de Pascal) ; en assembleur, on peut également placer en divers endroits l'incrémentation ; on peut enfin calculer une fois pour toutes, avant d'entrer dans la boucle, la valeur limite de la variable qui gère la boucle ou au contraire recalculer cette valeur limite avant de la tester etc...

Devant un tel nombre de codages possibles de la boucle Pour, nous jugeons préférable de déconseiller son emploi : il est certes possible de programmer sans erreur une boucle Pour, mais lors des modifications ultérieures, les risques d'erreur sont énormes, surtout si ces modifications sont faites par une autre personne que celle qui a programmé initialement ; en conséquence, au moment où vous mettez au point le pseudo-code d'un traitement destiné à un codage en assembleur, n'utilisez pas la boucle Pour ; et si un pseudo-code que vous avez écrit et mis au point en langage évolué doit être transformé en assembleur pour des questions de performances,

```
Si ce pseudo-code contient des boucles Pour
Alors   . transformez-les en boucles Itérer en explicitant
          l'initialisation, l'incrémentation et le test
        . codez
Finsi
```

Attention, ces transformations doivent être faites par écrit dans le pseudo-code avant d'entreprendre le codage pour deux raisons :

1° le code doit être le reflet exact du pseudo-code,

2° si vous ne modifiez pas le pseudo-code avant, vous ne le ferez pas après : vous oublierez, ou vous n'aurez pas le courage, ou vous n'aurez pas le temps.

Et puis, la règle 3 le dit bien : "il faut respecter scrupuleusement le pseudo-code lors du codage" ; en conséquence, si on doit modifier le pseudo-code, il faut le faire avant le codage !

Nous en avons terminé avec la codification des différentes structures de base que l'on rencontre dans le pseudo-code et nous pouvons maintenant passer à la phase suivante : le codage d'un module dans lequel figurent plusieurs alternatives et/ou plusieurs répétitives.

2 La programmation de structures multiples.

Certains lecteurs ont probablement eu la tentation de critiquer la méthode proposée en remarquant que bien souvent on avait plus d'une structure Si ou Itérer et que le même mot clé ne peut pas servir d'étiquette en deux endroits différents ! La réponse à cette critique fort judicieuse est simple : il suffit de numéroter les structures (et donc les mots réservés) du pseudo-code avant de passer au codage.

Nous allons illustrer la mise en pratique de ceci sur un exemple.

Supposons le problème suivant : lire une suite de caractères et n'afficher que les chiffres ; s'arrêter quand on lit un zéro (après l'avoir affiché). Le pseudo-code est simple :

```
Itérer
        . lire un caractère
        . l'afficher si c'est un chiffre
Sortirsi le caractère lu est un zéro
Finitérer
```

Sachant que le code ASCII étendu de 0 est $B0 et que celui de 9 est $B9, on arrive au pseudo-code suivant :

```
Itérer1
        . lire un caractère
        . Si1 le code du caractère lu est >= $B0
          Alors1 . Si2 le code du caractère lu est <= $B9
                   Alors2 . écrire le caractère lu
                   Finsi2
          Finsi1
Sortirsi1 le caractère lu est un zéro
Finitérer1
```

La structure Itérer a été numérotée, bien qu'étant unique, en prévision d'éventuels ajouts ultérieurs d'autres structures Itérer. Le codage se déduit très simplement, en utilisant RDKEY et COUT qui sont deux sous-programmes de lecture d'un caractère au clavier, le caractère lu se retrouvant dans l'accumulateur (read key) et de sortie d'un caractère à l'écran, le caractère sorti étant le contenu de l'accumulateur (character out) ;

```
ITERER1     :
            JSR     RDKEY
SI1         :
            CMP     #$B0
            BCC     FINSI1
ALORS1      NOP
SI2         :
            CMP     #$BA
            BCS     FINSI2
ALORS2      JSR     COUT
FINSI2      :
FINSI1      :
SORTIRSI1   CMP     #$B0
            BEQ     FINITERER1
            JMP     ITERER1
FINITERER1  :
```

Ce programme nécessite plusieurs commentaires. D'abord une remarque concernant le code assembleur proprement dit : la comparaison du SI2 a comme opérande la constante $BA alors que le pseudo-code compare à $B9 : cette différence est due au fonctionnement de l'instruction BCS ; pour que le code et le pseudo-code soient bien cohérents, il faut donc modifier le pseudo-code et écrire : Si2 le code lu est < $BA au lieu de : Si2 le code lu est <= $B9 ; on se trouve ici devant un de ces rares cas où les limitations du micro-processeur conduisent à de légères modifications du pseudo-code. Il ne faut pas oublier la règle 3 et il faut donc modifier le pseudo-code avant de coder.

En ce qui concerne la structuration de ce programme on peut remarquer deux choses : premièrement le fait que le traitement du Alors1 commence par une alternative Si2 et non par un traitement quelconque nous oblige à utiliser une instruction porte-étiquette (par convention, nous avons réservé la syntaxe ':' pour les débuts et fins de structures et l'instruction vide NOP pour les autres cas) ; deuxièmement, on constate que les deux instructions successives FINSI1 et FINSI2 ne sont séparées par aucun traitement ; il ne faudrait pas en déduire qu'on peut les rassembler en un seul FINSI qui serait global : c'est tout aussi interdit de le faire en assembleur qu'en pseudo-code : à tout Si doit correspondre un Finsi ; cette règle de la programmation structurée permet une bonne lisibilité et améliore la maintenabilité ; elle se traduit dans notre méthode par les règles 1 (tout mot réservé du pseudo-code doit obligatoirement apparaître comme étiquette) et 4 (...prévoir les modifications ultérieures...).

Dernières remarques en ce qui concerne la numérotation des structures : d'abord, il faut remarquer que le choix des numéros n'est pas critique et qu'il peut être absolument quelconque ; la plupart du temps, les numéros découleront des différentes étapes de l'analyse descendante et les numéros d'un même type de structure ne seront donc pas forcément dans l'ordre ; dans ce cas, même si la documentation finale ne conserve pas les différentes étapes du pseudo-code, il n'est pas obligatoire de procéder à une renumérotation ; enfin, il est probable que des modifications ultérieures viendront de toutes façons perturber la numérotation.

En cette fin de paragraphe, on peut donner un exemple permettant d'apprécier l'intérêt de la méthode au niveau de la maintenabilité : imaginons que l'on nous demande de modifier le traitement de façon à ne pas afficher le chiffre 0 qui termine la séquence d'entrée. Nous allons d'abord modifier le pseudo-code ; on obtient immédiatement :

```
Itérer1
      . lire un caractère
Sortirsi1 le caractère lu est un zéro
      . l'afficher si c'est un chiffre
Finitérer1
```

puis :

```
Itérer1
      . lire un caractère
Sortirsi1 le caractère lu est un zéro
      . Si1 le code du caractère lu est >= $B0
        Alors1 . Si2 le code lu est < $BA
                 Alors2 . écrire le caractère lu
                 Finsi2
        Finsi1
Finitérer1
```

Par rapport à la dernière version, il suffit donc de déplacer le bloc Si1 derrière le Sortirsi1 ; la modification se transpose immédiatement au niveau de l'assembleur : on transporte le bloc d'instructions comprises entre les étiquettes SI1 et FINSI1 (bornes comprises) derrière l'instruction BEQ FINITERER1. Attention, il ne faut évidemment pas mettre ce bloc entre les instructions SORTIRSI1 CMP #$B0 et BEQ FINITERER1 qui sont inséparables. On obtient immédiatement :

```
ITERER1     :
            JSR     RDKEY
SORTIRSI1   CMP     #$B0
            BEQ     FINITERER1
SI1         :
            CMP     #$B0
            BCC     FINSI1
ALORS1      NOP
SI2         :
            CMP     #$BA
            BCS     FINSI2
ALORS2      JSR     COUT
FINSI2      :
FINSI1      :
            JMP     ITERER1
FINITERER1  :
```

Ce petit exemple illustre clairement la facilité de modification d'un code structuré.

Maintenant que nous savons coder un module dans lequel figurent plusieurs structures, nous pouvons passer à l'étape suivante qui est l'intégration de plusieurs modules dans un même assemblage.

3 Codage de modules en vue d'un assemblage unique.

Dans la plupart des applications réelles, l'analyse conduit à découper le traitement en un certain nombre de modules dont on développe séparément le pseudo-code, mais qui sont ensuite assemblés ensemble. Le problème de la duplication des étiquettes à l'intérieur d'un module, que l'on avait résolu en numérotant les mots clés, se pose de nouveau. Nous allons le résoudre de manière semblable en choisissant d'identifier toutes les étiquettes d'un même module grâce à quelques caractères (rappelant la fonction du module) qui apparaîtront au début de chaque étiquette : par exemple, toutes les étiquettes d'un module de lecture commenceront par LC, toutes celles d'un module d'écriture par EC, etc...

Cela nous amène au problème de la longueur des étiquettes : dans tous les assembleurs, cette longueur est limitée. Dans le cas de LISA, les étiquettes ont au plus 6 caractères. Il faut donc partager ces six caractères entre les 3 informations que doit porter une étiquette :
- identification du module,
- mot réservé,
- numéro de la structure dans le module.

Nous avons choisi de réserver les deux premières positions de l'étiquette pour l'identification du module, les trois suivantes pour le mot réservé, et la dernière pour le numéro de la structure dans le module.

Il est inutile de réserver plus d'une position pour cette dernière information ; en effet, depuis plusieurs années, on a effectué des recherches théoriques et des mesures expérimentales sur la complexité du logiciel que l'on peut résumer ainsi :
- une bonne mesure de la complexité d'un module est le nombre total d'alternatives et de répétitives qu'il contient ;
- parmi les modules de complexité supérieure à 10, il y en a une proportion significative qui recèlent des erreurs détectées uniquement dans la phase d'exploitation. A la suite de ces travaux on essaye, lors de la décomposition modulaire, d'arriver à des modules dont la complexité ne dépasse pas trop la valeur 10.

Pour en terminer avec la normalisation des étiquettes, il suffit de préciser que chaque mot clé est toujours représenté de la même façon, selon la table ci-dessous :

SII	Si	ITR	Itérer
ALO	Alors	SSI	Sortirsi
SNO	Sinon	FIT	Finitérer
FSI	Finsi		

Avec ces conventions l'étiquette LCFIT2 indiquera la fin de la deuxième répétitive du module LC.

Certains lecteurs seront peut-être choqués de la faible distance entre SII et SSI. Nous en sommes conscients mais, d'une part nous n'avons pas trouvé mieux et d'autre part, l'expérience nous a montré que ce choix n'était pas pénalisant.

Enfin, chaque module doit être délimité par deux instructions vides, l'une d'entrée et l'autre de sortie, dont les étiquettes sont également codifiées :

DEB	Entrée
FIN	Sortie

Cette codification des étiquettes permet de localiser rapidement toutes les instructions d'un même module, son point d'entrée et son point de sortie, ce qui améliore encore la lisibilité.

Le listage de la figure 3.1 donne la version finale du petit exemple d'acquisition de caractères avec affichage des chiffres exposé au paragraphe 2.

```
LCDEB    :
LCITR1   :
         JSR     RDKEY
LCSII1   :
         CMP     #$B0
         BCC     LCFSI1
LCALO1   NOP
LCSII2   :
         CMP     #$BA
         BCS     FSI2
LCALO2   JSR     COUT
LCFSI2   :
LCFSI1   :
LCSSI1   CMP     #$B0
         BEQ     LCFIT1
         JMP     LCITR1
LCFIT1   :
         BRK
RDKEY    EQU     $FD0C
COUT     EQU     $FDED
LCFIN    :
```

Fig. 3.1 Listage en assembleur LISA.

A première vue, ce listage peut paraître touffu... et peu lisible !!! C'est vrai à première vue et cela tient à trois raisons : premièrement, dans cet exemple les instructions de traitement sont beaucoup moins nombreuses que les instructions d'organisation, ce qui n'est pas le cas général dans la pratique ; deuxièmement, il faut évidemment une (courte) période d'adaptation pour s'habituer à la norme des mots-clés ; enfin, les deux alternatives imbriquées proviennent en fait d'une condition double dans laquelle intervient un ET logique, nous verrons au paragraphe 5.2 une manière plus légère de coder ce genre de condition.

Cette remarque conduit directement au paragraphe suivant dans lequel on commence à discuter a priori des avantages et des inconvénients de la méthode (on y reviendra quand on l'aura pratiquée).

4 Evaluation de la méthode.

Cette méthode de codage en assembleur présente un certain nombre d'avantages qu'il suffit d'énumérer : c'est ce que nous verrons d'abord ; par contre, certains points sont moins évidents (performances en temps d'exécution et en place mémoire) : nous en discuterons dans la deuxième partie de ce paragraphe.

4.1 Les avantages évidents.

D'abord, c'est une méthode : "Démarche ordonnée de l'esprit pour parvenir à un certain but" (Larousse). Et il faut bien dire qu'il y a peu de méthodes concurrentes : en général les gens qui programment en assembleur ont tout au plus des habitudes personnelles. Cette méthode a un certain nombre de caractéristiques agréables : elle est facile à comprendre, facile à enseigner, facile à mettre en œuvre. De plus, cette méthode parvient bien aux buts fixés : le code obtenu conserve tous les avantages de la structuration : **lisibilité, modularité, fiabilité** et il présente donc toutes les garanties au niveau de la **maintenabilité**.

Enfin, il est très facile, au vu du code produit, de vérifier que cette méthode a été correctement utilisée : il suffit de faire des vérifications lexicales simples sur les étiquettes, et des vérifications structurelles (syntaxiques) répertoriées et tout aussi simples sur les instructions d'une même structure de base. En conséquence, lorsqu'il a été décidé, au niveau d'un groupe quelconque de développeurs, d'utiliser cette méthode, tout un chacun peut en vérifier la bonne utilisation, ou au contraire détecter très rapidement une tendance schismatique, qu'elle soit consciente ou involontaire. Ceci est très important, car il serait inutile d'avoir une bonne méthode si on était incapable d'en contrôler immédiatement l'application correcte. Au contraire, ici, on peut s'assurer, dès la production du code, que toutes les précautions ont été prises pour en faciliter la maintenance ultérieure.

4.2 Les performances.

On est conduit à utiliser l'assembleur essentiellement pour l'une ou/et l'autre des deux raisons suivantes : problème de place en mémoire pour loger le code exécutable, ou problème de temps d'exécution. Il est donc important de voir quelles sont les conséquences de l'utilisation de cette méthode sur les deux plans de l'encombrement et du temps d'exécution.

Il est indéniable que le code produit n'est pas le meilleur qu'on puisse imaginer. Essayons de tirer les conséquences de cette constatation.

La simple inspection du code obtenu, mais aussi l'expérience que nous avons accumulée, nous permettent d'affirmer que les coefficients d'expansion en place ou en temps sont inférieurs à 1.1 par rapport au code optimal produit par un programmeur chevronné et "astucieux", pour un algorithme donné quelconque. Autrement dit, on aboutit à quelque chose qui est au pire 10% plus encombrant, et 10% plus lent.

Nous allons maintenant supposer que des spécifications précises ont été formulées lors de l'étude initiale du projet à développer. Des contraintes sur l'encombrement et le temps d'exécution ont donc été énoncées. Alors, de trois choses l'une :
- Le code produit satisfait aux contraintes : c'est parfait. Cela ne servirait à rien d'essayer d'améliorer des performances satisfaisantes, au détriment de la maintenabilité.
- Le code produit est loin de satisfaire aux contraintes : si une hypothétique amélioration de 10% n'est pas suffisante, alors, c'est au niveau de l'algorithme qu'il faut agir, et non pas au niveau du codage.
- Le code produit est satisfaisant à quelques pour-cent près : alors, il faut essayer de faire quelque chose : à partir du code obtenu, supprimer des instructions inutiles, éliminer les parties vides, remplacer les NOP par des : , etc...

On peut maintenant faire un certain nombre de remarques : d'abord, les simples lois de la statistique permettent d'affirmer que le troisième cas n'est pas le plus fréquent : dans la majorité des cas, ou bien le codage obtenu sera satisfaisant, ou bien il faudra revoir au niveau de l'algorithme ; ensuite, dans le cas où une optimisation du code produit permet d'espérer une amélioration satisfaisante, on peut affirmer, sans crainte de démenti, que **il est plus facile d'optimiser un programme qui tourne, que de mettre au point un programme "optimisé" qui ne tourne pas.**

Enfin, dans le cas où on est conduit à écrire une version condensée, il faut conserver la version brute, de façon à faciliter la maintenance : en cas de modification ultérieure, on commence par s'attaquer au pseudo-code, puis on modifie le code brut (en vérifiant si, par hasard, il n'est pas devenu satisfaisant par rapport aux nouveaux critères d'encombrement et de temps d'exécution, ... on ne sait jamais), et on cherche ensuite à améliorer ce nouveau code brut, en s'inspirant bien sûr des optimisations conduites sur les versions antérieures.

4.3 Et les commentaires ?

Il faut d'abord rappeler que notre méthode est complètement différente de celle qui consiste à insérer dans le code, sous forme de commentaires, tout ou partie du pseudo-code. Dans notre cas, les mots-clés sont utilisés comme étiquettes et jouent donc un rôle lors de l'assemblage du programme et en conséquence lors de son exécution.

A notre avis, un programme en assembleur ne doit comporter aucun commentaire ; par contre, il doit être accompagné d'une documentation complète rédigée avant son codage : c'est un des principes de la programmation structurée sur lequel s'appuie notre méthode. Si le pseudo-code n'existe pas, on peut évidemment appliquer nos règles sur le choix des étiquettes, mais cela ne donnerait pas de bons résultats : en effet, c'est dans les différentes versions successives et de plus en plus détaillées du pseudo-code que l'on trouve les renseignements sur la stratégie globale du traitement mis en œuvre dans les différents modules. Il est fondamental de comprendre le but d'un traitement avant d'essayer de comprendre les instructions utilisées pour y parvenir.

La seule question à trancher concerne la présentation du pseudo-code et du listage ; il y a deux possibilités.

- Présentation séparée : l'ensemble des pseudo-codes et des tables des identificateurs des différents modules est relié dans un document unique, puis le listage des programmes correspondants est sorti également dans un document unique ; ces deux documents

séparés forment alors l'ensemble de la documentation du programme, à laquelle s'ajouteront plus tard le document résumant les tests et la notice d'utilisation.

- Présentation intercalée : on réalise un seul document dans lequel on trouve pour chaque module le pseudo-code et la table des identificateurs suivis immédiatement par le listage du programme.

Il ne semble pas qu'il y ait un argument décisif en faveur de l'une ou de l'autre et le choix entre ces deux présentations dépend plutôt de la puissance des éditeurs de texte disponibles et des préférences des utilisateurs.

Après cette discussion sur l'efficacité prévisible de la méthode, certains voudront passer directement au chapitre suivant dans lequel on montre son utilisation. Il est tout-à-fait possible de sauter le dernier paragraphe de ce chapitre dans lequel nous allons présenter quelques points de détail ; le lecteur qui déciderait de passer au chapitre suivant pourra toujours faire un retour en arrière lorsque le besoin s'en fera sentir.

5 Détails complémentaires.

Dans la pratique, la mise en œuvre de la méthode soulève quelques problèmes simples à résoudre que nous décrivons ci-dessous. Des exemples concrets d'utilisation sont exposés dans les chapitres suivants.

5.1 La structure Itérer avec plusieurs sorties.

Il arrive que l'on soit conduit à utiliser dans le pseudo-code une structure Itérer avec plusieurs sorties :

Itérer
    . traitement 1
Sortirsi condition 1
    . traitement 2
Sortirsi condition 2
    . traitement 3
Finitérer

Pour citer un cas d'application fréquent, c'est ainsi que l'on code une condition composée dans laquelle apparaît un OU :

Sortirsi condition 1 ou condition 2

sera modifié en deux Sortirsi successifs :

Sortirsi condition 1
Sortirsi condition 2

Le codage ne pose aucune difficulté, il faut simplement préciser le choix des étiquettes des différents Sortirsi : on ne peut évidemment pas les numéroter, puisque les numéros sont réservés aux différentes structures ; on a donc choisi de les différencier par les lettres A, B, C etc... : Sortirsi1B sera donc la deuxième possibilité de sortir de l'Itérer1. Quant au code réduit à 3 caractères, au lieu d'utiliser SSI, nous proposons d'utiliser SSA, SSB, SSC etc...

Le codage des conditions composées soulève d'ailleurs un certain nombre de problèmes auxquels on peut apporter des solutions normalisées.

5.2 Le codage des conditions composées.

Une condition composée peut faire intervenir l'opérateur ET ou l'opérateur OU, et on peut la trouver dans un Si ou dans un Sortirsi. En tout, cela fait quatre cas différents, mais l'un d'eux a été traité ci-dessus ; il reste à traiter les trois autres. On traitera successivement le Si avec un ET, le Si avec un OU enfin le Sortirsi avec un ET.

Une structure Si avec une condition dans laquelle apparaît un ET a déjà été codée au paragraphe 2 dans l'exposé d'un exemple où il fallait afficher uniquement les chiffres. Il suffit de coder deux Si imbriqués ; pour bien fixer les idées, soit à coder :

```
Si1 le registre X contient 0 et la mémoire $900 également 0
Alors1  . exécuter T1
Sinon1  . exécuter T2
Finsi1
```

on obtient :

```
Si1 le registre X contient 0
Alors1  . Si2 la mémoire $900 contient 0
          Alors2 . exécuter T1
          Sinon2 . exécuter T2
          Finsi2
Sinon1  . exécuter T2
Finsi1
```

d'où le code :

```
SI1       :
          CPX    #0
          BNE    SINON1
ALORS1    NOP
SI2       :
          LDA    $900
          CMP    #0
          BNE    SINON2
ALORS2    JSR    T1
          JMP    FINSI2
SINON2    JSR    T2
FINSI2    :
          JMP    FINSI1
SINON1    JSR    T2
FINSI1    :
```

On peut simplifier ce code et le rendre plus lisible en utilisant une nouvelle étiquette ETSI :

```
SI1       :
          CPX    #0
          BNE    SINON1
ETSI      LDA    $900
          CMP    #0
          BNE    SINON1
ALORS1    JSR    T1
          JMP    FINSI1
SINON1    JSR    T2
FINSI1    :
```

Pour être tout-à-fait cohérent au niveau des étiquettes, le ETSI devrait en fait être ETSI1 pour bien montrer son lien avec le Si1. Par ailleurs, il faut une étiquette normalisée à trois caractères. Nous avons choisi d'utiliser ESI dans nos programmes. Le cas d'une condition dans laquelle apparaît plusieurs ET sera traité dans un des exemples du chapitre 4.

Pour en finir avec le Si, il nous reste le Si avec un OU. On va exposer la méthode sur l'exemple suivant :

Si1 le registre X ou la mémoire $900 contient 0
Alors1 . exécuter T1
Sinon1 . exécuter T2
Finsi1

Par analogie avec ce qui précéde, on va utiliser une étiquette OUSI, en fait une étiquette OUSI1 :

```
SI1         :
            CPX     #0
            BEQ     ALORS1
OUSI1       LDA     $900
            CMP     #0
            BNE     SINON1
ALORS1      JSR     T1
            JMP     FINSI1
SINON1      JSR     T2
FINSI1      :
```

Nous avons choisi d'utiliser l'étiquette normalisée OSI dans nos programmes.

Le cas du Sortirsi avec un ET nécessite une étape de préparation du test de sortie : on positionne un indicateur appelé EXIT ; la sortie de la boucle dépend ensuite de la valeur de cet indicateur. Pour préciser les choses, soit le pseudo-code :

Itérer1
. exécuter T1
Sortirsi1 le registre X contient 0 et la mémoire $900 contient 0
. exécuter T2
Finitérer1

Il faut commencer par détailler la préparation du test de sortie dans le pseudo-code :
. positionner EXIT à FAUX
. Itérer1
. exécuter T1
. Si1 le registre X contient 0 et la mémoire $900 contient 0
Alors1 . positionner EXIT à VRAI
Finsi1
Sortirsi1 EXIT est VRAI
. exécuter T2
Finitérer1

Le codage ne pose pas de difficulté :

```
            LDA     # FAUX
            STA     EXIT
ITERER1     :
            JSR     T1
SI1         :
            CPX     #0
            BNE     FINSI1
ETSI1       LDA     $900
            CMP     #0
            BNE     FINSI1
ALORS1      LDA     # VRAI
            STA     EXIT
FINSI1
```

```
SORTIRSI1   LDA     EXIT
            CMP     # VRAI
            BEQ     FINITERER1
            JSR     T2
            JMP     ITERER1
FINITERER1  :
```

Pour en terminer avec le codage des conditions multiples, on peut conclure que ce n'est pas simple. C'est vrai, mais il n'est intrinsèquement pas simple de manipuler des conditions multiples, et de plus la manipulation implicite du théorème de DE MORGAN n'est pas quelque chose de facile.

Le mérite de notre méthode est de proposer une norme, ce qui est d'autant plus utile que les choses sont compliquées.

5.3 Le codage de la structure Cas où.

Nous avons vu qu'il existait deux types de structure Cas où, selon qu'on examine les différentes valeurs d'un sélecteur ou qu'on évalue successivement plusieurs expressions logiques. En assembleur, on pourra coder aussi facilement l'un que l'autre.

En ce qui concerne le choix des étiquettes, on a choisi une solution analogue à celle adoptée pour les sorties multiples de l'Itérer : CSA, CSB, CSC etc..., les mots réservés Cas où, Autres cas et Fincas étant codés CAS, ACS et FCA. Les étiquettes CSA, CAS, ACS ne sont pas visuellement très différentes, mais avec trois lettres, il est difficile de faire mieux.

A titre d'exemple, supposons que l'on ait à coder le bout de pseudo-code suivant correspondant au deuxième Cas où d'un module MN (menu).

```
Cas où2 caractère lu
    A :. afficher le texte
    S :. supprimer une ligne
    I :. insérer une ligne
    Autres cas2 :. afficher un message d'erreur
Fincas2
```

Si le caractère lu se trouve dans une variable appelée COM, et si les modules correspondant aux quatre actions s'appellent AFFICHER (AF), SUPPRIMER (SU), INSERER (IN) et PREVENIR (PR), on obtient le code suivant :

```
CAS2        :
            LDA     COM
CSA2        CMP     "A
            BNE     CSB2
            JSR     AFFICHER
            JMP     FCA2
CSB2        CMP     "S
            BNE     CSC2
            JSR     SUPPRIMER
            JMP     FCA2
```

```
CSC2        CMP     "I
            BNE     ACS2
            JSR     INSERER
            JMP     FCA2
ACS2        JSR     PREVENIR
FCA2        :
```

On peut faire un certain nombre de remarques au sujet de ce codage.

1 - Il ne faudrait pas remplacer la suite d'instructions :

```
            BNE     CSB2
            JSR     AFFICHER
            JMP     FCA2
      CSB2  ...     ...
```

par les seules instructions

```
            BEQ     AFFICHER
      CSB2  ...     ...
```

car cela aurait bien évidemment pour effet de ne plus considérer le module AFFICHER comme un sous-programme et après son exécution, on ne reviendrait pas au Fincas.

2 - Nous n'avons pas jugé utile de répéter l'instruction LDA COM en tête de chaque cas, ce qui est bien conforme au pseudo-code qui met en jeu un sélecteur : en assembleur, on place le sélecteur dans l'accumulateur et il y reste jusqu'à ce qu'on ait trouvé une comparaison satisfaisante.

3 - Supposons que le pseudo-code ne comporte pas d'Autres cas, il faudrait supprimer l'instruction AUTRESCAS2 JSR PREVENIR, et on se trouverait en présence d'une instruction JMP à une étiquette qui est celle de l'instruction suivante, ceci pourrait paraître lourd et on pourrait être tenté de supprimer cette dernière instruction JMP FINCAS2. Pour des raisons de programmation automatique et défensive, il ne faudrait surtout pas supprimer cette instruction JMP FINCAS2. Programmation automatique en ce sens que les différents cas doivent alors être programmés exactement par la même suite d'instructions : CMP, BNE, JSR et JMP, ce qui donne un code plus simple à vérifier. Programmation défensive en ce sens que si l'on est conduit ultérieurement à ajouter un nouveau cas après le dernier, on ne risque pas d'oublier le JMP FINCAS qu'il faudrait rajouter puisqu'il y sera déjà.

5.4 Référence à des étiquettes lointaines.

Dans le 6502, les instructions de branchement ont comme partie opérande un déplacement par rapport à l'adresse courante, et non pas l'adresse effective à laquelle on veut se brancher. Le déplacement étant codé sur un octet, et pouvant être positif ou négatif, sa valeur absolue maximale est de l'ordre de 127 (pour être plus précis, ce déplacement est calculé par rapport à l'adresse de l'instruction suivante et il est donc limité à un nombre d'octets compris entre -126 et +129), et il arrivera parfois, lors de l'assemblage, qu'une tentative de branchement à une adresse trop éloignée provoque une erreur "BRANCH TOO LONG" : ce sera le cas par exemple de l'instruction B.. FINSI qui précède l'instruction SINON lorsque le traitement correspondant au Sinon est volumineux. Dans ce cas on peut appliquer la technique suivante qui consiste à utiliser

un branchement inconditionnel à l'adresse effective à laquelle on veut se brancher.

Version initiale non assemblable :

```
            BCC     FINSI2
SINON2      ...
            ...
            ...
            ...
            ...
FINSI2      :
```

Version modifiée :

```
            BCS     SINON2
            JMP     FINSI2
SINON2      ...
            ...
            ...
            ...
            ...
FINSI2      :
```

Une situation analogue peut se rencontrer avec l'instruction B.. FINITERER ; dans ce cas il faut utiliser une instruction qui servira de relais. Voyons d'abord le principe de fonctionnement de ce relais, nous verrons ensuite la règle concernant le choix des étiquettes.

Version initiale non assemblable :

```
            BCC     FINITERER1
            ...
            ...
            ...
            ...
            ...
FINITERER1  :
```

Version modifiée :

```
            BCS     RELAIS1
            JMP     FINITERER1
RELAIS1     ...
            ...
            ...
            ...
            ...
FINITERER1  :
```

Quelles sont les étiquettes normalisées que nous proposons ? Dans la mesure où un relais ne peut être que dans un Itérer, et dans la mesure où, exceptionnellement, plusieurs relais peuvent apparaître dans le même module, nous proposons la norme suivante :

REL3 Relais du Itérer3

Conclusion

La méthode est exposée, son principe est compris, les détails de mise en œuvre également, il ne reste plus qu'à passer aux applications pratiques : c'est le sujet des deux chapitres suivants. Le chapitre 4 expose une dizaine de petites applications, tandis que le chapitre 5 montre une application plus conséquente, qui soulève en plus le problème des échanges d'information entre modules que nous n'avons pas abordé dans ce chapitre.

EXEMPLES SIMPLES

4

Le but de ce chapitre est d'appliquer sur quelques exemples la méthode exposée dans le chapitre précédent et d'en préciser certains points.

Introduction.

On va voir successivement dix exemples. La lecture de ces exemples permet de se familiariser avec la méthode que nous venons de proposer. On suppose donc que le lecteur va lire en détail les solutions que nous proposons et non pas tenter de les retrouver de lui-même. Par contre, à la fin de certains exemples, nous avons suggéré quelques propositions de travaux personnels : ce sont en général des modifications à l'exemple qui vient d'être traité, qui peuvent nécessiter un travail d'un quart d'heure à une demi-journée.

Chaque exemple commence par l'énoncé précis du traitement à programmer ; on trouve ensuite quelques mots sur la méthode de résolution qui va être programmée et parfois certaines informations techniques. Ensuite, on trouve le pseudo-code et la table des identificateurs. Rappelons à ce sujet que ces deux documents doivent être développés en parallèle grâce à des affinements successifs : c'est ce que l'on appelle l'analyse descendante. Les premiers exemples sont tellement simples qu'ils ne nécessitent pas d' "affinements". Par contre, les suivants sont déjà plus compliqués et nous avons essayé de montrer au maximum l'application de cette technique d'analyse descendante ; nous avons pris les conventions suivantes :
- à chaque niveau de développement du pseudo-code, on écrit en caractères gras et italiques les actions qui seront développées au niveau ultérieur, par exemple :

```
. Itérer1
      . lire un caractère
  Sortirsi1 on vient de lire un zéro
      . l'afficher si c'est un chiffre
  Finitérer1
```

- on développe ensuite séparément les parties à développer en les encadrant de deux lignes qui reprennent en caractères gras entre parenthèses le texte de l'action développée, par exemple :

```
(début de l'afficher si c'est un chiffre)
. Si1 le code du caractère lu est >= $B0
 Alors1 . Si2 le code du caractère lu est <=$B9
          Alors2 . afficher le caractère lu
          Finsi2
 Finsi1
(fin de l'afficher si c'est un chiffre)
```

- lorsqu'on a terminé l'analyse descendante, on donne une version finale du pseudo-code en remplaçant les parties à développer par leur développement, par exemple :

```
. Itérer1
    . lire un caractère
 Sortirsi1 on vient de lire un zéro
    (début de l'afficher si c'est un chiffre)
    . Si1 le code du caractère lu est >= $B0
     Alors1 . Si2 le code du caractère lu est <=$B9
              Alors2 . afficher le caractère lu
              Finsi2
     Finsi1
    (fin de l'afficher si c'est un chiffre)
  Finitérer1
```

- par contre, afin d'alléger la présentation, on donne une seule version de la table des identificateurs ; répétons encore une fois que cette table est en pratique développée parallèlement au pseudo-code : chaque fois qu'on introduit un objet nouveau dans le pseudo-code, on le décrit immédiatement dans la table des identificateurs ; ici, on a placé la table complète avant le développement du pseudo-code, afin que le lecteur puisse s'y référer lorsqu'il lit le pseudo-code ; dans cette table, les objets sont rangés selon leur ordre d'apparition dans le pseudo-code.

Pour chaque exemple on trouve ensuite la traduction du pseudo-code en pseudo-assembleur (mots réservés non codifiés et pas de déclaration des variables) et finalement une version de ce programme en assembleur Lisa. Le code a été écrit dans un esprit de programmation défensive ; par exemple, si une variable est chargée dans l'accumulateur et qu'on doive la comparer à une constante dix instructions plus loin, on charge l'accumulateur avant de faire la comparaison même si l'accumulateur n'a pas été modifié : cela en prévision d'une éventuelle modification ultérieure. L'optimisation de ce code permettrait, si le besoin s'en faisait sentir, de gagner du temps et de la place.

Exercice 1

1.1 Enoncé.

Un tableau appelé TORG comprenant au plus 256 octets est implanté en mémoire à partir de l'adresse $900 ; le dernier octet de ce tableau a pour valeur 0. Recopier ce tableau dans un tableau TDES implanté à partir de l'adresse $A00.

1.2 Méthode de résolution.

Les éléments du tableau origine seront recopiés l'un après l'autre jusqu'au dernier.

1.3 Informations techniques.

Le format des registres d'index est de un octet : par conséquent l'adressage absolu indexé permet de 'balayer' tous les éléments d'un tableau d'au plus 256 octets.

1.4 Pseudo-code et table des identificateurs.

Id.	Signification	Type	Nb oct.	VI	VF	Vérif.
TORG	tableau à recopier	tableau	<= 256	?	?	
TDES	tableau recopié	tableau	<= 256	?	?	
X	indice de l'élément traité	index	1	0	?	

pseudo-code

```
. initialiser X, indice de l'élément à recopier, à 0
. Itérer1
        . recopier l'élément d'indice X
  Sortirsi1 l'élément recopié est le dernier
        . incrémenter X
  Finitérer1
```

1.5 Traduction du pseudo-code.

```
DEBUT       :
            LDX     #0
ITERER1     :
            LDA     TORG,X
            STA     TDES,X
SORTIRSI1   LDA     TORG,X
            CMP     #0
            BEQ     FINITERER1
            INX
            JMP     ITERER1
FINITERER1  :
            BRK
FIN         :
```

1.6 Programme d'essai en LISA sur APPLE IIe.

```
DEB :
            LDX     #0
ITR1 :
            LDA     TORG,X
            STA     TDES,X
SSI1        LDA     TORG,X
            CMP     #0
            BEQ     FIT1
            INX
            JMP     ITR1
FIT1 :
            BRK
TORG        EQU     $900
TDES        EQU     $A00
FIN :
            END
```

1.7 Proposition de travaux personnels.

a) afficher le nombre d'éléments recopiés.

b) ne pas recopier le dernier élément nul.

c) bien que l'énoncé nous assure que le tableau origine contient au moins un élément nul il serait prudent de vérifier que ce programme ne boucle pas.

Exercice 2

2.1 Enoncé.

Lire 4 chiffres au clavier et afficher le plus petit sur la ligne suivante. On suppose qu'il n'y a pas d'erreur de frappe et que seuls des chiffres sont entrés.

2.2 Informations techniques.

On dispose de trois sous-programmes :
- RDKEY (read key) : lecture au clavier d'un caractère ; on retrouve son code ASCII étendu dans l'accumulateur
- COUT (character out) : affichage du caractère dont le code ASCII étendu se trouve dans l'accumulateur
- CROUT (carriage return out) : passage à la ligne suivante.

Le code ASCII étendu des chiffres va de \$B0 à \$B9 ; on peut donc se contenter de comparer les codes ASCII plutôt que les valeurs puisque les ordres sont les mêmes.

2.3 Pseudo-code et table des identificateurs. (version 1)

Id.	**Signification**	**Type**	**Nb oct.**	**VI**	**VF**	**Vérif.**
K	nombre de chiffres restant à lire	Naturel	1	4	0	I C U
MIN	code ASCII du plus petit chiffre déjà traité	Caractère	1	?	?	

pseudo-code

```
. initialiser K, nombre de chiffres restant à lire, à 4
. lire et afficher le premier chiffre et le recopier dans MIN
. décrémenter K
. Itérer1
    . lire et afficher un chiffre
    . décrémenter K
    . Si1  le chiffre lu est strictement inférieur à MIN
      Alors1
        . le recopier dans MIN
      Finsi1
  Sortirsi1  le dernier chiffre est traité (K = 0)
  Finitérer1
. aller à la ligne et afficher MIN
```

2.4 Traduction du pseudo-code. (version 1)

```
DEBUT      :
           LDA        #4
           STA        K
           JSR        LIRE UN CARACTERE AU CLAVIER
           JSR        AFFICHER UN CARACTERE
           STA        MIN
           DEC        K
ITERER1    :
           JSR        LIRE UN CARACTERE AU CLAVIER
           JSR        AFFICHER UN CARACTERE
           DEC        K
SI1        :
           CMP        MIN
           BCS        FINSI1
ALORS1     STA        MIN
FINSI1     :
SORTIRSI1  LDA        K
           CMP        #0
           BEQ        FINITERER1
           JMP        ITERER1
FINITERER1 :
           JSR        ALLER A LA LIGNE
           LDA        MIN
           JSR        AFFICHER UN CARACTERE
           BRK
FIN        :
```

Remarque : le test Si1 compare le caractère lu au MIN déjà trouvé ; mais l'instruction DEC K se trouve entre l'instruction qui place le caractère lu dans l'accumulateur (JSR LIRE UN CARACTERE AU CLAVIER) et celle qui effectue la comparaison (CMP MIN) ; or un bon principe de programmation défensive veut que ces deux instructions soient indissociées afin de ne pas courir le risque de modifier involontairement l'accumulateur. Il est donc souhaitable de placer la décrémentation de K après la structure Si1 ; c'est possible, mais il faut alors modifier le pseudo-code en conséquence ainsi que la table des identificateurs car en toute rigueur la variable K devient le nombre de chiffres restant à **traiter**. Cette remarque nous conduit à proposer une deuxième version dans laquelle les modifications sont portées en caractères gras.

2.5 Pseudo-code et table des identificateurs. (version 2)

Id.	Signification	Type	Nb oct.	VI	VF	Vérif.
K	nombre de chiffres restant à **traiter**	Naturel	1	4	0	I C U
MIN	code ASCII du plus petit chiffre déjà traité	Caractère	1	?	?	

pseudo-code

. initialiser K, nombre de chiffres restant à **traiter,** à 4
. lire et afficher le premier chiffre et le recopier dans MIN
. décrémenter K
. Itérer1
  . lire et afficher un chiffre
  . Si1 le chiffre lu est strictement inférieur à MIN
    Alors1
      . le recopier dans MIN
    Finsi1
  . **décrémenter K**
 Sortirsi1 le dernier chiffre est traité (K = 0)
 Finitérer1
. aller à la ligne et afficher MIN

2.6 Traduction du pseudo-code. (version 2)

```
DEBUT       :
            LDA     #4
            STA     K
            JSR     LIRE UN CARACTERE AU CLAVIER
            JSR     AFFICHER UN CARACTERE
            STA     MIN
            DEC     K
ITERER1     :
            JSR     LIRE UN CARACTERE AU CLAVIER
            JSR     AFFICHER UN CARACTERE
SI1         :
            CMP     MIN
            BCS     FINSI1
ALORS1      STA     MIN
FINSI1      :
            DEC     K
SORTIRSI1   LDA     K
            CMP     #0
            BEQ     FINITERER1
            JMP     ITERER1
FINITERER1  :
            JSR     ALLER A LA LIGNE
            LDA     MIN
            JSR     AFFICHER UN CARACTERE
            BRK
FIN         :
```

2.7 Programme d'essai en LISA sur APPLE IIe.

```
DEB :
        LDA     #4
        STA     K
        JSR     RDKEY
        JSR     COUT
        STA     MIN
        DEC     K
ITR1 :
        JSR     RDKEY
        JSR     COUT
SII1 :
        CMP     MIN
        BCS     FSI1
ALO1    STA     MIN
FSI1 :
        DEC     K
SSI1    LDA     K
        CMP     #0
        BEQ     FIT1
        JMP     ITR1
FIT1 :
        JSR     CROUT
        LDA     MIN
        JSR     COUT
        BRK
K       DFS     1
MIN     DFS     1
RDKEY   EQU     $FD0C
COUT    EQU     $FDED
CROUT   EQU     $FD8E
FIN :
        END
```

2.8 Proposition de travaux personnels.

a) tester si les caractères lus sont des chiffres et ne prendre en compte que les chiffres. Ne pas oublier de préciser la nouvelle signification de K et d'en tirer les conséquences au niveau de sa décrémentation.

b) calculer la SOMME des quatre chiffres lus et afficher sa valeur en hexadécimal. Pour afficher en hexadécimal le contenu de l'accumulateur on dispose du sous-programme PRBYTE.

Exercice 3

3.1 Enoncé.

Lire 9 caractères au clavier, les recopier en écho sur l'écran, compter le nombre de 0 (zéro) et de 1 (un) et afficher ces résultats sur les deux lignes suivantes.

3.2 Informations techniques.

On dispose d'un sous-programme PRBYTE (print byte) qui affiche la valeur hexadécimale de l'accumulateur.

3.3 Pseudo-code et table des identificateurs.

Id.	Signification	Type	Nb oct.	VI	VF	Vérif.
CPT0	nombre de '0' lus	naturel	1	0	?	I C U
CPT1	nombre de '1' lus	naturel	1	0	?	I C U
K	nombre de caractères restant à traiter	naturel	1	9	0	I C U

pseudo-code

```
. initialiser les compteurs CPT0 et CPT1 à 0
. initialiser K, nombre de caractères restant à traiter, à 9
. Itérer1
    . lire un caractère et le recopier à l'écran
    . Si1 le caractère lu est '0'
      Alors1
        . incrémenter CPT0
      Finsi1
    . Si2  le caractère lu est '1'
      Alors2
        . incrémenter CPT1
      Finsi2
    . décrémenter K
  Sortirsi1  le dernier caractère est traité (K = 0)
  Finitérer1
. aller à la ligne et afficher CPT0
. aller à la ligne et afficher CPT1
```

Remarque : la présentation avec deux alternatives successives est plus simple que celle avec deux alternatives imbriquées ; le code obtenu est bien sûr moins efficace.

3.4 Traduction du pseudo-code.

```
DEBUT       :
            LDA     #0
            STA     CPT0
            STA     CPT1
            LDA     #9
            STA     K
ITERER1     :
            JSR     LIRE UN CARACTERE AU CLAVIER
            JSR     AFFICHER UN CARACTERE
SI1         :
            CMP     "0
            BNE     FINSI1
ALORS1      INC     CPT0
FINSI1      :
SI2         :
            CMP     "1
            BNE     FINSI2
ALORS2      INC     CPT1
FINSI2      :
            DEC     K
SORTIRSI1   LDA     K
            CMP     #0
            BEQ     FINITERER1
            JMP     ITERER1
FINITERER1  :
            JSR     ALLER A LA LIGNE
            LDA     CPT0
            JSR     AFFICHER UN NOMBRE
            JSR     ALLER A LA LIGNE
            LDA     CPT1
            JSR     AFFICHER UN NOMBRE
            BRK
FIN         :
```

3.5 Programme d'essai en LISA sur APPLE IIe.

```
DEB :
            LDA     #0
            STA     CPT0
            STA     CPT1
            LDA     #9
            STA     K
ITR1 :
            JSR     RDKEY
            JSR     COUT
SII1 :
            CMP     "0
            BNE     FSI1
ALO1        INC     CPT0
FSI1 :
```

```
SII2 :
        CMP     "1
        BNE     FSI2
ALO2    INC     CPT1
FSI2 :
        DEC     K
SSI1    LDA     K
        CMP     #0
        BEQ     FIT1
        JMP     ITR1
FIT1 :
        JSR     CROUT
        LDA     CPT0
        JSR     PRBYTE
        JSR     CROUT
        LDA     CPT1
        JSR     PRBYTE
        BRK
CPT0    DFS     1
CPT1    DFS     1
K       DFS     1
PRBYTE  EQU     $FDDA
CROUT   EQU     $FD8E
COUT    EQU     $FDED
RDKEY   EQU     $FD0C
FIN :
        END
```

3.6 Proposition de travail personnel.

Donner également le nombre de caractères lus différents de 0 ou de 1 .

Exercice 4

4.1 Enoncé.

Lire un nombre décimal de un ou deux chiffres le recopier en écho à l'écran et le convertir en binaire. Si l'opérateur entre des caractères autres que des chiffres, il faudra les ignorer ; si l'opérateur entre plus de deux chiffres, il faut ne tenir compte que des deux premiers ; si l'opérateur n'entre aucun chiffre, afficher le caractère E. On peut illustrer ces règles à l'aide des exemples ci-après dans lesquels le retour chariot a été représenté par le symbole ®.

Suite de caractères saisis	Nombre retenu	Nombre binaire (NB)
2®	2	0000 0010
43®	43	0010 1011
5678®	56	0011 1000
A3BC2U®	32	0010 0000
D®	aucun, afficher le caractère E (erreur)	
®	aucun, afficher le caractère E (erreur)	

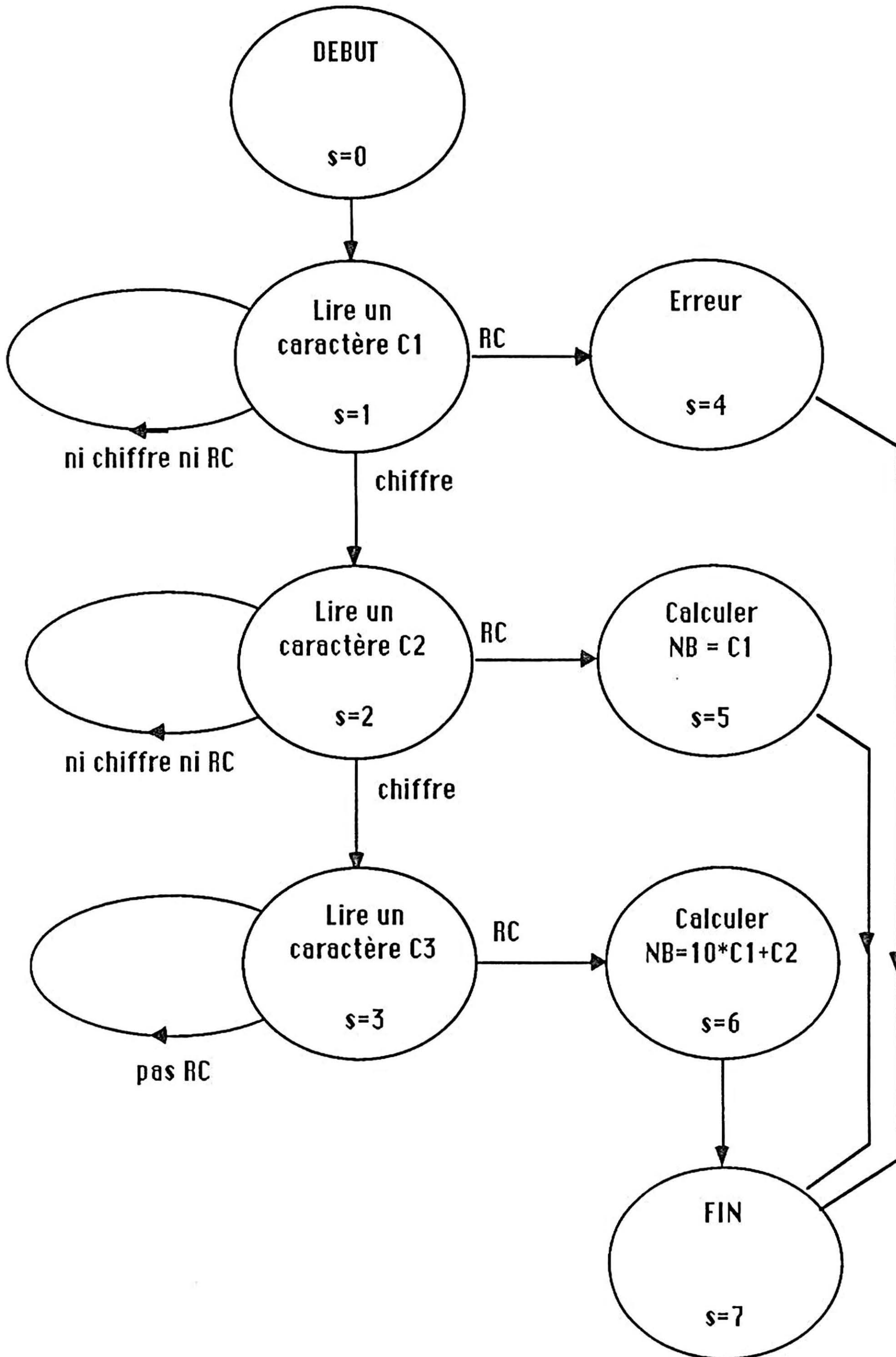
Remarquons qu'un nombre décimal de deux chiffres peut être représenté en binaire dans un seul octet.

4.2 Informations techniques.

Dans les opérations d'entrée et de sortie les chiffres sont considérés comme des caractères ; on les manipule donc par l'intermédiaire de leur code ASCII.

4.3 Méthode de résolution.

On peut traduire les règles de lecture sous la forme de l'automate de la page suivante, dont la traduction en pseudo-code est donnée ensuite.


DEBUT
s=0
Lire un
caractère C1
s=1
RC
Erreur
s=4
ni chiffre ni RC
chiffre
Lire un
caractère C2
s=2
RC
Calculer
NB = C1
s=5
ni chiffre ni RC
chiffre
Lire un
caractère C3
s=3
RC
Calculer
NB=10*C1+C2
s=6
pas RC
FIN
s=7

4.4 Pseudo-code et table des identificateurs.

Id.	Signification	Type	Nb oct.	VI	VF	Vérif.
constante						
RC	retour chariot	caractère	1	$8D	$8D	
variables						
NB	valeur binaire du nombre lu	naturel	1	?	?	
S	numéro de l'état	naturel	1	1	7	
C1	caractère lu dans l'état 1 ou 1er chiffre lu	caractère puis naturel	1	?	?	
C2	caractère lu dans l'état 2 ou 2ième chiffre lu	caractère puis naturel	1	?	?	
C3	caractère lu dans l'état 3	caractère	1	?	?	

niveau1

```
. initialiser S, numéro de l'état, à 1
. Itérer1
   . Cas où  S
      1 : . lire le caractère C1
          . Si1  C1 est un retour chariot
            Alors1   . affecter à S la valeur 4
            Sinon1
               . Si2  le caractère lu est un chiffre
                 Alors2
                    . afficher le chiffre lu
                    . affecter à S la valeur 2
                 Sinon2   . affecter à S la valeur 1
                 Finsi2
            Finsi1
```

```
2 : . lire le caractère C2
    . Si3  C2 est un retour chariot
      Alors3  . affecter à S la valeur 5
      Sinon3
           . Si4  le caractère lu est un chiffre
             Alors4
               . afficher le chiffre lu
               . affecter à S la valeur 3
             Sinon4  . affecter à S la valeur 2
             Finsi4
      Finsi3
3 : . lire le caractère C3
    . Si5  C3 est un retour chariot
      Alors5  . affecter à S la valeur 6
      Sinon5  . affecter à S la valeur 3
      Finsi5
4 : . afficher le caractère E symbolisant l'erreur
    . affecter à S la valeur 7
5 : . convertir le chiffre C1 en nombre naturel
    . affecter à NB la valeur binaire de C1
    . affecter à S la valeur 7
6 : . convertir le chiffre C2 en nombre naturel
    . convertir le chiffre C1 en nombre naturel
    . affecter à NB la valeur binaire de l'expression 10*C1+C2
    . affecter à S la valeur 7
Fincas1
Sortirsi1  S est l'état final (S =7)
Finitérer1
```

niveau2 (version1)

(début de le caractère lu est un chiffre)
le caractère lu est supérieur à '0' et inférieur à '9'
(fin de le caractère lu est un chiffre)

Remarquons que la codification de la condition 'strictement inférieur' est plus simple en 6502 que celle de 'inférieur ou égal', ceci nous conduit à proposer une nouvelle version.

niveau2 (version2)

(début de le caractère lu est un chiffre)
le code ASCII du caractère lu est supérieur à $B0 et strictement inférieur à $BA
(fin de le caractère lu est un chiffre)

pseudo-code

```
. initialiser S, numéro de l'état, à 1
. Itérer1
    . Cas où  S
        1 : . lire le caractère C1
            . Si1  C1 est un retour chariot
              Alors1   . affecter à S la valeur 4
              Sinon1
                . Si2  (début de le caractère lu est un chiffre)
                        le code ASCII du caractère lu est supérieur ou égal à $B0 et
                        strictement inférieur à $BA
                       (fin de le caractère lu est un chiffre)
                  Alors2
                    . afficher le chiffre lu
                    . affecter à S la valeur 2
                  Sinon2  . affecter à S la valeur 1
                  Finsi2
              Finsi1
        2 : . lire le caractère C2
            . Si3  C2 est un retour chariot
              Alors3   . affecter à S la valeur 5
              Sinon3
                . Si4  (début de le caractère lu est un chiffre)
                        le code ASCII du caractère lu est supérieur ou égal à $B0 et
                        strictement inférieur à $BA
                       (fin de le caractère lu est un chiffre)
                  Alors4
                    . afficher le chiffre lu
                    . affecter à S la valeur 3
                  Sinon4  . affecter à S la valeur 2
                  Finsi4
              Finsi3
        3 : . lire le caractère C3
            . Si5  C3 est un retour chariot
              Alors5   . affecter à S la valeur 6
              Sinon5   . affecter à S la valeur 3
              Finsi5
        4 : . afficher le caractère E symbolisant l'erreur
            . affecter à S la valeur 7
        5 : . convertir le chiffre C1 en nombre naturel
            . affecter à NB la valeur binaire de C1
            . affecter à S la valeur 7
        6 : . convertir le chiffre C2 en nombre naturel
            . convertir le chiffre C1 en nombre naturel
            . affecter à NB la valeur binaire de l'expression 10*C1+C2
            . affecter à S la valeur 7
      Fincas1
  Sortirsi1  on est dans l'état final (S =7)
  Finitérer1
```

4.5 Traduction du pseudo-code.

```
DEBUT    :
         LDA     #1
         STA     S
ITERER1  :
CAS1     :
         LDA     S
CSA1     CMP     #1
         BNE     CSB1
         JSR     LIRE UN CARACTERE
         STA     C1
SI1      :
         LDA     C1
         CMP     # RC
         BNE     SINON1
ALORS1   LDA     #4
         STA     S
         JMP     FINSI1
SINON1   NOP
SI2      :
         LDA     C1
         CMP     #$B0
         BCC     SINON2
ETSI2    CMP     #$BA
         BCS     SINON2
ALORS2   LDA     C1
         JSR     AFFICHER UN CARACTERE
         LDA     #2
         STA     S
         JMP     FINSI2
SINON2   LDA     #1
         STA     S
FINSI2   :
FINSI1   :
         JMP     FINCAS1
CSB1     CMP     #2
         BNE     CSC1
         JSR     LIRE UN CARACTERE
         STA     C2
SI3      :
         LDA     C2
         CMP     # RC
         BNE     SINON3
ALORS3   LDA     #5
         STA     S
         JMP     FINSI3
SINON3   NOP
SI4      :
         LDA     C2
         CMP     #$B0
         BCC     SINON4
ETSI4    CMP     #$BA
```

```
        BCS    SINON4
ALORS4  LDA    C2
        JSR    AFFICHER UN CARACTERE
        LDA    #3
        STA    S
        JMP    FINSI4
SINON4  LDA    #2
        STA    S
FINSI4  :
FINSI3  :
        JMP    FINCAS1
CSC1    CMP    #3
        BNE    CSD1
        JSR    LIRE UN CARACTERE
        STA    C3
SI5     :
        LDA    C3
        CMP    #RC
        BNE    SINON5
ALORS5  LDA    #6
        STA    S
        JMP    FINSI5
SINON5  LDA    #3
        STA    S
FINSI5  :
        JMP    FINCAS1
CSD1    CMP    #4
        BNE    CSE1
        LDA    'E
        JSR    AFFICHER UN CARACTERE
        LDA    #7
        STA    S
        JMP    FINCAS1
CSE1    CMP    #5
        BNE    CSF1
        LDA    C1
        AND    #%00001111
        STA    NB
        LDA    #7
        STA    S
        JMP    FINCAS1
CSF1    CMP    #6
        BNE    FINCAS1
        LDA    C2
        AND    #%00001111
        STA    C2
        LDA    C1
        AND    #%00001111
        STA    C1
        ASL
        ASL
        ASL
        CLC
        ADC    C1
```

```
            ADC         C1
            ADC         C2
            STA         NB
            LDA         #7
            STA         S
            JMP         FINCAS1
FINCAS1     :
SORTIRSI1   LDA         S
            CMP         #7
            BEQ         FINITERER1
            JMP         ITERER1
FINITERER1  :
FIN         :
```

4.6 Programme d'essai en LISA sur APPLE IIe.

```
DEB :
            LDA         #1
            STA         S
ITR1 :
CAS1 :
            LDA         S
CSA1        CMP         #1
            BNE         CSB1
            JSR         RDKEY
            STA         C1
SII1 :
            LDA         C1
            CMP         #RC
            BNE         SNO1
ALO1        LDA         #4
            STA         S
            JMP         FSI1
SNO1        NOP
SII2 :
            LDA         C1
            CMP         #$B0
            BCC         SNO2
ESI2        CMP         #$BA
            BCS         SNO2
ALO2        LDA         C1
            JSR         COUT
            LDA         #2
            STA         S
            JMP         FSI2
SNO2        LDA         #1
            STA         S
FSI2 :
FSI1 :
            JMP         FCA1
```

```
CSB1      CMP       #2
          BNE       CSC1
          JSR       RDKEY
          STA       C2
SII3 :
          LDA       C2
          CMP       #RC
          BNE       SNO3
ALO3      LDA       #5
          STA       S
          JMP       FSI3
SNO3      NOP
SII4 :
          LDA       C2
          CMP       #$B0
          BCC       SNO4
ESI4      CMP       #$BA
          BCS       SNO4
ALO4      LDA       C2
          JSR       COUT
          LDA       #3
          STA       S
          JMP       FSI4
SNO4      LDA       #2
          STA       S
FSI4 :
FSI3 :
          JMP       FCA1
CSC1      CMP       #3
          BNE       CSD1
          JSR       RDKEY
          STA       C3
SII5 :
          LDA       C3
          CMP       #RC
          BNE       SNO5
ALO5      LDA       #6
          STA       S
          JMP       FSI5
SNO5      LDA       #3
          STA       S
FSI5 :
          JMP       FCA1
CSD1      CMP       #4
          BNE       CSE1
          LDA       "E
          JSR       COUT
          LDA       #7
          STA       S
          JMP       FCA1
```

```
CSE1    CMP     #5
        BNE     CSF1
        LDA     C1
        AND     #%00001111
        STA     NB
        LDA     #7
        STA     S
        JMP     FCA1
CSF1    CMP     #6
        BNE     FCA1
        LDA     C2
        AND     #%00001111
        STA     C2
        LDA     C1
        AND     #%00001111
        STA     C1
        ASL
        ASL
        ASL
        CLC
        ADC     C1
        ADC     C1
        ADC     C2
        STA     NB
        LDA     #7
        STA     S
        JMP     FCA1
FCA1 :
SSI1    LDA     S
        CMP     #7
        BEQ     FIT1
        JMP     ITR1
FIT1 :
        BRK
RC      EQU     $8D
NB      DFS     1
S       DFS     1
C1      DFS     1
C2      DFS     1
C3      DFS     1
RDKEY   EQU     $FD0C
COUT    EQU     $FDED
FIN :
        END
```

4.7 Proposition de travail personnel.

Si l'opérateur entre plus de deux chiffres, tenir compte des deux derniers et non plus des deux premiers.

Exercice 5

5.1 Enoncé.

Lire trois nombres décimaux L1, L2, L3 d'au plus deux chiffres. Indiquer si ces trois nombres peuvent représenter les longueurs des cotés d'un triangle ; c'est-à-dire vérifier la condition : L2+L3 >= L1 et L1+L2 >= L3 et L3+L1 >= L2.

5.2 Méthode de résolution.

Les nombres sont lus par le module 'Lire un nombre Décimal' décrit dans l'exercice 4, à la sortie duquel la valeur du nombre lu se trouve dans l'accumulateur. La seule difficulté consiste à évaluer le prédicat composé L1 <= L2+L3 et L3 <= L1+L2 et L2 <= L3+L1 ; on suppose a priori que le résultat de l'évaluation est FAUX, puis on évalue en cascade les trois prédicats L1 <= L2+L3, L3 <= L1+L2, L2 <= L3+L1 ; le résultat est VRAI si et seulement si ils sont vérifiés tous les trois.

5.3 Informations techniques.

Les instructions de comparaison du 6502 opèrent sur un seul octet, par conséquent les sommes L2+L3, L1+L2, L3+L1 doivent être strictement inférieures à 255, mais cette condition est garantie par l'hypothèse faite sur les nombres L1, L2, L3 qui sont inférieurs à $(99)_{10}$.

5.4 Pseudo-code et table des identificateurs.

Id.	Signification	Type	Nb oct	VI	VF	Vérif.
constantes						
FAUX		booléen	1	0	0	
VRAI		booléen	1	1	1	
variables						
L1	premier nombre lu	naturel <=99	1	?	?	
L2	deuxième nombre lu	naturel <=99	1	?	?	
L3	troisième nombre lu	naturel <=99	1	?	?	
TRIA	résultat de l'évaluation du prédicat L2+L3 >= L1 et L1+L2 >= L3 et L3+L1 >= L2	booléen	1	FAUX	?	I B U

pseudo-code

. lire les nombres L1, L2, L3
. initialiser TRIA, résultat de l'évaluation, à FAUX
. Si1 L2+L3 >= L1 et L1+L2 >= L3 et L3+L1 >= L2
Alors1
. recopier la valeur VRAI dans TRIA
Finsi1
. aller à la ligne et afficher TRIA

5.5 Traduction du pseudo-code.

```
DEBUT   :
        JSR     LIRE UN NOMBRE DECIMAL
        STA     L1
        JSR     LIRE UN NOMBRE DECIMAL
        STA     L2
        JSR     LIRE UN NOMBRE DECIMAL
        STA     L3
        LDA     #FAUX
        STA     TRIA
SI1     :
        LDA     L2
        CLC
        ADC     L3
        CMP     L1
        BCC     FINSI1
ETSIA1  LDA     L1
        CLC
        ADC     L2
        CMP     L3
        BCC     FINSI1
ETSIB1  LDA     L3
        CLC
        ADC     L1
        CMP     L2
        BCC     FINSI1
ALORS1  LDA     #VRAI
        STA     TRIA
FINSI1  :
        JSR     ALLER A LA LIGNE
        LDA     TRIA
        JSR     AFFICHER UN NOMBRE
        BRK
FIN     :
```

5.6 Programme d'essai en LISA sur APPLE IIe.

Le module LD est le programme décrit dans l'exercice 4. On verra au chapitre 5 que son étiquette de début est LDDEB. On suppose que ce module est sauvegardé dans le fichier 'LD'.

```
DEB :
          JSR     LDDEB
          STA     L1
          JSR     LDDEB
          STA     L2
          JSR     LDDEB
          STA     L3
          LDA     #FAUX
          STA     TRIA
SII1 :
          LDA     L2
          CLC
          ADC     L3
          CMP     L1
          BCC     FSI1
ESA1      LDA     L1
          CLC
          ADC     L2
          CMP     L3
          BCC     FSI1
ESB1      LDA     L3
          CLC
          ADC     L1
          CMP     L2
          BCC     FSI1
ALO1      LDA     #VRAI
          STA     TRIA
FSI1 :
          JSR     CROUT
          LDA     TRIA
          JSR     PRBYTE
          BRK
FAUX      EQU     1
VRAI      EQU     0
TRIA      DFS     1
L1        DFS     1
L2        DFS     1
L3        DFS     1
CROUT     EQU     $FD8E
PRBYTE    EQU     $FDDA
FIN :
          ICL     'LD'
```

5.7 Proposition de travail personnel.

Ne pas accepter les triangles plats.

Exercice 6

6.1 Enoncé.

Pour tous les nombres relatifs compris entre -128 et +127, afficher la table suivante : dans la colonne de gauche les valeurs hexadécimales de $00 à $FF et dans la colonne de droite les valeurs décimales correspondantes.

La valeur décimale est exprimée sous la forme d'un signe suivi de la valeur absolue sur trois chiffres ; le nombre nul a été choisi arbitrairement positif.

Exemple :	$00	+000
	---	----
	$61	+097
	---	----
	$8E	-114
	---	----
	$FF	-001

6.2 Méthode de résolution.

On produit successivement chaque ligne de cette table en partant de la valeur hexadécimale que l'on convertit en décimal. On examine en premier lieu son signe puis on calcule son complément à deux dans le cas d'un nombre négatif. (le complément à deux est obtenu en ajoutant 1 au complément logique).

Pour afficher la valeur absolue en décimal on détermine et on affiche successivement le chiffre des centaines, des dizaines et des unités.

6.3 Informations techniques.

On ne dispose pas de sous-programme permettant d'afficher un chiffre décimal : par conséquent on doit d'abord convertir ce chiffre en caractère puis faire appel au sous-programme COUT d'affichage d'un caractère.

6.4 Pseudo-code et table des identificateurs.

Id.	Signification	Type	Nb oct.	VI	VF	Vérif
NR	nombre relatif à convertir	relatif	1	$00	$FF	
VA	initialement la valeur absolue de NR finalement le nombre d'unités de NR	naturel	1	?	?	
ND	nombre de dizaines de NR	naturel	1	0	?	I U C

niveau 1

. initialiser NR, nombre à convertir, à $00
. Itérer1
. afficher la valeur hexadécimale de NR
. afficher un espace
. afficher le signe de NR et calculer VA, sa valeur absolue , en binaire
. afficher VA en décimal
Sortirsi1 le dernier nombre est converti (NR = $FF)
. incrémenter NR
. aller à la ligne
Finitérer1

niveau 2

(début de afficher le signe de NR et calculer VA, sa valeur absolue, en binaire)
. Si1 NR est un nombre positif
Alors1
. afficher le caractère '+'
. affecter la valeur de NR à VA
Sinon1
. afficher le caractère '-'
. affecter le "complément logique" de NR à VA
. ajouter 1 à VA
Finsi1
(fin de afficher le signe de NR et calculer VA, sa valeur absolue, en binaire)

(début de afficher VA en décimal)
. déterminer et afficher le chiffre des centaines de VA
. déterminer et afficher le chiffre des dizaines
. déterminer et afficher le chiffre des unités
(fin de afficher VA en décimal)

niveau 3

(début de déterminer et afficher le chiffre des centaines de VA)
. Si2 VA est supérieure ou égale à 100_{10}
Alors2
. afficher le caractère '1'
. retrancher 100_{10} à VA
Sinon2
. afficher le caractère '0'
Finsi2
(fin de déterminer et afficher le chiffre des centaines de VA)

(début de déterminer et afficher le chiffre des dizaines)
. initialiser ND, nombre de dizaines, à 0
. Itérer2
Sortirsi2 VA est strictement inférieure à 10_{10}
. incrémenter ND
. retrancher 10_{10} à VA
Finitérer2
. convertir ND en caractère
. afficher ce caractère
(fin de déterminer et afficher le chiffre des dizaines)

(début de déterminer et afficher le chiffre des unités)
. convertir VA, nombre d'unités, en caractère
. afficher ce caractère
(fin de déterminer et afficher le chiffre des unités)

pseudo-code

. initialiser NR, nombre à convertir, à $00
. Iterer1
. afficher la valeur hexadécimale de NR
. afficher un espace
(début de afficher le signe de NR et calculer VA, sa valeur absolue, en binaire)
. Si1 NR est un nombre positif
Alors1
. afficher le caractère '+'
. affecter la valeur de NR à VA
Sinon1
. afficher le caractère '-'
. affecter le "complément logique" de NR à VA
. ajouter 1 à VA
Finsi1
(fin de afficher le signe de NR et calculer VA, sa valeur absolue, en binaire)

(début de afficher VA en décimal)
(début de déterminer et afficher le chiffre des centaines de VA)
. Si2 VA est supérieure ou égale à 100_{10}
Alors2
. afficher le caractère '1'
. retrancher 100_{10} à VA
Sinon2
. afficher le caractère '0'
Finsi2
(fin de déterminer et afficher le chiffre des centaines de VA)
(début de déterminer et afficher le chiffre des dizaines)
. initialiser ND, nombre de dizaines à 0
. Itérer2
Sortirsi2 VA est strictement inférieure à 10_{10}
. incrémenter ND
. retrancher 10_{10} à VA
Finitérer2

. convertir ND en caractère
. afficher ce caractère
(fin de déterminer et afficher le chiffre des dizaines)
(début de déterminer et afficher le chiffre des unités)
. convertir VA, nombre d'unités, en caractère
. afficher ce caractère
(fin de déterminer et afficher le chiffre des unités)
(fin de afficher VA en décimal)
Sortirsi1 le dernier nombre est converti (NR = $FF)
. incrémenter NR
. aller à la ligne
Finitérer1

6.4 Traduction du pseudo-code.

```
DEBUT    :
         LDA   #$00
         STA   NR
ITERER1  :
         LDA   NR
         JSR   AFFICHER EN HEXADECIMAL
         LDA   "
         JSR   AFFICHER UN CARACTERE
SI1      :
         LDA   NR
         BMI   SINON1
ALORS1   LDA   "+
         JSR   AFFICHER UN CARACTERE
         LDA   NR
         STA   VA
         JMP   FINSI1
SINON1   LDA   "-
         JSR   AFFICHER UN CARACTERE
         LDA   NR
         EOR   #%11111111
         STA   VA
         INC   VA
FINSI1   :
SI2      :
         LDA   VA
         CMP   #100
         BCC   SINON2
ALORS2   LDA   "1
         JSR   AFFICHER UN CARACTERE
         LDA   VA
         SEC
         SBC   #100
         STA   VA
         JMP   FINSI2
SINON2   LDA   "0
         JSR   AFFICHER UN CARACTERE
FINSI2   :
```

```
                LDA         #0
                STA         ND
ITERER2         :
SORTIRSI2       LDA         VA
                CMP         #10
                BCC         FINITERER2
                INC         ND
                LDA         VA
                SEC
                SBC         #10
                STA         VA
                JMP         ITERER2
FINITERER2      :
                LDA         ND
                ORA         #$B0
                JSR         AFFICHER UN CARACTERE
                LDA         VA
                ORA         #$B0
                JSR         AFFICHER UN CARACTERE
SORTIRSI1       LDA         NR
                CMP         #$FF
                BEQ         FINITERER1
                INC         NR
                JSR         ALLER A LA LIGNE
                JMP         ITERER1
FINITERER1      :
                BRK
FIN             :
```

6.5 Programme d'essai en LISA sur APPLE IIe.

```
DEB :
                LDA         #$00
                STA         NR
ITR1 :
                LDA         NR
                JSR         PRBYTE
                LDA         "
                JSR         COUT
SII1 :
                LDA         NR
                BMI         SNO1
ALO1            LDA         "+
                JSR         COUT
                LDA         NR
                STA         VA
                JMP         FSI1
SNO1            LDA         "-
                JSR         COUT
                LDA         NR
                EOR         #%11111111
                STA         VA
                INC         VA
FSI1 :
```

```
SII2 :
            LDA         VA
            CMP         #100
            BCC         SNO2
ALO2        LDA         "1
            JSR         COUT
            LDA         VA
            SEC
            SBC         #100
            STA         VA
            JMP         FSI2
SNO2        LDA         "0
            JSR         COUT
FSI2 :
            LDA         #0
            STA         ND
ITR2 :
SSI2        LDA         VA
            CMP         #10
            BCC         FIT2
            INC         ND
            LDA         VA
            SEC
            SBC         #10
            STA         VA
            JMP         ITR2
FIT2 :
            LDA         ND
            ORA         #$B0
            JSR         COUT
            LDA         VA
            ORA         #$B0
            JSR         COUT
SSI1        LDA         NR
            CMP         #$FF
            BEQ         FIT1
            INC         NR
            JSR         CROUT
            JMP         IR1
FIT1 :
            BRK
NR          DFS         1
VA          DFS         1
ND          DFS         1
COUT        EQU         $FDED
PRBYTE      EQU         $FDDA
CROUT       EQU         $FD8E
FIN :
            END
```

6.6 Proposition de travail personnel.

Ne pas afficher les zéros des centaines et des dizaines non significatifs. (attention : problème pour 106 par exemple).

Exercice 7

7.1 Enoncé.

Calculer le produit PROD de deux nombres naturels(1 octet) MPD et MPR.

7.2 Méthode de résolution.

La méthode utilisée est inspirée de celle employée pour effectuer une multiplication à la main en décimal. Rappelons son principe :

```
        1 2 3   multiplicande
   x    1 3 2   multiplicateur
   ____________
        2 4 6   produit du multiplicande par 2
      3 6 9     produit du multiplicande par 3, décalé à gauche de  1 position
    1 2 3       produit du multiplicande par 1, décalé à gauche de  2 positions
   ____________
    1 6 2 3 6   somme des produits partiels
```

Pour effectuer la multiplication de deux nombres naturels(1 octet) en assembleur, on doit prendre en compte les remarques suivantes :
- on ne sait pas sommer plus de deux nombres en une seule opération ; par conséquent on cumulera les produits partiels petit à petit
- les opérations sont effectuées en binaire, donc le produit du multiplicande par un chiffre du multiplicateur est soit nul soit égal au multiplicande ; le cumul sera donc effectué uniquement si le produit partiel est non nul
- pour obtenir chaque chiffre du multiplicateur on va effectuer des décalages à droite, et récupérer les bits successifs dans le carry
- dans une multiplication de deux nombres de même format il faut un format double pour représenter le produit ; par conséquent le produit PROD de deux nombres naturels(1) est un naturel(2)
- pour calculer les produits partiels, on utilise une seule variable MPD de type naturel(2) initialisée avec le multiplicande, les produits partiels successifs étant obtenus par décalages
- dans le code donné ci-dessous, on suppose que les valeurs de MPD et MPR sont introduites dans le programme.

7.3 Pseudo-code et table des identificateurs.

Id.	Signification	Type	Nb oct.	VI	VF	Vérif.
MPD	initialement le multiplicande ensuite les produits partiels	naturel	2	MPD		
MPR	initialement le multiplicateur	naturel	1	MPR		

Id.	Signification	Type	Nb oct.	VI	VF	Vérif.
PROD	cumul des produits partiels finalement le produit	naturel	2	0	MPD*MPR	
K	nombre de bits restant à traiter	naturel	1	8	0	IUC

pseudo-code

```
. initialiser les opérandes MPD et MPR
. initialiser l'octet de poids fort de MPD à $00
. initialiser le résultat PROD à $0000
. initialiser K, nombre de bits à traiter, à 8
. Itérer1
    . lire le bit courant du multiplicateur MPR
    . Si1  ce bit est '1'
      Alors1
        . ajouter MPD à PROD
      Finsi1
    . décaler à gauche le produit partiel MPD
    . décrémenter K
  Sortirsi1  le dernier bit est traité (K = 0)
  Finitérer1
. afficher la produit PROD
```

7.4 Traduction du pseudo-code.

```
DEBUT       :
            LDA     #12
            STA     MPD
            LDA     #6
            STA     MPR
            LDA     #0
            STA     MPD+1
            STA     PROD
            STA     PROD+1
            LDA     #8
            STA     K
ITERER1     :
            LSR     MPR
SI1         :
            BCC     FINSI1
ALORS1      LDA     MPD
            CLC
            ADC     PROD
            STA     PROD
            LDA     MPD+1
            ADC     PROD+1
            STA     PROD+1
FINSI1      :
```

```
            ASL        MPD
            ROL        MPD+1
            DEC        K
SORTIRSI1   LDA        K
            CMP        #0
            BEQ        FINITERER1
            JMP        ITERER1
FINITERER1  :
            LDA        PROD+1
            JSR        AFFICHER EN HEXADECIMAL
            LDA        PROD
            JSR        AFFICHER EN HEXADECIMAL
            BRK
FIN         :
```

7.5 Programme d'essai en LISA sur APPLE IIe.

```
DEB :
            LDA        #12
            STA        MPD
            LDA        #6
            STA        MPR
            LDA        #0
            STA        MPD+1
            STA        PROD
            STA        PROD+1
            LDA        #8
            STA        K
ITR1 :
            LSR        MPR
SII1 :
            BCC        FSI1
ALO1        LDA        MPD
            CLC
            ADC        PROD
            STA        PROD
            LDA        MPD+1
            ADC        PROD+1
            STA        PROD+1
FSI1 :
            ASL        MPD
            ROL        MPD+1
            DEC        K
SSI1        LDA        K
            CMP        #0
            BEQ        FIT1
            JMP        ITR1
FIT1 :
            LDA        PROD+1
            JSR        PRBYTE
            LDA        PROD
            JSR        PRBYTE
            BRK
```

```
MPD       DFS       2
MPR       DFS       1
PROD      DFS       2
K         DFS       1
PRBYTE    EQU       $FDDA
FIN :
          END
```

7.6 Proposition de travail personnel.

Lire les données MPD et MPR à l'aide du code de l'exercice 4.

Exercice 8

8.1 Enoncé.

Lire au clavier un tableau TAB de N caractères et les recopier en écho sur l'écran. Classer ces caractères par ordre croissant suivant leur code ASCII et les afficher. (N est une constante non nulle inférieure à 255 ; dans cet exercice, elle a été choisie égale à 10).

8.2 Méthode de résolution.

Supposons que les X-1 premiers éléments soient à leur place définitive dans le tableau classé ; on cherche alors le plus petit élément de la partie non classée et on le permute avec le Xième élément ; les X premiers éléments sont alors à leur place définitive dans le tableau classé.

8.3 Pseudo-code et table des identificateurs.

Id.	**Signification**	**Type**	**Nb oct.**	**VI**	**VF**	**Vérif.**
constante						
N	nombre d'éléments du tableau	naturel	1	N	N	
variables						
IMIN	indice de MIN	naturel	1	?	?	
MIN	plus petit élément de la partie non classée	caractère	1	?	?	
TAB	tableau à classer puis classé	tableau de caractères	N	?	?	
X	indice du 1ier élément non classé ou indice de parcours du tableau	index	1	0	N-1	
Y	indice de l'élément en cours de comparaison avec le plus petit élément déjà trouvé	index	1	?	?	

niveau 1

. lire et afficher le tableau à classer
. classer le tableau
. afficher le tableau classé

niveau2

(début de lire et afficher le tableau à classer)
. initialiser X, indice de parcours du tableau, à 0
. Itérer1
  . lire l'élément d'indice X
  . afficher l'élément lu
 Sortirsi1 le dernier élément est lu (attention : X = N-1)
  . incrémenter X
 Finitérer1
. aller à la ligne
(fin de lire et afficher le tableau à classer)

(début de classer le tableau)
. initialiser X, indice de parcours du tableau, à 0
. Itérer2
  . classer l'élément d'indice X
 Sortirsi2 tous les éléments sont classés (attention : X = N-2)
  . incrémenter X
 Finitérer2
(fin de classer le tabeau)

(début de afficher le tableau classé)
. initialiser X, indice de parcours du tableau, à 0
. Itérer3
  . afficher l'élément d'indice X
 Sortirsi3 le dernier élément est affiché (X = N-1)
  . incrémenter X
 Finitérer3
(fin de afficher le tableau classé)

niveau3

(début de classer l'élément d'indice X)
  . chercher le plus petit élément de la partie non classée
  . placer cet élément en tête de la partie non classée
   par permutation avec le premier élément de cette partie
(fin de classer l'élément d'indice X)

niveau4

(début de chercher le plus petit élément de la partie non classée)
. initialiser IMIN, indice de l'élément cherché, à X
. initialiser MIN, élément cherché avec l'élément d'indice X
. initialiser Y, indice de l'élément à comparer, à (X+1)

```
. Itérer4
   . Si1  TAB(Y) < MIN
  Alors1
      . affecter TAB(Y) à MIN
      . affecter Y à IMIN
    Finsi1
  Sortirsi4  le dernier élément est comparé (Y = N-1)
   . incrémenter Y
  Finitérer4
(fin de chercher le plus petit élément de la partie non classée )
```

pseudo-code

```
(début de lire et afficher le tableau à classer)
. initialiser X à 0
. Itérer1
   . lire l'élément d'indice X
   . afficher l'élément lu
  Sortirsi1  le dernier élément est lu (X = N-1)
   . incrémenter X
  Finitérer1
. aller à la ligne
(fin de lire et afficher le tableau à classer)
. initialiser X à 0
. Itérer2
   (début de classer l'élément d'indice X)
   (début de chercher le plus petit élément de la partie non
    classée)
   . initialiser IMIN, indice de l'élément cherché, à X
   . initialiser MIN, élément cherché avec l'élément d'indice X
   . initialiser Y, indice de l'élément à comparer, à (X+1)
   . Itérer4
      . Si1  TAB(Y) < MIN
        Alors1
         . affecter TAB(Y) à MIN
         . affecter Y à IMIN
        Finsi1
     Sortirsi4  le dernier élément est comparé (Y = N-1)
      . incrémenter Y
     Finitérer4
   (fin de chercher le plus petit élément de la partie non
    classée )
   . placer cet élément en tête de la partie non classée
     (par permutation avec le premier élément de cette partie)
   (fin de classer l'élément d'indice X)
  Sortirsi2  tous les éléments sont classés (X = N-2)
   . incrémenter X
  Finitérer2
(début de afficher le tableau classé)
 initialiser X à 0
```

. <u>Itérer3</u>
  . afficher l'élément d'indice X
 <u>Sortirsi3</u> le dernier élément est affiché (X = N-1)
  . incrémenter X
 <u>Finitérer3</u>
(fin de afficher le tableau classé)

8.4 <u>Traduction du pseudo-code.</u>

```
DEBUT       :
            LDX     #0
ITERER1     :
            JSR     LIRE UN CARACTERE AU CLAVIER
            STA     TAB, X
            JSR     AFFICHER UN CARACTERE
SORTIRSI1   CPX     #N-1
            BEQ     FINITERER1
            INX
            JMP     ITERER1
FINITERER1  :
            JSR     ALLER A LA LIGNE
            LDX     #0
ITERER2     :
            STX     IMIN
            LDA     TAB, X
            STA     MIN
            TXA
            TAY
            INY
ITERER4     :
SI1         :
            LDA     TAB, Y
            CMP     MIN
            BCS     FINSI1
ALORS1      LDA     TAB, Y
            STA     MIN
            STY     IMIN
FINSI1      :
SORTIRSI4   CPY     #N-1
            BEQ     FINITERER4
            INY
            JMP     ITERER4
FINITERER4  :
            LDA     TAB, X
            LDY     IMIN
            STA     TAB,Y
            LDA     MIN
            STA     TAB, X
SORTIRSI2   CPX     #N-2
            BEQ     FINITERER2
            INX
            JMP     ITERER2
FINITERER2  :
            LDX     #0
```

```
ITERER3      :
             LDA        TAB, X
             JSR        AFFICHER UN CARACTERE
SORTIRSI3    CPX        #N-1
             BEQ        FINITERER3
             INX
             JMP        ITERER3
FINITERER3   :
             BRK
FIN          :
```

8.5 Programme d'essai en LISA sur APPLE IIe.

```
DEB :
             LDX        #0
ITR1  :
             JSR        RDKEY
             STA        TAB,X
             JSR        COUT
SSI1         CPX        #N-1
             BEQ        FIT1
             INX
             JMP        ITR1
FIT1  :
             JSR        CROUT
             LDX        #0
ITR2  :
             STX        IMIN
             LDA        TAB,X
             STA        MIN
             TXA
             TAY
             INY
ITR4  :
SII1  :
             LDA        TAB,Y
             CMP        MIN
             BCS        FSI1
ALO1         LDA        TAB,Y
             STA        MIN
             STY        IMIN
FSI1  :
SSI4         CPY        #N-1
             BEQ        FIT4
             INY
             JMP        ITR4
FIT4  :
```

```
          LDA     TAB,X
          LDY     IMIN
          STA     TAB,Y
          LDA     MIN
          STA     TAB,X
SSI2      CPX     #N-2
          BEQ     FIT2
          INX
          JMP     ITR2
FIT2 :
          LDX     #0
ITR3 :
          LDA     TAB,X
          JSR     COUT
SSI3      CPX     #N-1
          BEQ     FIT3
          INX
          JMP     ITR3
FIT3 :
          BRK
N         EQU     $A
IMIN      DFS     1
MIN       DFS     1
TAB       DFS     N
RDKEY     EQU     $FD0C
COUT      EQU     $FDED
CROUT     EQU     $FD8E
FIN :
          END
```

8.6 Proposition de travail personnel.

Classer les caractères par ordre décroissant. Le lecteur qui trouve ce travail trop facile pourra toujours classer les caractères dans le désordre.

Exercice 9

9.1 Enoncé.

Lire au clavier un tableau TAB de N caractères distincts et les recopier en écho sur l'écran. Classer ces caractères par ordre croissant suivant leur code ASCII dans un autre tableau TCL. (N est une constante non nulle inférieure à 255)

9.2 Méthode de résolution.

Dans un tableau classé par ordre croissant il y a K éléments plus petits que celui qui occupe la $(K+1)^{ième}$ place ; par exemple :

tableau à classer	tableau classé
P	.
M	.
J	.
L	M
A	.

dans le tableau à classer il y a 3 éléments plus petits que 'M', par conséquent le caractère M occupera dans le tableau classé la $(3+1)^{ième}$ place c'est-à-dire la place d'indice 3 si le premier élément est indicé par 0. Pour classer un tableau, il suffit donc de calculer pour chaque élément le nombre d'éléments plus petits que lui-même ; puis de ranger chaque élément à sa place dans la tableau classé.

9.3 Pseudo-code et table des identificateurs.

Id.	Signification	Type	Nb oct.	VI	VF	Vérif.
constante						
N	nombre d'éléments du tableau	naturel	1	N	N	
variables						
NB	nombre d'éléments inférieurs à TAB(X)	naturel	1	0	?	I U C
TAB	tableau à classer	tableau de caractères	N	?	?	
TCL	tableau classé	tableau de caractères	N	?	?	
X	indice de parcours du tableau	index	1	0	N-1	
Y	indice de l'élément à comparer à TAB(X)	index	1	0	N-1	

niveau1

. lire et afficher le tableau à classer
. classer le tableau
. afficher le tableau classé

niveau2

(début de classer le tableau)
. initialiser X, indice de parcours du tableau TAB, à 0
. Itérer2
   . calculer NB, nombre de caractères inférieurs à TAB(X)
   . ranger ce caractère dans le tableau classé TCL à la place d'indice NB
 Sortirsi2 le dernier élément est classé (X = N-1)
   . incrémenter X
 Finitérer2
(fin de classer le tableau)

niveau3

(début de calculer NB, nombre de caractères inférieurs à TAB(X))
. initialiser NB à 0
. initialiser Y, indice dans TAB de l'élément à comparer, à 0
. Itérer4
   . Si1 TAB(Y), élément à comparer, est inférieur à TAB(X)
     Alors1
       . incrémenter NB
     Finsi1
 Sortirsi4 le dernier élément est comparé (Y = N-1)
   . incrémenter Y
 Finitérer4
(fin de calculer NB, nombre de caractères inférieurs à TAB(X))

pseudo-code

(début de lire et afficher le tableau à classer)
. initialiser X, indice de parcours du tableau TAB, à 0
. Itérer1
   . lire l'élément d'indice X du tableau TAB
   . afficher l'élément lu
 Sortirsi1 le dernier élément est lu (X = N-1)
   . incrémenter X
 Finitérer1
. aller à la ligne
(fin de lire et afficher le tableau à classer)

(début de classer le tableau)
. initialiser X, indice de parcours du tableau TAB, à 0
. <u>Itérer2</u>
  (début de calculer NB, nombre de caractères inférieurs à TAB(X))
  . initialiser NB à 0
  . initialiser Y, indice dans TAB de l'élément à comparer, à 0
  . <u>Itérer4</u>
    . <u>Si1</u> TAB(Y) est inférieur à TAB(X)
      <u>Alors1</u>
        . incrémenter NB
      <u>Finsi1</u>
    <u>Sortirsi4</u> le dernier élément est comparé (Y = N-1)
      . incrémenter Y
    <u>Finitérer4</u>
  (fin de calculer NB, nombre de caractères inférieurs à TAB(X))
  . ranger ce caractère dans le tableau classé TCL à la place d'indice NB
 <u>Sortirsi2</u> le dernier élément est classé (X = N-1)
  . incrémenter X
 <u>Finitérer2</u>
(fin de classer le tableau)
(début de afficher le tableau classé)
. initialiser X, indice de parcours du tableau TCL, à 0
. <u>Itérer3</u>
  . afficher l'élément d'indice X du tableau TCL
 <u>Sortirsi3</u> le dernier élément est affiché (X = N-1)
  . incrémenter X
 <u>Finitérer3</u>
(fin de afficher le tableau classé)

9.4 <u>Traduction du pseudo-code.</u>

```
DEBUT      :
           LDX     #0
ITERER1    :
           JSR     LIRE UN CARACTERE AU CLAVIER
           STA     TAB, X
           JSR     AFFICHER UN CARACTERE
SORTIRSI1  CPX     #N-1
           BEQ     FINITERER1
           INX
           JMP     ITERER1
FINITERER1 :
           JSR     ALLER A LA LIGNE
           LDX     #0
ITERER2    :
           LDA     #0
           STA     NB
           LDY     #0
```

```
ITERER4     :
SI1         :
            LDA         TAB, Y
            CMP         TAB, X
            BCS         FINSI1
ALORS1      INC         NB
FINSI1      :
SORTIRSI4   CPY         #N-1
            BEQ         FINITERER4
            INY
            JMP         ITERER4
FINITERER4  :
            LDA         TAB, X
            LDY         NB
            STA         TCL, Y
SORTIRSI2   CPX         #N-1
            BEQ         FINITERER2
            INX
            JMP         ITERER2
FINITERER2  :
            LDX         #0
ITERER3     :
            LDA         TCL, X
            JSR         AFFICHER UN CARACTERE
SORTIRSI3   CPX         #N-1
            BEQ         FINITERER3
            INX
            JMP         ITERER3
FINITERER3  :
            BRK
FIN         :
```

9.5 Programme d'essai en LISA sur APPLE IIe.

```
DEB :
            LDX         #0
ITR1 :
            JSR         RDKEY
            STA         TAB,X
            JSR         COUT
SSI1        CPX         #N-1
            BEQ         FIT1
            INX
            JMP         ITR1
FIT1 :
            JSR         CROUT
            LDX         #0
ITR2 :
            LDA         #0
            STA         NB
            LDY         #0
```

```
ITR4 :
SII1 :
          LDA     TAB,Y
          CMP     TAB,X
          BCS     FSI1
ALO1      INC     NB
FSI1 :
SSI4      CPY     #N-1
          BEQ     FIT4
          INY
          JMP     ITR4
FIT4 :
          LDA     TAB,X
          LDY     NB
          STA     TCL,Y
SSI2      CPX     #N-1
          BEQ     FIT2
          INX
          JMP     ITR2
FIT2 :
          LDX     #0
ITR3 :
          LDA     TCL,X
          JSR     COUT
SSI3      CPX     #N-1
          BEQ     FIT3
          INX
          JMP     ITR3
FIT3 :
          BRK
N         EQU     $A
NB        DFS     1
TAB       DFS     N
TCL       DFS     N
RDKEY     EQU     $FD0C
COUT      EQU     $FDED
CROUT     EQU     $FD8E
FIN :
          END
```

9.6 Proposition de travail personnel.

Utiliser un troisième tableau pour mémoriser les différentes valeurs de NB et en profiter alors pour diviser par deux le nombre de comparaisons en ne comparant chaque élément qu'avec ceux qui le suivent.

Exercice 10

10.1 Enoncé.

Un tableau appelé TORG comprenant un nombre entier de pages NBPA plus un certain nombre d'octets REST est implanté en mémoire à partir de l'adresse $1000 ; ce tableau contient donc NBPA*256+REST octets.

Recopier TORG dans un tableau TDES implanté à partir de l'adresse $2000. (NBPA et REST sont des constantes éventuellement nulles).

10.2 Méthode de résolution.

Les éléments du tableau origine sont recopiés l'un après l'autre en commençant par les pages complètes et en terminant par le reste.

10.3 Informations techniques.

L'adressage absolu indexé ne permet pas de balayer tous les éléments d'un tableau de plus d'une page.

Pour résoudre ce problème nous allons utiliser un mode d'adressage indexé dans lequel la base peut être modifiée pour adresser n'importe quelle page ; ce mode est disponible sur le micro-processeur 6502, c'est le mode indirect post-indexé par Y.

10.4 Pseudo-code et table des identificateurs.

Id.	Signification	Type	Nb oct.	VI	VF	Vérif.
constantes						
NBPA	nombre de pages complètes à recopier	naturel	1			
REST	nombre d'octets de la dernière page	naturel	1			
variables						
TORG	tableau à recopier	tableau d'octets	NBPA*256+REST			
ORG	adresse de début de la page à recopier	pointeur	2	$1000		
TDES	tableau recopié	tableau d'octets	NBPA*256+REST			
DES	adresse de début de la page recopiée	pointeur	2	$2000		

Id.	Signification	Type	Nb oct.	VI	VF	Vérif.
K	nombre de pages complètes restant à recopier	naturel	1	NBPA	0	I U C
Y	indice dans la page de l'élément à recopier	index	1	0	REST	

niveau1

. *recopier les pages complètes*
. *recopier le reste*

niveau2

(début de recopier les pages complètes)
. initialiser ORG, adresse de la page à recopier avec l'adresse de TORG
. initialiser DES, adresse de la page recopiée avec l'adresse de TDES
. initialiser K, nombre de pages complètes restant à recopier à NBPA
. Itérer1
 Sortirsi1 la dernière page complète est recopiée (K = 0)
  . *recopier la page d'adresse ORG*
  . décrémenter K
  . mettre à jour ORG
  . mettre à jour DES
 Finitérer1
(fin de recopier les pages complètes)

(début de recopier le reste)
. initialiser Y, indice dans la dernière page de l'élément à recopier, à 0
. Itérer2
 Sortirsi2 le dernier élément est recopié (Y =REST)
  . recopier l'élément d'indice Y
  . incrémenter Y
 Finitérer2
(fin de recopier le reste)

niveau3

(début de recopier la page d'adresse ORG)
. initialiser Y, indice dans la page de l'élément à recopier, à 0
. Itérer3
  . recopier l'élément d'indice Y
 Sortirsi3 le dernier élément de la page est recopié (Y = 255)
  . incrémenter Y
 Finitérer3
(fin de recopier la page d'adresse ORG)

pseudo-code

```
(début de recopier les pages complètes)
. initialiser ORG, adresse de la page à recopier avec l'adresse de TORG
. initialiser DES, adresse de la page recopiée avec l'adresse de TDES
. initialiser K, nombre de pages complètes restant à recopier à NBPA
. Itérer1
  Sortirsi1 la dernière page complète est recopiée (K = 0)
    (début de recopier la page d'adresse ORG)
    . initialiser Y, indice dans la page de l'élément à recopier, à 0
    . Itérer3
        . recopier l'élément d'indice Y
      Sortirsi3 le dernier élément de la page est recopié (Y = 255)
        . incrémenter Y
      Finitérer3
    (fin de recopier la page d'adresse ORG)
    . décrémenter K
    . mettre à jour ORG
    . mettre à jour DES
  Finitérer1
(fin de recopier les pages complètes)
(début de recopier le reste)
. initialiser Y, indice dans la dernière page de l'élément à recopier, à 0
. Itérer2
  Sortirsi2 le dernier élément est recopié (Y = REST)
    . recopier l'élément d'indice Y
    . incrémenter Y
  Finitérer2
(fin de recopier le reste)
```

10.5 Traduction du pseudo-code.

```
DEBUT       :
            LDA     #TORG
            STA     ORG
            LDA     /TORG
            STA     ORG + 1
            LDA     #TDES
            STA     DES
            LDA     /TDES
            STA     DES + 1
            LDA     #NBPA
            STA     K
ITERER1     :
SORTIRSI1   LDA     K
            CMP     #0
            BEQ     FINITERER1
            LDY     #0
```

```
ITERER3     :
            LDA         (ORG), Y
            STA         (DES), Y
SORTIRSI3   CPY         #255
            BEQ         FINITERER3
            INY
            JMP         ITERER3
FINITERER3  :
            DEC         K
            INC         ORG + 1
            INC         DES + 1
            JMP         ITERER1
FINITERER1  :
            LDY         #0
ITERER2     :
SORTIRSI2   CPY         #REST
            BEQ         FINITERER2
            LDA         (ORG), Y
            STA         (DES), Y
            INY
            JMP         ITERER2
FINITERER2  :
            BRK
FIN         :
```

10.6 Programme d'essai en LISA sur APPLE IIe.

```
DEB :
            LDA         #TORG
            STA         ORG
            LDA         /TORG
            STA         ORG+1
            LDA         #TDES
            STA         DES
            LDA         /TDES
            STA         DES+1
            LDA         #NBPA
            STA         K
ITR1  :
SSI1        LDA         K
            CMP         #0
            BEQ         FIT1
            LDY         #0
```

```
ITR3 :
        LDA     (ORG),Y
        STA     (DES),Y
SSI3    CPY     #255
        BEQ     FIT3
        INY
        JMP     ITR3
FIT3 :
        DEC     K
        INC     ORG+1
        INC     DES+1
        JMP     ITR1
FIT1 :
        LDY     #0
ITR2 :
SSI2    CPY     #REST
        BEQ     FIT2
        LDA     (ORG),Y
        STA     (DES),Y
        INY
        JMP     ITR2
FIT2 :
        BRK
NBPA    EQU     $1
REST    EQU     $A
TORG    EQU     $1000
ORG     EPZ     $42
TDES    EQU     $2000
DES     EPZ     $44
K       DFS     1
FIN :
        END
```

LA PROGRAMMATION MODULAIRE

La décomposition d'un programme en modules améliore sa maintenabilité et sa lisibilité ; en effet, la programmation modulaire permet l'écriture et la lecture d'un programme à différents niveaux d'abstraction, elle diminue ainsi le degré de complexité à chaque niveau. Le but de ce chapitre n'est ni l'exposé d'une méthode de décomposition modulaire ni l'étude des mécanismes d'appel et de retour de sous-programme en assembleur, mais d'esquisser une méthode de normalisation des rapports entre objets et modules en assembleur. On voit successivement les notions de partage, de localité et leurs incidences sur le choix des identificateurs, les problèmes de communication entre modules et enfin le développement complet d'une petite application.

1 Partage et localité : choix des identificateurs.

Les notions de partage et de localité sont opposées, nous allons commencer par définir la localité, la notion de partage s'en déduira naturellement ; nous préciserons ensuite les conventions esquissées au chapitre 1 § 2.7 sur le choix des identificateurs.

Dans ce chapitre on convient d'appeler module, soit un programme appelant, soit un programme appelé. Une variable est locale à un module si elle est connue uniquement de celui-ci ; plus précisément, si ce module est le seul qui peut effectuer une opération d'écriture ou de lecture sur cette variable ; par opposition, une variable partagée peut être lue ou affectée dans plusieurs modules. En assembleur, déclarer une variable consiste à lui allouer un emplacement en mémoire, c'est-à-dire à établir une correspondance biunivoque entre son identificateur et son adresse en mémoire ; cette allocation a lieu une seule fois lors de l'assemblage de tous les modules (on n'aborde pas ici les techniques d'assemblage séparé) ; par conséquent, la notion de localité est inconnue de l'assembleur : toutes les variables ont la même durée de vie, qui est celle du programme, et la même portée, qui s'étend à l'ensemble des modules.

Cependant, la notion de localité peut prendre une signification en programmation assembleur si le programmeur respecte certaines conventions sur le choix des identificateurs, conventions analogues à celle qui a été prise pour les étiquettes (toutes les étiquettes utilisées dans un module portent les initiales du module). Nous proposons les définitions et règles suivantes :

<u>définition 1</u> une variable locale à un module peut être lue ou affectée uniquement dans ce module ;

<u>définition 2</u> une variable partagée peut être lue et affectée par tous les modules sans exception ;

<u>règle 1</u> les deux premiers caractères de l'identificateur d'un module et d'une variable locale à ce module sont identiques ;

<u>règle 2</u> les variables locales à un module sont déclarées à la suite des instructions du module ;

<u>règle 3</u> les deux premiers caractères de l'identificateur d'une variable partagée sont les lettres VP ;

<u>règle 4</u> les variables partagées sont déclarées en dernier lieu dans un module reservé à cet effet et appelé VP.

La règle 1 évite les duplications d'identificateurs et facilite le contrôle par le programmeur du bon usage des variables locales ; la stricte application de cette règle interdit l'usage de registre comme variable locale, or cet usage s'impose naturellement pour les variables de type index, aussi on admet dans ce cas précis une exception, en prenant toutefois la précaution, quand cela est nécessaire, de sauvegarder le registre au début du module et de le restituer à la fin. La règle 2 améliore la lisibilité du programme en regroupant instructions et objets locaux, de plus elle permet l'insertion du texte source d'un module à partir d'une bibliothèque. La règle 3 permet de vérifier immédiatement que les seuls objets utilisés dans un module (PE par exemple) sont soit des objets partagés (VP...) soit des objets locaux (PE...).

Les notions de partage et de localité étant précisées, on va examiner successivement la communication par variable partagée et par passage de paramètre.

2 Communication entre modules en assembleur.

Des modules participant au même traitement doivent pouvoir communiquer entre eux ; dans le cas le plus général, le module appelé traite les données transmises par le module appelant, en retour il communique les résultats de son traitement. Cette communication est par conséquent bi-directionnelle. Elle peut être réalisée par deux méthodes : la première consiste à utiliser une variable partagée, la deuxième fait appel à la notion de paramètre.

2.1 <u>Communication par variable partagée.</u>

Une variable partagée permet de résoudre le problème de la communication entre modules ; en effet, une conséquence de la définition 2 est que l'on s'autorise à lire ou à affecter une variable partagée dans un module appelant et dans le module appelé ; si le module appelant place une donnée dans une variable partagée, le module appelé peut la lire et éventuellement, après traitement, y écrire un résultat. Cette méthode a un avantage certain : sa facilité de mise en oeuvre, mais elle présente un inconvénient majeur : le module appelé n'est pas réutilisable car il est intimement lié au contexte extérieur ; son

traitement, même s'il présente un intérêt général, peut s'appliquer uniquement sur les variables partagées propres à ce contexte.

On réservera donc l'emploi des variables partagées aux situations où plusieurs modules opérant sur le même contexte manipulent un même groupe réduit d'informations liées à ce contexte.

2.2 Communication par passage de paramètre.

On vient de le constater, les variables partagées engendrent des modules qui ne sont pas réutilisables ; pour réaliser une communication qui préserve la généralité du traitement effectué par le module appelé, on peut utiliser deux objets locaux à chacun des modules appelant et appelé ; ces deux objets sont sémantiquement indissociables, on les nomme respectivement paramètre effectif et paramètre formel. Le paramètre formel, connu uniquement du module appelé, est utilisé pour la définition de ce module ; le paramètre effectif, connu uniquement de l'appelant, est utilisé lors de l'appel.

Le problème de communication s'exprime alors en termes de passage de paramètres ; la communication est bi-directionnelle, par conséquent, un paramètre doit pouvoir passer une donnée à l'appelé, un résultat à l'appelant ou ces deux informations. Dans le premier cas, le paramètre est importé, dans le deuxième cas il est exporté ; on notera que les qualificatifs 'importé' et 'exporté' s'appliquent au point de vue du module appelé.

La notion de paramètre est précisée, on vient d'indiquer le chemin suivi par l'information, on va maintenant indiquer quelle information doit être effectivement transmise et étudier les techniques utilisées en assembleur pour réaliser ces transferts.

2.2.1 Mode de passage.

L'information effectivement transmise dans un passage de paramètre est soit la valeur du paramètre soit son adresse ; on convient d'appeler argument ce qui est transmis. L'effet d'un passage de la valeur est globalement équivalent à une affectation entre paramètre formel et paramètre effectif ; pour un paramètre importé, l'affectation équivalente est 'paramètre formel := paramètre effectif', pour un paramètre exporté, l'affectation 'paramètre effectif := paramètre formel' et bien sûr ces deux affectations pour un paramètre importé-exporté.

Dans un passage de la valeur l'information est transmise à l'appel ou au retour ; dans un passage de l'adresse l'information, c'est-à-dire l'adresse du paramètre effectif, est uniquement transmise à l'appel, le paramètre formel doit alors être considéré comme un moyen d'accès au paramètre effectif. Ce mode de passage ne permet donc pas de distinguer les trois configurations : import, export, import-export ; aussi son emploi est potentiellement dangereux : le module appelé peut modifier accidentellement un paramètre importé. Le passage par adresse, bien que dangereux, est efficace pour passer des paramètres de type structuré, par exemple, pour passer un tableau il suffit de transmettre l'adresse de son premier élément alors qu'un passage par valeur nécessiterait le passage par valeur de chaque élément du tableau.

On connait le chemin suivi par l'information import ou export, sa nature valeur ou adresse, on peut aborder les techniques utilisées en assembleur pour réaliser ces transferts d'information.

2.2.2 Techniques de passage.

On ne distingue pas dans ce paragraphe le passage de valeur du passage d'adresse, les problèmes techniques sont de même nature si ce n'est que l'information transmise est la valeur de l'adresse. On a vu qu'un passage de valeur est globalement équivalent à une affectation entre paramètre formel et paramètre effectif ; le problème à résoudre est donc de réaliser une affectation entre deux variables qui sont locales à des modules distincts. La solution consiste à utiliser un relais et le choix de ce relais conditionne la technique de passage ; le programmeur a deux possibilités, soit utiliser les ressources disponibles dans la machine : registres et pile, soit réserver une table en mémoire à ce seul usage. Dans chaque cas, la correspondance paramètre/relais doit être documentée dans le module appelé.

On examine successivement les trois techniques : passage par les registres, passage par la pile, passage par une table.

2.2.2.1 Passage par les registres.

Les transferts d'information sont réalisés par l'intermédiaire des registres ; à chaque paramètre est associé un ou plusieurs registres.

La stricte application de cette technique peut entraîner des lourdeurs : un paramètre formel s'avère inutile si le module appelé peut opérer directement sur les registres associés ; la suppression d'un paramètre formel outre le gain de mémoire et de temps qu'elle procure lève un des obstacles à la réentrance en éliminant une variable locale (on suppose que le programme de traitement de l'interruption sauvegarde et restaure les registres). On rappelle qu'un module réentrant peut être appelé par un programme de traitement d'interruption qui l'a interrompu ; une technique pour obtenir un module réentrant est d'implanter tous les objets qu'il manipule dans des zones mémoire dépendant du module appelant. La suppression d'un paramètre formel a cependant un inconvénient : le module est moins lisible ; d'autre part, la règle 1 qui permet le contrôle du bon usage des variables locales n'est plus applicable. Ces arguments font que l'on réservera cette pratique aux modules courts et de faible complexité.

La technique de passage par les registres est simple ; elle réalise le passage des informations indépendamment du contexte extérieur cependant elle n'est pas universelle, de plus son emploi en 6502 est limité par le nombre restreint de registres ainsi que par leur fréquente indisponibilité.

2.2.2.2 Passage par la pile.

Le module appelant empile les arguments et dépile les résultats ; le module appelé peut accéder à la pile soit par ses primitives d'accès soit en la considérant comme une table.

Dans le premier cas, l'appelé commence par sauvegarder l'adresse de retour qui se trouve au sommet de la pile puis il la restitue après avoir empilé les valeurs à exporter.

Si l'appelé accède à la pile par un adressage indexé, on doit supposer que son traitement laisse la pile globalement invariante, il est alors inutile de sauvegarder

l'adresse de retour ; l'appelant commence par réserver la place nécessaire aux valeurs exportées, en empilant des valeurs quelconques, puis empile les arguments. Chaque module appelant possède ainsi sa propre table pour passer les arguments.

2.2.2.3 Regroupement des arguments dans une table.

Dans cette dernière technique, l'appelant copie les arguments dans une table ; il suffit alors de passer l'adresse de cette table au module appelé qui peut ainsi accéder à chaque argument. Le passage de l'adresse de la table est réalisé soit par les registres soit par la pile.

Cette technique permet le passage d'un grand nombre de paramètres ; deux variantes sont utilisées : la table est implantée dans une zone mémoire réservée, ou directement dans la zone programme du module appelant. La mise en œuvre de la première variante est simple, précisons la deuxième : les arguments sont copiés dans la zone mémoire qui suit immédiatement l'instruction d'appel, par conséquent dans une zone propre à l'appelant ; or, on voit que lors de l'appel l'adresse de retour est sauvegardée dans la pile ; le module appelé trouve donc l'adresse de la table au sommet de la pile, en contre partie il doit calculer l'adresse réelle de retour ce qui alourdit bien sûr le traitement.

2.3 Conclusion.

On vient de voir qu'en appliquant certaines conventions sur le choix des identificateurs, les notions de partage et de localité peuvent être exploitées en programmation assembleur pour résoudre les problèmes de communication entre modules. Le partage d'information est facile à mettre en oeuvre mais il engendre des modules uniquement utilisables dans un environnement particulier, l'emploi de paramètres est plus délicat, il exige plus de place et de temps mais il produit des modules réutilisables et de plus certaines techniques permettent d'obtenir des modules réentrants.

L'emploi de variable partagée est réservé aux situations où plusieurs modules opérant sur le même contexte manipulent un même groupe d'information ; dans tous les autres cas, le choix de paramètres est plus approprié.

Après ces considérations théoriques, passons à la pratique.

3 Réalisation d'un éditeur de texte.

Il n'est pas question dans ce paragraphe de développer un véritable éditeur de texte, mais d'illustrer dans un but pédagogique toutes les étapes du développement d'une application et en particulier sa décomposition modulaire. On voit successivement le cahier des charges, les structures des données, l'automate de l'éditeur, la décomposition modulaire, et enfin la réalisation de chaque module.

3.1 Cahier des charges.

Réaliser un éditeur de texte de 16 lignes de 16 caractères comportant les fonctions suivantes :

SAISIE d'un texte,
AFFICHAGE du texte,
SUPRESSION d'une ligne,
INSERTION d'une ligne.

L'utilisateur de cet éditeur commence par saisir complètement le texte puis il tape une commande précisant s'il veut afficher le texte, insérer une ligne, supprimer une ligne ou quitter l'éditeur ; les commandes disponibles sont affichées en permanence sur la première ligne de l'écran.

Le texte comporte au plus 16 lignes ; lors de la saisie initiale, un texte de moins de 16 lignes est terminé par le caractère contrôle E (désigné par CE) tapé en début de ligne ; ce caractère de sortie de la saisie ne fait pas partie du texte. Une ligne comporte au plus 16 caractères ; si elle en comporte moins de 16, il faut terminer la saisie de la ligne par un retour chariot (désigné par RC ; ce caractère fait partie de la ligne). Par contre, si on saisit une ligne de 16 caractères, l'éditeur passe automatiquement à la ligne suivante. Le numéro de la ligne (compris entre 1 et 16) à supprimer ou à insérer est entré au clavier en décimal sous la forme d'un ou deux chiffres suivis d'un retour chariot, la suppression d'une ligne s'accompagne du tassement des lignes suivantes, l'insertion d'une ligne s'accompagne du déplacement des lignes suivantes, précisons enfin que l'affichage permet uniquement d'afficher le texte complet.

3.2 Structure des données.

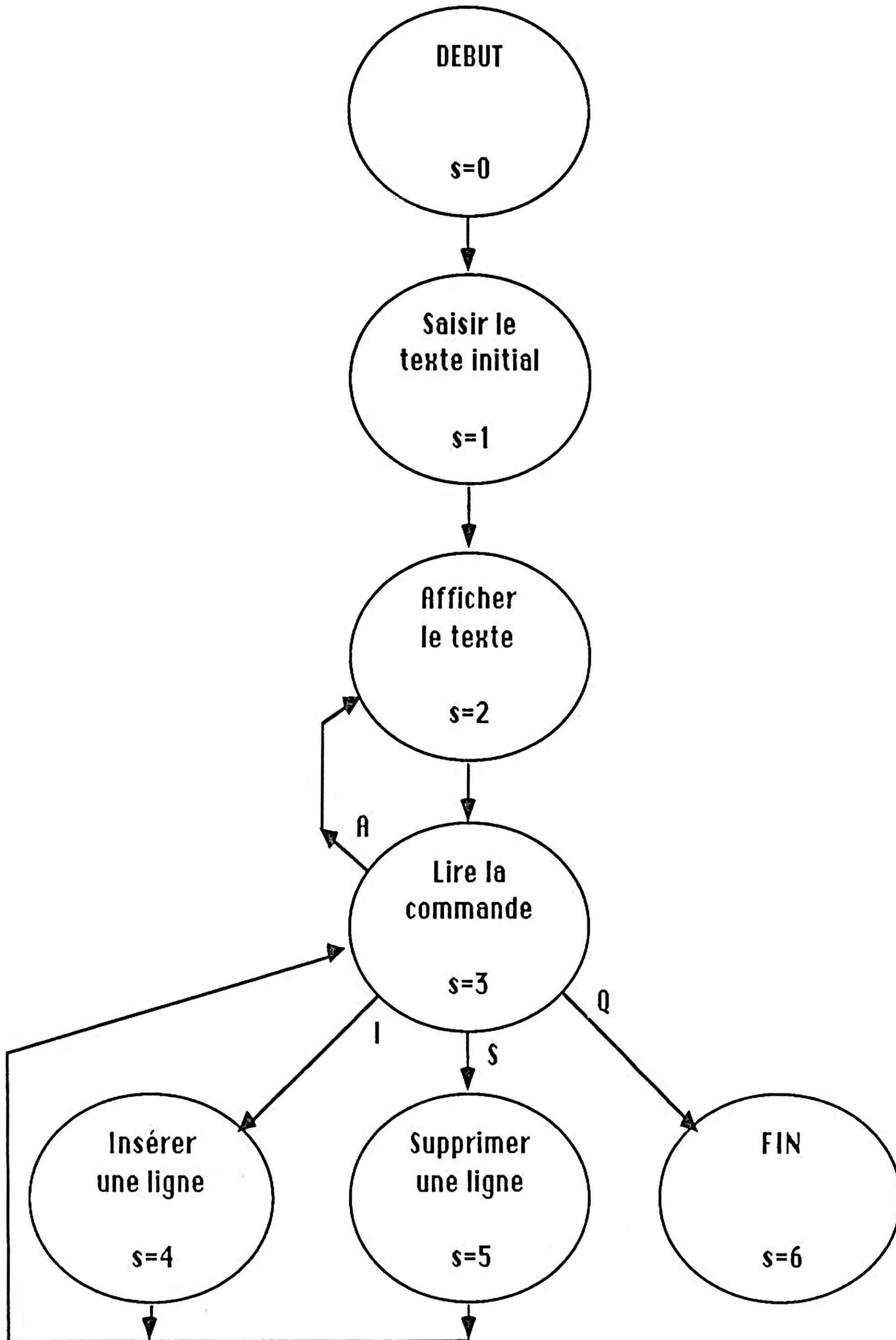
Le texte est mémorisé dans la variable partagée TEXT de type tableau de 256 caractères. Les 16 premiers caractères sont réservés à la première ligne, les 16 suivants à la deuxième ligne, etc.,. Si une ligne comporte un nombre n de caractères strictement inférieur à 16, le (n+1) ^{ième} est un retour chariot et les 16-(n+1) derniers sont indéfinis.

Le nombre de lignes du texte est mémorisé dans la variable partagée NBLN qui est initialisée lors de la saisie initiale du texte et mise à jour après chaque suppression ou insertion . D'où la table des Variables Partagées :

Id.	Signification	Type	Nb oct.	VI	VF	Vérif.
objets partagés						
TEXT	texte traité par l'éditeur	tableau de caractères	256	?	?	
NBLN	nombre de lignes du texte	naturel	1	0	?	

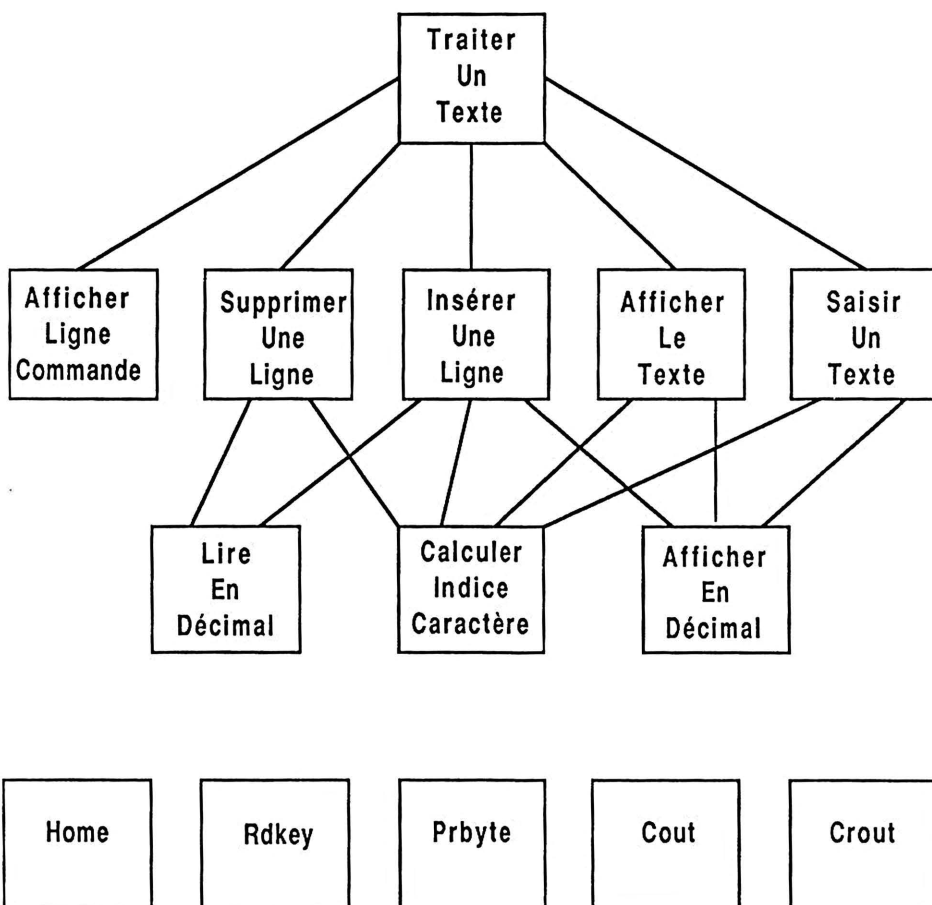
3.3 Automate de l'éditeur.

Rappelons que l'automate d'états finis est un des outils qui permet de supporter la pensée dans la phase initiale de développement d'une application ; il fournit ici une description normalisée du processus suivi par l'éditeur.


DEBUT
s=0
Saisir le texte initial
s=1
Afficher le texte
s=2
A
Lire la commande
s=3
I
S
Q
Insérer une ligne
s=4
Supprimer une ligne
s=5
FIN
s=6

3.4 Décomposition modulaire.

Le graphe ci-dessous présente une décomposition modulaire des tâches accomplies par l'éditeur ; on peut observer quatre niveaux (de haut en bas) : le niveau de contrôle, le niveau fonctionnel plus deux niveaux d'utilitaires, application et système. Les deux premiers niveaux découlent naturellement de l'automate d'états finis, les deux niveaux d'utilitaires résultent de l'analyse descendante appliquée aux modules fonctionnels ; précisons enfin que, par définition, les modules système sont disponibles dans le moniteur de l'Apple II.



La fonction de chaque module est décrite succintement ci-dessous, elle sera détaillée plus loin.
TRAITER UN TEXTE (TT) : contrôler la saisie initiale puis les modifications et l'affichage d'un texte en fonction des commandes entrées par l'utilisateur.
SAISIR UN TEXTE INITIAL (SA) : saisir un texte d'au plus 16 lignes d'au plus 16 caractères.
AFFICHER LE TEXTE (AF) : afficher le texte complet.

SUPPRIMER UNE LIGNE (SU) : supprimer la ligne dont on donne le numéro ; si cette ligne n'existe pas, la commande est ignorée.
INSERER UNE LIGNE (IN) : insérer une ligne dont on donne le numéro ; l'insertion dans un texte complet de 16 lignes est ignorée, l'insertion peut se faire à l'intérieur ou après la dernière ligne.
AFFICHER LA LIGNE DE COMMANDE (AC) : afficher, sur la première ligne de l'écran, la ligne des commandes disponibles.

Modules application :
CALCULER L'INDICE (CI) : calculer l'indice dans le texte d'un caractère connaissant son numéro de ligne et son numéro dans la ligne.
LIRE UN NOMBRE DECIMAL (LD) : lire un nombre décimal d'un ou deux chiffres et le convertir en binaire.
AFFICHER EN DECIMAL (AD) : convertir de binaire en décimal un nombre entier naturel inférieur ou égal à 99_{10} et l'afficher.

Modules système :
EFFACER L'ECRAN (HOME) : effacer l'écran et positionner le curseur en haut de la page à gauche.
LIRE UN CARACTERE (RDKEY) : lire un caractère au clavier et placer son code ASCII dans l'accumulateur.
AFFICHER EN HEXADECIMAL (PRBYTE) : afficher la valeur hexadécimale de l'accumulateur.
AFFICHER UN CARACTERE (COUT) : afficher le caractère dont le code ASCII est dans l'accumulateur.
ALLER A LA LIGNE (CROUT) : placer le curseur de l'écran au début de la ligne suivante.

3.5 Réalisation de chaque module.

On réalise les modules ci-dessus, auxquels on ajoutera le module VP pour les objets partagés. La compréhension d'un module appelant nécessite la lecture dans le module appelé des informations relatives à sa fonction et au passage des arguments.

3.5.1 Traiter un texte (module TT).

3.5.1.1 Fonction.

Le module TT a pour fonction de contrôler la saisie initiale puis les modifications et l'affichage du texte en fonction des commandes entrées par l'utilisateur : après la saisie initiale, on affiche en permanence la ligne des commandes ; l'utilisateur est sollicité par un caractère '?', il peut alors choisir une opération d'affichage, de suppression, d'insertion ou quitter l'éditeur.

La commande choisie est mémorisée dans la variable COM qui peut prendre les valeurs suivantes :

A pour Afficher le texte complet,
S pour Supprimer une ligne,
I pour Inserer une ligne,
Q pour Quitter l'éditeur ;

la saisie d'un caractère hors de l'alphabet des commandes (A, S, I, Q) est ignorée ; après un affichage, une suppression ou une insertion, l'utilisateur est de nouveau sollicité.

3.5.1.2 Informations techniques.

Pour afficher en permanence la ligne des commandes suivie d'une ligne vide on peut verrouiller les deux premières lignes de l'écran. Pour cela il suffit d'initialiser la variable WTOP (window top), implantée à l'adresse $22, avec le nombre de lignes à verrouiller : en fait le module HOME efface l'écran à partir de la ligne n+1, n étant la valeur de WTOP.

3.5.1.3 Pseudo-code et table des identificateurs.

Id.	Signification	Type	Nb oct.	VI	VF	Vérif.
objets locaux						
variables						
WTOP	nombre de lignes verrouillées	naturel	1	2	2	
COM	initiale de la commande	caractère	1	?	?	

Le pseudo-code du module de contrôle TT se déduit directement de l'automate d'états finis, comme cela est détaillé au chapitre 1 § 4.2 ; on présente ici une deuxième solution, qui suit la même démarche mais de façon moins rigide, en particulier la variable S désignant l'état courant n'est plus utilisée ; le code engendré est plus performant, mais en contrepartie, on perd au niveau de la maintenabilité.

niveau 1

```
. saisir un texte
. afficher la ligne de commande sur la première ligne de l'écran
. verrouiller les deux premières lignes de l'écran
. afficher le texte
. Itérer1
      . saisir une commande
 Sortirsi1  la commande est Q(uitter
      . traiter la commande saisie
 Finitérer1
```

niveau 2

```
(début de saisir une commande)
. aller à la ligne
. afficher le caractère de sollicitation '?'
. lire et afficher COM, initiale de la commande
(fin de saisir une commande)

(début de traiter la commande saisie)
. Cas où1  commande
     A  :   . afficher le texte
     S  :   . supprimer une ligne
     I  :   . insérer une ligne
 Fincas1
(fin de traiter la commande saisie)
```

d'où le pseudo-code :

```
. saisir un texte
. afficher la ligne de commande sur la première ligne de l'écran
. verrouiller les deux premières lignes de l'écran
. afficher le texte
. Itérer1
     (début de saisir une commande)
     . aller à la ligne
     . afficher le caractère de sollicitation '?'
     . lire et afficher COM, initiale de la commande
     (fin de saisir une commande)
 Sortirsi1  la commande est Q(uitter
     (début de traiter la commande saisie)
     . Cas où1  commande
          A  :   . afficher le texte
          S  :   . supprimer une ligne
          I  :   . insérer une ligne
      Fincas1
     (fin de traiter la commande saisie)
 Finitérer1
```

3.5.1.4 Traduction du pseudo-code.

```
DEBUT      :
           JSR     SAISIR UN TEXTE
           JSR     AFFICHER LES COMMANDES
           LDA     #2
           STA     WTOP
           JSR     AFFICHER LE TEXTE
ITERER1    :
           JSR     ALLER A LA LIGNE
           LDA     "?
           JSR     AFFICHER UN CARACTERE
           JSR     LIRE UN CARACTERE
           JSR     AFFICHER UN CARACTERE
           STA     COM
SORTIRSI1  LDA     COM
           CMP     "Q
           BEQ     FINITERER1
CAS1       :
           LDA     COM
CSA1       CMP     "A
           BNE     CSB1
           JSR     AFFICHER LE TEXTE
           JMP     FINCAS1
CSB1       CMP     "S
           BNE     CSC1
           JSR     SUPPRIMER UNE LIGNE
           JMP     FINCAS1
CSC1       CMP     "I
           BNE     FINCAS1
           JSR     INSERER UNE LIGNE
           JMP     FINCAS1
FINCAS 1   :
           JMP     ITERER1
FINITERER1 :
           BRK
FIN        :
```

3.5.1.5 Programme d'essai en LISA sur APPLE IIe.

```
TTDEB :
           JSR     SADEB
           JSR     ACDEB
           LDA     #2
           STA     TTWTOP
           JSR     AFDEB
TTITR1 :
           JSR     CROUT
           LDA     "?
           JSR     COUT
           JSR     RDKEY
           JSR     COUT
           STA     TTCOM
```

```
TTSSI1   LDA      TTCOM
         CMP      "Q
         BEQ      TTFIT1
TTCAS1 :
         LDA      TTCOM
TTCSA1   CMP      "A
         BNE      TTCSB1
         JSR      AFDEB
         JMP      TTFCA1
TTCSB1   CMP      "S
         BNE      TTCSC1
         JSR      SUDEB
         JMP      TTFCA1
TTCSC1   CMP      "I
         BNE      TTFCA1
         JSR      INDEB
         JMP      TTFCA1
TTFCA1 :
         JMP      TTITR1
TTFIT1 :
         BRK
TTWTOP   EPZ      $22
TTCOM    DFS      1
TTFIN :
```

3.5.2 Saisir un texte (module SA)

3.5.2.1 Fonction.

Le module SA a pour fonction de saisir un texte d'au plus 16 lignes d'au plus 16 caractères, si une ligne comporte moins de 16 caractères, il faut terminer la saisie de la ligne par un retour chariot (désigné par RC ; qui fait partie de la ligne) ; par contre, si on saisit une ligne de 16 caractères, l'éditeur passe automatiquement à la ligne suivante ; un texte de moins de 16 lignes sera terminé par le caractère Contrôle E, suivi d'un RC, tapé en début de ligne (le caractère de sortie CE ne fait pas partie du texte). Lors de la saisie l'utilisateur est assisté dans la mise en page par l'affichage du numéro de la ligne courante.

3.5.2.2 Pseudo-code et table des identificateurs.

Id.	Signification	Type	Nb oct.	VI	VF	Vérif.
objets partagés						
TEXT	texte traité par l'éditeur	tableau de caractères	256	?	?	
NBLN	nombre de lignes du texte	naturel	1	0	?	

objets locaux
constantes

RC	retour chariot	caractère	1	$8D	$8D
CE	contrôle E	caractère	1	$85	$85

variables

NULI	numéro de la ligne à saisir	naturel	1	1	?
NUCA	numéro dans la ligne du caractère à saisir	naturel	1	1	?
X	indice dans le texte du caractère à saisir	index	1	?	?

niveau 1

. effacer l'écran
. initialiser NULI, numéro de la ligne à saisir, à 1
. Itérer1
    . afficher en décimal NULI suivi d'un espace
    . saisir la ligne de numéro NULI, la recopier dans TEXT et l'afficher
  SortirsiA1 le premier caractère de la ligne est CE
  SortirsiB1 la ligne affichée est la 16ième ligne
    . incrémenter NULI
  Finitérer1
. mettre à jour NBLN , nombre de lignes du texte

niveau 2

(début de saisir la ligne de numéro NULI, la recopier dans TEXT et l'afficher)
. initialiser NUCA, numéro dans la ligne du caractère à saisir, à 1
. calculer l'indice X du 1ier caractère de la ligne de numéro NULI
. Itérer2
    . lire un caractère et l'afficher
    . ranger le caractère lu dans TEXT à la place d'indice X
  SortirsiA2 le caractère saisi est un retour chariot
  SortirsiB2 le caractère affiche est le 16ième de la ligne
    . incrémenter NUCA et X
  Finitérer2
. Si1 le dernier caractère n'est pas un retour chariot
  Alors1 . aller à la ligne
  Finsi1
(fin de saisir la ligne de numéro NULI, la recopier dans TEXT et l'afficher)

(début de mettre à jour NBLN)
. affecter à NBLN la valeur de NULI
. Si2 le 1^ier^ caractère de la ligne est un CE
Alors2
. décrémenter NBLN
Finsi2
(fin de mettre à jour NBLN)

d'où le pseudo-code :

. effacer l'écran
. initialiser NULI, numéro de la ligne à saisir, à 1
. Itérer1
. afficher en décimal NULI suivi d'un espace
(début de saisir la ligne de numéro NULI, la recopier dans TEXT et l'afficher)
. initialiser NUCA, numéro dans la ligne du caractère à saisir, à 1
. calculer l'indice X du 1^ier^ caractère de la ligne de numéro NULI
. Itérer2
. lire un caractère et l'afficher
. ranger le caractère lu dans TEXT à la place d'indice X
SortirsiA2 le caractère saisi est un retour chariot
SortirsiB2 le caractère affiché est le 16ième de la ligne
. incrémenter NUCA et X
Finitérer2
. Si1 le dernier caractère n'est pas un retour chariot
Alors1 . aller à la ligne
Finsi1
(fin de saisir la ligne de numéro NULI, la recopier dans TEXT et l'afficher)
SortirsiA1 le premier caractère de la ligne est CE
SortirsiB1 la ligne affichée est la 16ième ligne
. incrémenter NULI
Finitérer1
(début de mettre à jour NBLN)
. affecter à NBLN la valeur de NULI
. Si2 le 1^ier^ caractère de la ligne est un CE
Alors2
. décrémenter NBLN
Finsi2
(fin de mettre à jour NBLN)

3.5.2.3 Traduction du pseudo-code.

```
DEBUT     :
          JSR     EFFACER L'ECRAN
          LDA     #1
          STA     NULI
ITERER1   :
          LDA     NULI
```

```
            JSR     AFFICHER EN DECIMAL
            LDA     "
            JSR     AFFICHER UN CARACTERE
            LDA     #1
            STA     NUCA
            LDX     NULI
            LDY     NUCA
            JSR     CALCULER INDICE
ITERER2     :
            JSR     LIRE UN CARACTERE
            JSR     AFFICHER UN CARACTERE
            STA     TEXT,X
SORTIRSIA2  LDA     TEXT,X
            CMP     #RC
            BEQ     FINITERER2
SORTIRSIB2  LDA     NUCA
            CMP     #16
            BEQ     FINITETER2
            INC     NUCA
            INX
            JMP     ITERER2
FINITERER2  :
SI1         :
            LDA     TEXT,X
            CMP     #RC
            BEQ     FINSI1
ALORS1      JSR     ALLER A LA LIGNE
FINSI1      :
SORTIRSIA1  LDX     NULI
            LDY     #1
            JSR     CALCULER INDICE
            LDA     TEXT,X
            CMP     #CE
            BEQ     FINITERER1
SORTIRSIB1  LDA     NULI
            CMP     #16
            BEQ     FINITERER1
            INC     NULI
            JMP     ITERER1
FINITERER1  :
            LDA     NULI
            STA     NBLN
SI2         :
            LDX     NULI
            LDY     #1
            JSR     CALCULER INDICE
            LDA     TEXT,X
            CMP     #CE
            BNE     FINSI2
ALORS2      DEC     NBLN
FINSI2      :
            RTS
FIN         :
```

3.5.2.4 Programme d'essai en LISA sur APPLE IIe.

```
SADEB :
        JSR     HOME
        LDA     #1
        STA     SANULI
SAITR1 :
        LDA     SANULI
        JSR     AFDEB
        LDA     "
        JSR     COUT
        LDA     #1
        STA     SANUCA
        LDX     SANULI
        LDY     SANUCA
        JSR     CIDEB
SAITR2 :
        JSR     RDKEY
        JSR     COUT
        STA     VPTEXT,X
SASSA2  LDA     VPTEXT,X
        CMP     #SARC
        BEQ     SAFIT2
SASSB2  LDA     SANUCA
        CMP     #16
        BEQ     SAFIT2
        INC     SANUCA
        INX
        JMP     SAITR2
SAFIT2 :
SASII1 :
        LDA     VPTEXT,X
        CMP     #SARC
        BEQ     SAFSI1
SAALO1  JSR     CROUT
SAFSI1 :
SASSA1  LDX     SANULI
        LDY     #1
        JSR     CIDEB
        LDA     VPTEXT,X
        CMP     #SACE
        BEQ     SAFIT1
SASSB1  LDA     SANULI
        CMP     #16
        BEQ     SAFIT1
        INC     SANULI
        JMP     SAITR1
SAFIT1 :
        LDA     NULI
        STA     NBLN
SASII2 :
        LDX     SANULI
        LDY     #1
        JSR     CIDEB
```

```
              LDA      VPTEXT,X
              CMP      #SACE
              BNE      SAFSI2
      SAALO2  DEC      SANBLN
      SAFSI2 :
              RTS
      SARC    EQU      $8D
      SACE    EQU      $85
      SANULI  DFS      1
      SANUCA  DFS      1
      SAFIN :
```

3.5.3 Afficher le texte (module AF)

3.5.3.1 Fonction.

Le module AF a pour fonction d'afficher le texte complet en faisant précéder chaque ligne de son numéro.

3.5.3.2 Pseudo-code et table des identificateurs.

Id.	Signification	Type	Nb oct.	VI	VF	Vérif.
objets partagés						
TEXT	texte traité par l'éditeur	tableau de caractères	256	?	?	
NBLN	nombre de lignes du texte	naturel	1	0	?	
objets locaux						
constante						
RC	retour chariot	caractère	1	$8D	$8D	
variables						
NULI	numéro de la ligne à afficher	naturel	1	1	?	
NUCA	numéro dans la ligne du caractère à afficher	naturel	1	1	?	
X	indice dans le texte du caractère à afficher	index	1	?	?	

```
niveau 1
      . aller à la ligne
      . initialiser NULI, numéro de la ligne à afficher, à 1
      . Itérer1
       Sortirsi1  la dernière ligne est affichée (NBLN < NULI)
          . afficher en décimal NULI suivi d'un espace
          . afficher la ligne de numéro NULI
          . incrémenter NULI
       Finitérer1

niveau 2
      (début de afficher la ligne de numéro NULI)
      . initialiser NUCA, numéro dans la ligne du caractère à afficher, à 1
      . calculer l'indice X du 1ier caractère de la ligne de numéro NULI
      . Itérer2
          . afficher le caractère d'indice X
       SortirsiA2 le caractère affiché est un retour chariot
       SortirsiB2 le caractère affiché est le 16ième de la ligne
          . incrémenter NUCA
          . incrémenter X
       Finitérer2
      . Si1 le dernier caractère n'est pas un retour chariot
       Alors1 . aller à la ligne
       Finsi1
      (fin de afficher la ligne de numéro NULI)
```

d'où le pseudo-code :

```
      . aller à la ligne
      . initialiser NULI, numéro de la ligne à afficher, à 1
      . Itérer1
       Sortirsi1  la dernière ligne est affichée (NBLN < NULI)
          . afficher en décimal NULI suivi d'un espace
          (début de afficher la ligne de numéro NULI)
          . initialiser NUCA, numéro dans la ligne du caractère à afficher, à 1
          . calculer l'indice X du 1ier caractère de la ligne de numéro NULI
          . Itérer2
              . afficher le caractère d'indice X
           SortirsiA2 le caractère affiché est un retour chariot
           SortirsiB2 le caractère affiché est le 16ième de la ligne
              . incrémenter NUCA
              . incrémenter X
           Finitérer2
          . Si1 le dernier caractère n'est pas un retour chariot
           Alors1 . aller à la ligne
           Finsi1
          (fin de afficher la ligne de numéro NULI)
          . incrémenter NULI
       Finitérer1
```

3.5.3.3 Traduction du pseudo-code.

```
DEBUT       :
            JSR     ALLER A LA LIGNE
            LDA     #1
            STA     NULI
ITERER1     :
SORTIRSI1   LDA     NBLN
            CMP     NULI
            BCC     FINITERER1
            LDA     NULI
            JSR     AFFICHER EN DECIMAL
            LDA     "
            JSR     AFFICHER UN CARACTERE
            LDA     #1
            STA     NUCA
            LDX     NULI
            LDY     NUCA
            JSR     CALCULER INDICE
ITERER2     :
            LDA     TEXT,X
            JSR     AFFICHER UN CARACTERE
SORTIRSIA2  LDA     TEXT,X
            CMP     #RC
            BEQ     FINITERER2
SORTIRSIB2  LDA     NUCA
            CMP     #16
            BEQ     FINITERER2
            INC     NUCA
            INX
            JMP     ITERER2
FINITERER2  :
SI1         :
            LDA     TEXT,X
            CMP     #RC
            BEQ     FINSI1
ALORS1      JSR     ALLER A LA LIGNE
FINSI1      :
            INC     NULI
            JMP     ITERER1
FINITERER1  :
            RTS
FIN         :
```

3.5.3.4 Programme d'essai en LISA sur APPLE IIe.

```
AFDEB :
            JSR     CROUT
            LDA     #1
            STA     AFNULI
```

```
AFITR1 :
AFSSI1   LDA      VPNBLN
         CMP      AFNULI
         BCC      AFFIT1
         LDA      AFNULI
         JSR      ADDEB
         LDA      "
         JSR      COUT
         LDA      #1
         STA      AFNUCA
         LDX      AFNULI
         LDY      AFNUCA
         JSR      CIDEB
AFITR2 :
         LDA      VPTEXT,X
         JSR      COUT
AFSSA2   LDA      VPTEXT,X
         CMP      #AFRC
         BEQ      AFFIT2
AFSSB2   LDA      AFNUCA
         CMP      #16
         BEQ      AFFIT2
         INC      AFNUCA
         INX
         JMP      AFITR2
AFFIT2 :
AFSII1 :
         LDA      VPTEXT,X
         CMP      #AFRC
         BEQ      AFFSI1
AFALO1   JSR      CROUT
AFFSI1 :
         INC      AFNULI
         JMP      AFITR1
AFFIT1 :
         RTS
AFRC     EQU      $8D
AFNULI   DFS      1
AFNUCA   DFS      1
AFFIN :
```

3.5.4 Supprimer une ligne (module SU)

3.5.4.1 Fonction.

Le module SU a pour fonction de supprimer la ligne dont on donne le numéro ; si cette ligne n'existe pas, la commande est ignorée.

3.5.4.2 Méthode de résolution.

Si la ligne à supprimer n'est pas la dernière ligne du texte, on remonte d'un 'cran' toutes les lignes suivantes, on écrase ainsi la ligne à supprimer (pour simplifier le traitement, on suppose ici qu'une ligne contient 16 caractères). Dans tous les cas, on décrémente la variable partagée NBLN qui contient le nombre de lignes du texte.

3.5.4.3 Pseudo-code et table des identificateurs.

Id.	Signification	Type	Nb oct.	VI	VF	Vérif.
objets partagés						
TEXT	texte traité par l'éditeur	tableau de caractères	256	?	?	
NBLN	nombre de lignes du texte	naturel	1	1	?	
objets locaux variables						
NULI	numéro de la ligne à supprimer	naturel	1	?	?	
DERN	indice dans le texte du 16ième caractère de la dernière ligne du texte initial	naturel	1	?	?	
X	indice dans le texte du caractère à remonter	index	1	?	?	

niveau 1

. lire en décimal NULI, numéro de la ligne à supprimer
. Si1 la suppression est possible (NULI >= 1 et NBLN >= NULI)
  Alors1
    . Si2 la ligne à supprimer n'est pas la dernière (NULI <>NBLN)
      Alors2
        . remonter les lignes dont le numéro va de NULI+1 à NBLN
      Finsi2
    . décrémenter NBLN
  Finsi1

niveau 2

(début de remonter les lignes dont le numéro va de NULI+1 à NBLN)
. calculer l'indice DERN du 16ième caractère de la dernière ligne
. calculer l'indice X du 1ier caractère de la ligne de numéro NULI+1

. Itérer1
    . remonter d'une ligne le caractère d'indice X
  Sortirsi1 la dernière ligne est remontée (X = DERN)
    . incrémenter X
  Finitérer1
(fin de remonter les lignes dont le numéro va de NULI+1 à NBLN)

On obtient finalement le pseudo-code :

. lire en décimal NULI, numéro de la ligne à supprimer
. Si1 la suppression est possible (NULI >= 1 et NBLN >= NULI)
  Alors1
    . Si2 la ligne à supprimer n'est pas la dernière (NULI <> NBLN)
      Alors2
        (début de remonter les lignes dont le numéro va de NULI+1 à NBLN)
        . calculer l'indice DERN du 16ième caractère de la dernière ligne
        . calculer l'indice X du 1ier caractère de la ligne de numéro NULI+1
        . Itérer1
            . remonter d'une ligne le caractère d'indice X
          Sortirsi1 la dernière ligne est remontée (X = DERN)
            . incrémenter X
          Finitérer1
        (fin de remonter les lignes dont le numéro va de NULI+1 à NBLN)
      Finsi2
    . décrémenter NBLN
  Finsi1

3.5.4.4 Traduction du pseudo-code.

```
DEBUT     :
          JSR       LIRE EN DECIMAL
          STA       NULI
SI1       :
          LDA       NULI
          CMP       #1
          BCC       FINSI1
ETSI1     LDA       NBLN
          CMP       NULI
          BCC       FINSI1
ALORS1    NOP
SI2       :
          LDA       NULI
          CMP       NBLN
          BEQ       FINSI2
```

```
ALORS2      LDX     NBLN
            LDY     #16
            JSR     CALCULER INDICE
            STX     DERN
            LDX     NULI
            INX
            LDY     #1
            JSR     CALCULER L'INDICE
ITERER1     :
            LDA     TEXT,X
            STA     TEXT-16,X
SORTIRSI1   CPX     DERN
            BEQ     FINITERER1
            INX
            JMP     ITERER1
FINITERER1  :
FINSI2      :
            DEC     NBLN
FINSI1      :
            RTS
FIN         :
```

3.5.4.5 Programme d'essai en LISA sur APPLE IIe.

```
SUDEB :
            JSR     LDDEB
            STA     SUNULI
SUSII1 :
            LDA     SUNULI
            CMP     #1
            BCC     SUFSI1
SUESI1      LDA     VPNBLN
            CMP     SUNULI
            BCC     SUFSI1
SUALO1      NOP
SUSII2 :
            LDA     SUNULI
            CMP     VPNBLN
            BEQ     SUFSI2
SUALO2      LDX     VPNBLN
            LDY     #16
            JSR     CIDEB
            STX     SUDERN
            LDX     SUNULI
            INX
            LDY     #1
            JSR     CIDEB
SUITR1 :
            LDA     VPTEXT,X
            STA     VPTEXT-16,X
SUSSI1      CPX     SUDERN
            BEQ     SUFIT1
            INX
            JMP     SUITR1
SUFIT1 :
```

```
SUFSI2 :
         DEC      VPNBLN
SUFSI1 :
         RTS
SUNULI   DFS      1
SUDERN   DFS      1
SUFIN :
```

3.5.5 Insérer une ligne (module IN)

3.5.5.1 Fonction.

Le module IN a pour fonction d'insérer une ligne dont on donne le numéro et de déplacer éventuellement les lignes suivantes ; l'insertion dans un texte complet de 16 lignes est ignorée, l'insertion peut se faire à l'intérieur du texte ou après la dernière ligne.

3.5.5.2 Méthode de résolution.

Pour insérer une ligne à l'intérieur du texte, on commence par libérer la place de la nouvelle ligne en déplacant d'un 'cran' vers le bas les lignes suivantes, puis on insère la nouvelle ligne ; pour insérer une ligne après la dernière ligne, on ne touche pas le texte existant, on se contente d'ajouter une ligne à la fin du texte. Dans les deux cas, on incrémente la variable partagée NBLN qui contient le nombre de lignes du texte.

3.5.5.3 Pseudo-code et table des identificateurs.

Id.	Signification	Type	Nb oct.	VI	VF	Vérif.
objets partagés						
TEXT	texte traité par l'éditeur	tableau de caractères	256	?	?	
NBLN	nombre de lignes du texte	naturel	1	0	?	
objets locaux						
constante						
RC	retour chariot	caractère	1	$8D	$8D	
variables						
NULI	numéro de la ligne à insérer	naturel	1	?	?	
NUCA	numéro dans la ligne du caractère à insérer	naturel	1	1	?	

PREM	indice dans le texte du premier caractère de la ligne de numéro NULI	index	1	?	?
X	indice dans le texte du caractère à déplacer ou à insérer	index	1	?	?

niveau 1
. lire en décimal NULI, numéro de la ligne à insérer
. Si1 l'insertion est possible (NULI >= 1 et NBLN+1 >= NULI et NBLN < 16)
Alors1
. Si2 la ligne de numéro NULI existe dans le texte (NBLN >= NULI)
Alors2
. descendre les lignes dont le numéro va de NBLN à NULI
Finsi2
. saisir et insérer la ligne de numéro NULI
. incrémenter NBLN
Finsi1

niveau 2
(début de descendre les lignes dont le numéro va de NBLN à NULI)
. calculer l'indice PREM du 1^{ier} caractère de la ligne de numéro NULI
(le dernier caractère à descendre)
. calculer l'indice X du $16^{ième}$ caractère de la dernière ligne
. Itérer1
. descendre d'une ligne le caractère d'indice X
Sortirsi1 la ligne de numéro NULI est descendue (X = PREM)
. décrémenter X
Finitérer1
(fin de descendre les lignes dont le numéro va de NBLN à NULI)

(début de saisir et insérer la ligne de numéro NULI)
. aller à la ligne
. afficher en décimal NULI suivi d'un espace
. initialiser NUCA, numéro dans la ligne du caractère à saisir, à 1
. calculer l'indice X du 1^{ier} caractère de la ligne de numéro NULI
. Itérer2
. lire un caractère et l'afficher
. ranger le caractère lu dans le texte
SortirsiA2 ce caractère est un retour chariot
SortirsiB2 le caractère affiché est le $16^{ième}$ de la ligne
. incrémenter X et NUCA
Finitérer2
. Si3 le dernier caractère n'est pas un retour chariot
Alors3 . aller à la ligne
Finsi3
(fin de saisir et insérer la ligne de numéro NULI)

d'où le pseudo-code :

. lire en décimal NULI, numéro de la ligne à insérer
. Si1 l'insertion est possible (NULI >= 1 et NBLN+1 >= NULI et NBLN < 16)
Alors1
. Si2 la ligne de numéro NULI existe dans le texte (NBLN >= NULI)
Alors2
(début de descendre les lignes dont le numéro va de NBLN à NULI)
. calculer l'indice PREM du 1ier caractère de la ligne de numéro NULI
(le dernier caractère à déplacer)
. calculer l'indice X du 16ième caractère de la dernière ligne
. Itérer1
. descendre d'une ligne le caractère d'indice X
Sortirsi1 la ligne de numéro NULI est descendue (X = PREM)
. décrémenter X
Finitérer1
(fin de descendre les lignes dont le numéro va de NBLN à NULI)
Finsi2
(début de saisir et insérer la ligne de numéro NULI)
. aller à la ligne
. afficher en décimal NULI suivi d'un espace
. initialiser NUCA, numéro dans la ligne du caractère à saisir à 1
. calculer l'indice X du premier caractère de la ligne de numéro NULI
. Itérer2
. lire un caractère et l'afficher
. ranger le caractère lu dans le texte
SortirsiA2 ce caractère est un retour chariot
SortirsiB2 le caractère affiché est le 16ième de la ligne
. incrémenter X et NUCA
Finitérer2
. Si3 le dernier caractère n'est pas un retour chariot
Alors3 . aller à la ligne
Finsi3
(fin de saisir et insérer la ligne de numéro NULI)
. incrémenter NBLN
Finsi1

L'indice du 1ier caractère de la ligne de numéro NULI est évalué une première fois dans 'Alors2' puis une seconde fois après 'Finsi2'. On peut éviter cette répétition en plaçant une seule évaluation comme première action du 'Alors1' ; néanmoins, si on choisit cette solution, il est indispensable d'écrire une nouvelle version du pseudo-code avant de passer à la traduction.

3.5.5.4 Traduction du pseudo-code.

```
DEBUT     :
          JSR       LIRE EN DECIMAL
          STA       NULI
SI1       :
          LDA       NULI
          CMP       #1
          BCC       FINSI1
ETSIA1    LDY       NBLN
          INY
```

```
            CPY         NULI
            BCC         FINSI1
ETSIB1      LDA         NBLN
            CMP         #16
            BCS         FINSI1
ALORS1      NOP
SI2         :
            LDA         NBLN
            CMP         NULI
            BCC         FINSI2
ALORS2      LDX         NULI
            LDY         #1
            JSR         CALCULER INDICE
            STX         PREM
            LDX         NBLN
            LDY         #16
            JSR         CALCULER INDICE
ITERER1     :
            LDA         TEXT,X
            STA         TEXT+16,X
SORTIRSI1   CPX         PREM
            BEQ         FINITERER1
            DEX
            JMP         ITERER1
FINITERER1  :
FINSI2      :
            JSR         ALLER A LA LIGNE
            LDA         NULI
            JSR         AFFICHER EN DECIMAL
            LDA         "
            JSR         AFFICHER UN CARACTERE
            LDA         #1
            STA         NUCA
            LDX         NULI
            LDY         #1
            JSR         CALCULER INDICE
ITERER2     JSR         LIRE UN CARACTERE
            JSR         AFFICHER UN CARACTERE
            STA         TEXT,X
SORTIRSIA2  LDA         TEXT,X
            CMP         #RC
            BEQ         FINITERER2
SORTIRSIB2  LDA         NUCA
            CMP         #16
            BEQ         FINITERER2
            INX
            INC         NUCA
            JMP         ITERER2
FINITETER2  :
SI3         :
            LDA         TEXT,X
            CMP         #RC
            BEQ         FINSI3
ALORS3      JSR         ALLER A LA LIGNE
```

```
FINSI3      :
            INC         NBLN
FINSI1      :
            RTS
FIN         :
```

3.5.5.5 Programme d'essai en LISA sur APPLE IIe.

Le codage de ce module pose un problème particulier : si on applique la méthode utilisée jusqu'à présent, on obtient lors de l'assemblage le message d'erreur BRANCH TOO LONG. Ceci vient du fait que la distance dans le pseudo-code entre SI1 et FINSI1 est très grande ; en conséquence lors de l'assemblage du BCC FINSI1 cela pose un problème.

Une solution à ce problème a été présentée au chapitre 3 : elle consiste à utiliser un branchement inconditionnel à l'adresse absolue à laquelle on veut se brancher ; dans le cas présent la modification apportée au code est la suivante :

version initiale non assemblable

```
INSII1   :
         .
         BCC   INFSI1
INESA1   .
         .
         BCC   INFSI1
INESB1
         .
         .
         BCS   INFSI1
INALO1   .

INFSI1   :
```

version modifiée

```
INSII1   :
         .
         BCS   INESA1
         JMP   INFSI1
INESA1   .
               .
         BCS   INESB1
         JMP   INFSI1
INESB1   .
         .
         BCC   INALO1
         JMP   INFSI1
INALO1   .
         .
INFSI1   :
```

On obtient finalement le code ci-dessous :

```
INDEB :
           JSR       LDDEB
           STA       INNULI
INSII1 :
           LDA       INNULI
           CMP       #1
           BCS       INESA1
           JMP       INFSI1
INESA1     LDY       VPNBLN
```

```
            INY
            CPY         INNULI
            BCS         INESB1
            JMP         INFSI1
INESB1      LDA         VPNBLN
            CMP         #16
            BCC         INALO1
            JMP         INFSI1
INALO1      NOP
INSII2  :
            LDA         VPNBLN
            CMP         INNULI
            BCC         INFSI2
INALO2      LDX         INNULI
            LDY         #1
            JSR         CIDEB
            STX         INPREM
            LDX         VPNBLN
            LDY         #16
            JSR         CIDEB
INITR1  :
            LDA         VPTEXT,X
            STA         VPTEXT+16,X
INSSI1      CPX         INPREM
            BEQ         INFIT1
            DEX
            JMP         INITR1
INFIT1  :
INFSI2  :
            JSR         CROUT
            LDA         INNULI
            JSR         ADDEB
            LDA         "
            JSR         COUT
            LDA         #1
            STA         INNUCA
            LDX         INNULI
            LDY         #1
            JSR         CIDEB
INITR2      JSR         RDKEY
            JSR         COUT
            STA         VPTEXT,X
INSSA2      LDA         VPTEXT,X
            CMP         #INRC
            BEQ         INFIT2
INSSB2      LDA         INNUCA
            CMP         #16
            BEQ         INFIT2
            INX
            INC         INNUCA
            JMP         INITR2
INFIT2  :
INSII3  :
            LDA         VPTEXT,X
```

```
            CMP       #INRC
            BEQ       INFSI3
INALO3      JSR       CROUT
INFSI3 :
            INC       VPNBLN
INFSI1 :
            RTS
INRC        EQU       $8D
INNULI      DFS       1
INNUCA      DFS       1
INPREM      DFS       1
INFIN :
```

3.5.6 Calculer l'indice (module CI)

3.5.6.1 Fonction et échange d'information.

Le module CI a pour fonction de calculer l'indice dans le texte d'un caractère connaissant son numéro de ligne et son numéro dans la ligne en évaluant l'expression :

16*(numéro de ligne-1)+(numéro dans la ligne-1).

Les paramètres importés NULI et NUCA sont passés respectivement par les registres X et Y , le paramètre exporté IND est passé par le registre X.

3.5.6.2 Pseudo-code et table des identificateurs.

Id.	Signification	Type	Nb oct.	VI	VF	Vérif.
paramètres importés						
NULI	numéro de la ligne	naturel	1	?	?	
NUCA	numéro du caractère dans la ligne	naturel	1	?	?	
paramètre exporté						
IND	indice dans le texte du caractère	naturel	1	?	?	

Le traitement de ce module est simple, par conséquent le pseudo-code n'est pas détaillé.

3.5.6.3 Traduction du pseudo-code.

```
DEBUT     :
          STX       NULI
          STY       NUCA
          DEC       NULI
          LDA       NULI
          ASL
          ASL
          ASL
          ASL
          DEC       NUCA
          CLC
          ADC       NUCA
          STA       IND
          LDX       IND
          RTS
FIN       :
```

Remarquons que le traitement du module est très simple et n'utilise pas les registres X et Y, on pourrait donc, comme cela est suggéré au paragraphe 2.2.2.1, supprimer les paramètres formels NULI, NUCA et IND.

3.5.6.4 Programme d'essai en LISA sur APPLE IIe.

```
CIDEB :
          STX       CINULI
          STY       CINUCA
          DEC       CINULI
          LDA       CINULI
          ASL
          ASL
          ASL
          ASL
          DEC       CINUCA
          CLC
          ADC       CINUCA
          STA       CIIND
          LDX       CIIND
          RTS
CINULI    DFS       1
CINUCA    DFS       1
CIIND     DFS       1
CIFIN :
```

Le traitement de ce module CI est si peu important qu'on peut se poser la question de la nécessité d'en faire un module. Le seul argument valable est que le but de ce chapitre est d'illustrer la programmation modulaire et le passage des arguments.

3.5.7 Lire un nombre décimal (module LD)

3.5.7.1 Fonction et échange d'information.

Le module LD a pour fonction de lire un nombre décimal de un ou deux chiffres, le recopier en écho à l'écran et le convertir en binaire. On lit un chiffre, deux chiffres, plus de deux chiffres, et dans ce cas il faut ne tenir compte que des premiers chiffres entrés ; de plus, si l'opérateur entre des caractères autres que des chiffres, il faudra les ignorer.

Ce module est l'objet de l'exercice 4 du chapitre 4, pour le transformer en module il faut normaliser les étiquettes et les identificateurs et remplacer l'instruction BRK par RTS. L'objet NB, représentant la valeur binaire du nombre lu, devient un paramètre exporté et il est passé par l'accumulateur.

3.5.8 Afficher en décimal (module AD)

3.5.8.1 Fonction et échange d'information.

Le module AD a pour fonction de convertir de binaire en décimal un nombre entier naturel inférieur ou égal à $(99)_{10}$ et de l'afficher.

Le paramètre importé NBR, représentant le nombre à convertir et à afficher, est passé par l'accumulateur.

3.5.8.2 Méthode de résolution.

On va afficher du DCB. Attention : il ne faut pas confondre le code DCB avec le code Binaire par exemple $(98)_{10} = (01100010)_2 = (10011000)_{DCB}$

On calcule le nombre de dizaines puis le nombre d'unités ; on détermine ensuite la valeur Décimale Codée Binaire en codant le chiffre des dizaines sur les 4 bits de poids forts et le chiffre des unités sur les 4 bits de poids faibles.

3.5.8.3 Pseudo-code et table des identificateurs.

Id.	Signification	Type	Nb oct.	VI	VF	Vérif.
paramètre importé						
NBR	initialement le nombre à convertir et à afficher finalement le nombre d'unités	naturel (<=99)	1	?	?	
objet local						
DIZ	nombre de dizaines	naturel	1	0	?	I U C

niveau 1

. recopier la valeur de l'accumulateur dans NBR
. calculer le nombre de dizaines DIZ et le nombre d'unités
. déterminer la valeur Décimale Codée Binaire du nombre à traiter
. afficher la valeur Décimale Codée Binaire du nombre à traiter

niveau 2

(début de calculer le nombre de dizaines DIZ et le nombre d'unités)
. initialiser DIZ, nombre de dizaines, à 0
. Itérer1
Sortirsi1 NBR est strictement inférieur à 10_{10}
. retrancher 10_{10} à NBR
. incrémenter DIZ
Finitérer1
(fin de calculer le nombre de dizaines DIZ et le nombre d'unités)

(début de déterminer la valeur Décimale Codée Binaire du nombre à traiter)
. placer le chiffre des dizaines sur les quatre bits de poids forts
. placer le chiffre des unités sur les quatre bits de poids faibles
(fin de déterminer la valeur Décimale Codée Binaire du nombre à traiter)

d'où le pseudo-code :

. recopier la valeur de l'accumulateur dans NBR
(début de calculer le nombre de dizaines DIZ et le nombre d'unités)
. initialiser DIZ, nombre de dizaines, à 0
. Itérer1
Sortirsi1 NBR est strictement inférieur à 10_{10}
. retrancher 10_{10} à NBR
. incrémenter DIZ
Finitérer1
(fin de calculer le nombre de dizaines DIZ et le nombre d'unités)

(début de déterminer la valeur Décimale Codée Binaire du nombre à traiter)
. placer le chiffre des dizaines sur les quatre bits de poids forts
. placer le chiffre des unités sur les quatre bits de poids faibles
(fin de déterminer la valeur Décimale Codée Binaire du nombre à traiter)
. afficher la valeur Décimale Codée Binaire du nombre à traiter

3.5.8.4 Traduction du pseudo-code.

```
DEBUT      :
           STA      NBR
           LDA      #0
           STA      DIZ
ITERER1    :
SORTIRSI1  LDA      NBR
           CMP      #10
           BCC      FINITERER1
           LDA      NBR
           SEC
           SBC      #10
           STA      NBR
           INC      DIZ
           JMP      ITERER1
FINITERER1 :
           LDA      DIZ
           ASL
           ASL
           ASL
           ASL
           ORA      NBR
           JSR      AFFICHER EN HEXADECIMAL
           RTS
FIN        :
```

3.5.8.5 Programme d'essai en LISA sur APPLE IIe.

```
ADDEB :
           STA      ADNBR
           LDA      #0
           STA      ADDIZ
ADITR1 :
ADSSI1     LDA      ADNBR
           CMP      #10
           BCC      ADFIT1
           LDA      ADNBR
           SEC
           SBC      #10
           STA      ADNBR
           INC      ADDIZ
           JMP      ADITR1
ADFIT1 :
```

```
              LDA       ADDIZ
              ASL
              ASL
              ASL
              ASL
              ORA       ADNBR
              JSR       PRBYTE
              RTS
    ADNBR     DFS       1
    ADDIZ     DFS       1
    ADFIN :
```

3.5.9 Afficher la ligne de commande (module AC)

3.5.9.1 Fonction.

Le module AC a pour fonction d'afficher, sur la première ligne de l'écran, la ligne des commandes disponibles suivie d'une ligne vide.

3.5.9.2 Pseudo-code et table des identificateurs.

Id.	Signification	Type	Nb oct.	VI	VF	Vérif.
objets locaux						
LICO	libellé de la ligne de commande	tableau de caractères		?	?	

On suppose que le premier élément du tableau LICO contient la longueur de la chaine à afficher.

Id.	Signification	Type	Nb oct.	VI	VF	Vérif.
X	indice du caractère à afficher	index	1	1	?	

niveau 1

. effacer l'écran
. ***afficher le libellé de la ligne de commande***
. aller à la ligne
. aller à la ligne pour afficher une ligne vide

niveau 2

(début de afficher le libellé de la ligne de commande)
. initialiser X, indice du caractère à afficher, à 1
. Itérer1
    . afficher le caractère d'indice X
 Sortirsi1 le dernier caractère est affiché
    . incrémenter X
 Finitérer1
(fin de afficher le libellé de la ligne de commande)

d'où le pseudo-code :

. effacer l'écran
(début de afficher le libellé de la ligne de commande)
. initialiser X, indice du caractère à afficher, à 1
. Itérer1
    . afficher le caractère d'indice X
 Sortirsi1 le dernier caractère est affiché
    . incrémenter X
 Finitérer1
(fin de afficher le libellé de la ligne de commande)
. aller à la ligne
. aller à la ligne pour afficher une ligne vide

3.5.9.3 Traduction du pseudo-code.

```
DEBUT      :
           JSR      EFFACER L'ECRAN
           LDX      #1
ITERER1    :
           LDA      LICO,X
           JSR      AFFICHER UN CARACTERE
SORTIRSI1  CPX      LICO
           BEQ      FINITERER1
           INX
           JMP      ITERER1
FINITERER1 :
           JSR      ALLER A LA LIGNE
           JSR      ALLER A LA LIGNE
           RTS
FIN        :
```

3.5.9.4 Programme d'essai en LISA sur APPLE IIe.

```
ACDEB :
            JSR         HOME
            LDX         #1
ACITR1 :
            LDA         ACLICO,X
            JSR         COUT
ACSSI1      CPX         ACLICO
            BEQ         ACFIT1
            INX
            JMP         ACITR1
ACFIT1 :
            JSR         CROUT
            JSR         CROUT
            RTS
ACLICO      STR         "A(ff S(up I(ns Q(uit"
ACFIN :
```

3.5.10 Variables partagées (VP)

On termine cette application par le module VP qui contient les déclarations de variables partagées.

3.5.10.1 Table des identificateurs.

Id.	Signification	Type	Nb oct.	VI	VF	Vérif.
objets partagés						
TEXT	texte traité par l'éditeur	tableau de caractères	256	?	?	
NBLN	nombre de lignes du texte	naturel	1	?	?	

3.5.10.2 Programme d'essai en LISA sur APPLE IIe.

```
VPDEB :
VPTEXT      EQU         $1000
VPNBLN      DFS         1
VPFIN :
            END
```

ANNEXE
LISTE DES ABREVIATIONS POUR LES MOTS-CLES

SII	Si
ALO	Alors
SNO	Sinon
FSI	Finsi
ESI	Etsi
ESA, ESB, ESC,...	EtsiA, EtsiB, EtsiC,...
OSI	Ousi
OSA, OSB, OSC,...	OusiA, OusiB, OusiC,...
ITR	Itérer
SSI	Sortirsi
SSA, SSB, SSC,...	SortirsiA, SortirsiB, SortirsiC,...
FIT	Finitérer
CAS	Casoù
CSA, CSB, CSC,...	CasA, CasB, CasC,...
ACS	Autrescas
FCA	Fincas
DEB	Début
FIN	Fin

INDEX ALPHABETIQUE

Le premier chiffre indique le numéro du chapitre.

automate d'états finis 1.4 + 4.1 + 5.3.3
cahier des charges 5.3.1
codage 3.3
commentaires 3.4.3
communication entre modules 5.2
comparaisons numériques 2.3.5
conditions composées 3.5.2
décomposition modulaire 5.3.4
déplacement 2.3.7
documentation 1.1
étiquettes 3.Intro + 3.3 + Annexe
identificateurs 1.2.7 + 2.2.4 + 5.1
identificateurs (table des ...) 1.2 + 1.2.3 + 1.2.4 + 1.2.5
indicateurs 2.3.2
langage (assembleur, machine) 2.Intro.
méthode (évaluation de la ...) 3.4
méthode (principe de la ...) 3.Intro.
mode décimal 2.3.6
module 5.1
module (appel d'un ...) 2.3.1
mots-clés 3.3
nombres naturels 2.3.5
nombres relatifs 2.3.5
objets locaux 5.1
objets partagés 5.1 + 5.2.1 + 5.3.5.10
octet (contenu d'un ...) 2.2.1
passage de paramètres 5.2.2
pile du 6502 2.2.3
programmation modulaire 5.Intro.
pseudo-code 1.1
séquence 1.3.1
structures alternatives 1.3.2 + 3.1
  .simplifiée 3.1.1
  .complète 3.1.2
  .cas-où 3.5.3
structure des données 2.2 + 5.3.2
structure des traitements 1.3 + 2.3
structure multiple 3.2
structures répétitives 1.3.3 + 1.3.4
  .itérer 3.1.5 + 3.5.1
  .pour 3.1.6
  .répéter 3.1.4
  .tantque 3.1.3
tableaux 2.2.2
types 2.2.4
variables partagées 5.2.1

PROGRAMMATION STRUCTURÉE EN ASSEMBLEUR 6502

Par J.-P. MALENGÉ, L. ANDRÉANI, P. COLLARD

Le but de cet ouvrage est de montrer qu'il est parfaitement possible de normaliser la programmation en assembleur de façon à concevoir des programmes lisibles qui puissent être améliorés et maintenus par n'importe quel membre d'une équipe et non plus uniquement par celui qui les a écrits.

Après deux chapitres de rappels sur la programmation structurée et le langage assembleur, le chapitre 3 présente les bases de la programmation structurée du 6502 sous forme d'une suite d'exemples. Les problèmes abordés au chapitre 4 expliquent en détail les routines de base de la programmation en assembleur (sur Apple II et avec l'assembleur LISA). Le dernier chapitre fournit au lecteur la possibilité de se lancer dans une application pratique en écrivant un petit éditeur de texte.

Cet ouvrage s'adresse à un public composé d'étudiants et de professionnels de l'informatique. Il a été conçu et rédigé par une équipe d'enseignants du département informatique de l'IUT de Nice.

ISBN : 2-225-81114-8