2

MICRO APPLICATION

LE LIVRE DU LANGAGE MACHINE DE L'ATARI ST

ATARI

UN LIVRE DATA BECKER

Suivant la puissance de l'assembleur, le programmeur dispose d'un nombre plus ou moins grand d'opérateurs et de fonctions pour former un valeur donnée. Il ne s'agit là en aucun cas d'opérations que l'assembleur traduirait en instructions machine. Ces opérations sont utilisées exclusivement pour calculer les opérandes d'instructions machine.

Exemples: 10+$0A
LIGNE+1
NOT 10

En principe, les assembleurs 68000 vous offrent pour former des expressions toutes les opérations de base ainsi que certaines opérations logiques.

Symboles et constantes système

Nous avons déjà souvent évoqué l'emploi des symboles et des labels. Nous allons maintenant vous présenter les règles de syntaxe de la programmation assembleur symbolique. L'assembleur doit pouvoir distinguer clairement les symboles du reste du texte source. C'est pourquoi le programmeur doit respecter certaines limites dans la création des symboles. En général, les symboles se composent d'une série continue de LETTRES, CHIFFRES et quelques CARACTERES SPECIAUX. Les espaces sont interdits. Les symboles doivent en effet le plus souvent être séparés des autres parties du texte source par des espaces. Les espaces peuvent cependant être en général négligés quand la séparation est réalisée de façon nette par d'autres caractères, par exemple par des opérateurs mathématiques. Un symbole ne doit jamais commencer par un chiffre pour que la distinction entre symboles et constantes puisse se faire. Il est interdit d'utiliser des NOMS RESERVES comme symboles mais la plupart des assembleurs permettent qu'un nom réservé soit contenu dans un symbole.

2

MICRO APPLICATION

LE LIVRE DU LANGAGE MACHINE DE L'ATARI ST

ATARI

UN LIVRE DATA BECKER

Distribué par : MICRO APPLICATION
13, Rue Sainte Cécile
75009 PARIS

et

EDITION RADIO
3, Rue de l'Eperon
75006 PARIS

ISBN : 2-86899-030-4

Traduction Française assurée par Pascal HAUSMANN

édité par Frédérique BEAUDONNET

Table des matières

PREFACE

Le présent ouvrage est conçu particulièrement pour les débutants qui souhaitent s'attaquer au langage machine. Nous partons cependant du principe qu'ils maîtrisent déjà les bases d'un langage de programmation. Mais comme une bonne connaissance de la structure et du mode de fonctionnement d'un ordinateur est nécessaire pour programmer en langage machine, nous avons consacré une partie de cet ouvrage à l'explication de ces notions de base.

Les chapitres qui présentent des exemples pratiques en langage machine prennent relativement peu de place bien qu'ils traitent de ce qui est au fond le sujet de cet ouvrage. Nous n'avons cependant pas cherché à donner le plus grand nombre possible d'exemples pratiques mais bien plutôt à fournir une introduction systématique à la programmation en langage machine. C'est pourquoi nous avons en quelque sorte "développé" nos exemples dans le cours de l'ouvrage.

En dernière analyse, la programmation en assembleur ne se distingue pas fondamentalement de la programmation dans un langage évolué. Les instructions de l'assembleur et donc du microprocesseur sont seulement beaucoup moins "puissantes" que celles d'un langage évolué. Cela signifie qu'il faut plus d'instructions pour un programme en assembleur que pour le programme équivalent écrit par exemple en Logo. Les programmes assembleur sont par contre en général plus rapides et permettent de résoudre des problèmes qui sont insolubles avec un langage évolué.

Cet ouvrage a été écrit spécialement pour l'Atari ST. C'est pourquoi les exemples fournis peuvent être exécutés directement sur cette machine.

Il peut arriver cependant si vous utilisez d'autres utilitaires de programmation en assembleur que nous, que vous ayez à modifier légèrement les programmes (adresse de départ, etc.).
Vous trouverez toutes les informations nécessaires à ce sujet dans le manuel d'utilisation de votre assembleur. Par ailleurs, nous vous recommandons de faire exécuter les exemples comme application TOS.

"Last but not least", nous souhaitons remercier chaleureusement tous ceux qui nous ont aidés dans la réalisation et la correction de cet ouvrage. Nous souhaitons nommer tout particulièrement Andreas Lucht qui nous a permis d'effectuer certaines corrections.

Berlin, Août 1985

Bernd Grohmann
Petra Seidler
Harald Slibar

Nous souhaitons encore signaler que de nombreux termes tels que "GEM" (Digital Research) sont protégés et ne peuvent donc être utilisés librement.

BASES DU TRAITEMENT ELECTRONIQUE DES DONNEES

1) Introduction

2) Représentation des données

3) Opérations logiques et manipulation de bits

4) Processus de réalisation d'un programme

Ce chapitre a pour objet de décrire les bases de la programmation. Le lecteur déjà familiarisé avec ces notions peut donc le lire en diagonale. Nous pensons cependant que même les lecteurs expérimentés peuvent y découvrir des sujets très intéressants.

Qu'est-ce que la programmation?

On recherche une solution à un problème donné. Une fois celle-ci trouvée, on essaiera une procédure permettant par étapes successives d'arriver à la solution générale du problème. Cette procédure étape par étape est appelée ALGORITHME. On essaie souvent également de décomposer d'abord le problème global en problèmes partiels significatifs, clairs et limités qui seront faciles ou du moins plus faciles à résoudre.

Un algorithme constitue une procédure d'après laquelle un problème peut être résolu étape par étape. Cet algorithme ne peut bien sûr se composer que d'un nombre fini d'étapes. L'algorithme peut être représenté dans n'importe quel langage ou symbolique. Voici un exemple simple d'algorithme:

1.Allumer le lecteur de cassette
2.Introduire la cassette dans le lecteur
3.Sélectionner le volume voulu
4.Mettre le lecteur de cassette sur "marche"

Dès lors que la solution d'un problème existe sous la forme d'un algorithme, on peut la traduire dans un langage ou symbolique compréhensible par l'ordinateur. Le français ou d'autres langages "naturels" ne conviennent pas à l'écriture d'un programme. La raison en est que toute langue naturelle comprend de nombreuses ambiguïtés syntaxiques que l'ordinateur ne peut comprendre. On peut cependant créer un langage artificiel dont les concepts soient bien définis.

Les concepts peuvent aussi être dérivés d'un langage naturel pour être plus aisément compréhensibles par l'homme. Un tel langage artificiel est appelé LANGAGE DE PROGRAMMATION.

L'ordinateur ne peut toutefois normalement pas comprendre un langage de programmation. L'ordinateur ne maîtrise qu'un langage qu'on appelle LANGAGE MACHINE. Des programmes auxiliaires sont donc encore nécessaires pour traduire un programme écrit dans un langage de programmation dans le langage machine de l'ordinateur.

Il est cependant également possible de convertir l'algorithme directement en langage machine. Comme la représentation interne des instructions en langage machine est très difficile à maîtriser, on a créé ce qu'on appelle des instructions assembleur. Chaque instruction assembleur correspond exactement à une instruction langage machine. Avec un programme d'assembleur (en jargon technique, nous dirons: avec un assembleur), le programme écrit en instructions assembleur est traduit en langage machine. Nous expliquerons dans les chapitres suivants les autres services que peut rendre un assembleur.

REPRESENTATION DES DONNEES

Tout programme a à traiter des données, à les interpréter ou à les produire d'une manière ou d'une autre. C'est pourquoi les données doivent être représentées sous une forme compréhensible et utilisable par l'ordinateur. Bien sûr, les données de programmes utilitaires peuvent également être traduites sous une telle forme.

Représentation de données numériques:

Pour comprendre la représentation des données numériques dans l'ordinateur, il est utile de commencer par se pencher sur le "mode de fonctionnement" de notre représentation des nombres. C'est pourquoi nous allons décrire ici d'abord le système décimal.

Dans le système décimal, un nombre est exprimé par une série de chiffres. Le nom de système décimal signifie que ce système repose sur l'emploi de dix chiffres différents: 0, 1, 2, ... 8, 9. Un chiffre a une valeur qui est fonction de son emplacement dans le nombre. Le principe est que la valeur d'un chiffre en n-ième position dans un nombre est dix fois plus grande que la valeur du même chiffre en (n-1)-ième position. "1" a par exemple dans le nombre "1000" une valeur dix fois supérieure à celle de "1" dans le nombre "0100".

Le nombre 12345 est donc une abréviation de l'expression:

1 * 10000 + 2 * 1000 + 3 * 100 + 4 * 10 + 5 * 1

ou encore:

1 * 10^4 + 2 * 10^3 + 3 * 10^2 + 4 * 10^1 + 5 * 10^0

Ce système dans lequel la valeur d'un chiffre dépend de son emplacement dans le nombre ne va d'ailleurs absolument pas de soi. Son adoption a constitué une véritable révolution scientifique alors qu'il était par exemple ignoré des romains dont le système utilisait un chiffre différent pour représenter une valeur différente.

Le système de chiffres à "valeur d'emplacement" a des avantages considérables pour les opérations de calcul. Du fait que chaque chiffre décrit une zone particulière du nombre, il est possible de travailler chiffre par chiffre dans les opérations de calcul.

Une addition peut par exemple être ainsi effectuée:

```
  235
+582
-----
  817
```

On additionne tout d'abord la colonne de droite: 5 + 2 = 7.

Vient ensuite la colonne du milieu: 3 + 8 = 11. Cela peut également être représenté comme "1" avec une retenue de "1". La retenue sera prise en compte lors de l'addition de la colonne suivante: 2 + 5 + 1 = 8.

Cette procédure peut facilement être formulée comme algorithme:

1. Additionne la première colonne avec retenue
2. Additionne la colonne suivante avec, le cas échéant, la retenue (si nécessaire, produire à nouveau une retenue).
3. Répéter 2. jusqu'à ce que toutes les colonnes aient été traitées.

Cet algorithme ne prévoit pas le cas où une retenue se produirait lors de l'addition de la dernière colonne. Cela peut être par exemple obtenu de la façon suivante:

4. Si une retenue est apparue dans la dernière colonne, allonger le résultat d'un chiffre dans lequel s'écrira la retenue.

Si l'extension d'un chiffre n'est pas possible (ce qui sera souvent le cas), on peut également provoquer l'affichage d'une erreur. En faisant exécuter cet algorithme plusieurs fois successivement, on peut additionner plusieurs nombres.

Il conviendrait maintenant que nous nous demandions si la procédure choisie est au moins valable mathématiquement, c'est-à-dire si elle conduit toujours au bon résultat. Pour ce petit exemple, on voit bien que le résultat obtenu était correct. Il est cependant possible de démontrer la justesse de l'addition colonne par colonne:

$$235 = 2*10^2 + 3*10^1 + 5*10^0$$
$$582 = 5*10^2 + 8*10^1 + 2*10^0$$

$$235 + 582 =$$

$$(2*10^2 + 3*10^1 + 5*10^0)$$
$$+ \quad (5*10^2 + 8*10^1 + 2*10^0) =$$
$$(2+5)*10^2 + (3+8)*10^1 + (5+2)*10^0 =$$
$$7*10^2 + 11*10^1 + 7*10^0 =$$
$$7*10^2 + 1*10^2 + 1*10^1 + 7*10^0 =$$
$$8*10^2 + 1*10^1 + 7*10^0 =$$

817

Notre démonstration utilise notamment les lois de commutativité et d'associativité. Notre démonstration ne vaut cependant que pour cet exemple mais la démonstration générale pourrait être formulée de façon analogue.

Dans la pratique, on ne commence évidemment pas par démontrer mathématiquement chaque procédure avant de l'appliquer. De nombreux algorithmes parfaitement corrects ne peuvent d'ailleurs pas être démontrés du fait de leur trop grande complexité. Ou pour être plus précis: personne n'a jusqu'ici réussi (ou ne s'est essayé) à apporter une démonstration de ces algorithmes.

Il peut souvent être judicieux de contrôler la justesse mathématique d'un algorithme. On peut alors souvent découvrir à cette occasion des erreurs (par exemple des situations exceptionnelles).

Une multiplication peut être effectuée de manière semblable. La procédure en est bien connue:

```
              243 * 103
--------------------
                  729
                    0
+               243
--------------------
                25029
```

Avec cet algorithme on effectue d'abord des multiplications colonne par colonne. Les résultats de ces multiplications sont ensuite additionnés en tenant compte du poids de l'emplacement dans lequel chaque facteur figurait.

Il convient de noter le cas des multiplications partielles avec "0" et "1". Pour la multiplication partielle avec "0", on écrit simplement un "0". Pour la multiplication partielle avec "1", on écrit le facteur d'origine "243".

Cette brève introduction avait simplement pour but de décrire le système décimal de façon à pouvoir établir plus tard des analogies entre les systèmes de calcul des différents systèmes numériques.

Comment les données sont-elles réprésentées dans un ordinateur?

Il existe différents principes permettant de représenter les données. On pourrait affecter à chaque nombre une tension proportionnelle à sa valeur: 1.23 serait donc représenté par une tension de 1.23 volts. Ce principe est utilisé dans les ordinateurs analogiques.
L'inconvénient est évident: tous les composants de calcul ou de mémoire doivent travailler avec une extraordinaire précision. Pour obtenir une précision de calcul de trois chiffres seulement, ces composants devraient déjà avoir une marge d'imprécision inférieure à un millième. La zone de nombres décrite ne pourrait pas d'autre part être étendue à volonté. Pour une précision de 4 chiffres, il faudrait déjà pouvoir reconnaître et traiter avec précision des tensions allant de 0,01 volt à 10 volt. Un autre inconvénient des ordinateurs analogiques est qu'ils sont beaucoup plus difficiles à programmer et qu'ils ne peuvent en fait traiter que des valeurs numériques.

Pour représenter les données dans un ordinateur, il faut donc choisir un système numérique approprié, avec une écriture à valeur d'emplacement. On pourrait en principe utiliser le système décimal. Mais le problème qui se pose alors immédiatement est de savoir comment représenter un chiffre. Le problème est toujours de pouvoir distinguer entre les dix différents chiffres.

Remarquons ici qu'il est parfaitement possible de calculer quel nombre de possibilités par chiffre permet d'obtenir le système numérique le plus efficace. Des personnes très savantes l'ont fait et elles ont abouti à 2,7. En arrondissant, on obtiendrait donc 3. On disposerait donc des chiffres 0, 1 et 2. Mais comme il est plus simple et plus sûr de s'en tenir à seulement deux possibilités pour chaque chiffre, on a opté pour le système binaire.

En système binaire, chaque chiffre ne peut revêtir que deux états différents, "0" et "1". Ces deux états peuvent être très facilement représentés, par exemple par:

"0"	"1"
Absence de tension	Tension
Pas de courant	Passage du courant
Commutateur ouvert	Commutateur fermé
Lampe éteinte	Lampe allumée

Chiffre binaire se dit BINARY DIGIT en anglais, d'où l'abréviation BIT qui est généralement employée pour désigner un chiffre pouvant avoir deux valeurs différentes, "0" ou "1".

Comme en système décimal, les nombres plus grands sont exprimés par l'écriture à valeur d'emplacement. On place simplement plusieurs bits les uns à la suite des autres. La base du système binaire est bien sûr deux et non dix comme en système décimal.

Le nombre binaire "0101" est donc une abréviation de:

0*2^3 + 1*2^2 + 0*2^1 + 1*2^0

Le principe est exactement le même que pour le système décimal mais comme la base est "2", on a "2^x" au lieu de "10^x" comme en système décimal. Le nombre binaire "0101" peut être ainsi directement converti en système décimal:

0*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 5

Les nombres binaires plus importants peuvent bien sûr également être convertis en nombres décimaux, prenons par exemple le nombre "01101110":

```
        0 * 2^7  =  0 * 128  =    0
+       1 * 2^6  =  1 *  64  =   64
+       1 * 2^5  =  1 *  32  =   32
+       0 * 2^4  =  0 *  16  =    0
+       1 * 2^3  =  1 *   8  =    8
+       1 * 2^2  =  1 *   4  =    4
+       1 * 2^1  =  1 *   2  =    2
+       0 * 2^0  =  0 *   1  =    0
-----------------------------------
                              110
```

Lorsqu'une confusion sera possible entre les différents systèmes numériques, nous désignerons les nombres binaires en les faisant précéder d'un signe "%" (par exemple %01101110 = 110). Les nombres décimaux ne seront marqués par aucun signe.

Un nombre décimal peut bien sûr être converti en un nombre binaire; prenons à nouveau l'exemple de 110. On peut utiliser à cet effet la procédure suivante:

```
110 / 2 = 55Reste: 0
 55 / 2 = 27Reste: 1
 27 / 2 = 13Reste: 1
 13 / 2 =  6Reste: 1
  6 / 2 =  3Reste: 0
  3 / 2 =  1Reste: 1
  1 / 2 =  0Reste: 1
```

La colonne des restes, lue de bas en haut, donne le nombre binaire recherché, ici %1101110. %1101110 = %01101110. Nous avons ainsi réussi le test pour notre première conversion.

Avec les nombres binaires, on peut utiliser les mêmes procédures qu'avec les nombres décimaux. Il faut simplement toujours tenir compte du fait qu'on ne dispose que de deux chiffres. Il y a donc retenue dès qu'on dépasse "1" et non lorsqu'on dépasse "9".

Une addition de nombres binaires se présente donc ainsi:

```
    0110
+   1011
---------
   10001          ( 6 + 11 = 17 )
```

Nous vous conseillons maintenant de prendre vous-même différents exemples et d'effectuer des calculs en système binaire ainsi que des conversions entre les systèmes décimal et binaire, dans les deux sens.

La multiplication de nombres binaires s'effectue également d'après le même principe que la multiplication de nombres décimaux. Mais comme il n'y a que les chiffres "0" et "1", on additionne toujours soit zéro, soit le facteur, décalé. Nous avions déjà rencontré ce principe pour la multiplication de nombres décimaux.

Calculons à titre d'exemple %0110 * %1011:

```
      0110  *  1011
--------------------
               0110
              0110
.              0
      +      0110
      -------------
             1000010              ( 6 * 11 = 66 )
```

Une soustraction s'effectue également de la manière habituelle, compte tenu de la différence de base. Une autre possibilité d'effectuer la soustraction consiste à additionner le nombre négatif. Il faut cependant pour cela se demander d'abord comment représenter de façon judicieuse un nombre négatif en binaire. Faisons donc le raisonnement suivant:

Si on ajoute %1 au nombre %1111 sans tenir compte de la retenue, on obtient %0000. Par conséquent, par le raisonnement inverse, en ôtant %1 de %0000, nous obtiendrions %1111.

%1111 correspond donc à "-1". On appelle d'ailleurs le bit de plus grande valeur bit de signe. Si ce bit vaut "1", on a un nombre négatif, s'il vaut "0", on a un nombre positif.

L'important lorsqu'on effectue des calculs avec des nombres binaires est de veiller à ce que les deux nombres aient le même nombre de chiffres (ou bits).

Voici un exemple de nombres positifs et négatifs en écriture binaire:

%1100	%1101	%1110	%1111	%0000	%0001	%0010	%0011
-4	-3	-2	-1	0	1	2	3

Si vous comptez, vous constaterez maintenant que pour un nombre donné de bits, il y a toujours un nombre négatif de plus que les nombres positifs. Quatre bits permettent de représenter les nombres -8 à 7, huit bits les nombres -128 à 127.

Un nombre négatif binaire peut être aisément calculé à partir d'un nombre négatif décimal. Le nombre positif correspondant doit d'abord être converti en un nombre binaire. Tous les bits du nombre binaire ainsi obtenu doivent alors être inversés, "1" devenant "0" et "0" "1". Cette méthode est appelée formation du "complément à 1". On additionne enfin "1" à ce complément à 1. Le résultat est un nombre binaire négatif appelé COMPLEMENT A DEUX.

Exemple:

-5 :	5 =	%0101	
Complément à un de	%0101 =		%1010
	+%0001 =		%1011
Donc:	-5 =		%1011

```
5 - 4 :                    3 - 6 :

(5)        %0101           (3)        %0011
(-4)     + %1100           (-6)     + %1010
---------------            ---------------
(1)        0001            (-3)       1101
```

Si l'on ne prend pas en compte la retenue lors de l'addition de 1 (exemple 5-4), on obtient visiblement le bon résultat. Avec ce mode de calcul, il est toutefois nécessaire de délimiter la zone de nombres concernée. Comme la zone de nombres utilisée est toujours limitée dans les microordinateurs, cela n'est pas gênant dans la pratique.

Sans apporter de démonstration systématique, nous nous contenterons de constater ici que ce mode de représentation fonctionne et qu'il permet d'additionner et de soustraire correctement (par addition du nombre négatif correspondant) des nombres avec signe.

Pour convertir un nombre binaire de 4 bits en un nombre binaire de 8 bits, il faut tenir compte de la valeur du bit de plus grande valeur (le bit de signe). Si ce bit vaut 0, les quatre bits supérieurs du nombre binaire de 8 bits seront remplis par "0000", sinon par "1111".

Exemples:

Nombre binaire 4 bits	Nombre binaire 8 bits
%0110	%00000110
%1111	%11111111
%1001	%11111001
%0100	%00000100

Lors de la conversion de nombres binaires de 8 bits en nombres binaires de 4 bits, il suffit de supprimer les quatre bits supérieurs. Il faut toutefois vérifier auparavant si l'intervale des nombres de 4 bits suffit à représenter le nombre à convertir.

Ce principe ne vaut naturellement pas uniquement pour les nombres binaires de 4 et 8 bits. Il peut être étendu à toutes les autres combinaisons.

Il est évident que la représentation des nombres sous forme de nombres binaires est peu intelligible. Difficile de se représenter à quoi correspond par exemple %01101110 alors que nous voyons tout de suite ce que représente le nombre décimal équivalent 110. En effet pour nous êtres humains, peu importe en réalité à quel système numérique un nombre appartient, c'est toujours son nombre de chiffres plus ou moins important qui détermine si nous sommes capable de le retenir, d'y associer certaines choses.

C'est pourquoi il est utile de réunir plusieurs chiffres binaires entre eux pour obtenir une meilleure représentation et une meilleure compréhension (par les hommes). En réunissant entre eux plusieurs chiffres binaires, l'ordinateur pourra continuer à travailler en binaire mais nous pourrons utiliser de façon externe des nombres comportant moins de chiffres et donc plus faciles à saisir.

En réunissant des chiffres binaires, il n'est malheureusement pas possible de retomber sur le système décimal qui nous est si familier. En effet, dix n'est pas une puissance entière de 2. Or toutes les possibilités binaires représentées par les chiffres à réunir doivent trouver un répondant dans un nouveau chiffre. Si l'on réunissait trois bits, toutes les combinaisons des trois bits pourraient être exprimées par les chiffres 0 à 7. Comme les chiffres 8 et 9 ne seraient pas alors utilisés, on ne pourrait évidemment pas employer des nombres décimaux normaux.

Si l'on réunissait par contre quatre bits, les combinaisons de bits %0000 à %1001 pourraient être représentées par les chiffres 0 à 9 mais les autres combinaisons n'auraient pas de répondant en système décimal.

Il est ainsi évident que nous ne pouvons nous en tenir au systèmes numériques qui nous sont déjà connus. Il nous faut trouver comme base du système numérique recherché un nombre qui soit une puissance de deux. Historiquement, deux nombres ont été envisagés: 8 et 16.

Si l'on veut utiliser le "8", il faut réunir trois bits puisque trois bits offrent 2^3 = 8 possibilités de combinaison. Le système numérique ainsi obtenu est appelé système octal.
Sur les anciens (et très gros) ordinateurs, ce système était très répandu. La raison en est entre autre que les chiffres du système octal ne constituent qu'un sous-ensemble des chiffres du système décimal. C'est pourquoi il n'est pas nécessaire d'introduire de nouveaux chiffres et d'autre part n'importe quelle imprimante peut sortir les chiffres décimaux (par exemple les imprimantes des calculatrices de bureau) donc aussi les chiffres octaux. Le tableau suivant présente les équivalences entre chiffres binaires, octaux et décimaux:

Décimal	Binaire	Octal
0	000	0
1	001	1
2	010	2
3	011	3
4	100	4
5	101	5
5	110	6
7	111	7
8	1000	10
10	1010	12
15	1111	17
16	10000	20

En système octal, les calculs s'effectuent comme dans les autres systèmes numériques. Le lecteur intéressé peut fort bien essayer de faire quelques exercices dans ce système. Pour contrôler les résultats obtenus, il suffit de les convertir d'abord en système binaire puis en système décimal et inversement.

L'inconvénient principal du système octal tient à ce qu'il n'est pas possible de sauvegarder n'importe quel nombre décimal dans le champ prévu pour un nombre octal. Pour représenter les chiffres "8" et "9", un second chiffre octal serait nécessaire.

On n'a plus cet inconvénient si l'on réunit toujours quatre bits. Comme quatre bits offrent 2^4 = 16 combinaisons, la base du système ainsi obtenu sera 16. Ce système numérique devrait normalement s'appeler "SYSTEME SEDECIMAL" mais ce terme a été évincé par le terme américain "HEXADECIMAL".

Pour représenter les seize combinaisons de ce système il faut employer, outre les dix chiffres "normaux", six nouveaux caractères. On a tout simplement eu recours pour cela aux six premières lettres de l'alphabet a...f. Voici maintenant une table d'équivalence des différents systèmes:

Décimal	Binaire	Octal	Hexadécimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
23	10111	27	17
24	11000	30	18
32	100000	40	20

La conversion du système hexadécimal dans le système décimal se fait selon le même principe que la conversion du système binaire en système décimal. Mais comme on a très souvent recours à cette conversion, nous allons en donner ici un exemple:

Le nombre hexadécimal "C57A" est une abréviation de:

C*16^3 + 5*16^2 + 7*16^1 + A*16^0

soit:

12*16^3 + 5*16^2 + 7*16^1 + 10*16^0 = 50554

Il y a deux possibilités pour convertir dans l'autre sens. Le plus simple est de convertir le nombre décimal en un nombre binaire puis de convertir ce nombre binaire en hexadécimal. Cette procédure nécessite toutefois un nombre relativement important d'étapes intermédiaires.

La seconde procédure est plus compliquée mais elle mène plus rapidement au résultat recherché:

> Le nombre à convertir est divisé par la plus grande puissance de 16 inférieure à ce nombre. Le reste est alors multiplié par 16 autant de fois que l'implique la puissance obtenue.

Convertissons par exemple le nombre 50554:

La plus grande puissance de 16 inférieure à 50554 est 16^3 = 4096, 16^4 vaut en effet déjà 65536.

50554 / 4096 = 12,34228516 ==: 12 (déc) = C (hexa)
0,342285 * 16 = 5,476562499 ==: 5 (déc) = 5 (hexa)
0,47656 * 16 = 7,624999987 ==: 7 (déc) = 7 (hexa)

0,625 * 16 = 10 ==: 10 (déc) = A (hexa)

Nous nous arrêtons ici puisque nous avons divisé par la troisième puissance de 16. Le résultat (attendu) est "C57A".

Nous désignerons les nombres hexadécimaux en les faisant précéder d'un $. Nous écrirons donc le nombre hexadécimal "C57A" $C57A.

On peut effectuer les calculs avec les nombres hexadécimaux comme avec les nombres décimaux. Il faut simplement faire attention au fait que la retenue ne se produit pas dès qu'on dépasse "9" mais seulement quand on dépasse "F".

Exemple d'addition en hexadécimal:

```
      C5D9
+     13EA
-----------
      D9C3
```

Cela peut sembler compliqué au premier abord mais on peut facilement l'expliquer:

$9 + $A = 9 + 10 = 19 = 16+3 = S3 et je retiens $1
retenue $1+$D+$E = 1+13+14 = 28 = 16 + 12 = $C et je retiens $1
retenue $1+$5+$3 = 1+5+3 = 9 = $9
$C + $1 = 12+1 = 13 = $D

Vous rencontrerez souvent de tels calculs car les ordinateurs travaillent uniquement avec des bits et ceux-ci sont presque toujours convertis en nombres hexadécimaux pour les besoins humains. La multiplication des nombres hexadécimaux est le plus souvent exécutée comme une multiplication de la représentation binaire des facteurs.

On peut bien sûr écrire et donc traiter également sous forme d'un nombre hexadécimal des nombres binaires négatifs en représentation en complément à 2.

Par exemple:

-95 = $A1 = %101000001

Pour le calcul avec les nombres négatifs hexadécimaux en représentation en complément à deux, les mêmes règles s'appliquent que pour le calcul avec les nombres négatifs binaires en représentation en complément à deux.

Il y aurait encore beaucoup à dire sur le maniement des nombres hexadécimaux. Nous réservons cependant certaines explications, notamment sur le calcul avec le microprocesseur MC68000, pour plus tard de façon à ne pas vous surcharger de notions nouvelles.

Représentation de fractions décimales

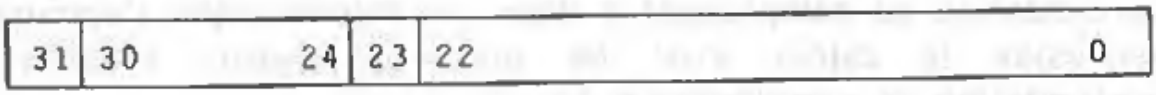
Nous n'avons pas encore indiqué comment l'ordinateur peut représenter des fractions décimales. Nous ne vous fournirons ici cependant que quelques exemples car une explication systématique dépasserait le cadre de cet ouvrage. Une telle explication n'est d'ailleurs pas nécessaire lorsqu'on "débute" avec la programmation en langage machine.

Une forme fréquemment utilisée de la représentation des fractions décimales dans l'ordinateur est la représentation exponentielle normalisée. Pour le nombre 0,0000234, quatre chiffres seulement sont nécessaires pour indiquer où se trouve la virgule. Ce nombre pourrait également être écrit 0,234*10^(-5). Le nombre -100 pourrait être écrit -0,1*10^3, 41,23 s'écrirait 0,4123*10^2.

Le 0,4123 est appelé MANTISSE, le 2 est appelé EXPOSANT.

Ce mode de représentation peut également être étendu aux nombres binaires. Le nombre binaire 1010,011 peut être écrit 0,1010011*2^4.

Les nombres binaires en écriture exponentielle peuvent être représentés dans l'ordinateur selon différents formats. Un format fréquemment utilisé utilise 32 bits:



SE Exposant SM Mantisse

SE: signe de l'exposant
SM: signe de la mantisse

Dans cet exemple, la mantisse ainsi que l'exposant sont représentés en complément à deux. L'exposant peut donc prendre les valeurs -128 à 127. 24 bits sont disponibles pour la mantisse. Puisque le premier bit indique le signe, 23 bits indiquent la grandeur de la mantisse.

Il s'agit bien sûr uniquement d'un exemple de représentation en format exponentiel normalisé. Le nombre de bits employés peut évidemment varier.

Lors de la conversion de nombres décimaux en nombres binaires ou hexadécimaux ou inversement, il y a toujours des erreurs de conversion dûes à l'ordinateur. La représentation binaire ne se prête donc pas à des applications (par exemple comptabilité) où il est nécessaire d'avoir une exactitude absolue des calculs effectués. Pour pouvoir faire de tels calculs on a donc dû introduire un autre mode de représentation:

La représentation en BCD:

BCD est l'abréviation de "Binary Coded Decimal" soit "décimal codé en binaire". Avec la représentation en BCD, chaque groupe de quatre bits représente un chiffre décimal. Il faut cependant veiller à ne jamais appliquer les règles de calcul qui valent pour les nombres binaires aux nombres BDC. Les nombres BCD ont la même forme que les nombres binaires mais ils ont d'autres propriétés.

Le nombre 735 serait donc représenté de la façon suivante:

735 = 0111 0011 0101 (BCD)

Il faut bien sûr faire attention au fait que par exemple le nombre

1101 0111 0101

n'est pas un nombre BCD puisque 1101 ne représente aucun chiffre décimal (mais le 13 ou $D).

Pour représenter des nombres BCD de façon pratique, on utilise souvent encore d'autres "astuces". Le premier chiffre BCD pourrait par exemple indiquer combien le nombre comporte de chiffres. Le second chiffre peut alors déterminer s'il s'agit d'un nombre positif ou négatif (%0000 = +, %0001 = -):

```
0011 0001 0111 0011 0101
  3    -    7    3    5
```

D'après la convention que nous venons d'indiquer, cela représenterait le nombre -735 (3 chiffres, négatif, grandeur = 735).

Les fractions décimales peuvent être également représentées de façon comparable. Entre les deux premiers groupes de quatre bits, on insère un chiffre BCD qui indique en quelle position (en comptant à partir de la droite) se trouve la virgule:

```
0011 0010 0001 0111 0011 0101
  3    2    -    7    3    5
```

Cela correspondrait donc à -7,35 (3 chiffres, virgule en seconde position en partant de la droite, négatif, grandeur = 735).

On peut certainement trouver encore de nombreuses formes de représenation semblables mais nous voulions ici indiquer simplement le principe.
On pourrait par exemple utiliser pour le signe un bit du champ "nombre de chiffres". On peut également utiliser deux chiffres BCD pour le champ nombre de chiffres si l'on veut traiter des nombres plus grands.

QUARTETS, OCTETS, etc.

On utilise souvent plusieurs bits ensemble. Des noms se sont imposés pour différentes "réunions de bits". Les noms les plus importants (avec l'équivalent anglais entre parenthèses) sont:

QUARTET (NIBBLE)	soit	4 BITS
OCTET (BYTE)	soit	8 BITS
MOT	soit	16 BITS
LONG MOT	soit	32 BITS

Un octet contient donc deux quartets, un mot en contient quatre, etc... Un quartet peut être écrit sous la forme d'un chiffre hexadécimal. Les réunions de bits plus importantes sont habituellement représentées avec plusieurs chiffres hexadécimaux.

Tous les groupes de bits peuvent naturellement être représentés en binaire mais cette méthode est déjà peu intelligible lorsqu'on arrive au niveau de l'octet. Le système octal ne se prête pas bien à l'indication des valeurs car ni 4, ni 8, 16 ou 32 ne sont divisibles par 3 et les chiffres octaux ne seraient donc pas pleinement utilisés.

REPRESENTATION DE CARACTERES D'ECRITURE

Comme nous le savons maintenant, l'ordinateur sauvegarde toujours les données sous une forme binaire. Pour pouvoir traiter les caractères d'écriture avec l'ordinateur, il faut donc définir un code qui affecte à chaque caractère une combinaison de bits unique.

Un tel code existe déjà depuis assez longtemps: c'est le code des télex. Le code télex utilise cinq bits ce qui donne 32 combinaisons différentes. Bien qu'on n'utilise avec les télex que les lettres minuscules, ce nombre de combinaisons est insuffisant. On a en effet besoin d'au moins 26 lettres (sans les accents) et de 10 chiffres. C'est pourquoi on a eu recours pour le télex à une astuce.

Il y a parallèlement deux niveaux de code. Le premier contient toutes les lettres, le second les autres caractères (tels que ":", "=", etc...). Deux caractères de commande spéciaux qui ne sont jamais imprimé permettent d'effectuer la commutation entre les modes lettres d'une part et chiffres et caractères spéciaux de l'autre. Ces deux caractères de commande ainsi que les caractères de commande pour retour de chariot, ligne suivante et espace existent dans les deux niveaux de code avec les mêmes combinaisons de bits. Si l'on envoie par exemple la combinaison de bits %01100, c'est un "i" qui sera imprimé si c'est le niveau de code "lettres" qui est activé.

On peut ensuite commuter sur le niveau "chiffres et caractères spéciaux" avec le code %11011. Le code %01100 provoquera alors l'impression du caractère "8". Le code %00100 représente un espace, quel que soit le niveau de code.

La commutation entre les niveaux de code rend le code télex très peu pratique à utiliser. L'absence des majuscules ainsi que de certains caractères spéciaux usuels (par exemple le caractère " ") est d'autre part un inconvénient sérieux. Il a été cependant intéressant pendant un temps pour les amateurs d'utiliser d'anciens télex comme imprimantes. En 1982, on pouvait avoir un télex d'occasion pour souvent moins de 1000 francs. Alors qu'une imprimante coûtait au minimum 4500 francs.

Mais avec l'augmentation constante des ventes en microinformatique domestique les prix des imprimantes ont tellement baissé qu'il n'est plus intéressant aujourd'hui d'accepter les inconvénients (notamment le bruit) que comporte un télex.

Un autre code permettant la représentation de tous les caractères usuels est le code EBCDIC de la société IBM. Ce code n'est cependant intéressant que si l'on veut travailler sur de grosses installations informatiques IBM.

Le code de loin le plus répandu est le code ASCII. ASCII est l'abréviation de "American Standard Code for Information Interchange". L'inconvénient principal de ce code pour nous est indiqué par son nom même: il s'agit d'un code américain qui ne contient donc normalement pas d'accents. Mais nous y reviendrons plus tard.

Il faut représenter 26 lettres majuscules et autant de minuscules. Il faut en outre 10 chiffres et environ 20 caractères de commande.

Cela donne un nombre global de 82 caractères à représenter. Pour représenter un tel nombre de caractères par des combinaisons binaires sans avoir recours comme pour le télex à une commutation, 7 bits sont nécessaires. Le code ASCII est effectivement un code sur 7 bits.
Les combinaisons restant disponibles sont employées comme caractères de commande pour ligne suivante, retour de chariot, etc... Il y a en outre un certain nombre de codes de commande qui régissent la transmission de données avec les périphériques.

Les caractères ASCII sont cependant représentés le plus souvent par huit bits, un caractère ASCII ayant ainsi la même largeur qu'un octet ce qui présente des avantages décisifs pour le stockage des données.
Le huitième bit est souvent employé pour le contrôle des erreurs, pour la commutation entre différents jeux de caractères ou pour les tailles d'écriture, etc...

Déc.	Hexa	ASCII	CTRL	Déc.	Hexa	ASCII
0	00	NUL	@	32	20	Space
1	01	SOH	A	33	21	!
2	02	STX	B	34	22	"
3	03	ETX	C	35	23	#
4	04	EOT	D	36	24	$
5	05	ENQ	E	37	25	%
6	06	ACK	F	38	26	&
7	07	BEL	G	39	27	'
8	08	BS	H	40	28	(
9	09	HT	I	41	29	)
10	0A	LF	J	42	2A	*
11	0B	VT	K	43	2B	+
12	0C	FF	L	44	2C	,
13	0D	CR	M	45	2D	-
14	0E	SO	N	46	2E	.
15	0F	SI	O	47	2F	/
16	10	DLE	P	48	30	0
17	11	DC1	Q	49	31	1
18	12	DC2	R	50	32	2
19	13	DC3	S	51	33	3
20	14	DC4	T	52	34	4
21	15	NAK	U	53	35	5
22	16	SYN	V	54	36	6
23	17	ETB	W	55	37	7
24	18	CAN	X	56	38	8
25	19	EM	Y	57	39	9
26	1A	SUB	Z	58	3A	:
27	1B	ESC	[	59	3B	;
28	1C	FS	\	60	3C	<
29	1D	GS	]	61	3D	=
30	1E	RS	^	62	3E	>
31	1F	US	_	63	3F	?

Caractères de commande Caractères et chiffres

Déc.	Hexa	ASCII	Déc.	Hexa	ASCII
64	40	@	96	60	`
65	41	A	97	61	a
66	42	B	98	62	b
67	43	C	99	63	c
68	44	D	100	64	d
69	45	E	101	65	e
70	46	F	102	66	f
71	47	G	103	67	g
72	48	H	104	68	h
73	49	I	105	69	i
74	4A	J	106	6A	j
75	4B	K	107	6B	k
76	4C	L	108	6C	l
77	4D	M	109	6D	m
78	4E	N	110	6E	n
79	4F	O	111	6F	o
80	50	P	112	70	p
81	51	Q	113	71	q
82	52	R	114	72	r
83	53	S	115	73	s
84	54	T	116	74	t
85	55	U	117	75	u
86	56	V	118	76	v
87	57	W	119	77	w
88	58	X	120	78	x
89	59	Y	121	79	y
90	5A	Z	122	7A	z
91	5B	[	123	7B	{
92	5C	\	124	7C	|
93	5D	]	125	7D	}
94	5E	^	126	7E	~
95	5F	_	127	7F	DEL

Majuscules Minuscules

Les caractères de commande sont souvent également indiqués par ce qu'on appelle un code de contrôle. Le caractère de commande "BEL" peut par exemple être sorti avec le code CTRL-G (touche CONTROLE en même temps que touche G.

Les combinaisons de la première colonne représentent les caractères de commande. Un bon nombre d'entre eux sont cependant rarement utilisés dans les faits. C'est pourquoi nous ne présenterons ici que les codes les plus importants pour l'étude du langage machine:

BEL	Bell	Bip
BS	Back Space	Pas en arrière
LF	Line Feed	Passage à la ligne suivante
FF	Formular Feed	Passage à la page suivante
CR	Carriage Return	Retour de chariot (le plus souvent sans passage automatique à la ligne)

Le code ASCII standard ne contient pas les accents français. On a donc dû redéfinir pour la France certains caractères peu utilisés. Mais ces modifications sont bien sûr contraires au standard original et ne sont pas elles-mêmes toujours identiques pour tous les constructeurs. Le mieux est donc de se reporter à la documentation fournie pour chaque ordinateur. Il y a malheureusement aussi des sociétés qui utilisent un autre code que le code ASCII pour la représentation interne des caractères mais le plus souvent ce code maison consiste simplement à interchanger des colonnes du code tel que nous vous l'avons présenté.

OPERATIONS LOGIQUES ET MANIPULATION DE BITS

Outre les opérations arithmétiques bien connues (addition, multiplication, etc.), il existe ce qu'on appelle les opérations logiques. Les opérations logiques, contrairement aux opérations arithmétiques, influencent les différents bits isolément. On distingue en outre des opérations monadiques qui ne nécessitent qu'un opérande et des opérations diadiques à deux opérandes. Il y a quatre opérations logiques particulièrement importantes:

NON (NOT)

A	nA
0	1
1	0

ET (AND)

A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

OU (OR)

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

OU EXCLUSIF (EXOR)

A	B	A exor B
0	0	0
0	1	1
1	0	1
1	1	0

Nous pourrions dire à la place de "0" "faux" et à la place de "1", "vrai". Il apparaît ainsi que les opérateurs logiques présentés ici correspondent tout à fait aux opérateurs logiques usuels en mathématiques.

Un exemple d'opérations logiques:

Un "0" en A indique que le temps est mauvais; un "1" qu'il fait beau.

Un "1" en B indique que c'est le week-end; un "0" que ce n'est pas le week-end.

Si l'on ne veut aller à la piscine que le week-end et que s'il fait beau, on pourrait dire également que (A AND B) doit être égal à 1.

L'affirmation (A OR B) signifierait qu'on irait à la piscine dès qu'un au moins des critères serait rempli. L'affirmation (A EXOR B) signifierait qu'on irait à la piscine uniquement si un critère et un seul était rempli.

On peut naturellement à nouveau combiner ces opérations logiques entre elles pour former de nouvelles opérations logiques qui relieront ainsi trois bits ou plus.

Pour revenir à notre exemple: nous n'allons pas à la piscine si "non (A and B)" vaut "1". L'équivalent de "non (A and B)" est l'affirmation: ((non (A)) or (non (B))), ce qui peut parfaitement être démontré mathématiquement.

Les opérations logiques peuvent être appliquées à des octets, des mots ou d'une manière générale à des groupes de bits. L'opération est alors simplement exécutée bit par bit avec les bits correspondants des opérateurs.

Nous en avons déjà vu un exemple lorsque nous avons formé le complément à un d'un nombre binaire. Nous avons alors simplement inversé les différents bits de ce nombre. Cela correspond à une exécution bit par bit de la fonction NOT.

Un autre exemple:

non (0010) = 1101

Voici maintenant des exemples avec d'autres opérations logiques:

```
.   (0010)        (0010)        (0010)
or  (1011)   and  (1011)   exor (1011)
----------   ----------   ----------
=    1011    =    0010    =    1001
```

Ces opérations sont souvent utilisées notamment en langage machine pour obtenir une modification sélective et conditionnelle de certains bits. Les opérateurs sont alors cependant souvent écrits autrement. Nous expliquerons ce mode d'écriture un peu plus loin.

Opérations de décalage de bits:

Outre les opérations logiques, il existe également des opérations qui ont pour effet de décaler les différents bits d'un nombre binaire. Le 68000 possède pour cela une série d'instructions. Nous n'en donnerons ici que quelques exemples.

Soit le nombre binaire %00101100.

Ce nombre décalé d'une position sur la droite deviendrait %0010110 (pour autant que l'emplacement libéré soit rempli par un zéro). Si l'on considère la valeur du nombre, on constate que celle-ci a été divisée par deux. Nous pouvons donc poser comme règle: un décalage sur la droite correspond à la division d'un nombre binaire par deux.

Ce même nombre décalé vers la gauche donnerait %01011000. La valeur a maintenant doublé. Donc: un décalage vers la gauche correspond à une multiplication d'un nombre binaire par deux.

Il existe encore des opérations qui permettent de faire subir une rotation aux bits. %10011000 donne par exemple, après rotation vers la gauche, %00110001.

PROCESSUS DE REALISATION D'UN PROGRAMME

Au début de la réalisation d'un programme, il faut définir clairement le problème qu'il s'agit de résoudre. Il faut pour cela d'abord définir quelles données devront être sorties. Rien que de logique jusqu'ici, mais qu'entend-on exactement par "données"?

Voici une définition tirée d'un dictionnaire:

> Données
> Sur l'ordinateur, on entend par données toute forme de représentation de caractères, symboles et lettres devant être traités sous une forme quelconque.

Ajoutons simplement que les données peuvent également consister en décisions ou états logiques, etc.

Une fois cette étape terminée, il faut examiner d'où viennent les données qui devront être traitées.

On passe ensuite généralement à l'établissement d'un plan de flux des données précis. Ce plan de flux des données peut revêtir presque n'importe quelle forme. L'essentiel est qu'il indique précisément ce que doivent devenir les données.
Il faut indiquer le déroulement du traitement des données ainsi que l'entrée/sortie et la sauvegarde, etc. Il faut alors développer les algorithmes précis qui permettront de remplir les fonctions voulues. Il arrive aussi qu'on trouve des algorithmes tout prêts qui permettent de résoudre le problème de façon satisfaisante. Pourquoi chercher alors à réinventer la poudre?

Une fois que tout cela est fait, on réalise un organigramme. Nous vous expliquons au chapitre 5 ce qu'est un organigramme. L'organisation déjà existante est affinée à cette occasion et on doit parfois reformuler les algorithmes.

Ce n'est normalement qu'après avoir accompli ce travail de préparation qu'on devrait passer à la formulation du problème dans le langage de programmation choisi. Très souvent, certaines parties du problème seront écrites dans un langage évolué (par exemple BASIC, Pascal ou C) alors que d'autres parties seront écrites en langage machine (c'est-à-dire avec des instructions assembleur).

On écrira notamment souvent en langage machine les parties de programme pour lesquelles la rapidité d'exécution joue un rôle important car les langages évolués sont généralement plus lents.

Ce n'est qu'une fois que le programme est écrit qu'il devrait normalement être testé sur l'ordinateur. La programmation étape par étape suivant le principe "une erreur, une solution de détail" n'est pas très pratique. L'important est en effet de tester le programme sous le plus grand nombre d'angles possibles. Les erreurs ne se produisent souvent que dans des conditions apparemment inenvisageables.

Une fois que le programme proprement dit est terminé, il faut réaliser la documentation du programme. Elle doit permettre au programmeur ainsi qu'aux autres utilisateurs de pouvoir éventuellement modifier le programme ultérieurement. L'expérience démontre en effet de façon catégorique qu'on ne comprend plus même ses propres programmes au bout d'un an à moins de disposer d'une bonne documentation.

Il est important de réaliser un manuel d'utilisation de qualité. C'est en effet du manuel que dépendra notamment la possibilité de bien employer un programme. Il faut bien entendu prendre en compte à cet égard la catégorie d'utilisateurs visée. Un programme qui constitue une aide à la programmation devra être expliqué différemment d'un programme que devra utiliser une secrétaire sans formation informatique.

Le travail sur un programme n'est cependant pas encore achevé lorsqu'un programme est "terminé". Il faut en effet corriger les erreurs rencontrées par l'utilisateur ou tenir compte de certaines modifications extérieures (par exemple un changement du taux de la TVA). La pratique montre souvent quelles fonctions manquent encore au programme. Toutes ces tâches sont regroupées sous le terme d'entretien du programme.

Les phases du développement d'un programme peuvent donc être ainsi formulées en résumé:

1) Définition précise du problème
2) Réalisation du plan de flux des données
3) Ecriture de l'organigramme
4) Ecriture du programme dans un langage de programmation
5) Entrée, test et correction du programme
6) Documentation du programme
7) Manuel d'utilisation
8) Entretien du programme

Certaines phases peuvent bien sûr se confondre dans la pratique mais il est recommandé de suivre systématiquement ce schéma. Un bon programmeur n'est pas quelqu'un qui s'assoit devant l'ordinateur pour ne se lever qu'une fois que le programme est terminé mais bien plutôt quelqu'un qui suit ce schéma de façon judicieuse. Il arrive souvent que certaines phases se déroulent inconsciemment dans la tête du programmeur (surtout pour les petits programmes).

Cette méthode doit s'appliquer, pour les programmes d'envergure, séparément à chaque partie du problème. Il devient ainsi possible que plusieurs personnes collaborent sans se gêner mutuellement. Dans ce cas, les tâches doivent être définies précisément auparavant.

Le principal problème est que seulement 10% des programmeurs sont réellement en mesure de travailler sans méthode bien définie mais que 90% croient faire partie de ces 10%.

STRUCTURE D'UN MICRO ORDINATEUR

1) Introduction
2) Mémoire
3) Unité centrale
4) Entrée/sortie

Un ordinateur se compose essentiellement de trois éléments:

Mémoire
Unité centrale
Unités d'entrée/sortie

Ces trois éléments ne constituent cependant pas seulement trois composants et l'Atari comporte par exemple un nombre beaucoup plus grand de composants. Tous ces branchements électroniques sont dûs au fait qu'il n'existe encore aucun composant pouvant exécuter toutes les fonctions dont dispose l'Atari ST.

La mémoire

Il faut distinguer entre deux types principaux de mémoire:

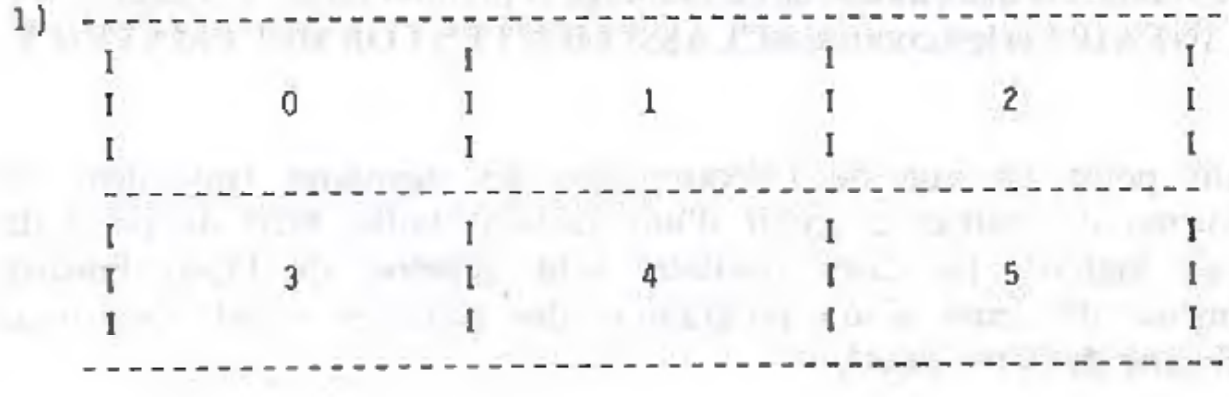
1) la mémoire de lecture/écriture

2) la mémoire de lecture seulement (Read Only Memory ou ROM)

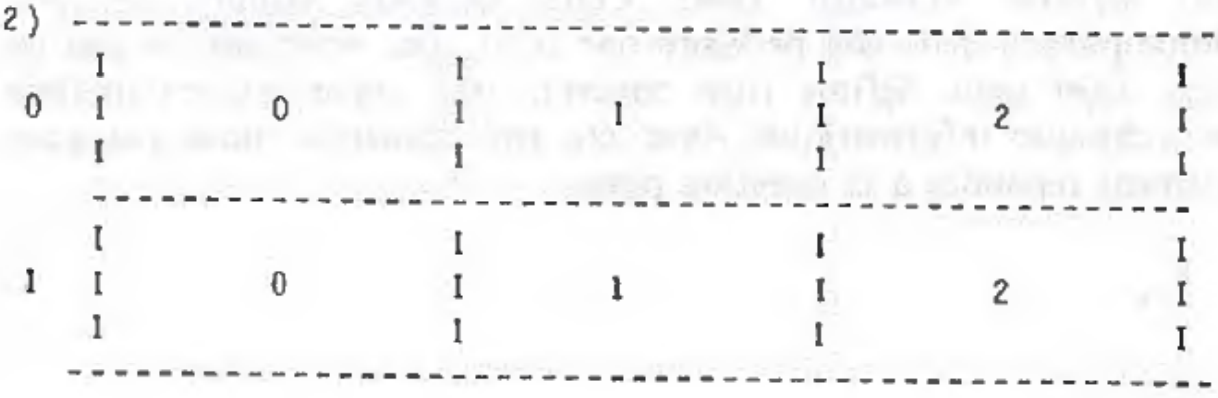
La mémoire de lecture/écriture est généralement désignée sous le terme de RAM (Random Access Memory) ce qui signifie mémoire à accès sélectif. L'accès sélectif ne signifie pas seulement qu'on peut accéder librement à n'importe quelle adresse de la mémoire; cela est en effet possible aussi bien en ROM qu'en RAM. Cela signifie plutôt qu'on peut choisir le mode d'accès à la mémoire, lecture ou écriture.

Une mémoire RAM est comme une commode à plusieurs tiroirs. Supposons que notre commode ait six tiroirs répartis en deux rangées de 3:

Nous pourrons alors désigner les six tiroirs de deux façons différentes:



Nous avons ainsi les tiroirs 0 à 5 que nous pourrons clairement distinguer grâce à leurs numéros respectifs. Il est à noter à ce propos ques les informaticiens numérotent presque toujours en partant de 0 ce qui a souvent des avantages dans les programmes. Il y a encore une autre méthode pour désigner les tiroirs:



Nous arrivons ainsi aux noms (0,0), (0,1), (0,2),
(1,0), (1,1), (1,2).

Cette méthode permet également de distinguer clairement entre les différents tiroirs.

Retenons ces deux modes de numérotage: le premier est le CLASSEMENT LINEAIRE et le second est le CLASSEMENT EN FORME DE MATRICE.

Du point de vue de l'électronique, les mémoires travaillent en format de matrice à partir d'une certaine taille; mais du point de vue logiciel, les cases mémoire sont appelées de façon linéaire (même d'ailleurs si on programme des matrices - mais cela nous éloigne de notre sujet).

Nous avons donc six tiroirs que nous savons distinguer entre eux. Nous voulons maintenant les utiliser. Nous ouvrons le tiroir zéro et nous regardons ce qu'il contient:

RIEN

Pourquoi? Parce que nous n'y avons encore rien mis. Mais est-ce que rien signifie vraiment rien? Cette question semble un peu philosophique mais elle nécessite que nous nous écartions un peu de notre sujet pour définir trois concepts très importants en matière de technique informatique. Avec ces trois concepts, nous pourrons aisément répondre à la question posée.

L'informaticien distingue trois caractéristiques d'une information:

1) les aspects syntaxiques,
2) sémantiques et
3) pratiques d'une information donnée.

Que signifient ces termes grecs?

La "SYNTAXE" désigne la structure fondamentale d'une information.

Prenons par exemple le mot "HOMME". Si nous pensions "HOMME" mais écrivions OM, personne ne nous comprendrait et c'est pourquoi nous nous en tenons aux règles (règles de syntaxe!) de la langue française et écrivons HOMME.

Prenons maintenant l'exemple des feux aux carrefours. Leur syntaxe est:

rouge
vert
orange

Si le rouge et le vert apparaissaient en même temps, personne ne saurait plus interpréter cette information car elle serait d'une SYNTAXE incorrecte.

La SEMANTIQUE désigne le contenu de l'information auquel a pensé l'émetteur de l'information ou, en le formulant autrement, la syntaxe désigne ce qui devrait normalement se passer au vu de cette information.

L'aspect sémantique du feu rouge est: ARRET.

Le troisième aspect d'une information, la PRATIQUE désigne ce qui pousse l'utilisateur d'une information à l'utiliser ou la façon dont il utilise en fait une information pour lui-même. Suivant les automobilistes, la vue du feu orange entraînera par exemple la réflexion "en appuyant sur l'accélérateur je passe" ou "trop tard, je m'arrête".

Ces définitions nous permettent maintenant de répondre assez facilement sur le plan de la technique informatique à notre question qui serait insoluble sur le plan philosophique. Nous en étions resté à "nous ne voyons rien dans le tiroir que nous venons d'ouvrir."

RIEN, QUE NOUS PUISSIONS EXPLOITER UTILEMENT

Aidons-nous de nos trois définitions:

1) La syntaxe de l'information est correcte; il y a certainement quelque chose dans le tiroir, ne serait-ce que du vide.

2) La sémantique ne pose pas non plus de problème, nous n'avons rien mis dans le tiroir et nous ne pouvons donc rien en retirer.

3) Sur le plan pratique, le tiroir est bien vide pour nous (bien qu'il contienne quelque chose, à savoir RIEN!), ce qui veut dire que nous pouvons y mettre quelque chose.
Si nous le faisons et si personne ne retire quoi que ce soit du tiroir, nous retrouverons ce que nous y avons mis chaque fois que nous ouvrirons le tiroir.

Nous allons maintenant formuler la procédure décrite ci-dessus sur le plan technique:

Nous appellerons les numéros qui nous servent à désigner les tiroirs:

A D R E S S E S

Chaque tiroir peut contenir exactement huit bits !!!

Ouvrons maintenant un tiroir, par exemple le numéro 3. Nous appellerons à l'avenir cette procédure d'ouverture des tiroirs

A D R E S S A G E

et nous appellerons

LECTURE le fait de regarder dans un tiroir,
ECRITURE le fait d'y placer quelque chose
et nous désignerons sous le nom de case mémoire le tiroir lui-même.

Cela représente beaucoup de termes nouveaux en une fois mais avec le temps nous apprendrons à les manier.

Nous allons donc ADRESSER la CASE MEMOIRE 3 et la LIRE. Que contient-elle? La réponse est rien puisque nous n'y avons encore rien écrit.

Comme nous l'avons appris au chapitre précédent, seuls des 0 et des 1 peuvent se trouver dans une case mémoire car nous n'avons que deux chiffres à notre disposition. Par ailleurs, nous avons dit que rien ne signifie pas forcément rien. Il se pourrait par exemple que notre case mémoire contienne en fait le nombre %01101100 ($6C). Pourquoi pas?

Cela s'explique par le fait que des irrégularités internes des canaux conducteurs des composants font que la mémoire contient des combinaisons de bits aléatoires après la mise sous tension de l'ordinateur. Et ces combinaisons de bits peuvent être une quelconque des 256 possibilités de combinaison de huit zéros ou uns, de %00000000 à %11111111.

Supposons donc que nous ayons un ordinateur devant nous sur lequel la case mémoire 3 contiendrait le nombre hexadécimal $6C après la mise sous tension.

Nous pourrons bien sûr lire et relire ce nombre à tout moment car il figure effectivement dans cette case mémoire. Mais l'aspect pratique de cette information est pour nous utilisateur très peu supérieur à zéro car il s'agit d'une information purement aléatoire, c'est-à-dire dûe au hasard !!!

Nous allons donc écrire quelque chose d'intéressant dans cette case mémoire. Choisissons pour cela "$D7". Nous adressons toujours notre case mémoire numéro 3 et y ECRIVONS maintenant le nombre $D7.

Que se passe-t-il alors?

Le nombre aléatoire $6C disparaît de la case mémoire et le nombre $D7 y est STOCKE. Si nous accédons maintenant ultérieurement à nouveau à la case mémoire numéro 3, nous lirons toujours le nombre $D7 jusqu'à ce que nous écrivions un nouveau nombre dans cette case mémoire ou jusqu'à ce que nous mettions l'ordinateur hors tension.

Dans le premier cas, le $D7 serait naturellement effacé et remplacé par le nouveau nombre, dans le second cas, nous obtiendrions à nouveau une information aléatoire après mise sous tension.

N'y a-t-il donc pas de possibilité de retrouver même après arrêt de la machine une information stockée ou de protéger une information contre son effacement et son remplacement par une autre?

Cela n'est possible que par une intervention dans les circuits de l'ordinateur mais la RAM est justement une mémoire de lecture/écriture. Si nous voulons une mémoire qui nous fournisse des informations que nous puissions exploiter sans que ces informations puissent être effacées, il nous faut employer une mémoire de lecture seulement.

I.es mémoires de lecture seulement:

Les mémoires de lecture seulement sont appelées ROM (Read Only Memory). Il existe différents types de ROM qu'on distingue par les lettres précédant le terme ROM, par exemple EPROM, EEROM, EAROM, PROM, IPROM. Mais une EPROM déterminée s'appellera par exemple, selon la nomenclature du fabricant, "2716".

Toutes les ROM ont ceci en commun: l'ordinateur ne peut pas SANS AIDE PARTICULIERE modifier les informations qu'elles contiennent. Pour certains composants, les informations sont placées dans le composant dès sa fabrication de sorte qu'il nous est ensuite impossible de les modifier.

Unité centrale

L'unité centrale se charge de toute la gestion de l'ordinateur. Elle adresse la mémoire disponible, elle traite les données en mémoire et elle gère les périphériques. Elle est donc le coeur de tout ordinateur. Si elle est défectueuse, l'ordinateur est en panne totale.

Elle comprend normalement quelques cases mémoire internes, appelées registres, qui ont par rapport au reste de la mémoire l'avantage de pouvoir être appelées par l'unité centrale beaucoup plus rapidement.

La CPU (Central Processing Unit ou unité centrale d'exécution) comprend également une logique complexe pour le calcul, des chassoirs pour commander les mémoires, des canaux d'entrée/sortie ainsi que des mémoires intermédiaires ne pouvant pas être adressées directement.

Vers l'extérieur, l'unité centrale envoie des signaux d'adresse pour commander les autres éléments, les canaux de données pour le transfert (lecture ou écriture) des données ainsi que quelques signaux de contrôle qui indiquent l'état de l'unité centrale comme par exemple un canal de lecture/écriture qui indique si les données doivent être envoyées ou lues.

L'Atari ST possède une unité centrale MC68000. Celle-ci dispose de 24 canaux d'adresse. On peut ainsi adresser en mémoire 2^24 octets différents. L'Atari ST n'a bien sûr pas une mémoire d'une taille aussi considérable et de grandes zones d'adresses sont inutilisées.

L'unité centrale MC 68000 est un microprocesseur 16 bits ce qui signifie que le bus de données a une largeur de 16 bits. C'est pourquoi la mémoire est également large de 16 bits. Dans la mémoire, il y a donc toujours des groupes solidaires de deux octets, c'est-à-dire d'un mot. Chacun de ces deux octets a une adresse propre car le 68000 est ce qu'on appelle une machine-octet. Mais comme le bus de données peut transporter deux octets en une fois, il est possible d'écrire ou lire dans la mémoire un mot entier (deux octets) avec un seul accès, à la condition toutefois que l'accès à ce mot soit exécuté sur une adresse PAIRE.

ENTREE / SORTIE

Pour qu'un ordinateur puisse entrer en contact avec le "monde extérieur", il dispose de composants d'entrée/sortie. Le clavier ou la souris constituent des exemples d'entrée. Les sorties de l'Atari ST vont par exemple sur l'écran ou sur une imprimante. Mais les sons qu'il peut produire sont également des sorties.

A travers un composant spécial, le floppy controller, un ou deux lecteurs de disquette sont connectés à l'Atari ST. Ce floppy controller est également un composant d'entrée/sortie.

Les lecteurs de disquette ne font pas partie de la mémoire normale car les données sur disquette ne peuvent pas être adressées directement comme celles de la zone mémoire normale.

Les données peuvent être transmises de la périphérie dans la mémoire de deux façons:

1) Le processeur lit lui-même les données dans le périphérique ou il les transmet lui-même à ce périphérique. Il accède normalement à cet effet à un registre du périphérique. Cet accès est identique à l'accès à une case mémoire normale. La transmission peut être initiée de deux façons. Soit le microprocesseur demande à intervales réguliers si des données sont disponibles ou si des données sont nécessitées (cette technique est appelée "polling"); soit le processeur travaille sur un endroit quelconque d'un programme et il est interrompu par un signal électronique appelé interruption chaque fois que des données sont disponibles ou nécessaires.

   Cette technique peut être comparée à la sonnerie du téléphone suivie d'une conversation, interrompant ainsi la tâche que vous étiez en train d'accomplir, par exemple la vaisselle. Avec la première méthode, on regarderait par exemple toutes les minutes si le téléphone sonne. L'inconvénient de cette méthode par rapport à la seconde est évident: même lorsque le téléphone ne sonne pas, on perd du temps à le vérifier.

2) Le processeur se contente d'initialiser la transmission des données (dans un controller DMA). Il détermine alors quelles données devront être transmises à partir de quelle zone de la mémoire et vers où ou bien où les données reçues doivent être placées (dans quelle zone de la mémoire). La transmission proprement dite de la mémoire à un périphérique ou inversement est prise en charge par le controller DMA (DMA: Direct Memory Access, accès direct à la mémoire).

Sur un signal du périphérique, le controller DMA prend le contrôle du bus système et exécute lui-même la transmission des données. Le processeur est ainsi déchargé et le système entier en devient plus puissant. Ce n'est qu'une fois que la transmission est totalement terminée que le processeur principal reçoit un signal du controller DMA. Le processeur peut bien sûr également demander si la transmission est déjà terminée. Comme il n'y a plus à effectuer de travail de traitement pendant la transmission, la transmission s'effectue généralement plus rapidement.

Cette technique de transmission des données est appelée transmission DMA. On peut illustrer la transmission DMA par la comparaison suivante: on indique à une tierce personne quelle conversation téléphonique elle devra avoir et où doivent être envoyées les informations à transmettre dans cette conversation téléphonique ou d'où viendront ces informations. On peut ensuite se consacrer tranquillement à une autre tâche. Une fois que la conversation téléphonique se sera déroulée, la tierce personne viendra nous en avertir.

LE MICROPROCESSEUR 68000

1) Introduction

2) Structure des registres et organisation des données

3) Etats de travail

4) Modes d'adressage

5) Aperçu du jeu d'instructions

Ce chapitre se propose de donner un bref aperçu de la structure du microprocesseur 68000 qui est utilisé sur l'Atari ST. Nous essaierons cependant simplement d'expliquer toutes les notions absolument nécessaires à la compréhension de la suite de cet ouvrage. Une description systématique de toutes les instructions du 68000 dépasserait en effet le cadre du présent ouvrage. Bien entendu nous expliquerons toutes les instructions que nous utiliserons et qui sont d'ailleurs les plus usuelles.

Nous ne pouvons en tout cas que vous conseiller d'étudier également un ouvrage consacré au processeur et à ses instructions. Même une fois que vous aurez appris à programmer en langage machine, vous aurez encore besoin d'un ouvrage de référence dans lequel vous trouverez également les instructions que nous n'utiliserons pas ici. Nous vous conseillons "Le livre du processeur 68000" (également chez Micro Application - Data Becker) auquel nous nous sommes notamment référé.

Le 68000 dispose de deux modes de travail différents. Ces modes sont appelés mode Supervisor et mode User. Les instructions ne sont toutes autorisées qu'en mode supervisor. Cette structure permet de créer des systèmes avec plusieurs utilisateurs sans risque de "planter" la machine.

On peut par exemple faire travailler le système d'exploitation en mode supervisor ou superviseur mais les programmes utilisateur ne peuvent travailler qu'en mode user ou utilisateur. Sur l'Atari ST, la mémoire se trouvant tout en bas de la zone d'adresse et la zone périphérique ne peuvent être appelées que si le 68000 se trouve en mode superviseur alors qu'en mode utilisateur le programme en cours de déroulement serait interrompu et qu'une routine de traitement des erreurs serait appelée.

Il existe également sur d'autres systèmes des composants appelés MMUs (Memory Management Units, unités de gestion de la mémoire) qui surveillent à quelle adresse on accède. Si l'on tente d'accéder à une zone "interdite", la MMU interrompt le programme.

Structure des registres et organisation des données

L'utilisateur dispose sur le 68000 de huit registres de données. Chacun de ces registres de données a une largeur de 32 bits. C'est pourquoi le 68000 est également souvent qualifié dans la publicité de processeur 32 bits. Mais comme son bus de données a une largeur de 16 bits, il doit être considéré comme un processeur 16 bits. Les registres de données sont numérotés de D0 à D7.

Il existe en outre sept registres d'adresse et un compteur de programme. Ces registres sont également larges de 32 bits. C'est ainsi un domaine adressable de 4 giga-octets dont dispose ce processeur. Mais comme le bus d'adresse n'a qu'une largeur de 24 bits, l'utilisateur ne peut adresser "que" 16 mégaoctets. Les registres d'adresse sont désignés par les noms A0 à A6, le compteur de programme (Program Counter) est appelé PC.

Du fait du grand nombre de registres utilisables, il est possible de conserver de nombreuses variables dans les registres et de limiter ainsi le nombre d'accès à la mémoire dans le cours d'un programme. Comme les registres ont une largeur de 32 bits, ils peuvent également contenir une adresse mémoire complète.

La reconstitution d'adresses d'accès à partir de différents éléments telle qu'elle était nécessaire sur le 6502 devient donc heureusement inutile ici.

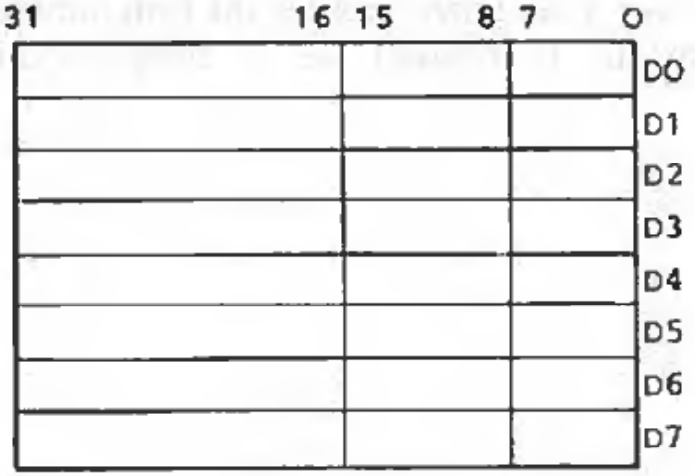
Il y a encore deux pointeurs de pile (stack pointer), un pour le mode superviseur et un pour le mode mode utilisateur. Suivant le mode de travail de l'unité centrale, c'est alternativement l'un des deux pointeurs de pile qui sera activé. Les pointeurs de pile sont désignés sous les noms de "A7" et "A7-". Ce nom montre bien qu'on peut également appeler le pointeur de pile activé comme registre d'adresse 7. Même avec des instructions qui servent à appeler un registre d'adresse en général et non pas le pointeur de pile de façon implicite, on ne peut utiliser que le pointeur de pile actuellement activé. L'instruction "MOVE USP" fait cependant exception. L'existence de deux pointeurs de pile permet de construire très aisément des piles différentes pour les modes supérviseur et utilisateur.

Le compteur de programme du 68000 a une largeur de 32 bits. "Seuls" 24 de ces 32 bits sont placés sur le bus d'adresse. Les huit bits restants ont été réservés pour une extension ultérieure (avec laquelle le 68000 possèderait un domaine adressable de 4 giga-octets).

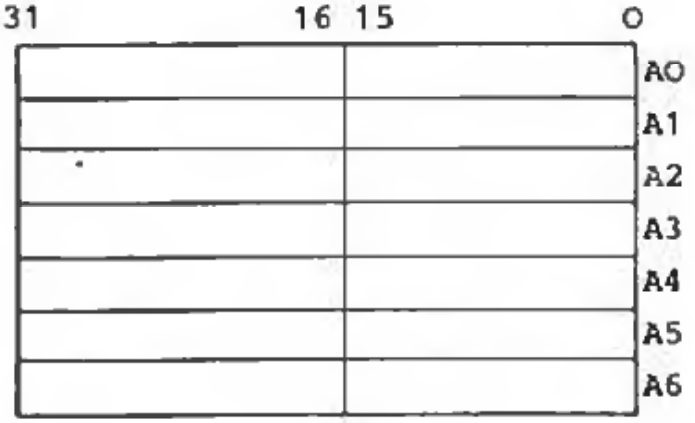
"Last but not least", le registre d'état a une largeur de 16 bits. Il est subdivisé en un octet utilisateur (bits 0 à 7) et un octet système (bits 8 à 15). En mode utilisateur on ne peut écrire que dans l'octet utilisateur. Ainsi, seul le superviseur peut changer le mode de travail de l'unité centrale (cela n'aurait pas de sens sinon). Dans l'octet utilisateur se trouvent les flags qui seront interrogés dans les instructions de saut conditionnelles (instructions de branchement).

Les flags indiquent par exemple après de nombreuses opérations si le résultat est nul, négatif ou si une retenue s'est produite, etc. C'est ainsi par exemple qu'un saut à un autre endroit du programme ne se fera qu'à la condition que le résultat de la comparaison précédente ait été nul.

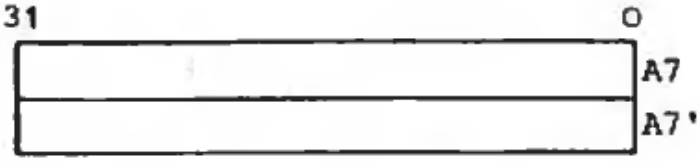
LE JEU DE REGISTRES DU 68000



Huit
registres de données

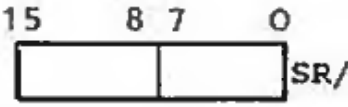


Sept
registres d'adresse



Deux
pointeurs de pile

Compteur de
programme



Registre d'états

Octet système Octet utilisateur

Formats d'opérande

Le format d'opérande est soit contenu de façon implicite dans l'instruction, soit indiqué de façon explicite dans l'instruction. Voici les formats d'opérandes qui sont définis:

1 long mot	correspond à	32 bits
1 mot	correspond à	16 bits
1 octet	correspond à	8 bits

Il est également possible de traiter des opérandes BCD. Le traitement se fait par "groupes", c'est-à-dire que ce sont chaque fois deux nombres BCD qui sont traités ensemble dans un octet. Il existe en outre des instructions de manipulation de bits.

Un mot est le format d'opérande standard puisque le 68000 dispose d'un bus de données 16 bits.

Pour les registres de données, tous les formats sont autorisés. Les opérandes d'un octet occupent les 8 bits inférieurs, les opérandes d'un mot les 16 bits inférieurs alors que les opérandes d'un long mot occupent la totalité des 32 bits.

Lorsqu'un registre de données est utilisé comme opérande source ou opérande objet et que le format d'opérande n'est pas de 32 bits, seule sera modifiée la partie du registre appelée. Le reste du registre n'est ni utilisé ni modifié.

Pour les registres d'adresse et les deux pointeurs de pile, seuls les opérandes d'un mot ou d'un long mot sont autorisés. Les registres d'adresse et les pointeurs de pile ne travaillent pas avec des données de l'ordre de l'octet ou du bit.

Les pointeurs de pile désignent en permanence les dernières données valables et ils "croissent" en descendant vers les adresses les plus basses.

Lorsqu'un registre d'adresse est indiqué comme opérande source, on utilise, en fonction du format d'opérande choisi, soit le registre entier, soit seulement le mot inférieur. Si un registre d'adresse est utilisé comme opérande objet, le registre entier sera affecté, quel que soit le format d'opérande choisi. Si le format est du type mot, tous les opérandes seront étendus sur 32 bits en tenant compte de leur signe.

Registre d'états

Le registre d'états se compose d'un octet utilisateur et d'un octet système. Il est ainsi structuré:

Octet utilisateur:

Bit 0 :	Flag de retenue	Carry	C
Bit 1 :	Flag de dépassement	Overflow	V
Bit 2 :	Flag zéro	Zéro	Z
Bit 3 :	Flag négatif	Négatif	N
Bit 4 :	Flag d'extension	Extension	X
Bit 5 :	inutilisé		-
Bit 6 :	inutilisé		-
Bit 7 :	inutilisé		-

Octet système:

Bits 8 à 10	Masque d'interruption	I0, I1, I2
Bit 1 :	Inutilisé	-
Bit 2 :	Inutilisé	-
Bit 3 :	Etat superviseur	S
Bit 4 :	Inutilisé	-
Bit 5 :	Mode trace	T

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
T	-	S	-	-	I2	I1	I0	-	-	-	X	N	Z	V	C

Retenue:

Le flag de retenue est toujours mis sur un lorsqu'une retenue se produit lors d'une opération arithmétique dans le bit le plus élevé de l'opérande objet. Il est également utilisé pour la retenue négative de la soustraction.

Dépassement:

Le flag de dépassement est utilisé pour indiquer à l'utilisateur que l'intervale numérique couvert a été dépassé lors d'une opération arithmétique.

Cela se produit par exemple lorsque lors d'une addition de deux nombres positifs le résultat déborde des limites du registre et qu'on obtient donc un nombre négatif en complément à deux. Le flag de dépassement est également utilisé pour indiquer lors d'opérations de division que le quotient deviendrait plus grand que 16 bits ou que le dividende est trop grand.

Flag zéro:

Le flag zéro est mis lorsqu'après une opération le bit de plus grande valeur du résultat est mis, c'est-à-dire lorsque le résultat représente un nombre négatif en complément à deux.

Flag d'extension:

Le flag d'extension se comporte exactement comme le flag de retenue. Il n'est toutefois pas affecté par toutes les instructions qui affectent le flag de retenue. Il est par exemple traité comme le flag de retenue pour les additions et les soustractions mais pas dans tous les cas avec les instructions de rotation. Une liste des instructions indique précisément quand le flag d'extension est utilisé. Le flag d'extension peut être utilisé pour conserver le contenu du flag Carry pendant plusieurs opérations auxiliaires. Le flag d'extension constitue une particularité du 68000.

En lisant les autres bits (inutilisés) de l'octet utilisateur du registre d'états, on obtient toujours des zéros, même lorsqu'on y écrit d'autres valeurs. Cela vaut également pour les bits inutilisés de l'octet système du registre d'états. Voici maintenant la description des bits de l'octet système.

Masque d'interruption:

Le 68000 possède sept niveaux d'interruption (numérotés de 1 à 7). Une interruption externe n'est autorisée que si la valeur du masque d'interruption est inférieure au rang de priorité de l'interruption. Le niveau d'interruption 7 ne peut toutefois pas être verrouillé. C'est ce qu'on appelle la NMI (Non Maskable Interruption = interruption non masquable). La modification du masque d'interruption permet donc de verrouiller ou de libérer les interruptions.

Etat superviseur:

Ce bit permet de commuter le processeur entre les états superviseur et utilisateur. Un zéro correspond à l'état utilisateur et un un à l'état superviseur. La commutation permet de faire fonctionner des systèmes à plusieurs utilisateurs sans les "planter".

Mode trace:

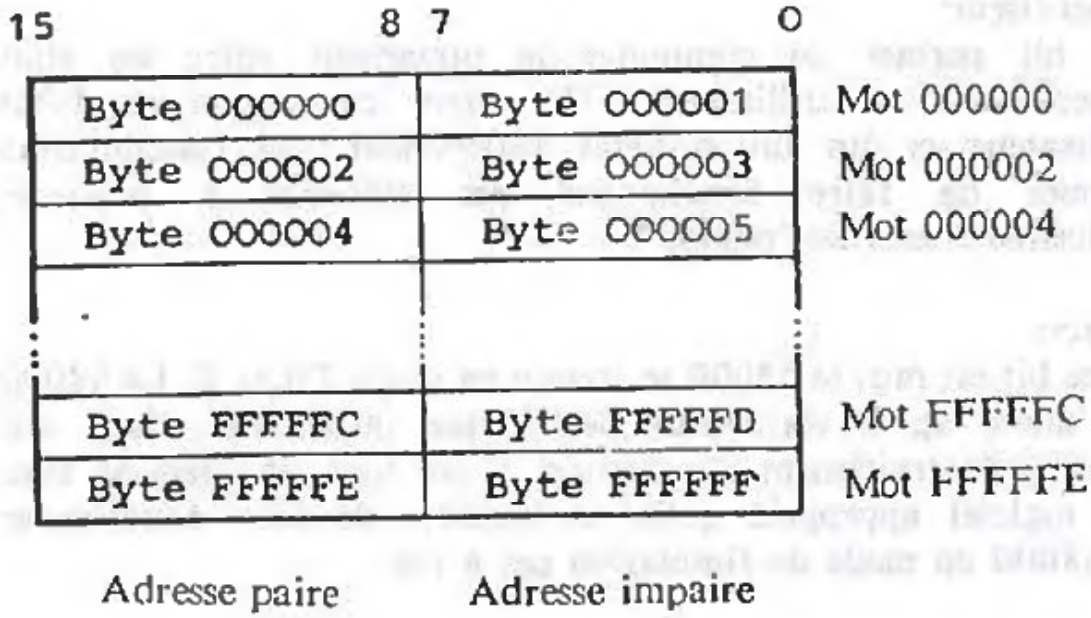
Si ce bit est mis, le 68000 se trouve en mode TRACE. Le 68000 va alors, après traitement de chaque instruction, dans une routine de traitement d'exception. C'est donc uniquement avec un logiciel approprié qu'il est possible de faire fonctionner le 68000 en mode de simulation pas à pas.

Organisation des données dans la mémoire

Bien que le 68000 soit un processeur 16 bits, il travaille comme une machine huit bits. Cela signifie que chaque mot est divisé en deux octets et que chacun de ces deux octets a une adresse propre. Le 68000 permet bien sûr d'appeler simultanément les deux octets d'un mot dans la mémoire puisque le bus de données a une largeur de 16 bits. Chaque octet occupe une adresse dans la zone mémoire, chaque mot s'étend donc sur deux adresses.

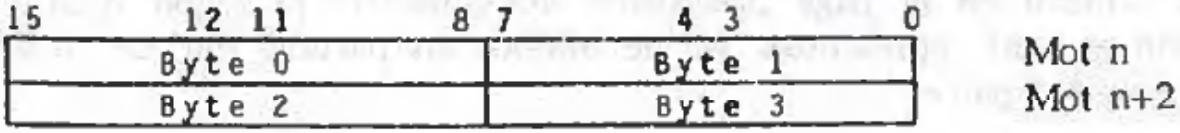
L'octet fort (octet de plus grande valeur) d'un mot (du mot d'adresse "n") figure à l'adresse la plus basse (l'adresse "n") alors que l'octet faible (octet de moindre valeur) figure à l'adresse la plus haute (adresse "n+1").

Le tableau suivant illustre cette situation:

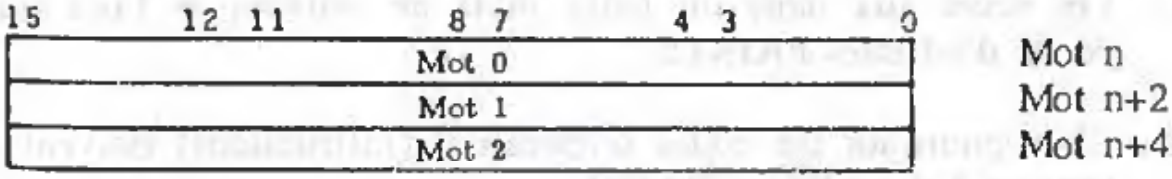


L'organisation des données dans la mémoire

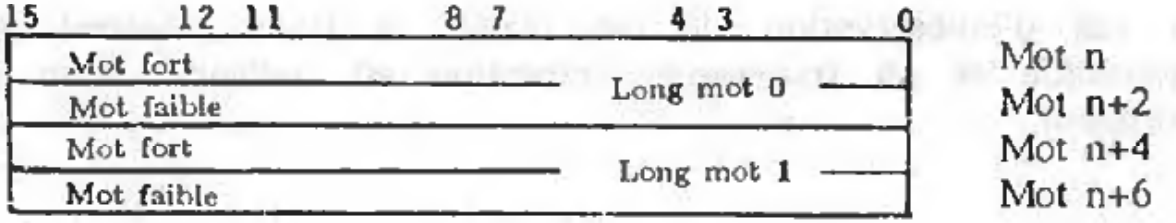
(Organisation des mots)



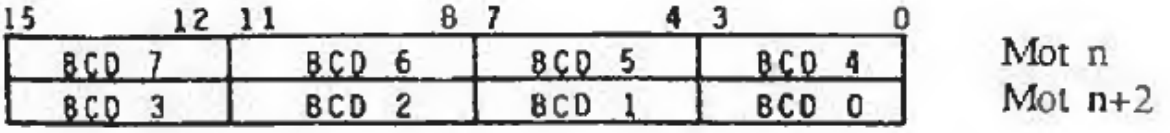
Données entières (octet)



Données entières (Mot)



Données entières (Long mot) BCD 7



BCD 7 : chiffre de plus grande valeur
BCD 0 : chiffre de plus petite valeur

Données décimales (codées en BCD)

Le tableau de la page précédente vous montre la façon dont les données sont représentées sur le 68000, en partant toujours d'une adresse "n" paire.

La structure du 68000 entraîne les règles suivantes pour l'accès à la mémoire:

1) Les accès aux mots ou longs mots ne peuvent se faire qu'à partir d'adresses PAIRES.

2) C'est pourquoi les codes d'opération (instructions) doivent se trouver à des adresses PAIRES

3) Les accès en format octet peuvent se faire aussi bien à partir d'adresses paires qu'à partir d'adresses impaires.

En cas d'inobservation de ces règles, le travail normal est interrompu et un traitement d'exception est déclenché dans le processeur.

Etats de travail

Suivant l'état du bit superviseur de l'octet système du registre d'états, le 68000 travaille soit en état superviseur (bit superviseur = 1) soit en état utilisateur. L'état choisi détermine quelles opérations sont autorisées.

En état utilisateur, quelques instructions sont interdites et si on essaie de les utiliser malgré tout, cela conduit à un traitement d'exception. En état superviseur, le pointeur de pile A7- est utilisé, en état utilisateur, c'est le pointeur de pile A7. Il est d'ailleurs indifférent à cet égard que le pointeur de pile soit utilisé implicitement par une instruction (par exemple PEA) ou que le registre A7 soit indiqué explicitement comme objet ou comme source dans l'instruction.

Le 68000 connaît trois modes de travail fondamentaux:

1) Traitement normal

2) Traitement d'exception

3) Etat d'arrêt

L'exécution normale d'instructions constitue donc le premier état. Un cas exceptionnel de cet état est l'état stop de l'unité centrale. Cet état stop est provoqué par l'instruction STOP. Dans cet état, aucun autre accès à la mémoire n'est plus possible.

L'état d'arrêt est déclenché par des erreurs graves, lorsqu'on doit considérer que le système n'est plus en état de fonctionner. Le processeur n'abandonne ensuite cet état qu'aprés un signal externe de réinitialisation (RESET). L'état d'arrêt est provoqué par exemple par une erreur de bus lors du traitement d'exception d'une erreur de bus précédente (double erreur de bus). L'état d'arrêt ne doit cependant pas être confondu avec l'état stop!

L'état d'exception est déclenché par les interruptions, les instructions TRAP, le mode de travail trace ou d'autres conditions exceptionnelles.

En développant les états d'exception, il est possible de prescrire au processeur des réactions définies même dans des situations d'erreur ou dans des situations imprévues. Des interruptions de la périphérie peuvent par exemple demander au processeur de traiter un caractère reçu.

Le traitement d'exception se fait entièrement en mode superviseur. Lorsqu'un traitement d'exception est commencé, le processeur sauve l'ancien mot d'états sur la pile et met le bit superviseur. En état superviseur, toutes les instructions sont autorisées.

Le traitement d'exception peut être provoqué de l'intérieur ou de l'extérieur. Les erreurs d'adresse (accès à un mot à une adresse impaire), la division par zéro, des instructions directes (instructions TRAP) ou le mode trace peuvent par exemple provoquer de l'intérieur un traitement d'exception. Les exceptions peuvent être provoquées de l'extérieur par des interruptions, des erreurs de bus (erreur dans l'électronique du bus) ou le RESET.

Le 68000 dispose d'un grand nombre d'exceptions ce qui est d'ailleurs une de ses forces. Les exceptions permettent en effet de faire réagir le processeur d'un manière bien définie à des situations d'exception ou à des erreurs. En cela le 68000 surclasse beaucoup d'unités centrales de miniordinateurs et la plupart des autres microprocesseurs.

Les différentes situations d'exception sont numérotées et, suivant le cas qui se présente, le processeur va rechercher un vecteur d'exception dans la mémoire. Ce vecteur consiste en une adresse 32 bits stockée comme toutes les autres adresses. Les 1024 octets (ou 512 mots) inférieurs de la mémoire (domaine adressable) sont utilisés comme une table pour les 256 vecteurs.

Au début du traitement d'exception, le registre d'états est sauvé. Le bit superviseur est ensuite mis. Le bit trace est par ailleurs annulé pour qu'un traitement d'exception ne soit pas à nouveau engagé après la première instruction du traitement d'exception. Lorsque c'est une interruption qui a provoqué l'exception (ce qui n'est possible que si l'interruption avait un rang de priorité supérieur à celui indiqué par le masque d'interruption dans le registre d'états), le masque d'interruption du registre d'états est en outre fixé sur la nouvelle valeur. L'adresse de retour et le contenu de l'ancien registre d'états sont placés sur la pile du superviseur.

Le processeur peut obtenir le numéro de vecteur de deux façons différentes. Soit il la produit de façon interne (par exemple en cas d'erreurs de bus ou d'adresse mais aussi en cas de vecteurs internes d'interruption), soit, lors de ce qu'on appelle des interruptions de vecteur externe, il obtient le numéro de vecteur à travers le bus (directement ou indirectement de la machine qui a provoqué l'interruption). Le 68000 multiplie le numéro de vecteur par quatre (par un double décalage vers la gauche des bits du numéro de vecteur). Le numéro calculé est alors utilisé comme adresse. De cette adresse, il charge un long mot et le transfère dans le compteur de programme. Il commence alors l'exécution de l'instruction indiquée par le (nouveau) compteur de programme, passant ainsi au traitement de l'exception.

Pour pouvoir appeler en état superviseur des routines du système d'exploitation à partir de programmes utilisateur qui tournent en mode utilisateur, il existe 16 exceptions particulières, ce qu'on appelle les traps. Une instruction "TRAP #n" (avec un numéro "n" allant de 0 à 15) provoque un traitement d'exception. C'est alors dans la routine indiquée par le vecteur d'exception que la fonction du système d'exploitation correspondante sera exécutée.

De cette façon il est d'une part possible d'appeler de façon sélective des sections de programme qui tournent avec l'état superviseur sans que d'autre part le principe de protection des différents modes du 68000 soit affecté.

Nous n'allons cependant pas entrer dans une description détaillée des différentes exceptions car celles-ci ne sont pas indispensables à l'apprentissage et à la compréhension du langage machine.

Aperçu des modes d'adressage

Il faut indiquer dans les instructions quels opérandes doivent être utilisés. Les instructions 68000 se composent donc de deux parties:

1) le type d'opération à exécuter
2) l'adresse du ou des opérande(s)

Le terme d'adresse ne s'applique cependant pas ici uniquement aux adresses mémoire mais il peut également correspondre à un registre.

Les instructions peuvent déterminer l'adresse de l'opérande de trois façons différentes:

1) Spécification du registre: indication du numéro de registre dans l'instruction.

2) Adresse effective: on utilise différents modes d'adressage pour obtenir l'adresse. La sélection se fait à travers six bits de l'instruction (le champ correspondant à ces six bits est appelé adresse effective).

3) Indication implicite: l'opérande (un registre) est déjà contenu de façon implicite dans l'instruction.

Le 68000 possède 14 modes d'adressage utilisant les techniques de détermination de l'adresse de l'opérande que nous venons d'indiquer. Ces 14 modes d'adressage peuvent être répartis en six groupes principaux:

Registre direct:
On indique directement dans l'instruction un registre contenant l'opérande.

Registre indirect:
On indique dans l'instruction un registre contenant l'adresse de l'opérande.

Données absolu:
Des mots d'extension de l'instruction indiquent l'adresse de l'opérande dans la mémoire.

Relativement au compteur de programme:
On indique une distance par rapport au compteur de programme. Cela signifie qu'un mot ou un long mot sera additionné avec signe au compteur de programme. Le résultat sera l'adresse de l'opérande. Ces modes d'adressage permettent d'écrire des programmes pouvant tourner à n'importe quelle adresse du système et qui sont donc "relogeables".

Données immédiat:

Une extension de l'instruction (d'un ou deux mots) contient directement l'opérande.

Implicite:

L'opérande est indiqué de façon implicite par l'instruction. Les opérations de pile contiennent par exemple implicitement le pointeur de pile comme pointeur sur l'adresse de l'opérande.

Sur les 14 modes d'adressage, 13 produisent dans l'instruction ce qu'on appelle une "adresse effective". Cette adresse effective occupe un champ de deux fois trois bits dans le premier mot d'instruction. Suivant le mode d'adressage, le mot d'instruction peut être encore suivi d'autres mots qui déterminent l'opérande de façon plus précise.

Pour les instructions avec adresse effective, le mot d'instruction a la structure suivante:

Bits 0 à 2: Ces bits contiennent le champ du registre.
Bits 3 à 5: Ces bits contiennent le champ du mode. (Les bits 0 à 5 représentent l'adresse effective).
Bits 6 à 11: Ces bits contiennent soit l'adresse effective du second opérande, soit une partie du type d'instruction.
Bits 12 à 15: Ces bits contiennent le type d'instruction.

La table suivante présente les modes d'adressage qui peuvent être sélectionnés à travers une adresse effective. Nous avons utilisé les abréviations suivantes:

ARI: registre d'adresse indirect
An: numéro d'un registre d'adresse (3 bits, soit de 0 à 7)
Dn: numéro d'un registre de données (3 bits, soit de 0 à 7)
PC: compteur de programme (Program Counter)

Adresse effective

Mode	Registre	Mode d'adressage
000	Dn	Registre de données direct
001	An	Registre d'adresse direct
010	An	Registre d'adresse indirect
011	An	ARI avec postincrément
100	An	ARI avec prédécrément
101	An	ARI avec distance d'adresse
110	An	ARI avec distance d'adresse et index
111	000	Absolu court
111	001	Absolu long
111	010	relativement au PC avec distance d'adresse
111	011	relativement au PC avec distance d'adresse et index
111	100	Données immédiat

Aucune adresse effective n'est nécessaire avec l'adressage implicite.

Voici une description des modes d'adressage que nous n'avons pas encore décrits. Ces modes d'adressage seront certainement plus faciles à comprendre par les exemples de programme.

Registre d'adresse indirect avec postincrément:

L'adresse de l'opérande se trouve dans le registre d'adresse indiqué.

Après l'opération, le registre d'adresse est augmenté de 1, 2 ou 4 suivant la longueur de l'opérande. S'il s'agit du pointeur de pile, l'adresse est augmentée d'au moins 2 pour que le pointeur de pile conserve une valeur paire. Ce mode d'adressage permet de constituer d'autres piles.

Registre d'adresse indirect avec prédécrément:

Le registre indiqué est diminué de 1, 2 ou 4. Si le registre d'adresse est le pointeur de pile, il est diminué de 2 ou 4 pour rester pair. Cela empêche des exceptions d'erreur d'adresse. L'accès s'effectue à l'adresse se trouvant après la soustraction dans le registre d'adresse indiqué.

Registre d'adresse indirect avec distance d'adresse et index:

Ce mode d'adressage additionne un autre mot au contenu du registre d'adresse indiqué. Un mot d'extension est donc nécessaire. L'adresse de l'opérande est la somme du contenu du registre et de la valeur 16 bits signée de la distance d'adresse.

Registre d'adresse indirect avec distance d'adresse et index:

Ce mode d'adressage est semblable au précédent. Le mot d'extension est cependant organisé différemment. L'octet inférieur représente une distance d'adresse 8 bits qui sera additionnée avec signe. L'octet supérieur contient des indications sur le type du registre d'index (registre d'adresse ou de données), sur la taille de l'index (mot avec signe ou long mot) et le numéro du registre. Le contenu (ou contenu partiel) de ce registre est additionné avec signe.

Absolu court:

Avec ce mode d'adressage, il y a un mot d'extension qui contient l'adresse absolue sur 16 bits avec signe de l'opérande. Cette adresse 16 bits est étendue sur 32 bits, en tenant compte du signe, avant d'être utilisée.

Absolu long:

Avec ce mode d'adressage, il y a deux mots d'extension qui contiennent l'adresse de l'opérande. La partie de plus grande valeur de l'adresse 32 bits se trouve dans le premier mot d'extension, la partie de moindre valeur dans le second mot d'extension.

Relativement au compteur de programme (PC) avec distance d'adresse:

Pour ce mode d'adressage, un mot d'extension est nécessaire. L'adresse de l'opérande est donnée par l'addition du compteur de programme et de la valeur 16 bits avec signe de la distance contenue dans le mot d'extension.

Relativement au compteur de programme avec distance d'adresse et index:

Ce mode d'adressage est semblable au précédent. Le mot d'extension a cependant une structure différente. L'octet inférieur représente une distance d'adresse 8 bits qui sera additionnée avec signe. L'octet supérieur contient des indications sur le type de registre d'index (registre d'adresse ou de données), sur la taille de l'index (mot avec signe ou long mot) et le numéro du registre.

Le contenu (ou contenu partiel) de ce registre est additionné avec signe. Ce mode d'adressage ainsi que l'adressage simple relativement au compteur de programme permettent d'écrire des programmes pouvant tourner à n'importe quelle adresse, c'est-à-dire des programmes relogeables. Il est particulièrement intéressant dans les systèmes multi-postes de pouvoir faire tourner des programmes dans les zones de la mémoire système qui sont justement disponibles.

Données immédiat:

L'opérande suit directement l'instruction. Suivant la longueur de l'opérande, un ou deux mots d'extension sont nécessaires. Pour les opérations octet, c'est l'octet de moindre valeur du mot d'extension qui est utilisé. Pour les opérations long mot, le mot de plus grande valeur figure dans le premier mot d'extension, le mot de moindre valeur dans le second mot d'extension.

Aperçu du jeu d'instructions

Le jeu d'instructions du 68000 comprend 56 instructions. C'est peu comparé à d'autres processeurs. Le 68000 est cependant rendu très puissant par ses 14 modes d'adressage. Si chaque instruction avait un nom particulier, on compterait en réalité plus de 1000 instructions. Celles-ci ne peuvent être maîtrisées que grâce à la structure claire du 68000 avec ses modes d'adressage. Le jeu d'instructions assembleur soutient par ailleurs particulièrement la programmation modulaire et la programmation dans un langage évolué.

Le 68000 est une "vraie" machine deux adresses. Cela signifie qu'aussi bien la source que l'objet d'une opération peuvent être dans la mémoire. Il est ainsi possible d'effectuer des décalages dans la mémoire directement de case mémoire à case mémoire. Sur la plupart des autres processeurs, le contenu d'une case mémoire doit d'abord être chargé dans un registre du processeur puis être transféré avec une autre instruction dans une autre case mémoire.

Une instruction du 68000 se compose d'un à cinq mots, soit de deux à dix octets. La longueur et le type de l'instruction sont déterminés par le premier mot d'instruction. Les instructions du 68000 sont structurées de façon systématique.

On peut répartir le jeu d'instructions du 68000 dans les groupes suivants:

- opérations arithmétiques (avec des nombres entiers)
- instructions BCD
- instructions logiques
- instructions de décalage et de rotation
- instructions de manipulation de bits
- instructions de transfert de données
- instructions de commande du programme

De nombreuses instructions du 68000 peuvent traiter plusieurs types de données différents. L'instruction MOVE permet par exemple de transférer des octets, des mots ou des longs mots. Outre les différents modes d'adressage qu'offrent les instructions, on dispose donc également de différentes longueurs d'opérande.

STRUCTURES DE PROGRAMME ET DE MEMOIRE

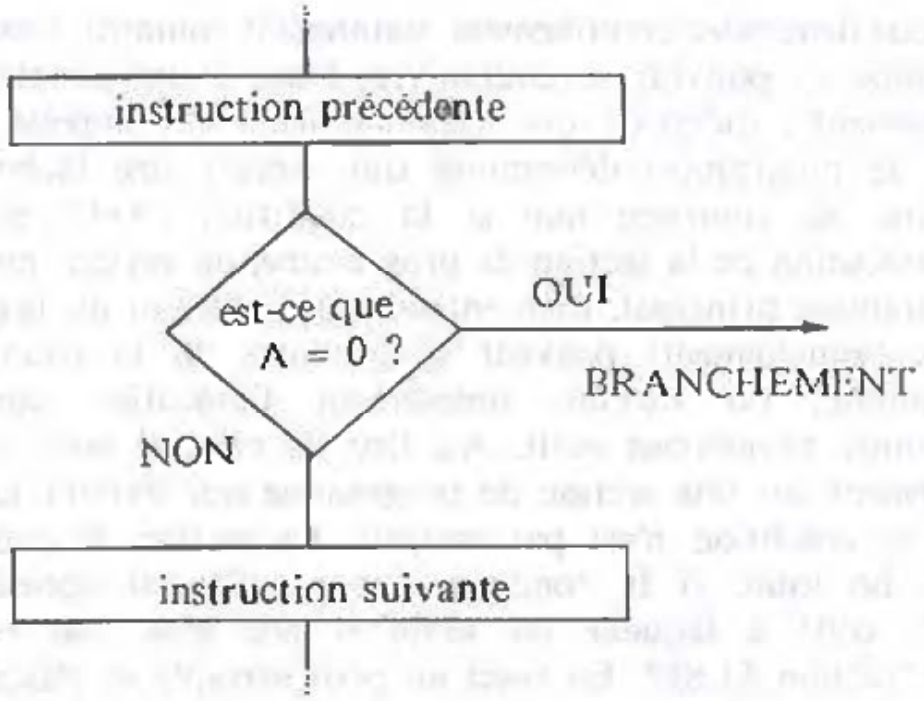
1) Introduction
2) Procédures et fonctions
3) Structures de mémoire

Sur les premiers ordinateurs, les données et le programme étaient rigoureusement séparés entre eux. Alors que les données se trouvaient dans les registres et les cases mémoire, le programme était amené de l'extérieur. Les ordinateurs étaient par exemple programmés à travers des cartes perforées.

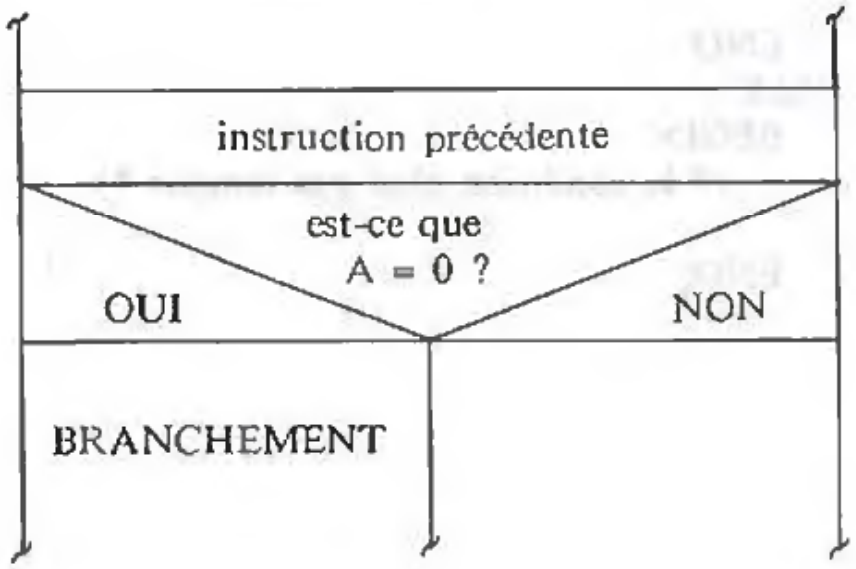
Les ordinateurs d'aujourd'hui, les ordinateurs digitaux sont ce qu'on appelle des ordinateurs "von Neumann" dont la caractéristique principale est que le programme et les données sont "conservés" dans la même mémoire. L'unité centrale possède simplement un registre (appelé compteur de programme) qui indique l'endroit de la mémoire dans lequel figure la prochaine instruction à exécuter. Les instructions figurent ainsi simplement les unes à la suite des autres dans la mémoire. En modifiant le contenu du compteur de programme, on peut amener l'unité centrale à sortir de l'ordre normal des instructions en mémoire. C'est ainsi que des sauts à l'intérieur du programme sont possibles. On peut ainsi écrire des programmes qui réagissent différemment suivant la situation PENDANT le déroulement du programme. C'est là qu'est l'avantage principal des ordinateurs "von Neumann" alors que sur les anciens ordinateurs le programme se déroulait toujours comme une longue chaîne unique d'instructions.

Nous allons vous montrer dans ce chapitre à titre d'exemples quelques possibilités de structure de programmes. Vous trouverez d'autres exemples et notamment des exemples pratiques dans les chapitres suivants de l'ouvrage.

Le fondement essentiel des structures à déroulement conditionnel de programmes est le branchement simple. A une condition déterminée, on saute à une section de programme déterminée. Si la condition n'est pas remplie, l'instruction suivante de la chaîne d'instructions (du programme) est exécutée. Nous allons par exemple demander si la variable "A" vaut zéro.



Organigramme (d'après la norme DIN 66001)



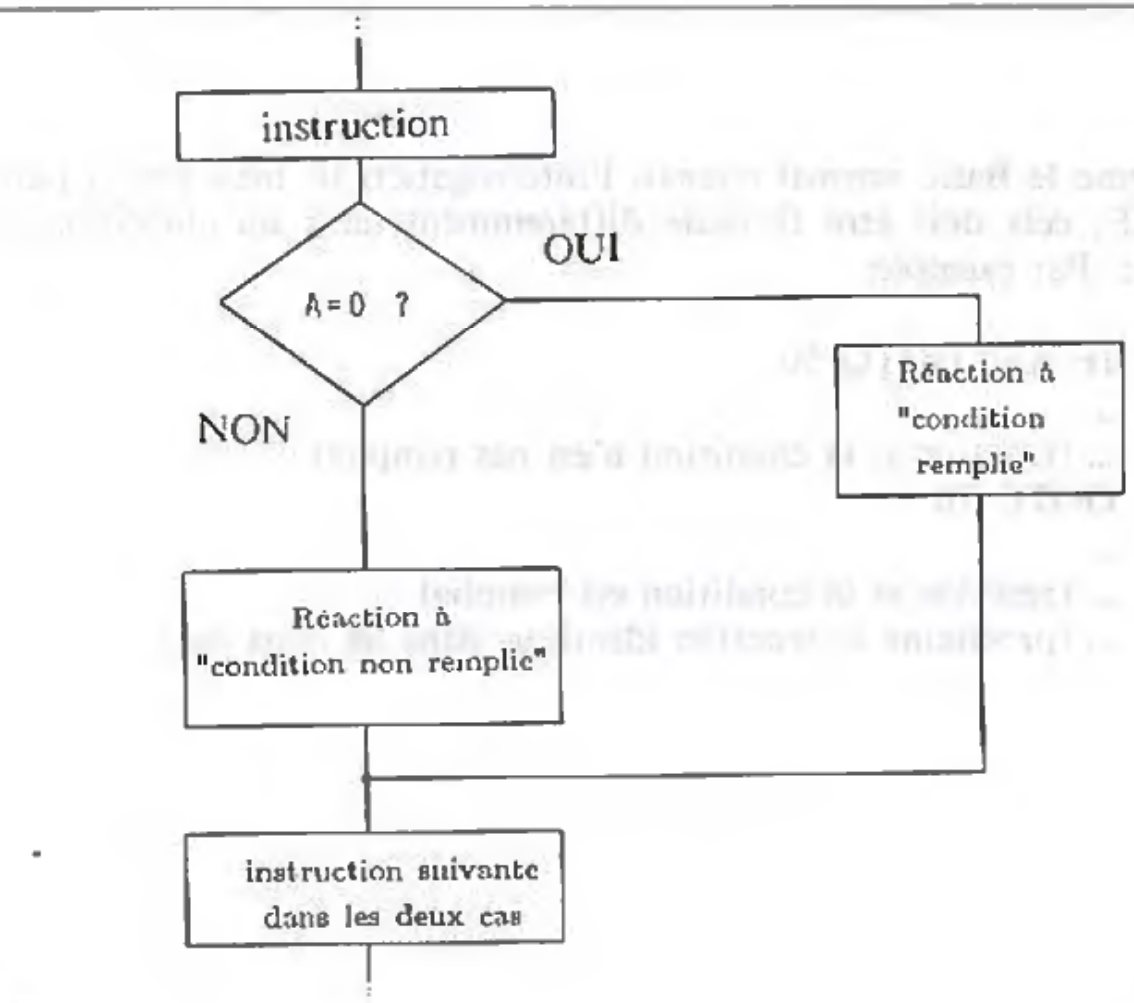
Structogramme d'après Nassi-Schneiderman

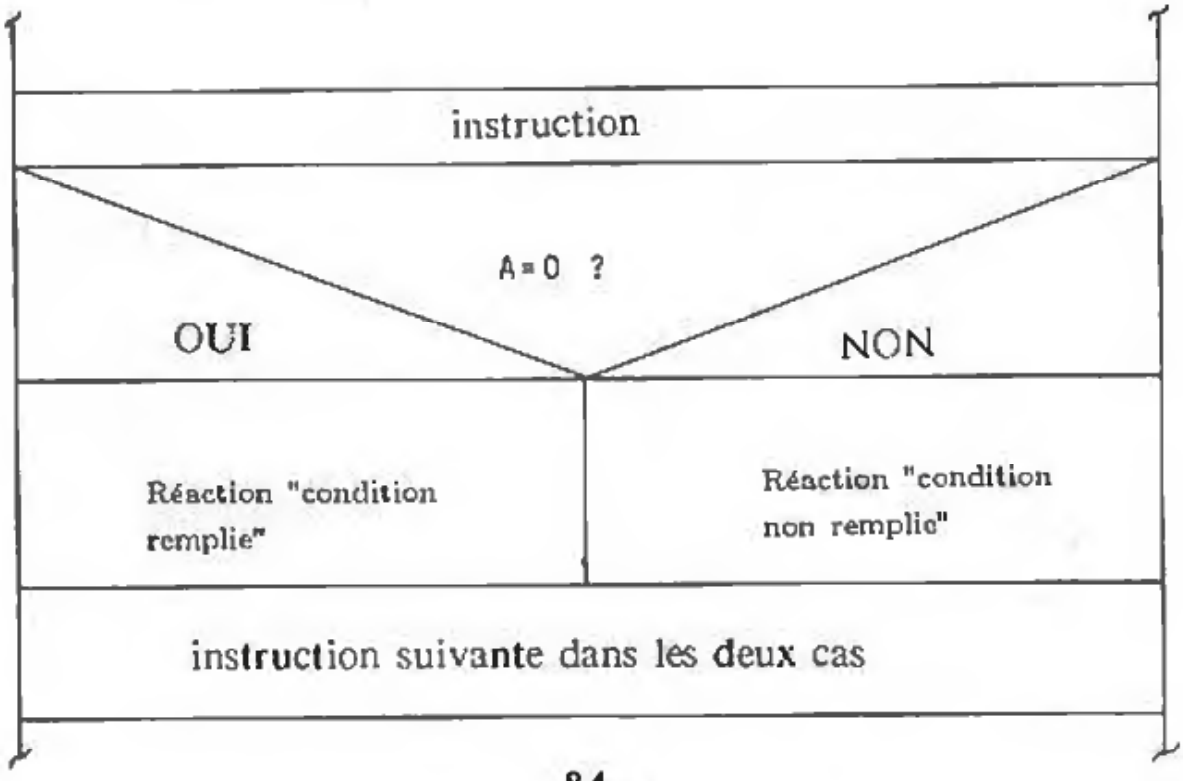
Vous vous demandez certainement maintenant comment l'exécution du programme va pouvoir se poursuivre. Nous avons simplement écrit "branchement", qu'est-ce que cela signifie? On appelle ainsi une section de programme déterminée qui exécute une tâche qui n'est nécessaire ou autorisée que si la condition (A=0) est remplie. Après exécution de la section de programme, on revient normalement au programme principal. Bien entendu, à l'intérieur du branchement, d'autres branchements peuvent se produire. Si la condition n'est pas remplie, on exécute simplement l'exécution suivante du programme, avons-nous écrit. Au lieu de cela, il peut y avoir un branchement sur une section de programme qui traitera justement le cas où la condition n'est pas remplie. La section de programme à laquelle on saute si la condition est remplie est appelée "section THEN", celle à laquelle on saute si elle n'est pas remplie est appelée "section ELSE". En voici un petit exemple en Pascal:

```
IF A=0 THEN
        BEGIN
        .  (* la condition est remplie *)
        .
        END
    ELSE
        BEGIN
        .  (* la condition n'est pas remplie *)
        .
        END;
```

Comme le Basic normal connaît l'interrogation IF mais pas la partie ELSE, cela doit être formulé différemment dans un algorithme en Basic. Par exemple:

```
10  IF A=0 GOTO 50
20  ...
30  ... (réaction si la condition n'est pas remplie)
40  GOTO 70
50  ...
60  ... (réaction si la condition est remplie)
70  ... (prochaine instruction identique dans les deux cas)
```


instruction
A=0 ?
OUI
NON
Réaction à "condition remplie"
Réaction à "condition non remplie"
instruction suivante dans les deux cas


instruction
A=0 ?
OUI
NON
Réaction "condition remplie"
Réaction "condition non remplie"
instruction suivante dans les deux cas

Il est également possible de construire des BOUCLES dans un programme avec des interrogations appropriées et des sauts qui ne seront exécutés que si la condition de l'interrogation est remplie (ou que si elle n'est pas remplie). On peut ainsi obtenir qu'une action soit exécutée un certain nombre de fois consécutives. On distingue deux types de boucles:

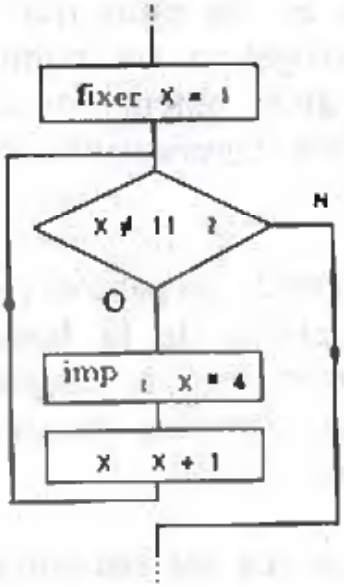
1) La condition est testée avant que ne commence le parcours de la boucle. Si elle est remplie, les instructions de la boucle seront exécutées une fois puis la condition sera à nouveau testée. Si par contre la condition n'est pas remplie, on saute à l'endroit du programme suivant la boucle.

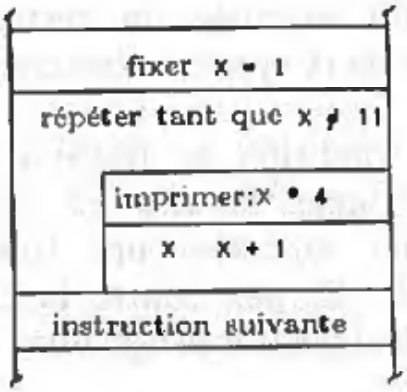
2) La condition est testée après que la boucle ait été parcourue. Les instructions de la boucle sont donc toujours exécutées au moins une fois. La condition est ensuite testée. Si elle est remplie, on exécute habituellement la prochaine instruction du programme. Sinon, on retourne à la première instruction de la boucle. La boucle est donc parcourue une seconde fois. La condition est ensuite à nouveau testée.

Pour toutes les boucles il faut faire attention à ce qu'elles s'achèvent bien à un moment ou un autre. Si la condition ne peut jamais être remplie du fait d'une erreur dans le programme ou lorsqu'une condition impossible à remplir est prescrite, l'exécution de la boucle ne s'arrête jamais.

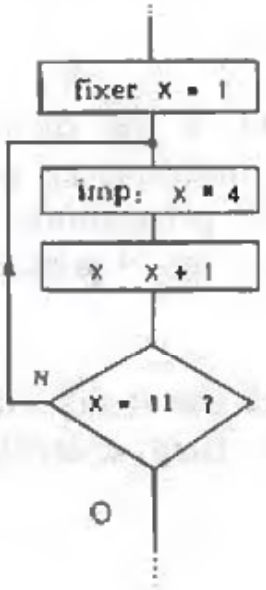
De nombreux langages de programmation possèdent des instructions spéciales pour construire des boucles. Voici à titre d'exemple l'impression en Basic de la table de 4:

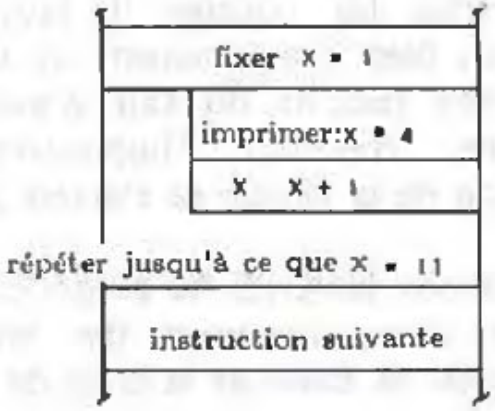
```
10 FOR X = 1 TO 10
20 PRINT 4 * X
30 NEXT X
```





INTERROGATION AVANT PARCOURS DE LA BOUCLE





INTERROGATION APRES PARCOURS DE LA BOUCLE

Les deux illustrations montrent les deux variantes fondamentales d'une boucle. Les programmes représentés par ces illustrations ont également pour but de calculer et imprimer la table de 4.

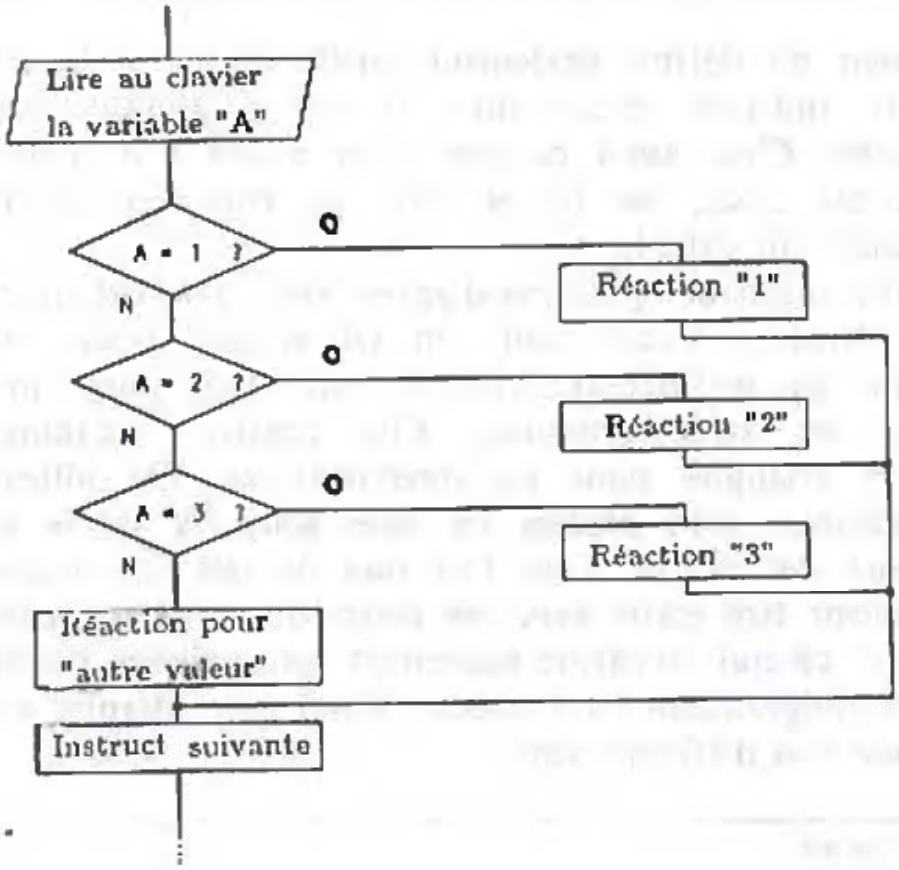
A l'examen de ces exemples, il apparaît qu'on teste une fois s'il y a égalité et une fois s'il y a inégalité. Dans le premier exemple, la boucle est interrompue si la condition n'est plus remplie. Si X atteint la valeur 11, la boucle doit être abandonnée. C'est pourquoi on demande si X est différent de 11. Si X n'est plus différent de 11, c'est-à-dire si X égale 11, la boucle est abandonnée.

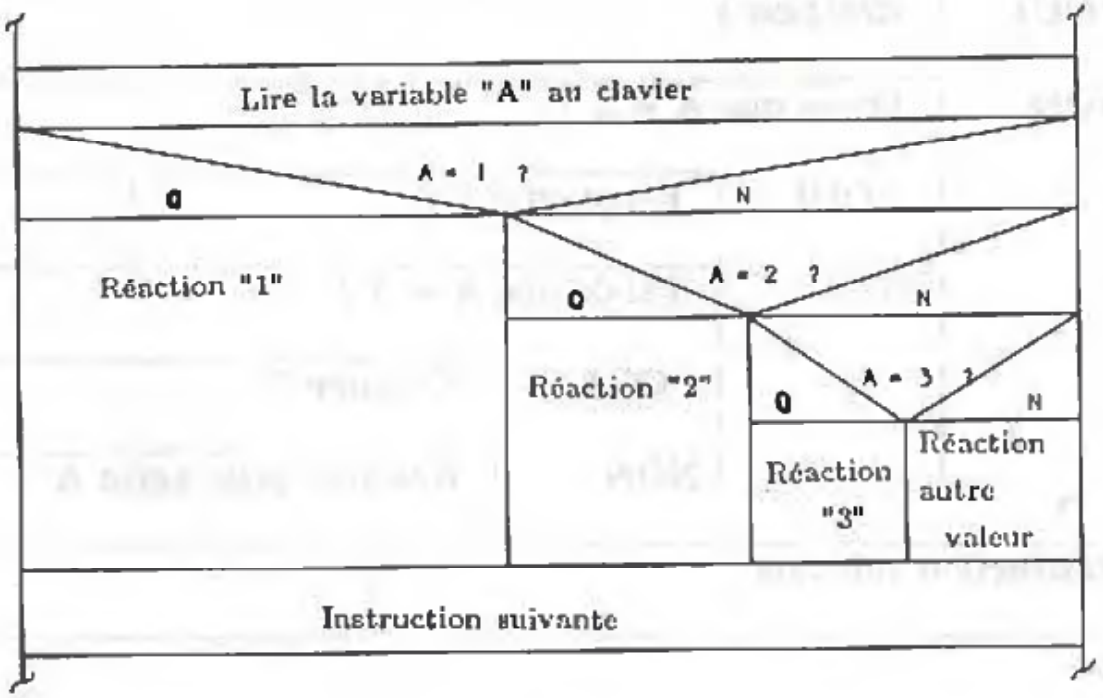
On peut imaginer encore de nombreuses autres variantes de boucles qui conduiront toutes au même résultat. Vous pourriez donc à titre d'exercice essayer de réaliser d'autres organigrammes. Vous pourriez par exemple commencer avec la valeur X=0 et augmenter X de un avant l'impression.

On voit souvent également des boucles au milieu desquelles on teste la condition d'interruption. Un tel procédé ne peut cependant que rarement être justifié. Le plus souvent cela provoque en effet une certaine inintelligibilité du programme. Par ailleurs des erreurs apparaissent ainsi fréquemment car il devient très difficile de suivre l'évolution exacte de la valeur actuelle de la variable de comptage (de la variable qui se modifie à chaque parcours de boucle). C'est pourquoi il est conseillé de ne modifier la variable de comptage qu'au début ou à la fin d'une boucle.

Il arrive souvent dans les programmes que le fait d'appuyer sur une touche déclenche une action déterminée du programme. Le code de la touche qui a été appuyée est généralement stocké dans une variable. Le programme doit tester toutes les variantes de la variable (ou de la touche enfoncée) et engager des réactions appropriées. Le plus souvent certaines procédures ou fonctions sont déclenchées en fonction de la touche qui a été enfoncée.

Souvent il arrive également qu'une seule section de programme soit exécutée, comme lors d'un branchement normal.


Lire au clavier
la variable "A"
A = 1 ?
O
N
Réaction "1"
A = 2 ?
O
N
Réaction "2"
A = 3 ?
O
N
Réaction "3"
Réaction pour
"autre valeur"
Instruct suivante


Lire la variable "A" au clavier
A = 1 ?
O
N
Réaction "1"
A = 2 ?
O
N
Réaction "2"
A = 3 ?
O
N
Réaction
"3"
Réaction
autre
valeur
Instruction suivante

Le plus souvent on définit également quelle doit être la réaction en cas d'état indéfini, c'est-à-dire si on a appuyé sur une mauvaise touche. C'est aussi ce que nous avons fait dans notre exemple. Souvent aussi, on lit et relit un caractère jusqu'à ce qu'on en obtienne un valable.
Comme on voit, un structogramme d'après Nassi Schneiderman peut vite devenir illisible. Avant tout, un tel schéma peut rarement être réalisé sur un ordinateur. Mais il existe une autre méthode pour dessiner les structogrammes. Elle consiste notamment à abandonner les triangles pour les interrogations. Par ailleurs les différentes variantes sont placées les unes sous les autres et non les unes à côté des autres. Cela fait que de tels structogrammes peuvent également être écrits avec des programmes de traitement de texte "normaux" ce qui constitue également un avantage décisif par rapport aux organigrammes DIN 66001. Voici notre dernier exemple écrit encore une fois différemment:

```
+-------------------------------------------------------------+
| Lire A au clavier                                           |
+-------------------------------------------------------------+
| Est-ce que A = 1 ?                                          |
|                                                             |
|  OUI  | Réaction 1                                          |
|                                                             |
| NON   | Est-ce que A = 2 ?                                  |
|       |                                                     |
|       |  OUI  | Réaction 2                                  |
|       |                                                     |
|       | NON   | Est-ce que A = 3 ?                          |
|       |       |                                             |
|       |       |  OUI  | Réaction 3                          |
|       |       |                                             |
|       |       | NON   | Réaction pour autre A               |
+-------------------------------------------------------------+
| Instruction suivante                                        |
+-------------------------------------------------------------+
```

Il existe encore d'autres techniques pour représenter des structogrammes de façon claire. L'exemple que nous avons choisi a encore l'inconvénient de nécessiter une feuille de travail assez large lorsqu'on veut représenter des interrogations imbriquées les unes dans les autres. Cependant le structogramme que nous avons choisi, conçu en fonction du travail sur ordinateur, ainsi que celui d'après Nassi-Schneiderman sont justement particulièrement appropriés à l'apprentissage du langage machine.

Les organigrammes DIN 66001 sont également tout à fait remarquables par leur clarté mais ils présentent l'inconvénient que, si on les utilise à l'exclusion de toute autre méthode, on obtient facilement un code spaghetti plutôt qu'un programme structuré. Il n'est certainement pas facile d'admettre que même les petits programmes doivent être programmés "proprement" mais finalement c'est à notre avis la meilleure possibilité qui vous est offerte d'apprendre les techniques nécessaires pour les programmes de dimensions plus importantes. Nous traiterons dans la section de chapitre suivante de certains aspects de la programmation structurée.

Procédures et fonctions

Les différentes interrogations, branchements et boucles permettent de façon évidente d'écrire déjà des programmes très puissants. Dans la pratique il arrive cependant souvent qu'une section de programme déterminée soit nécessaire en divers endroits du programme. Pourquoi alors réécrire bêtement le même texte de programme à tous les endroits où il est nécessaire?

C'est pourquoi on a créé la technique des SOUS-PROGRAMMES. Ceux-ci existent bien sûr également en Basic et ils sont employés de façon analogue en langage machine. Un exemple:

```
10  ...            (Programme)
20  GOSUB 100      (Appel de sous-programme)
30  ...            (Première instruction après le sous-programme)
100 ...            (Début du sous-programme)
110 ...            (Sous-programme)
120 RETURN         (Retour au programme appelant)
```

Un appel de sous-programme se fait comme un saut normal à un autre endroit du programme. Mais l'adresse de la prochaine instruction à exécuter après le retour du sous-programme est en outre placée sur la pile système (nous expliquerons bientôt ce qu'on entend par là). Dès que l'unité centrale reçoit l'instruction de retour au programme appelant, elle lit l'adresse figurant sur la piste système et sait ainsi où elle doit sauter.

Les adresses de retour sont simplement "empilées" lorsque plusieurs appels successifs sont imbriqués, c'est-à-dire lorsqu'un sous-programme en appelle lui-même un autre. Lors du retour, le processeur lit toujours l'adresse placée en haut de la pile en la retirant en même temps de la pile.

L'inconvénient de cette procédure est qu'aucun paramètre ne peut être transmis directement au sous-programme et que le sous-programme ne peut pas non plus restituer directement de valeur au programme appelant.

On peut certes écrire des programmes puissants même avec cet inconvénient, il n'est qu'à voir tout ce qui a déjà pu être programmé en Basic.

On aura cependant toujours des problèmes lorsqu'on voudra utiliser les routines d'un programme dans un autre programme.

Par ailleurs les programmes écrits de cette façon finissent presque toujours par devenir illisibles car on a du mal à suivre les paramètres. Par ailleurs un certain "chaos" dans les sauts (on s'en tient rarement dans la pratique à une utilisation exclusive des appels de sous-programme et retours) d'un programme peut facilement donner un produit incompréhensible et qu'on appelle code spaghetti dans le jargon des techniciens.

On peut cependant définir autrement les sous-programmes. En Pascal mais aussi dans d'autres langages tels que Ada, Modula, C etc., il existe à cet effet ce qu'on appelle des PROCEDURES et des FONCTIONS. Une fonction ou une procédure est appelée comme un sous-programme habituel (en Pascal on n'a bien sûr pas besoin d'indiquer le numéro de ligne comme en Basic, mais cela est sans importance ici). Mais lors de l'appel, des paramètres définis auparavant dans le programme sont transmis au sous-programme. Ces paramètres sont stockés dans le sous-programme dans ce qu'on appelle des variables locales et ils sont normalement effacés lors du retour au programme principal. Il est ainsi possible d'éviter des erreurs dûes à une utilisation incorrecte des variables dans des programmes de dimensions importantes. Et surtout, les procédures et fonctions peuvent aisément être transférées dans d'autres programmes.

A la fin de la procédure ou fonction, il y a retour à une adresse placée sur la pile, comme pour les sous-programmes ordinaires. En même temps, les variables locales, c'est-à-dire les variables qui n'ont été utilisées que dans le sous-programme, sont alors effacées. Pour une fonction, une valeur appelée VALEUR DE FONCTION est en outre envoyée en retour au programme appelant. C'est là qu'est la différence entre les procédures et les fonctions. On peut bien sûr utiliser également des groupes de valeurs comme valeur de fonction.

La valeur de fonction peut être considérée comme le "résultat" de la fonction.

Voici maintenant un exemple de fonction et un exemple de procédure, formulés en Pascal:

```
FUNCTION  CARRE (X: REAL): REAL;
    BEGIN
      CARRE := X*X;
    END;
```

Cette fonction calcule le carré d'un nombre. L'expression "CARRE (9)" donnerait par exemple "81". Voici maintenant une procédure qui sort un nombre déterminé de lignes vides. L'expression LIGNEV (10) donnerait par exemple 10 lignes vides.

```
PROCEDURE  LIGNEV (N: INT);
    VAR I: INT;
    BEGIN
      FOR I:=1 TO N DO
        WRITELN;
    END;
```

Le langage machine du 68000 ne soutient pas plus les procédures et fonctions que les autres processeurs. Il n'existe le plus souvent que des instructions d'appel de sous-programme et de retour. Le 68000 présente même l'avantage par rapport à d'autres processeurs de disposer d'instructions qui permettent de réserver ou de libérer de la place en mémoire pour des variables locales. Cependant, pour pouvoir programmer en langage machine avec des fonctions et des procédures, il faut se créer une CONVENTION DE PROCEDURE.

Une convention de procédure consiste à définir par avance comment les paramètres seront transmis au sous-programme et comment celui-ci renverra des valeurs de fonction. Une convention de procédure définit en outre le plus souvent quels registres devront à nouveau contenir leur ancienne valeur après le retour. Nous vous présentons au chapitre VII la convention de procédure que nous avons utilisée pour les exemples du présent ouvrage.

Structures de mémoire

Nous avons déjà indiqué au chapitre second que la mémoire est à une dimension du point de vue du logiciel. La mémoire apparaît comme une chaîne d'octets dont chacun possède une adresse qui lui est propre. Du point de vue de l'électronique, les octets sont toujours regroupés deux par deux puisque le 68000 est un processeur 16 bits. C'est pourquoi les accès à un mot ne peuvent se produire qu'à des adresses paires. Le 68000 n'a cependant besoin que d'un seul cycle de lecture ou d'écriture pour un accès de mot.

Dans la pratique, on a cependant souvent besoin non d'une mémoire à une dimension mais de tableaux à plusieurs dimensions. Ceux-ci peuvent aisément être réalisés avec un logiciel approprié.

Il faut pour cela limiter la taille du tableau. C'est ainsi que le tableau à plusieurs dimensions pourra être organisé dans la mémoire disponible. Pour un tableau à deux dimensions de $100 * $100 (256 * 256) éléments d'un octet, cela pourrait être réalisé ainsi: Nous partons du principe que le tableau doit commencer à l'adresse $10000.

Les différentes lignes du tableau se trouveront donc aux adresses suivantes:

Ligne $0: $10000..$100FF
Ligne $1: $10100..$101FF
Ligne $2: $10200..$102FF
.
.
Ligne $FE: $1FE00..$1FEFF
Ligne $FF: $1FF00..$1FFFF

L'élément $30 de la ligne $2 figurerait donc à l'adresse $10230. Les tableaux de plus grandes dimensions peuvent être définis également d'après ce même principe. Il faut noter qu'on ne doit jamais dépasser la limite d'une ligne car cela équivaudrait à accéder à la ligne suivante.

.

Nous avons déjà utilisé une mémoire en pile dans la dernière section de chapitre pour stocker les adresses de retour. Mais qu'est-ce donc exactement qu'une pile?

Un autre terme pour la pile est LIFO, abréviation de Last In First Out. Cela décrit en effet parfaitement le mode de fonctionnement d'une pile. Ce sont les données placées en dernier sur la pile qui en seront retirées les premières. C'est donc comme une pile de feuilles de notes qu'on constituerait en plaçant une nouvelle feuille sur la pile lorsqu'il s'agit de ranger des informations. On lirait ensuite les informations en retirant les feuilles de la pile, en commançant toujours par la feuille placée au sommet de la pile. Ce serait donc bien toujours la dernière feuille placée sur la pile qui en serait retirée en premier.

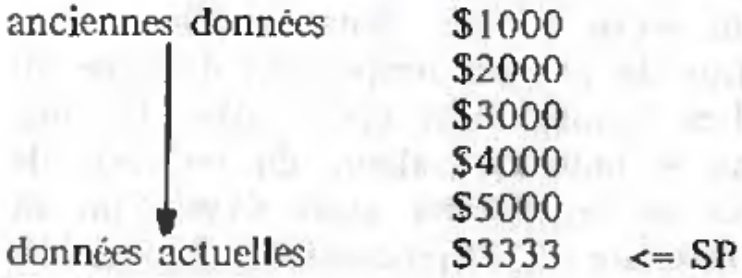
Pratiquement, une pile est généralement réalisée de la façon suivante. Une zone déterminée de la mémoire est réservée comme pile. Un registre de l'unité centrale, le pointeur de pile, désigne un mot de cette zone mémoire. Ce mot constitue un pointeur sur l'élément de la pile le plus élevé. Si des données doivent être placées sur la pile, le pointeur de pile est simplement diminué du nombre d'octets à écrire. Les données sont écrites dans la zone d'adresses entre les ancienne et nouvelle valeurs du pointeur de pile. La lecture des données de la pile est aussi simple. on lit simplement le contenu de la mémoire de l'emplacement indiqué par le pointeur de pile. Les données lues sur la pile sont alors déclarées non valables par le fait que le pointeur de pile est à nouveau augmenté du nombre correspondant d'octets lus.

anciennes données	$1000	
	$2000	
	$3000	
	$4000	
	$5000	
données actuelles	$3333	<= SP

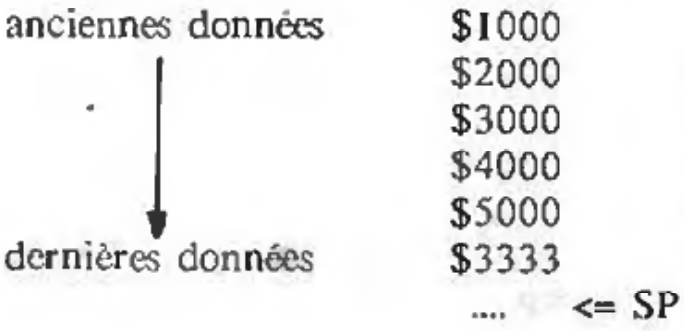
PRINCIPE D'UNE PILE SUR LE 68000

ECRITURE SUR LA PILE

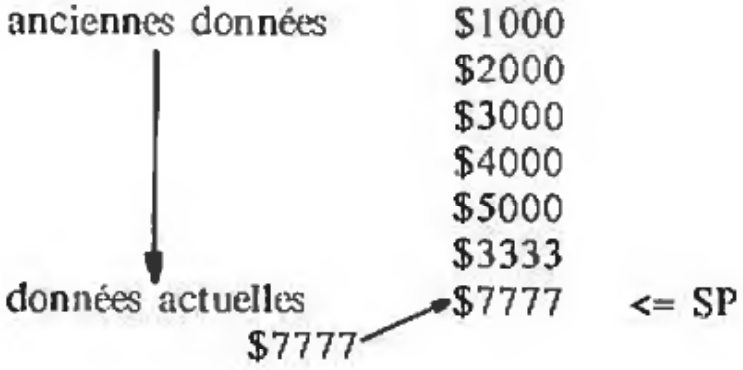
Etat originel



Diminuer le pointeur de pile

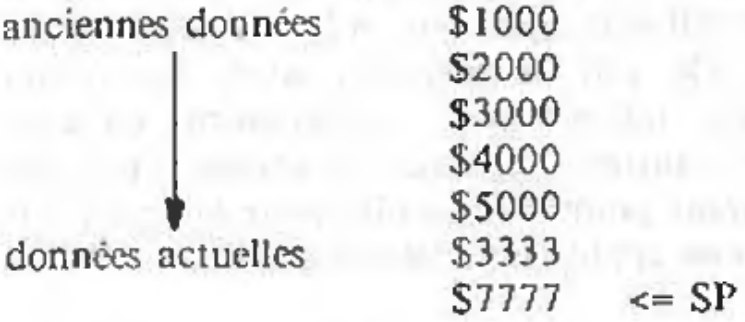


Ecrire les données

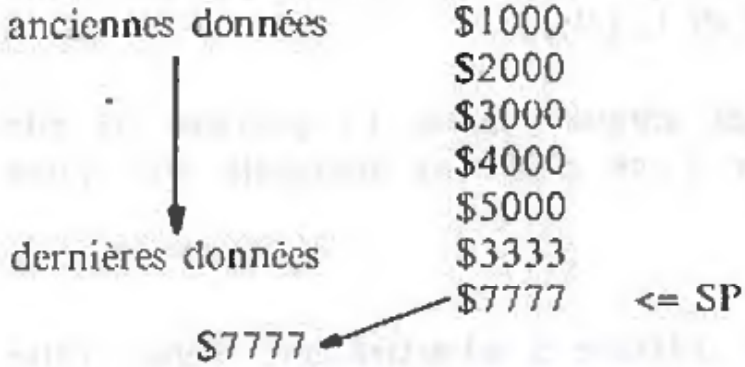


LECTURE SUR LA PILE

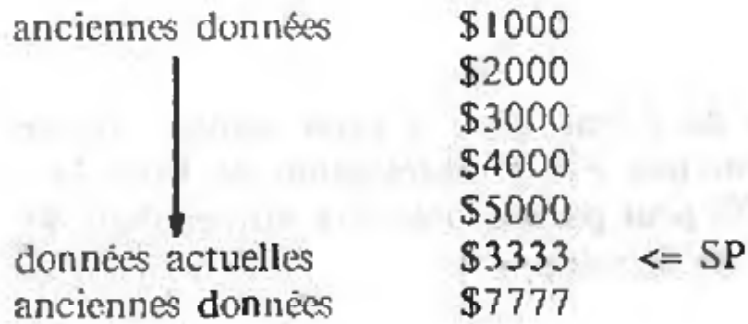
Etat originel



Lire les données



Augmenter le pointeur de pile



Sur le 68000, c'est le registre d'adresse A7 (ou A7- en mode superviseur) qui est défini comme pointeur de la pile système. Il est donc utilisé automatiquement pour stocker l'adresse de retour lors des appels de sous-programmes, par exemple après l'instruction "JSR". D'autres instructions utilisent également A7, ou A7- de façon implicite comme pointeur de pile. Cependant, avec les modes d'adressage "registre d'adresse indirect avec prédécrément" ou avec "postincrément", tous les autres registres d'adresse peuvent également être employés comme pointeurs de pile pour les piles que l'utilisateur a définies pour une application déterminée.

Sur le 68000, l'emploi du pointeur de pile est mise en oeuvre de la façon suivante:

1) Le pointeur de pile indique toujours l'entrée actuelle, celle qui figure tout en haut de la pile.

2) La pile "croît" vers les adresses basses. Le pointeur de pile est donc diminué lors d'une entrée et augmenté lors d'une lecture.

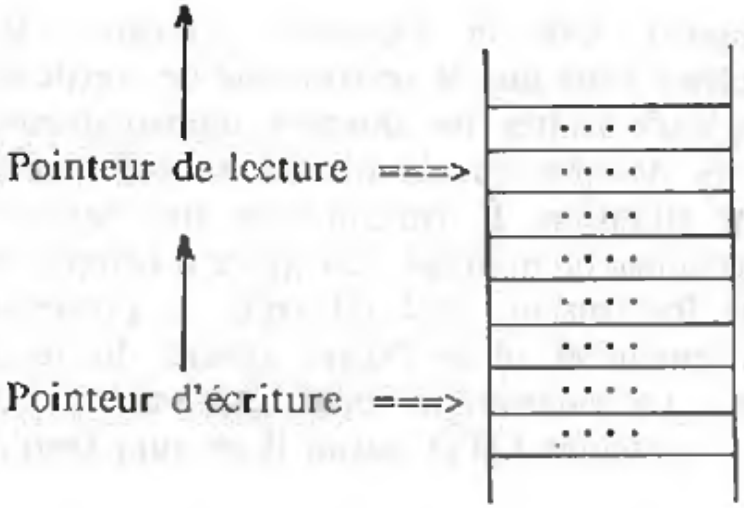
Il est certainement un peu difficile à admettre que d'après cette définition les données figurant physiquement "en bas" de la mémoire se trouvent logiquement "en haut" de la pile. Mais malheureusement de telles définitions doivent être prises et admises telles quelles.

Pour le stockage provisoire de données, on a assez souvent besoin de ce qu'on appelle des mémoires FIFO, abréviation de First In - First Out. Une mémoire FIFO peut par exemple être utilisée dans des programmes de transmission de données.

Il arrive souvent que le récepteur fournisse les données à intervales réguliers mais que le programme de réception lui-même ne puisse pas toujours traiter les données immédiatement et qu'il ne stocke donc des données sur la disquette que de temps en temps. Dans une telle situation, la transmission des données peut malgré tout être programmée de manière sûre grâce à l'emploi d'une mémoire FIFO. Chaque fois qu'un octet est reçu, le processeur interrompt son activité normale et place l'octet venant du récepteur dans la mémoire FIFO. Le programme principal ira ensuite chercher les données dans la mémoire FIFO quand il en aura besoin.

Cela entraîne bien sûr certains problèmes. Aucune mémoire FIFO ne peut avoir une taille illimitée. Il faut donc veiller à ce que les données soient suffisamment vite traitées et dirigées plus loin. Le stockage des données dans la mémoire FIFO doit en outre s'effectuer relativement rapidement. Sinon des pertes de données pourraient être provoquées par le fait que le stockage de données ne serait pas encore achevé que déjà les suivantes auraient été reçues.

Le principe de la mémoire FIFO pose un problème pratique. Une pile croît en effet dans une seule direction mais elle est toujours "défaite" en partant du haut. Un octet de la zone mémoire de la pile est donc utilisé et réutilisé en permanence. Avec la mémoire FIFO, par contre, chaque octet n'est écrit qu'un fois. La "chaîne" de données "se déplace" ensuite d'un emplacement.



MEMOIRE FIFO

Les mémoires FIFO sont réalisées pratiquement comme des MEMOIRE CYCLIQUES. Une mémoire cyclique est tout simplement un cercle d'octets. Chacun de ces octets a une adresse déterminée. Il existe pour cela deux pointeurs. Le POINTEUR D'ECRITURE indique toujours l'adresse à laquelle il faudra écrire la prochaine fois. Il est augmenté, après chaque écriture, du nombre d'octets écrits. Le POINTEUR DE LECTURE indique toujours l'adresse à laquelle le LECTEUR (le programme ou les programmes qui retirent les données de la mémoire FIFO) trouvera le prochain octet. Après qu'un octet ait été lu, le pointeur de lecture doit bien sûr également être augmenté d'un octet.

Du fait de cette structure, une situation d'erreur de plus qu'avec les mémoires LIFO doit être prise en compte: lorsque les LECTEUR et "INSCRIVEUR" travaillent à des vitesses très différentes, ils risquent de se rejoindre dans une mémoire cyclique. C'est pourquoi il faut, avant de modifier un pointeur, contrôler la valeur de l'autre.

Vous vous demandez certainement maintenant comment on peut faire une mémoire cyclique à partir d'une mémoire linéaire. On définit pour cela une zone déterminée de la mémoire comme mémoire cyclique ou mémoire FIFO. Lorsque le pointeur d'écriture ou de lecture atteint la plus haute adresse de cette zone mémoire, il est à nouveau fixé sur l'adresse de départ lors de sa prochaine élévation. Cela entraîne bien sûr une interrogation supplémentaire lors de la modification des pointeurs.

En outre, on arrive ainsi à une situation où la valeur du pointeur d'écriture est inférieure à celle du pointeur de lecture bien que, selon le principe de la mémoire cyclique ou mémoire FIFO, l'endroit où l'on écrit se trouve toujours "devant" l'endroit où l'on lit. Mais finalement, on aboutit bien ainsi, du point de vue fonctionnel, à une mémoire cyclique. Les nécessaires comparaisons permettant de détecter des erreurs dans les valeurs des pointeurs ne sont cependant pas toujours faciles à formuler.

SYSTEME D'EXPLOITATION ET PROGRAMMES

Un microprocesseur a toujours besoin d'un programme pour pouvoir faire quelque chose. En fait, l'état hors programmation n'existe absolument pas sur le 68000. Le 68000 essaie toujours de charger la prochaine instruction lorsqu'il n'a rien d'autre à faire et lorsqu'aucune autre machine ne réclame l'emploi du bus système. Le 68000 ne peut être amené que par une instruction dans un état dans lequel il restera inactif dans l'attente d'une interruption. Cet état est provoqué par l'instruction STOP et il est donc appelé état "stop". Si dans le travail du 68000 des erreurs si graves apparaissent que l'on doit considérer qu'aucun travail utile n'est plus possible (par exemple une double erreur de bus), l'état HALT est déclenché. Cet état ne peut plus être abandonné que par un RESET du processeur.

Comme un microprocesseur ne peut travailler sans programme, il faut toujours un programme pour qu'après la mise en marche de l'ordinateur, les instructions de l'utilisateur puissent être lues et que l'ordinateur sache ainsi ce qu'il doit faire, quel programme doit être exécuté. Sur de nombreux ordinateurs, c'est tout simplement le Basic intégré (donc un langage de programmation) qui est "démarré" après la mise sous tension.

Sur les systèmes un peu plus importants, notamment sur les systèmes avec lecteurs de disquette, on trouve également une structure développée à partir d'un logiciel système.

Un certain nombre de tâches doivent être exécutées dans n'importe quel programme. Presque chaque programme nécessite des entrées ou sorties de caractères avec le clavier, l'écran ou l'imprimante. D'autre part, tous les programmes doivent écrire et lire sur les disquettes de la même manière de façon à ce qu'une disquette puisse contenir des données appartenant à différents programmes.

Il est certainement peu utile de réécrire les routines nécessaires pour cela (sous-programmes, procédures ou fonctions) dans chaque programme. C'est pourquoi pratiquement tous les microordinateurs disposent de ce qu'on appelle un SYSTEME D'EXPLOITATION. Le système d'exploitation contient les programmes nécessaires pour pouvoir gérer la totalité de la périphérie. Les programmes se divisent le plus souvent en parties dépendant de la machine et en parties indépendantes de la machine. Si le système d'exploitation doit être transféré sur un autre ordinateur, seules les parties dépendant de la machine auront à être reprogrammées ou modifiées.

Les programmes utilisateur proprement dits (par exemple le traitement de texte ou le Basic) n'utilisent plus la périphérie qu'à travers le système d'exploitation. Comme le système d'exploitation peut être le même sur des ordinateurs différents, les programmes peuvent être utilisés sans modification sur différents ordinateurs, c'est-à-dire sur des ordinateurs différents du point de vue électronique.

Les systèmes d'exploitation importants contiennent souvent également parmi leurs fonctions la gestion de la mémoire de travail. Les programmes demandent chaque fois au système d'exploitation la place mémoire dont ils ont besoin.

L'utilisateur proprement dit n'entre pas en contact direct avec le système d'exploitation tant qu'il n'écrit pas lui-même des programmes au niveau du système d'exploitation. Une autre composante du système d'exploitation est constituée par la SURFACE UTILISATEUR. Font partie de la surface utilisateur les parties de programme permettant à l'utilisateur d'indiquer à l'ordinateur quel programme il doit exécuter, de lui indiquer qu'il doit supprimer des fichiers sur la disquette ou qu'il doit en imprimer, etc...

Sur l'Atari ST, c'est le système d'exploitation TOS avec la surface utilisateur GEM de Digital Research qui a été utilisé. TOS est sur de nombreux points identiques au CP/M68K de Digital Research. Il offre des routines pour l'utilisation de la périphérie avec des lecteurs de disquette. La structure de l'écran avec les fenêtres, l'utilisation de la souris sont tirées de GEM. Nous ne nous étendrons pas sur la structure de GEM et TOS car cela sortirait du cadre de cet ouvrage. Nous en expliquerons évidemment les fonctions chaque fois que cela sera nécessaire dans les chapitres suivants.

Nous avons jusqu'ici toujours considéré que l'ordinateur n'exécute jamais qu'un programme à la fois. Il existe cependant des systèmes d'exploitation qui permettent à l'ordinateur d'exécuter presque simultanément différents programmes. De tels systèmes d'exploitation sont appelés systèmes d'exploitation MULTI-TACHES. Presque simultanément signifie que l'ordinateur exécute un programme pendant quelques fractions de seconde pour passer ensuite chaque fois à l'exécution du programme suivant. Ce procédé est en général appelé "time sharing".

L'exploitation multi-tâches peut être exécutée de deux façons différentes:

1) Un programme travaille au premier plan. Ce programme peut travailler en interaction avec l'utilisateur. Tous les autres programmes ne travaillent qu'au second plan. Après qu'un programme de second plan ait été lancé, il travaille de façon autonome, sans plus recourir à l'écran ou au clavier. On peut par exemple faire se dérouler au second plan l'impression d'un fichier pendant qu'on continuerait au premier plan à écrire d'autres textes.

2) On ne distingue pas entre premier et second plans. L'écran peut être commuté par des instructions simples d'un travail sur un autre. On peut aussi ouvrir simplement une nouvelle fenêtre pour un autre travail. Cette technique est nettement plus pratique que la précédente mais elle nécessite des systèmes d'exploitation beaucoup plus compliqués. Moyennant quoi on peut réellement utiliser simultanément deux programmes. On pourrait par exemple écrire un rapport avec un programme de traitement de texte et résoudre les questions qui pourraient se poser grâce à un programme de banque de données fonctionnant de façon quasi-simultanée.

Le multi-tâches ne permet cependant pas uniquement d'exécuter simultanément des programmes différents au niveau de l'utilisateur. Les différents programmes peuvent aussi se composer à leur tour de programmes différents exécutés de façon quasi-simultanée. Un programme de traitement de texte pourrait se composer de deux parties: la première partie serait un programme de traitement de texte habituel alors que la seconde partie serait un programme qui effectuerait régulièrement des copies de sécurité du texte traité. Comme les deux programmes tourneraient simultanément, l'utilisateur ne se rendrait pas compte du déroulement de la réalisation des copies de sécurité.

Avec un microprocesseur, on peut également "servir" différents terminaux (écran et clavier). De même qu'un système d'exploitation multi-tâches (du second type) exécute quasi-simultanément plusieurs programmes sur un terminal, un système d'exploitation peut bien sûr également exécuter différents programmes sur différents terminaux. Il est ainsi possible que différents utilisateurs travaillent simultanément sur un même ordinateur. Les systèmes maîtrisant cette technique sont qualifiés de systèmes MULTI-USER ou multi-postes.

Les programmes

Un microprocesseur ne peut traiter que son langage machine. Il ne comprend pas les langages plus évolués (il y a cependant des processeurs qui contiennent un langage de programmation intégré, par exemple Forth ou Basic). Le langage machine est simplement constitué par le codage bit par bit des instructions et les données brutes d'un programme. Il n'est que très difficilement lisible par les hommes car il est difficile de faire des associations nettes entre les instructions langage machine et le code qui en résulte. C'est parce que le langage machine est aussi difficile à lire que les langages de programmation ont été créés. Avec des programmes appropriés, les programmes en langages évolués peuvent être traduits en langage machine ou être traités ligne par ligne en langage machine.

D'autres programmes auxiliaires sont utilisés pour l'écriture des textes de programme, pour tester ou corriger les logiciels. Nous souhaitons vous présenter quelques groupes de programmes de ce type pour expliquer certaines notions.

ASSEMBLEUR

Chaque instruction langage machine est affectée par le constructeur de l'ordinateur à ce qu'on appelle une mnémonique. L'assembleur traduit les mnémoniques en langage machine. Comme les sauts nécessitent par exemple l'indication d'une adresse mais qu'on ne sait pas encore lors de l'écriture du programme où se situeront finalement dans la mémoire les différentes parties du programme, l'assembleur calcule de lui-même les adresses qui conviennent. Le programmeur écrit pour cela des marques dans le programme, appelées "labels" ou étiquettes. On peut en outre donner des noms à des cases mémoire. Ces cases mémoire pourront par exemple contenir des variables. Voici en exemple un court extrait d'un programme assembleur:

```
addbcd:     MOVEA.L     #PTRSOURCE,A0
            MOVEA.L     #PTROBJET,A1
            MOVE.W      #LONGUEUR-1,D0
boucleadd:  ABCD        -(A0),-(A1)
            DBF         D0,boucleadd
```

En assembleur, les programmes sont finalement écrits en langage machine. La syntaxe de l'assembleur est cependant beaucoup plus facile à maîtriser que le langage machine lui-même car les mnémoniques constituent des "abréviations" et des aides-mémoire des instructions.

COMPILER

Un compiler est un programme qui traduit des textes de programme d'un langage de programmation plus élevé en langage machine. Le Pascal est par exemple traduit en langage machine par un compiler. Le compiler et l'assembleur sont très semblables dans le principe. Un compiler traduit cependant normalement une instruction d'un langage évolué avec plusieurs, voire de nombreuses instructions langage machine alors que l'assembleur traduit chaque instruction par exactement UNE instruction langage machine. Suivant le langage, le programme compilé effectue des contrôles plus ou moins nombreux durant le déroulement du programme (pendant le "RUN-TIME"). En Pascal par exemple, les limites des tableaux de variables sont surveillées de façon très pointilleuse. En "C" par contre, presque aucun contrôle n'est effectué. Cela rend certainement le programme plus rapide mais des erreurs graves peuvent aisément se produire.

INTERPRETEUR

Avec un interpréteur, un programme en langage évolué n'est pas traduit. Au lieu de cela, l'interpréteur exécute pour chaque instruction du langage évolué une séquence correspondante d'instructions en langage machine. Chaque instruction du langage évolué est donc lue, interprétée et exécutée. Cela entraîne cependant que l'instruction d'une boucle qui devra être parcourue 100 fois sera traitée 100 fois. L'avantage des interpréteurs est qu'après chaque modification du programme lors de la phase de tests, il n'est pas nécessaire de traduire chaque fois à nouveau la totalité du programme.

Il n'est souvent pas très facile dans les faits de délimiter la frontière entre compiler et interpréteur. Il y a par exemple des compilers Pascal (par exemple UCSD) qui ne traduisent pas en langage machine mais dans un langage intermédiaire qui ressemble au langage machine mais doit être exécuté par un interpréteur.

EDITEUR

Un éditeur est un programme avec lequel on peut écrire des textes de programme pour les langages de programmation. Dans de nombreux langages de programmation (par exemple Basic), l'éditeur fait déjà partie du programme de langage de programmation. Dans d'autres langages (par exemple "Pascal", "C"), on utilise cependant un éditeur séparé. Un programme de traitement de texte est en fait un éditeur qui n'a pas été conçu pour des textes de programme mais pour les lettres ou ouvrages ordinaires. Le présent ouvrage a lui aussi été écrit avec un programme de traitement de texte.

MONITEUR

Un moniteur est un programme qui permet d'examiner et de modifier directement les cases mémoire et les registres du processeur. Un moniteur contient d'autre part des fonctions pour retraduire en mnémoniques le langage machine, pour le DESASSEMBLER. Un programme de moniteur est le plus souvent utilisé pour contrôler, tester et corriger les programmes en langage machine.

DEBUGGER

Un moniteur est en fait également un débugger. Un débugger est un programme d'aide dans la recherche des erreurs dans les programmes. Il n'y a cependant pas que des débuggers qui travaillent, comme un moniteur, au niveau du langage machine mais il y en a également qui travaillent directement avec un langage évolué.

LINKER

Un linker est un programme qui fusionne en un seul programme plusieurs sections de programme qui ont été traduites individuellement en langage machine. Un linker permet également de relier des parties écrites en différents langages. C'est ainsi qu'on insèrera souvent, aux endroits devant absolument être exécutés très vite, des parties en assembleur dans des programmes en Pascal.

Outre les programmes que nous venons d'indiquer, on dispose souvent également de toute une série de ce qu'on appelle des TOOLs. C'est ainsi qu'il existe par exemple dans le système de développement de l'Atari ST un programme avec lequel on peut aisément concevoir les ICONS qui symbolisent les fichiers. D'autres Tools produisent des listes des variables utilisées dans un programme.

On peut ajouter à cela la liste presque illimitée des programmes utilisateur. Les plus importants sont certainement les programmes de traitement de texte, de gestion de fichiers (programmes de banque de données), de calcul en tableaux ou de dessin. Mais pour l'Atari ST notamment, on peut s'attendre à voir arriver sur le marché de nombreux autres programmes utilisateur, y compris certains nouveaux programmes.

LES BASES DE LA PROGRAMMATION EN ASSEMBLEUR

1) Introduction

2) L'éditeur

3) L'assembleur

4) Le débugger

5) Conventions de procédure

Dans ce chapitre, nous souhaitons vous faire découvrir la structure et le fonctionnement d'un assembleur 68000. Nous expliquerons des notions importantes qui appartiennent au "quotidien" de la programmation en assembleur et nous décrirons le mode d'emploi général ainsi que les règles de syntaxe spécifiques. Nous souhaitons ainsi vous permettre d'avoir un aperçu des possibilités d'un assembleur et de pouvoir résoudre des problèmes de programmation avec un assembleur.

Les auteurs disposent d'un système de développement Atari 520 ST. D'autres éléments d'expérience ont également été acquis sur un ordinateur UNIX et avec un Crossassembleur. Il n'existe malheureusement pas de normes contraignantes pour les assembleurs mais cependant des "standards" historiques ou des standards par firme se sont constitués. Nous nous fonderons dans ce qui suit sur ces standards sans toutefois prétendre à une quelconque universalité. Nous espérons vous faciliter le maniement de votre assembleur sans pouvoir vous dispenser de l'étude du manuel d'utilisation de votre assembleur.

Le système de développement Atari contient l'assembleur CP/M 68K (AS68) de Digital Research. Il n'est pas encore possible d'indiquer quels systèmes assembleur seront disponibles pour l'Atari ST ni quelle diffusion ils atteindront. Pour comprendre les exemples donnés dans cet ouvrage, vous n'avez pas besoin de disposer d'un assembleur. Si vous voulez toutefois résoudre vous-même des problèmes au moyen de la programmation en assembleur, vous ne pourrez faire l'économie d'un assembleur.

L'éditeur

Nous avons jusqu'ici toujours parlé d'un programme comme d'une représentation abstraite d'instructions qui traduit l'assembleur en instructions langage machine. Ces instructions ressemblent, pour prendre une analogie dans le mode de pensée humain, à un langage, le langage de programmation. Tous les langages de programmation ont ceci en commun que les programmes formulés dans l'un ou l'autre langage peuvent être représentés sous une forme écrite quelle qu'elle soit. Cette forme écrite est le texte de programme.

Le texte de programme doit être entré dans l'ordinateur, comme tout document écrit, avant de pouvoir être traité. On utilise à cet effet un programme spécial, l'éditeur.

Le texte de programme est stocké sous la forme d'un fichier dans la mémoire de masse (disquette ou disque dur). L'éditeur maîtrise toutes les possibilités de production et de modification d'un fichier de texte. L'aspect plus ou moins pratique de ces fonctions dépend bien sûr de l'éditeur utilisé.

Avec le système de développement Atari ST, nous disposons de l'éditeur MINCE. On peut au fond utiliser n'importe quel éditeur pour entrer un programme pourvu qu'il produise des fichiers de texte en code ASCII. C'est pourquoi il est possible d'utiliser un éditeur qui ne fasse pas partie du package assembleur du fabricant.

Cela peut être intéressant lorsque vous disposez d'un éditeur puissant dont les possibilités dépassent celles de l'éditeur de l'assembleur ou lorsque vous ne voulez pas avoir à vous habituer à deux modes d'utilisation différents.

La liste suivante doit vous donner une idée des fonctions d'un éditeur:

- production d'un fichier de texte sur la disquette ou le disque dur.
- modification d'un fichier de texte
- suppression d'un fichier de texte sur la disquette ou le disque
- réception des caractères du clavier
- affichage à l'écran du texte entré
- sortie d'un texte sur une imprimante
- exécution d'instructions d'édition
- déplacement du texte sur l'écran
- formatage du texte, par exemple d'après des tabulations

L'éditeur est en règle général lancé à partir du niveau du système d'exploitation (GEM ou TOS). Lors du démarrage de l'éditeur, il est possible d'indiquer le nom du fichier à traiter. Dans ce cas, l'éditeur ira immédiatement chercher le texte sur la disquette ou le disque dur et il l'affichera à l'écran. S'il s'agit d'un nouveau nom qui n'a pas encore été utilisé, un nouveau fichier de texte sera produit. Outre le texte en train d'être traité, l'éditeur affiche également quelques informations supplémentaires sur le texte.

Ces informations supplémentaires consistent en des indications sur l'état de travail de l'éditeur et le cas échéant en des explications sur les fonctions d'édition actuellement disponibles.

Quelques informations caractéristiques sont par exemple l'emplacement du curseur (page/ligne/colonne) ou le nom du fichier actuellement traité.

Une fois l'éditeur lancé, si un texte se trouve en traitement (même un nouveau texte), le programmeur se trouve alors dans le mode édition de l'éditeur. A ce niveau, il y a trois modes de travail principaux de l'éditeur:

- mode d'écriture
- mode de déplacement
- mode instruction

En mode d'écriture, toutes les lettres, chiffres et caractères spéciaux entrés directement sont insérés dans le texte. Le mode d'écriture est appelé automatiquement par le fait d'actionner une touche correspondante du clavier. Un caractère entré apparaît toujours dans l'emplacement actuel du curseur (emplacement d'écriture).

L'emplacement d'écriture peut être modifié à volonté dans le texte, au moyen de touches spéciales, les touches curseur. L'éditeur se trouve donc automatiquement en mode déplacement lorsqu'une touche curseur est actionnée.

Le mode instruction est en général atteint en appuyant simultanément sur la touche Control et sur une touche de lettre. Une ou plusieurs de ces combinaisons de touches ont pour effet de faire exécuter une instruction déterminée par l'éditeur. Il y a des instructions très simples ou très complexes qui rendent possible un traitement de texte commode.

Nous allons en conclusion vous donner simplement un aperçu des instructions d'édition les plus courantes:

Mode écriture: Chiffres, lettres, caractères spéciaux

Mode déplacement: Curseur gauche, droite, haut et bas, suppression d'un caractère (Delete, Back space), tabulateur

Mode instruction:

- Gestion de fichier: lecture, écriture, suppression de fichier
  sauvegarde (écrire, continuer)
  changement du nom d'un fichier, copie de fichier, affichage du contenu de la disquette, insertion d'un module de texte (sur disquette)

- Déplacement: Mot à gauche, à droite, début de la ligne, début de la ligne suivante, une ligne en arrière, une ligne en avant, une page en arrière, une page en avant, début du texte, fin du texte

- Suppression: Mot de gauche, mot de droite, du début de la ligne jusqu'au curseur, du curseur jusqu'à la fin de la ligne, suppression de ligne, suppression totale du texte

- Instructions de bloc:	Marquer le début du bloc
	marquer la fin du bloc
	supprimer le bloc marqué
	copier le bloc marqué
	décaler le bloc marqué
	sauvegarder le bloc marqué (sur disquette)
- Divers:	Recherche d'un terme dans le texte
	recherche d'un terme avec remplacement
	imprimer fichier de texte
	fixer ou supprimer tabulation
	fin du programme avec sauvegarde du texte
	interruption du programme
	appel d'un texte d'aide

L'assembleur

Dans le langage de tous les jours, le terme assembleur désigne le plus souvent un nombre plus ou moins important de programmes qui permettent à un programmeur de manier le langage machine d'un ordinateur. Un tel paquet de programmes remplit plusieurs fonctions qui sont regroupées en différents domaines de fonction. Des parties caractéristiques d'un assembleur sont par exemple un éditeur ou un débugger. Les frontières entre les différentes parties de programme sont floues et elles sont déterminées par les liens des différents programmes entre eux.

Nous étudierons par la suite ces différentes parties, telles que le débugger, de façon sélective. Nous allons cependant commencer par l'assembleur proprement dit. Lorsque nous parlerons désormais d'un assembleur nous penserons concrètement à un programme qui traduit des instructions langage machine écrites de façon symbolique en instructions pouvant être exécutées par le processeur.

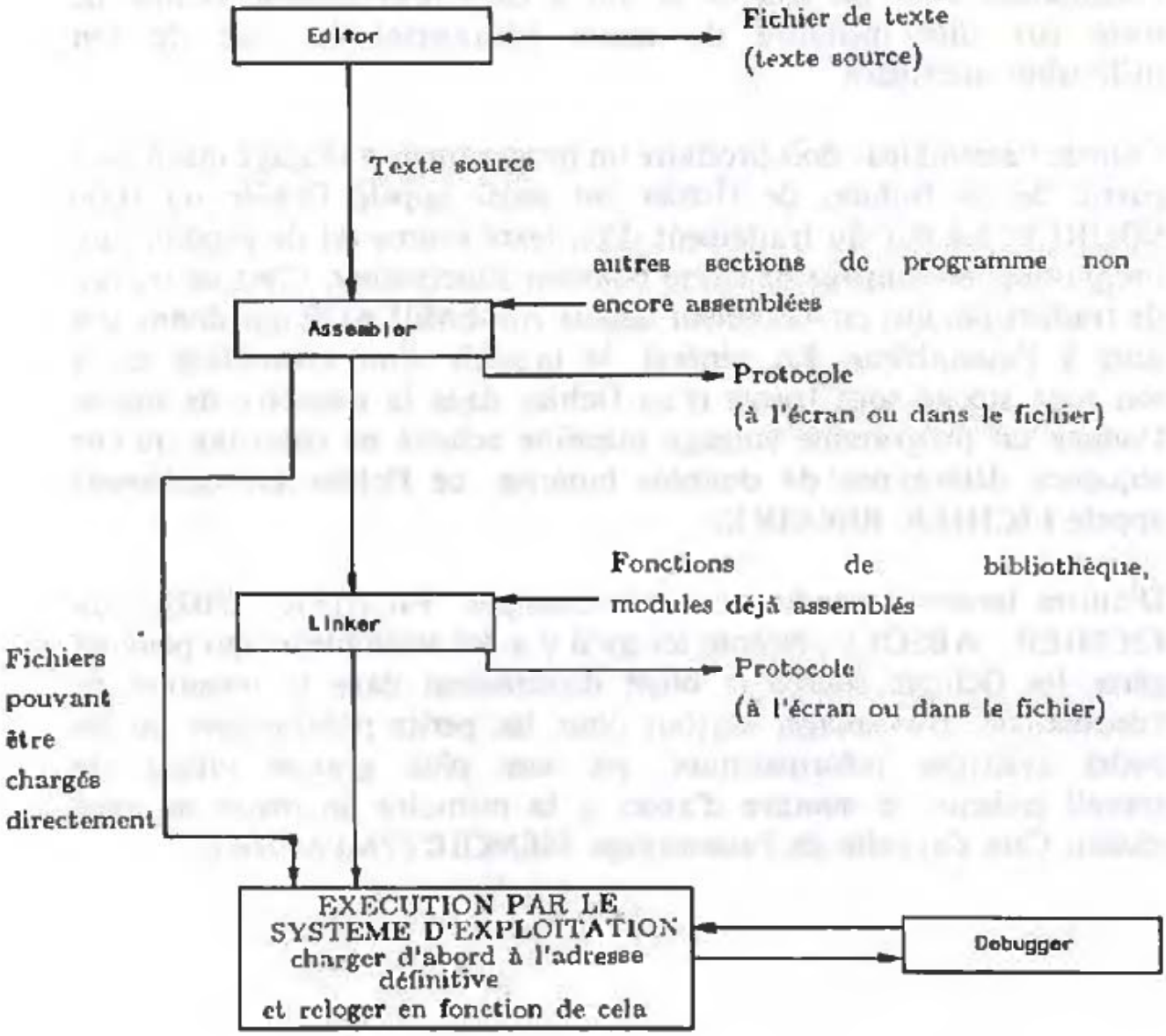
Fonction, structure et mode de travail de l'assembleur

Comme vous le savez déjà, un ASSEMBLEUR n'est lui aussi qu'un programme qui traite des données suivant des règles déterminées. Les données à traiter sont ici une liste d'instructions langage machine symboliques (mnémoniques).

Cette liste est un fichier qui a en général été entré dans l'ordinateur avec un éditeur et qui a été stocké comme fichier de texte sur une mémoire de masse (disquette) en vue de son utilisation ultérieure.

Comme l'assembleur doit produire un programme en langage machine à partir de ce fichier, ce fichier est aussi appelé fichier ou texte SOURCE. Le but du traitement d'un texte source est de produire un programme en langage machine pouvant fonctionner. C'est ce travail de traduction, qui est justement appelé ASSEMBLAGE.qui donne son nom à l'assembleur. En général, le produit d'un assemblage est à son tour stocké sous forme d'un fichier dans la mémoire de masse. Comme un programme langage machine achevé ne constitue qu'une séquence déterminée de données binaires, ce fichier est également appelé FICHIER BINAIRE.

D'autres termes répandus sont par exemple FICHIER OBJET ou FICHIER ABSOLU. Notons ici qu'il y a des assembleurs qui peuvent gérer les fichiers source et objet directement dans la mémoire de l'ordinateur. L'avantage, surtout pour les petits programmes ou les petits systèmes informatiques, est une plus grande vitesse de travail puisque le nombre d'accès à la mémoire de masse est plus réduit. Cela s'appelle de l'assemblage MEMOIRE/MEMOIRE.



SCHEMA DES ETAPES DE LA PROGRAMMATION EN ASSEMBLEUR

Pour les systèmes informatiques de la puissance de l'Atari ST, un bon assembleur permet également de réaliser des fichiers RELOGEABLES. Les fichiers relogeables constituent une forme particulière d'un résultat d'assemblage. Contrairement aux programmes langage machine pouvant fonctionner, qui sont en général liés à une zone d'adresses déterminée de la mémoire de l'ordinateur, les fichiers relogeables sont des programmes machine "semi-finis" auquels il manque simplement les indications sur la zone d'adresse utilisée. Ces indications d'adresse sont stockées séparément pour un fichier relogeable. Pour pouvoir produire un programme machine exécutable à partir d'un fichier relogeable, on a de nouveau besoin d'un programme utilitaire qui reliera la traduction "semi-finie" avec les indications d'adresse, produisant ainsi un fichier binaire exécutable ou chargeant même directement le programme machine en mémoire. Un tel programme utilitaire est appelé LOADER.

Le travail avec des fichiers relogeables apparaît très compliqué au départ mais il apporte des avantages considérablespour peu qu'on y regarde d'un peu plus près et qu'on l'applique de façon judicieuse. Comme le loader n'a qu'à rajouter des adresses absolues dans le code semi-fini, cette étape de travail est beaucoup plus vite exécutée qu'un assemblage complet. Le loader n'a pas à refaire le travail de traduction et de recherche des erreurs qui a déjà été fait par l'assembleur. Différents programmes machine sont produits de préférence sous une forme relogeable lorsqu'un programme doit être chargé dans différentes zones d'adresse. C'est souvent le cas pendant la phase de test d'un programme. Une autre possibilité est cependant pour le programmeur de concevoir son programme directement comme un CODE RELOGEABLE, en renonçant à l'adressage absolu.

Les fichiers relogeables présentent cependant d'autres aspects beaucoup plus importants. Avec un autre programme utilitaire, un LINKER, il est possible de fusionner en un seul programme des fichiers relogeables assemblés indépendamment les uns des autres.

Comme programmeur, vous pourrez utiliser la combinaison des fichiers relogeables et du linker pour de nombreuses applications. Si vous réalisez des programmes d'envergure, vous structurerez certainement le programme global en plusieurs petites solutions partielles, logiquement délimitées. Ces petits programmes devront être développés et testés séparément. Une fois que quelques-unes de ces solutions partielles seront terminées, vous réunirez ces MODULES pour former des solutions partielles un peu plus vastes. Vous testerez à leur tour ces solutions partielles, jusqu'à ce que vous ayez finalement achevé la totalité du programme.

En travaillant de cette façon, vous vous apercevrez que des solutions partielles qui fonctionnaient jusqu'ici se comportent soudain de façon incorrecte depuis que vous avez inséré un nouveau module. Si vous travailliez dans des cas semblables avec un seul texte source, auquel vous ajouteriez chaque fois le nouveau module, vous devriez assembler à nouveau la totalité du texte source pour chaque test, y compris donc les sections de programme terminées qui n'ont jusqu'ici révélé aucune erreur. Cela entraînerait un bien inutile gaspillage de temps de travail. Il est beaucoup plus agréable de travailler, pour des programmes de grande dimension, avec des petits textes source et des fichiers relogeables. Lors de la phase de test vous n'aurez alors à corriger et à assembler à nouveau que les modules suspects. Pour effectuer un test, vous pourrez produire rapidement un programme exécutable grâce au linker.

La modularisation a encore quelques effets secondaires intéressants. Vous pouvez par exemple, pour des programmes de dimension importante, sauvegarder les textes source des modules achevés sur une autre disquette en ne gardant sur votre disquette de travail que les fichiers relogeables correspondants. Vous économisez ainsi de la place mémoire sur la disquette, place mémoire que vous pouvez utiliser pour vos textes source actuels. Il est en effet à noter que le rapport entre les volumes occupés en mémoire respectivement par le texte source et le programme machine correspondant est en général environ de 10 à 1. Si vous veillez lors de la programmation des différents modules à respecter une répartition méthodique des tâches et des conventions de procédure bien définies - nous y reviendrons -, vous pourrez vous constituer avec le temps toute une BIBLIOTHEQUE de FONCTIONS que vous pourrez réemployer dans vos nouveaux programmes. Notons ici qu'il y a sur le marché des bibliothèques toutes prêtes. On propose par exemple des bibliothèques très complètes pour la gestion de fichier ou pour la solution de certains problèmes arithmétiques en mathématiques. L'utilisation de bibliothèques peut permettre à un programmeur de travailler de façon beaucoup plus efficace. Pour un utilisateur privé, ces bibliothèques sont cependant souvent hors de prix.

Une des applications importantes des fichiers relogeables consiste à résoudre seulement quelques problèmes spécifiques en langage machine alors que l'essentiel du programme principal est programmé dans un langage compilé. Le problème est de relier les sections de programme que vous avez résolues en langage machine avec les sections de programme qu'un compiler a lui-même traduites d'un langage évolué en langage machine. En général vous pouvez demander à un compiler de produire lui-même des fichiers relogeables. Le linker peut alors fusionner ces fichiers avec vos propres programmes machine.

Vous pouvez ainsi produire des programmes dont les modules ont été produits par différents compilers de langages évolués.

Venons-en brièvement à une autre fonction possible de l'assembleur: pendant la phase de développement et de test, vous utiliserez souventun DEBUGGER ouun PROGRAMME DE MONITEUR. Lors de la recherche des erreurs, vous pourrez ainsi examiner un programme machine directement en mémoire et le modifier aisément pour effectuer des tests.

Le débugger ou le moniteur vous offre à cet effet plusieurs possibilités. Comme le débugger ou le moniteur travaillent directement avec votre programme machine tout assemblé et qu'il est ainsi indépendant du texte source originel, il ne peut par exemple vous donner aucune information sur les noms de variable que vous avez utilisés pour différentes adresses mémoire. C'est pourquoi un programme machine devient très vite illisible pour le programmeur. Le programmeur ne peut donc compter que sur sa mémoire ou sur un protocole écrit (listing) du programme. Les packages assembleur puissants vous offrent cependant en cela une aide supplémentaire.

L'assembleur produit en effet ce qu'on appelle un FICHIER DES SYMBOLES ou FICHIER DES LABELS. Ce fichier contient tous les noms symboliques que le programmeur a définis dans son programme. L'assembleur place en outre dans ce fichier les informations qui correspondent aux valeurs absolues de ces symboles. Si vous utilisez un fichier relogeable, le fichier de symboles est bien sûr complété par le linker. Ainsi, toutes les informations essentielles sont à nouveau disponibles, en plus du programme machine prêt à fonctionner. Le débugger ou moniteur peut fournir à tout moment au programmeur, en exploitant le fichier symboles, des informations complètes sur son programme machine. Les systèmes les plus puissants fournissent même tout le texte source, avec les commentaires du programmeur.

Indiquons encore en conclusion que l'assembleur se charge également de produire un protocole, le LISTING. Le listing permet au programmeur de consulter son texte source en entier et de voir également quelle traduction en a été faite par l'assembleur. Dans la phase de test, il est très utile que l'assembleur produise en outre une TABLE DES SYMBOLES. Cette table des symboles contient tous les noms (symboles) définis par le programmeur avec les valeurs qui leur ont été affectées. Si en même temps sont indiqués tous les endroits du texte source qui contiennent ces noms, on parle alors d'une LISTE DE REFERENCES CROISEES.

Petite classification des assembleurs

Nous allons essayer maintenant d'expliquer quelques différences importantes qui peuvent exister entre les différents paquets de programmes d'assembleur. Lorsque vous aurez en effet à choisir un programme d'assembleur, il importe que vous sachiez à quoi correspondent les différentes caractéristiques qui permettent de définir un programme d'assembleur.

La forme la plus simple d'assembleur est l'assembleur DIRECT ou LINE BY LINE. Cette forme d'assembleur ne connaît pas de texte source ni de possibilités pratiques d'édition. Le programmeur est en dialogue direct avec l'assembleur. Il entre chaque instruction individuellement, directement dans l'assembleur. Celui-ci traduit immédiatement la ligne d'instruction complète en code binaire correspondant et place le résultat directement dans la mémoire. C'est pourquoi ce type d'assembleur n'offre pas non plus de traitement des symboles. Un assembleur line by line constitue rarement le coeur d'un package assembleur. Il est par contre un outil très utile dans un débugger ou un moniteur car il permet d'effectuer rapidement une petite modification sur un programme machine à tester, sans devoir passer pour cela par l'assembleur

proprement dit.
Mais il est pratiquement impossible de travailler uniquement avec un assembleur line by line pour des applications importantes.

Pour programmer des programmes de vastes dimensions, il est recommandé d'utiliser un ASSEMBLEUR ENTIEREMENT SYMBOLIQUE.
Ce type d'assembleur permet d'utiliser des noms symboliques pour les constantes, variables ou adresses du texte source. Nous traiterons par la suite des nombreuses possibilités offertes par l'utilisation des symboles. Indiquons simplement ici qu'un assembleur entièrement symbolique exécute deux phases de travail pour produire un programme machine. Dans la première phase de travail, l'assembleur examine le source pour y rechercher tous les symboles et calculer les valeurs effectives de ces symboles. Dans la seconde phase de travail, le programme machine est produit avec les valeurs qui sont maintenant entièrement connues. Cette particularité vaut à ce type d'assembleur l'appellation d'assembleur deux passages. Certains assembleurs comportent encore d'autres phases de travail lors de l'assemblage.

Un niveau supplémentaire de possibilités est offert par un MACROASSEMBLEUR. Un MACRO est un ensemble d'instructions machine qui peut être défini par le programmeur au moyen d'un SYMBOLE MACRO. On définit les macros essentiellement pour des séquences d'instructions qui se reproduisent souvent dans le source. En chaque endroit du texte source où le programmeur écrit un appel de MACRO, l'assembleur se comporte comme si le programmeur avait écrit à nouveau toute la séquence d'instructions. Les macros permettent de définir toute une bibliothèque de séquences d'instructions caractéristiques. Comme il ne s'agit pas là d'une technique de programmation, le traitement des macros est volontiers qualifié de "pure substitution textuelle".

Pour être complet et pour éviter toute confusion dans les termes, nous tenons encore à expliquer le terme d'ASSEMBLEUR CROISE. Un assembleur ne doit pas être nécessairement réalisé sur le système informatique pour lequel l'assembleur doit produire un programme machine. Le programme machine réalisé avec l'assembleur sur ces ordinateurs ne peut souvent même pas tourner sur ces ordinateurs. Pour pouvoir tester le programme machine dans de telles conditions, on utilise des programmes de SIMULATION. Les assembleurs croisés sont surtout utilisés là où le système objet, soit n'existe pas encore, (développement d'un ordinateur), soit n'est pas assez puissant pour permettre le développement du programme (commandes de microprocesseurs).

La structure des textes source

Comme nous l'avons déjà vu, un texte source constitue la forme symbolique d'un programme machine. Le texte source est orienté lignes, chaque ligne ne contenant en général qu'une seule instruction à traiter pour l'assembleur. La séquence logique de traitement à l'intérieur d'une ligne va de gauche à droite, et à l'intérieur du texte source, de la première (ligne supérieure) à la dernière ligne (ligne inférieure). Cette forme de représentation suit ainsi pleinement notre façon de penser lors de la programmation.

A l'intérieur d'un programme d'assembleur, on distingue différents groupes d'instructions qui remplissent des tâches déterminées et qui sont soumis à des règles de syntaxe différentes. Nous allons maintenant vous présenter ces groupes d'instructions avant de les expliquer chacun de façon détaillée.

Nous avons déjà indiqué que dans la programmation en assembleur, les instructions machine sont représentées par des mnémoniques.

Avec les opérandes et la représentation du mode d'adressage utilisé, les mnémoniques constituent une ligne de CODE D'OPERATION. L'assembleur produit ensuite une instruction machine à partir de chaque ligne de code d'opération, lors de l'assemblage. Une ligne de code d'opération peut être précédée par un LABEL. Un label est la désignation symbolique, choisie par le programmeur, de l'adresse à laquelle une instruction machine sera produite par l'assembleur. Le programmeur peut se référer à cette adresse dans son programme, simplement en la nommant, sans connaître la valeur de cette adresse. Mais nous reviendrons plus loin sur tous les aspects du traitement des symboles.

Une ligne de code d'opération peut également être complétée par un COMMENTAIRE. Cela augmente considérablement la lisibilité d'un programme et la compréhension des algorithmes utilisés, surtout pour les programmes complexes. Nous vous conseillons donc de commenter abondamment vos textes source car c'est un élément indispensable d'un travail "propre" de programmation. Un assembleur permet bien sûr également de ne doter une ligne de code d'opération que d'un label et/ou d'un commentaire. Cela peut augmenter la lisibilité du texte source mais cela n'a pas d'influence sur le fonctionnement du programme. De telles lignes peuvent être classées dans le groupe des lignes de code d'opération, même si ces lignes ne produisent pas d'instructions machine. Il en va de même pour les lignes vides.

Un autre groupe d'instructions assembleur comprend tous les types de définition directe des labels et symboles. Ces lignes sont appelées DECLARATIONS. D'autre appellations adéquates sont AFFECTATIONS DE VALEUR ou DEFINITIONS D'ADRESSE. En tout cas, une déclaration est l'affectation d'une valeur numérique à un symbole ou label que le programmeur peut définir librement dans le cadre de certaines contraintes de syntaxe.

Les déclarations sont utilisées par le programmeur pour qu'un programme reste le plus indépendant possible d'adresses ou de valeurs pouvant se modifier. Le programmeur appellera par exemple "colonne" un emplacement d'impression et affectera "10" à ce symbole. Il pourra toujours se référer dans le programme au symbole "colonne" lorsqu'il voudra avoir l'emplacement d'impression "10". Cela entraîne deux effets intéressants: d'une part le caractère auto-documenté du programme machine en est réhaussé, d'autre part, le programme devient plus aisé à modifier. Si le programmeur veut modifier plus tard la position d'impression, il lui suffira de modifier la constante dans la déclaration du symbole "colonne". L'assembleur utilisera ensuite automatiquement, lors de l'assemblage, la nouvelle valeur de position d'impression partout où le symbole "colonne" apparaît.

Nous vous avons déjà présenté une forme particulière de symboles, les labels. Au contraire des symboles qui désignent des valeurs variables et constantes, les labels sont utilisés exclusivement pour désigner des adresses. Toutes les adresses utilisées par le programmeur (but d'un saut) ne peuvent être définies par un label dans une ligne de code d'opération car certains de ces buts de saut ne figurent pas à l'intérieur du programme assembleur. On appelle ces adresses, qui doivent être définies au moyen d'une déclaration de symbole, LABELS EXTERNES.

Une spécialité des assembleurs puissants est la DECLARATION TEXTUELLE. Cela permet d'affecter des textes à certains noms de symbole réservés. Une telle déclaration textuelle peut par exemple être "D0 = COMPTEUR". "D0" est un SYMBOLE RESERVE qui désigne un registre de données dans la syntaxe assembleur du processeur 68000. Si le programmeur effectue cette déclaration dans son programme, l'assembleur saura, à partir de cette déclaration, que le programmeur veut toujours appeler le registre de données D0 lorsqu'il emploie le symbole "COMPTEUR". Cette fonction n'a pas non plus d'effet sur le mode de fonctionnement du programme machine.

Elle augmente simplement la lisibilité du texte source. Les bons programmeurs essaieront toujours d'organiser un programme machine de façon entièrement symbolique. Ce style de programmation est d'une valeur inappréciable pour des programmes complexes, lorsqu'il s'agit d'éliminer les erreurs ou d'adapter un programme machine. Nous ne saurions trop vous conseiller d'acquérir cette méthode "propre" de travail.

Le dernier et le plus vaste groupe d'instructions assembleur est constitué par les DIRECTIVES ASSEMBLEUR souvent également appelées PSEUDO-INSTRUCTIONS ou PSEUDO-CODES D'OPERATION. Les directives assembleur ne produisent pas en général d'instructions machine. Elles servent plutôt à commander le déroulement de l'assemblage, à choisir entre certaines options et à organiser les instructions machine dans la mémoire. On peut subdiviser les directives en différents groupes de fonction. La directive la plus importante dans la programmation assembleur est certainement celle qui indique à l'assembleur dans quelle zone mémoire doit s'effectuer l'assemblage. On peut également regrouper dans le même groupe toutes les directives qui indiquent à l'assembleur où il doit réserver dans la mémoire des zones d'adresses pour les zones de données ou bien où les tableaux doivent être définis. D'autres directives concernent l'organisation du listing ou indiquent à l'assembleur avec quels fichiers source il doit travailler. Nous allons maintenant décrire de façon détaillée les différentes directives.

Constantes et expressions arithmétiques

Dans la programmation assembleur, on ne distingue que des CONSTANTES numériques ou alphanumériques. Comme vous le savez déjà, un microprocesseur ne connait que des données binaires. Nos catégories ne désignent donc que différentes formes de représentation des données. Les constantes NUMERIQUES peuvent être selon l'opportunité représentées en différents systèmes numériques. Les bons assembleurs soutiennent différents systèmes numériques. Pour que l'assembleur sache, lors de l'interprétation d'une constante, quel système numérique le programmeur a choisi pour une constante donnée, quelques règles de syntaxe définissent l'indication du système numérique.

Le système numérique décimal qui nous est coutumier, représenté avec les chiffres 0 à 9, ne reçoit pas normalement de marque spéciale. Dans certains cas exceptionnels, le nombre décimal peut être précédé par un caractère "#".

Exemples: 100
#100

Certains assembleurs reconnaissent les constantes HEXADECIMALES automatiquement. Pour les distinguer des nombres décimaux et des symboles, il est simplement demandé que ces nombres hexadécimaux commencent par un 0. Cela serait de toute façon souvent le cas puisque un programmeur a tendance à écrire automatiquement les nombres hexadécimaux par groupes de chiffres correspondant à un octet ou à un mot. Une autre possibilité est le marquage par un caractère "$" (comme nous l'avons fait) ou par la lettre "H" qui ne constitue pas un chiffre du système hexadécimal. Cette marque peut être placée avant ou après le nombre.

Exemples: 0D
SF000
1000H

Certains assembleurs permettent aussi la représentation un peu dépassée des nombres OCTAUX. Les nombres octaux sont en général marqués par un "@" (arobas) placé avant ou après le nombre.

Exemples: 17
@10

Une possibilité très importante par contre est la représentation des nombres BINAIRES. Les nombres binaires sont marqués par un caractère "%" placé avant ou après le nombre.

Exemples: %1000000
%1010

Suivant le système représenté et le nombre de chiffres indiqués, l'assembleur traitera toujours les nombres en partant de leur droite. Le cas échéant il étendra de lui-même ou coupera les nombres pour arriver à un nombre de bits correspondant à un octet ou à un mot. Si cela entraîne la disparition de certains bits, un message d'avertissement est en général produit. Pour autant que l'assembleur puisse traiter des nombres signés, il tiendra compte de la formation du complément lors des extensions.

Certains assembleurs permettent l'emploi d'une base numérique variable. Le nombre est à cet effet suivi d'un "X" et de la base numérique choisie. La base numérique est indiquée en décimal.

Exemples: 1010x2 (nombre binaire) correspond à 10 déc.
1000x8 (nombre octal) correspond à 512 déc.
1250x10 (nombre décimal) correspond à 1250 déc.
2000x16 (nombre hexadécimal) correspond à 8192 déc.

Une séquence de caractères en CODE ASCII est appelée CONSTANTE ALPHANUMERIQUE. Il est d'ailleurs indifférent qu'il s'agisse de caractères imprimables ou de caractères de commande. Comme les caractères de commande ne peuvent en général être traités par un éditeur de texte, le programmeur doit les représenter sous la forme de constantes numériques. La représentation des caractères imprimables est par contre soutenue par l'assembleur. Pour que l'assembleur puisse distinguer les CHAINES DE CARACTERES ou STRINGS du texte source normal, les caractères formant une chaîne de caractères sont placés entre des CARACTERES DE SEPARATION ou DELIMITEURS. Un premier caractère de séparation marque le début de la chaîne de caractères alors qu'un second en marque la fin. Les guillemets ("), le slash (/) ou l'apostrophe (') constituent des caractères de séparation souvent utilisés.

Exemples: "Bonjour ATARI!"
/Bonjour lecteur!/

Des règles spéciales concernent la représentation des délimiteurs à l'intérieur d'une chaîne de caractères. Si le délimiteur est représenté par deux fois consécutives, sans autre caractère entre les deux, l'assembleur ne verra pas là la fin d'un chaîne de caractères mais il insèrera le délimiteur "une fois" dans la chaîne de caractères comme caractère à représenter.

Exemples: "Bonjour ""lecteur""" correspond à
'Bonjour "lecteur"'

/10//5=2/ correspond à "10/5=2"

Un ou plusieurs constantes, symboles ou FONCTIONS reliés par des SIGNES D'OPERATION constituent une EXPRESSION ARITHMETIQUE.
Dans une EXPRESSION MIXTE, il peut y avoir des constantes et/ou des symboles de types différents.

IMPRIMÉ EN FRANCE

Cet ouvrage constitue une introduction aisément compréhensible au langage machine du processeur 68 000. Il vous permettra enfin d'utiliser pleinement les incroyables possibilités de ce processeur 16/32 bits.

Contient notamment :

- Opérations logiques et manipulations de bits
- Processus de développement d'un programme
- Structure d'un micro ordinateur
- Le 68 000 sur l'Atari ST
- Structure des registres
- Modes de travail
- Structures de programme et de mémoire
- Procédures et fonctions
- Système d'exploitation et programmes
- Bases de la programmation en assembleur
- Editeur/Assembleur et débugger
- Programmation étape par étape
- Astuces pour intégrer des programmes assembleur dans des langages évolués
- Solution de problèmes caractéristiques

Réf. : ML 141
Prix : 149 FF

ISBN : 2-86899-030-4

Les noms réservés sont toutes les MNEMONIQUES, DIRECTIVES, NOMS DE FONCTION, et CONSTANTES SYSTEME. Les constantes système sont des noms de symbole fixés une fois pour toutes et dont l'assembleur gère automatiquement la valeur. Nous reviendrons plus tard plus en détail sur les constantes système. La longueur d'un symbole est le plus souvent limitée par un NOMBRE MAXIMAL DE CARACTERES. Pour des raisons historiques, le nombre de 6 caractères s'est maintenu jusqu'ici. Des assembleurs modernes autorisent cependant des noms d'une longueur illimitée. Mais seul un nombre déterminé de caractères seront évalués pour distinguer les symboles entre eux. Ces caractères sont appelés CARACTERES SIGNIFICATIFS. Du fait de ces limites, le programmeur est contraint de trouver pour ses symboles des avréviations courtes et parlantes. Pour augmenter la compréhension de ces symboles, certains assembleurs autorisent l'emploi de caractères spéciaux à l'intérieur des symboles. Le point (.), le souligné (_), le backslash (\) et le double point constituent des exemples caractéristiques de caractères spéciaux.

Exemples:
```
CHROUT
DATA_IN
BOUCLE1:
SPC.20
```

Les CONSTANTES SYSTEME constituent un groupe spécial de symboles. Les noms de symbole et leur nombre varient d'un assembleur à l'autre. Nous indiquons ici simplement quelques constantes système particulièrement caractéristiques.

Exemples:

CR	caractère de commande ($0D)
TRUE, HIGH	vrai ($FFFF)
FALSE, LOW	faux ($0000)
*	adresse actuelle

La dernière constante système donnée en exemple n'est pas vraiment une constante. La valeur de ce symbole est en effet actualisée au fur et à mesure de l'assemblage, au début de chaque ligne de programme. Sa valeur reste cependant constante à l'intérieur de chaque ligne. Ce symbole recouvre en permanence l'adresse à laquelle l'assembleur devrait placer la prochaine instruction machine traduite. Nous reviendrons sur la signification particulière de ce symbole lorsque nous traiterons du CALCUL D'ADRESSE.

Mnémoniques et extensions de mnémonique

Nous allons nous intéresser maintenant aux règles de syntaxe pour les MNEMONIQUES ou CODES D'OPERATION qui déterminent les instructions machine proprement dites.

Ces mnémoniques ne sont malheureusement pas normalisées. Il existe cependant des recommandations du constructeur auxquelles les développeurs des assembleurs se sont en général conformés. Nous indiquerons expressément les différences existantes.

Comme vous pouvez le voir facilement dans les pages suivantes, une mnémonique sélectionne toujours un groupe d'instructions déterminé à l'intérieur duquel l'assembleur retrouvera l'instruction machine proprement dite. Un groupe d'instructions ne regroupe que des instructions qui exécutent au fond la même fonction.

Dans la liste des mnémoniques, nous avons volontairement renoncé à une traduction en français. Nous pensons en effet que vous apprendrez plus aisément les mnémoniques, qui sont dérivées de l'anglais, si vous en connaissez la signification anglaise.

Nous expliquerons dans cet ouvrage toutes les instructions utilisées. La table suivante doit vous permettre d'avoir un aperçu et de comprendre les correspondances fondamentales. C'est pourquoi il n'est pas nécessaire que vous appreniez toutes les mnémoniques par coeur.

```
ABCD.B   OP1,OP2    Add binary coded decimals, extend
ADD.X    OP1,OP2    Add binary
ADDA.X   OP1,OP2    Add binary to addressregister
ADDI.X   OP1,OP2    Add Immediate
ADDQ.X   OP1,OP2    Add immediate quick
ADDX.X   OP1,OP2    Add binary with extended
AND.X    OP1,OP2    Logical AND
ANDI.X   OP1,OP2    Logical AND with Immediate value
ASL.X    OP1(,OP2)  Arithmetic shift left
ASR.X    OP1(,OP2)  Arithmetic shift right
Bcc.X    OP1        Branch if conditioncode true
BCHG.X   OP1,OP2    Test bit and change
BCLR.X   OP1,OP2    bit test and clear
BRA.X    OP1        branch always
BSET.X   OP1,OP2    Bit test and set
BSR.X    OP1        Branch to subroutine
BTST.X   OP1,OP2    Bit test
CHK.W    OP1,OP2    Check register against bounds
CLR.X    OP1        Clear
CMP.X    OP1,OP2    Compare
CMPA.X   OP1,OP2    Compare adressregister
CMPI.X   OP1,OP2    Compare immediate
CMPM.X   OP1,OP2    Compare in memory
DBcc.W   OP1,OP2    Decrement and Branch, conditionaly
DIVS.W   OP1,OP2    Divide signed
DIVU.W   OP1,OP2    Divide unsigned
EOR.X    OP1,OP2    Logical exclusive OR
EORI.X   OP1,DP2    Log. Excl. OR with immediate value
EXG.L    OP1,OP2    Exchange register
EXT.X    OP1        sign extend
JMP      OP1        Jump absolute
JSR      OP1        Jump to subroutine absolute
LEA.L    OP1,OP2    Load effectiv adr. to adr.-register
LINK     OP1,OP2    Link local basepointer
LSL.X    OP1,OP2    Logical shift left
LSR.X    OP1,OP2    Logical shift right
```

```
MOVE.X   OP1,OP2    Move sourcedata to destination
MOVEA.X  OP1,OP2    Move to adressregister
MOVEM.X  OP1,OP2    Move multiple register
MOVEP.X  OP1,OP2    Move from or to peripheral registers
MOVEQ.L  OP1,OP2    Move immediate quick
MULS.W   OP1,OP2    Multiply with sign
MULU.W   OP1,OP2    Multiply without sign
NBCD.B   OP1,OP2    Negate binary coded decimals
NEG.X    OP1        Negate
NEGX.X   OP1        Negate whith extend
NOP                 No operation
NOT.X    OP1        Logical NOT
OR.X     OP1,OP2    Logical OR
ORI.X    OP1,OP2    Logical OR with immediate value
PEA.L    OP1        Push effektive address
RESET               Reset external devices
ROL.X    OP1(,OP2)  Rotate left
ROR.X    OP1(,OP2)  Rotate right
ROXL.X   OP1(,OP2)  Rotate left with extended bit
ROXR.X   OP1(,OP2)  Rotate right with extended bit
RTE                 Return from exception
RTR                 Return and restore register
RTS                 Return from subroutine
SBCD.B   OP1,OP2    Subtract binary cod. dec. w. extend
Scc.B    OP1        Set byte according to conditioncode
STOP     OP1        Stop with conditioncode loaded
SUB.X    OP1,OP2    Subtract binary
SUBA.X   OP1,OP2    Subtract binary from addressreg.
SUBI.X   OP1,OP2    Subtract immediate
SUBQ.X   OP1,OP2    Subtract immediate qulck
SUBX.X   OP1,OP2    Subtract binary with extend
SWAP.W   OP1        Swap register halves
TAS.B    OP1        Test byte and set always bit 7
TRAP     OP1        software trap always
TRAPV    OP1        Trap on overflow
TST.X    OP1        Test byte
UNLK     OP1        Unlink local area
```

Vous avez certainement remarqué dans notre liste les abréviations ".X", ".B", ".W" et ".L". Ces EXTENSIONS DE MNEMONIQUE déterminent avec quelle largeur de traitement l'instruction machine doit être exécutée. Ces abréviations ont la signification suivante:

.B largeur de traitement BYTE, 8 BITS, 1 OCTET, 1/2 MOT
.W largeur de traitement WORD, 16 BITS, 2 OCTETS, 1 MOT
.L largeur de traitement LONG, 32 BITS, 4 OCTETS, 2 MOTS

Lorsque vous utilisez une instruction qui est marquée dans la table des mnémoniques par .B, .W ou .L, cela signifie que cette instruction ne peut être utilisée qu'avec cette largeur de traitement. Certains assembleurs permettent au programmeur de négliger pour ces mnémoniques l'indication de l'extension. Les instructions qui sont marquées par un .X peuvent travailler avec ces trois largeurs de traitement. Si vous négligez pour ces instructions l'indication de l'extension, l'assembleur supposera que le programmeur souhaite un traitement dans le format "MOT". Les instructions qui n'ont pas d'extension dans la table des mnémoniques contiennent la largeur de traitement de façon implicite.

Vous trouverez pour certaines instructions de la table des mnémoniques une forme particulière d'extension de mnémonique, représentée par deux petits c (cc). Ces instructions constituent des opérations qui testent une condition déterminée et qui exécutent une opération suivant le résultat du test. La condition posée par le programmeur est fixée par un CODE DE CONDITION. Le programmeur doit donc compléter la mnémonique avec de code de condition (cc).

A la suite des mnémoniques vous trouvez l'indication des OPERANDES de l'instruction. Certaines instructions n'ont pas d'indication d'opérandes. Elles contiennent le ou les opérandes de façon implicite. Un opérande placé entre parenthèses est facultatif.

Nous reviendrons plus loin plus en détail sur la formation des opérandes.

L'emploi des codes de condition

Dans la description suivante des codes de condition, nous n'expliquerons pas en détail toutes les possibilités des flags du registre de flags du processeur 68000. Nous voulons plutôt vous montrer la signification de "l'interrogation". En cas de besoin, nous vous invitons donc à vous reporter à un ouvrage consacré au processeur pour savoir comment les flags sont influencés par des opérations et à quelles conditions correspondent certaines positions de flags.

Les instructions 68000 qui interrogent les flags se réfèrent toujours à un ou plusieurs des flags suivants:

C Carry	Retenue
N Negative	Résultat négatif (complément à deux)
V Overflow	Dépassement produit par la dernière opération
Z Zero	Résultat nul (pour tous les bits)

A chacun de ces flags correspondent deux codes de condition suivant que le programmeur attend un flag mis (1) ou annulé (0). Nous allons vous présenter ces huit codes de condition dans l'exemple d'une instruction de saut conditionnelle. La forme générale de l'instruction est Bcc OP1, OP1 désignant le but du saut au cas où la condition serait remplie (voir modes d'adressage).

CC BCC OP1 Branch if Carry Clear Sauter si C=0
CS BCS OP1 Branch if Carry Set Sauter si C=1
PL BPL OP1 Branch if PLus Sauter si N=0
MI BMI OP1 Branch if MInus Sauter si N=1
VC BVC OP1 Branch if oVerflow Clear Sauter si V=0
VS BVS OP1 Branch if oVerflow Set Sauter si V=1
NE BNE OP1 Branch if Not Equal Sauter si Z=0
EQ BEQ OP1 Branch if EQual Sauter si Z=1

Les codes de condition indiqués précédemment sont en général utilisés lorsque le programmeur veut tester l'état des flags de façon sélective. L'état des flags a cependant une signification particulière lorsqu'on veut en déduire le résultat d'une opération de comparaison. Notez pour les explications suivantes qu'une opération de comparaison doit toujours précéder l'instruction de saut conditionnelle, par exemple CMP OP1, OP2. Pour les deux résultats égale et différent d'un tel test, on utilise les codes de condition déjà connus EQ et NE:

NE BNE OP1 Branch if Not Equal Sauter si OP1 = OP2
EQ BEQ OP1 Branch if EQual Sauter si OP1 <> OP2

Pour les conditions supérieur, inférieur, supérieur ou égal et inférieur ou égal, il nous faut revenir encore une fois sur la représentation des nombres binaires. Les nombres binaires peuvent être considérés aussi bien comme des nombres signés que comme des nombres non-signés. Un exemple nous permettra de mieux comprendre:

255 est supérieur à 0 (%11111111 est supérieur à %00000000)
-1 est inférieur à 0 (%11111111 est inférieur à %00000000)

Il faut bien sûr tenir compte de la formation du complément pour les nombres signés. Le processeur 68000 soutient le traitement des deux types de nombres, y compris pour les conditions. Les codes de

condition suivants sont utilisés pour les valeurs non-signées:

LO BLO OP1 Branch LOwer	Sauter si OP2 < OP1
LS BLS OP1 Branch Lower Same	Sauter si OP2 <= OP1
HI BHI OP1 Branch HIgher	Sauter si OP2 > OP1
HS BHS OP1 Branch Higher Same	Sauter si OP2 >= OP1

Pour les valeurs signées:

LT BLT OP1 Branch Less Than	Sauter si OP2 < OP1
LE BLE OP1 Branch Less/Equal	Sauter si OP2 <= OP1
GT BGT OP1 Branch Greater Than	Sauter si OP2 > OP1
GE BGE OP1 Branch Greater/Equal	Sauter si OP2 >= OP1

Indiquons ici encore une fois clairement que les instructions de comparaison fournies ici ne fonctionnent qu'en liaison avec l'instruction CMP. Il faut également noter que l'opérande 2 est toujours comparé à l'opérande 1. Nous allons maintenant évoquer les deux derniers codes de condition, qui sont particulièrement liés aux instructions DBcc.W OP1,OP2 et Scc.B OP1. Il s'agit d'instructions particulières qui influencent l'exécution de l'instruction en fonction de la condition. Nous reviendrons, pour autant que cela soit nécessaire, sur ces instructions de façon plus détaillée, lorsque nous les rencontrerons.

T	ST OP1	True: la condition est toujours remplie
F	DBF OP1,OP2	False: la condition n'est jamais remplie

Ces codes de condition n'ont aucune signification pour les instructions de saut. Pour l'instruction parfaitement imaginable "BT" (sauter toujours), il existe en effet une mnémonique spéciale (BRA) et la variante "BF" (ne jamais sauter) est interdite car le code d'opération correspondant serait identique à celui de BSR (appel relatif de sous-programme).

Syntaxe des modes d'adressage

Les modes d'adressage vous permettent de déterminer avec quels OPERANDES une opération doit être exécutée. Dans le chapitre "Le microprocesseur 68000", nous vous avons expliqué la fonction de ces 14 modes d'adressage. Nous allons maintenant vous montrer comment représenter et utiliser les différents modes d'adessage dans la programmation en assembleur.

Si vous examinez la liste des mnémoniques, vous constaterez qu'il y a quatre types d'instructions fondamentaux:

- Les instructions sans opérande
- Les instructions à un opérande
- Les instructions à deux opérandes
- Les instructions avec un opérande et un second opérande facultatif

Les instructions qui ne nécessitent aucun opérande constituent la première et la plus simple forme d'adressage. L'instruction contient en effet IMPLICITEMENT le mode d'adressage. Dans la syntaxe assembleur, l'adressage implicite est représenté en écrivant simplement la mnémonique, sans autre indication.

```
Exemples: NOP    ; pas d'opération
          RESET  ; réinitialiser la périphérie
          RTS    ; retour du sous-programme
```

Toutes les instructions qui nécessitent des opérandes utilisent pour cela une ou deux des 13 autres modes d'adressage que nous allons maintenant expliquer. Théoriquement, chacun de ces modes d'adressage devrait pouvoir être envisagé pour former un opérande.

Dans les faits, le jeu d'instructions du processeur 68000 connaît cependant certaines limites dans les possibilités de combinaison des instructions avec les différents modes d'adressage. Pour savoir quelles instructions peuvent être combinées avec quels modes d'adressage, nous vous invitons à vous reporter à la liste des instructions qui vous est fournie en annexe. L'explication suivante des autres modes d'adressage vous donne des exemples de ces combinaisons.

Un des modes d'adressage les plus importants est l'adressage registre direct. On distingue entre les registres de données et les registres d'adresse.

```
Exemples: CLR.L     D0      ; annuler le registre données 0
          ADD.L     D1,D0   ; D0 = D0 + D1 (addition)
          ADDA.L    D0,A1   ; A1 = A1 + D0 (addition)
          MOVEA.L   A0,A1   ; A1 = A0 (transfert)
```

Dans les exemples ci-dessus, nous avons toujours choisi "long mot" comme largeur de traitement, de façon à utiliser les 32 bits d'un registre. ".W" n'entraînerait par exemple l'utilisation que des 16 bits inférieurs. Ces exemples vous montrent également comment utiliser deux opérandes (séparation avec la virgule) et comment combiner deux modes d'adressage (registre d'adresse et registre de données direct).

Si des constantes sont nécessaires pour une opération, on utilise le mode d'adressage IMMEDIAT. Dans la syntaxe assembleur, on écrit alors un "#" (dièze) suivie d'une expression arithmétique quelconque.

```
Exemples: MOVE.L#30,D0  ; D0 = 30 (charger 30 dans D0)
          ADDI.W#$A0,D7; D7 = D7 + 160 (addition)
          CMPI.B #CR,D0 ; comparer D0 à CR
```

On connait pour beaucoup d'opérandes l'adresse à laquelle ils figurent dans la mémoire. Dans ce cas, le programmeur peut accéder directement à cette adresse. Ce type d'accès est appelé ADRESSAGE ABSOLU. Le 68000 distingue entre les modes d'adressage ABSOLU LONG et ABSOLU COURT. Lors de la formulation d'un programme en assembleur, le programmeur n'a cependant pas à tenir compte de cette distinction car l'assembleur sélectionne de lui-même le mode d'adressage qui convient lors de l'assemblage, au vu de la taille des opérandes. Dans la syntaxe assembleur, il suffit que le programmeur indique comme opérande l'adresse voulue au moyen d'une expression arithmétique.

Exemples:

```
MOVE.B    $00ABCDEF,D0  charger un octet (abs. long)
CLR.W     $1000         ; supprimer un mot (court)
```

Une autre forme de l'adressage d'un opérande est l'ADRESSAGE REGISTRE D'ADRESSE INDIRECT. Ce n'est pas ici l'adresse absolue qui est indiquée dans l'instruction mais seulement un registre d'adresse qui contient l'adresse absolue. Dans la syntaxe assembleur, cette situation est indiquée par la mise entre parenthèses du registre d'adresse.

```
Exemples: MOVE.L  D0,(A0)     ; D0 dans l'adresse en A0
          MOVE.B  (A0),(A1)   ; amener un octet de (A0) dans
          (A1)
```

Notez que dans le dernier exemple, le transfert d'un octet dont l'adresse est dans le registre d'adresse A0 à une adresse contenue dans le registre d'adresse A1 se fait sans passer par un autre registre.

Les modes POSTINCREMENT et PREDECREMENT constituent une extension de l'adressage registre d'adresse indirect.

Comme le montre le dernier exemple, le transfert d'un octet (ou de toute autre largeur de traitement) est très facile à réaliser.

Dans la pratique, ce sont cependant souvent des chaînes entières d'octets qui doivent être traitées. Il vous faut pour cela programmer une boucle dans laquelle les registres d'adresse seront chaque fois augmentés d'un nombre d'octets correspondant au nombre d'octets transférés (postincrément = augmentation a posteriori). Le mode d'adressage prédécrément (diminution a priori) constitue l'inverse du postincrément. Dans la syntaxe assembleur, ces modes d'adressage sont représentés en plaçant le signe plus après l'opérande ou le signe moins avant l'opérande.

```
Exemples: CLR.B    (A0)+      ; annuler octet et A0 = A0 + 1
          CLR.W    -(A1)      ; A1 = A1-2 et annuler un mot
          MOVE     (A0)+,(A1)+ ; décaler mot, adresses +2
          MOVE     (A0)+,(A1)- ; rotation de mots: A0 vers A1
          MOVE     (A0)+,D0   ; mot de A0 dans D0, A0 + 2
```

Une autre variante de l'adressage registre d'adresse indirect est l'ADRESSAGE REGISTRE D'ADRESSE INDIRECT AVEC DECALAGE.
Avec ce mode d'adressage, une valeur constante (décalage ou offset) est ajoutée à l'adresse proprement dite. Ce mode d'adressage permet au programmeur d'accéder très facilement aux éléments d'un tableau sans avoir à modifier le registre d'adresse lors de chaque accès. Dans la syntaxe assembleur, le décalage est placé, sous la forme d'une expression arithmétique, avant l'adressage indirect.

```
Exemples: CLR.B    0(A0)      ; octet adressé par A0
          CLR.B    1(A0)      ; octet suivant, A0 inchangé
          MOVE     (A0),1(A0) ; 1er octet après A0 dans second
```

Le mode d'adressage ADRESSAGE DE REGISTRE D'ADRESSE INDIRECT AVEC DECALAGE ET INDEX peut être considéré

comme une extension de l'adressage de registre d'adresse indirect avec décalage. Avec ce mode d'adressage, le contenu du registre d'adresse, le décalage et le contenu d'un autre registre de données ou d'adresse sont additionnés pour calculer l'adresse de l'opérande.
Ce mode d'adressage est utilisé pour pouvoir accéder dans le programme assembleur aux éléments d'un tableau, au moyen d'un pointeur variable (index). Une particularité de ce mode est la possibilité d'indiquer la largeur de traitement du registre d'index (registre de données ou d'adresse). C'est alors un mot (.W) ou un double mot (.L), c'est-à-dire le registre entier, qui sera utilisé chaque fois lors de l'addition. Dans la syntaxe assembleur, ce mode d'adressage est représenté en plaçant à la suite du registre d'adresse le registre d'index, dans les parenthèses qui indiquent l'accès indirect.

```
Exemples: NEG     1(A0,D0.L)  ;négation 2ème mot, indexé D0
          NEG     2(A0,A1)    ;négation 3ème mot avec index de
          mot
```

Une autre forme d'adressage est l'ADRESSAGE RELATIVEMENT AU COMPTEUR DE PROGRAMME ou adressage relatif. L'adresse relative est un index auquel le contenu actuel du compteur de programme est additionné pour calculer l'adresse mémoire effective. Ce mode d'adressage n'est utilisé que pour les instructions de saut conditionnelles (voir codes de condition) et pour deux instructions spéciales (BSR et DBcc). Comme l'adressage relatif peut être déduit immédiatement de la mnémonique utilisée, le programmeur n'a pas besoin de marquer cet adressage d'une façon spécifique dans le texte source. L'assembleur reconnaît non seulement le mode d'adressage mais il calcule en outre l'adresse relative (opérande) de l'instruction automatiquement lorsque le programmeur indique l'adresse du but du saut. Nous reviendrons sur cette fonction de l'assembleur lorsque nous traiterons du calcul d'adresse.

```
Exemples: BNE DIFFERENT      ;saut si OP1 <> OP2
          BSR CALCULER       ;saut relatif à un sous-programme
          DBF BOUCLE         ;saut au début de la boucle
```

Une forme particulière d'adressage relatif est constituée par l'ADRESSAGE RELATIVEMENT AU COMPTEUR DE PROGRAMME AVEC DECALAGE et l'ADRESSAGE RELATIVEMENT AU COMPTEUR DE PROGRAMME AVEC DECALAGE ET INDEX.
Ces deux derniers modes d'adressage correspondent exactement aux adressages de registres d'adresse indirect avec décalage et avec ou sans index. Ces deux modes d'adressage sont utilisés pour écrire des programmes relogeables. Ils ne peuvent cependant pas être combinés avec toutes les instructions machine importantes. Dans la syntaxe assembleur, ces deux modes d'adressage se distinguent de l'adresse de registre d'adresse indirect par l'indication de PC au lieu d'un registre d'adresse.

```
Exemples: CLR TABLEAU(PC)      ;annuler 1er mot du tableau
          CLR TABLEAU(PC,A0)   ;annuler tableau, indexé A0
```

Nous aurons dans les chapitres suivants à traiter encore souvent de l'utilisation de tous les modes d'adressage. Nous allons maintenant nous intéresser en conclusion aux particularités de syntaxe que présentent certaines instructions spéciales.

Comme vous le voyez d'après la description du dernier mode d'adressage, il y a encore certains noms réservés qui servent à désigner les registres spéciaux du 68000. Dans le cas de l'adressage relativement au compteur de programme, le compteur de programme a été désigné par "PC".

Comme vous le savez déjà, le registre d'adresse A7 est identique au POINTEUR DE LA PILE UTILISATEUR ou en mode superviseur au POINTEUR DE LA PILE SUPERVISEUR.

Pour accroître la lisibilité du texte source, la plupart des assembleurs autorisent l'utilisation de "USP" pour le pointeur de la pile utilisateur (USER STACK POINTER) et de "SSP" pour le pointeur de la pile superviseur (SUPERVISOR STACK POINTER). L'utilisation de "USP" avec l'instruction MOVE constitue une exception puisqu'elle produit des instructions différentes.

Le registre d'états ("SR" pour Status Register) constitue un autre registre spécial. Une partie du registre d'états est le registre code de condition ("CCR" pour Condition Code Register). Le registre d'états peut être fixé partiellement ou totalement sur une valeur définie au moyen d'une instruction MOVE spécifique. Un accès en lecture n'est possible qu'à la totalité du registre d'états. Ici aussi nous devons vous inviter à vous reporter pour une description détaillée à un ouvrage consacré au processeur. Nous nous contenterons donc de donner quelques exemples de la syntaxe assembleur.

```
Exemples: MOVE #0,CCR          ;registre code cond. est vrai
          MOVE #$1000,USP      ;initialiser pointeur de pile
          MOVE sr,-(A7)        ;sauver registre d'états sur pile
```

En ce qui concerne l'instruction MOVE, indiquons encore une formation de l'opérande tout à fait spéciale, la LISTE DES REGISTRES. L'instruction permet de charger ou de stocker certains ou tous les registres de données et/ou d'adresse à partir d'une adresse, en grandeur d'un mot ou d'un long mot. Cette instruction nécessite comme opérande un masque bits dans lequel chaque bit représente un registre déterminé. L'assembleur forme lui-même ce masque bits lorsqu'une liste de registres lui est communiquée.

Dans la syntaxe assembleur, une liste de registres est constituée par les noms de registre, dans un ordre quelconque, séparés entre eux par un slash ("/") ou bien une série de registres dont sont seuls indiqués les premier et dernier registres, séparés par un trait d'union.

Exemples:

```
MOVEM D0,-(A7)          ;1 registre sur la pile
MOVEM D0/A0,-(A7)       ;2 registres sur la pile
MOVEM (A7)+,D0-D7       ;reg. données retirés de la pile
MOVEM (A7)+,D0-D7/A0-A7;tous reg. retirés de la pile
```

Nous allons en résumé vous donner à nouveau, à la page suivante, un récapitulatif des règles de syntaxe des modes d'adressage avec indication des exceptions sur certains assembleurs.

Implicite	———
Registre de données direct	Dn
Registre d'adresse direct	An
Immédiat	#données.X
Absolu long	adresse.W
Absolu court	adresse.L
Registre d'adresse indirect	(An)
Postincrément	(An)+
Prédécrément	-(An)
Registre d'adresse indirect avec décalage	D16(An)
Reg.d'adresse indir. avec décalage et index	D8(An,Rn.X)
Relatif	Offset (décalage)
Relatif avec décalage	D16(PC)
Relatif avec décalage et index	D8(PC,Rn.X)
Liste de registres	Di-Dj/Ai-Aj
Pointeur de pile utilisateur	USP
Pointeur de pile superviseur	SSP
Registre d'états	SR
Registre de code de condition	CCR
Compteur de programme	PC

Légende:		
	Dn	Registres de données 0-7
	An	Registres d'adresse 0-7
	Rn	Dn ou An
	Données	Constante .B, .W ou .L
	Adresse	Constante .W ou .L
	Offset	Constante .B ou .W
	D8	Constante .B
	D16	Constante .W
	i, j, n	Numéros de registre 0-7
	.B	Octet
	.W	Mot
	.L	Long mot
	.X	.B, .W ou .L

Exceptions:	SP	correspond à USP
(rares	$	correspond à PC
cependant)	adresse	correspond à adresse
	Ri, Xi	correspond à Rn
	D	correspond à D8 ou D16
	A7	correspond à SSP

Les directives assembleur

Chaque assembleur dispose d'un certain nombre de directives assembleur (PSEUDO CODES D'OPERATION). Les directives ne produisent pas en général d'instructions machine. Pour la clarté de l'exposé, nous répartirons les directives les plus importantes dans les groupes suivants:

- calcul d'adresse, gestion et organisation de la mémoire
- gestion du texte source et commande du déroulement
- tableaux et zones de données
- déclaration des symboles
- traitement des macros
- formats de sortie et options

Toutes les directives sont représentées dans la syntaxe assembleur par un point (".") suivi d'une abréviation. L'abréviation peut être suivie de plusieurs opérandes. Pour la formation des opérandes, ce sont en général les mêmes règles qui s'appliquent que pour la formation des expressions arithmétiques.

La directive la plus importante du premier groupe est l'instruction "ORG". Cette directive indique à l'assembleur l'adresse à laquelle votre programme est censé tourner plus tard. C'est normalement la première instruction dans un texte source.

L'assembleur doit en tout cas recevoir une instruction ORG avant que la première ligne de code d'opération ou le premier tableau n'ait à être assemblé, de façon à ce que l'assembleur puisse stocker le code machine produit à une adresse définie.

Exemples:

```
.ORG $1000          ;indication absolue d'adresse
.ORG ADRDEBUT       ;le symbole doit avoir été défini
```

On veut parfois réserver de la place mémoire dans un programme, par exemple pour y stocker des valeurs internes. On utilise à cet effet l'instruction DS.X. Cette instruction réserve un certain nombre d'octets, mots ou longs mots dans le programme, en fonction de la largeur de traitement utilisée (.x). Cela entraîne la production d'un nombre correspondant de caractères de remplissage, à partir de la prochaine adresse disponible. C'est le plus souvent $00 qui est utilisé comme caractère de remplissage. Indiquons ici que le programmeur peut également doter une ligne pseudo code d'opération d'un label. Dans le programme assembleur, on pourra ainsi, avec un mode d'adressage approprié, accéder aisément et de façon entièrement symbolique à la zone d'adresses produite.

Exemples:

```
      .DS.B 256          ;réservation de 128 mots
TAB   .DS.W 128          ;réservation de 128 mots
DATA  .DS.L  64          ;réservation de 128 mots
OCTET .DS.B   1          ;réservation d'un octet
```

Le lecteur attentif aura reconnu dans le dernier exemple un problème caractéristique de la création de tableaux. Comme vous le savez déjà, une instruction machine doit toujours commencer à une adresse paire. Lorsque le programmeur définit des tableaux qui contiennent au total un nombre impair d'octets, la prochaine instruction machine devrait normalement se trouver à une adresse impaire.

Dans ce cas, on peut utiliser la directive EVEN qui placera le compteur d'adresse si nécessaire sur la prochaine adresse paire. Indiquons également qu'il y a des assembleurs qui tiennent automatiquement compte de la règle de l'accès par mot entier lors du traitement de tableaux.

Exemples:

```
DATA    .DS.B 3              ;réserver 3 octets
        .EVEN                ;accès à un mot entier
DEBUT   MOVE.B D0,DATA       ;remplir un tableau
```

Nous avons dit plus haut que la directive DS remplit la zone réservée de $00. Il y a cependant également des assembleurs qui permettent de sélectionner un autre caractère de remplissage ou des assembleurs qui ne produisent pas de caractère de remplissage et qui ne modifient pas la zone réservée lors du "linkage". Pour définir un caractère de remplissage, on utilise la directive FILL.

```
Exemples:   .FILL $20        ;remplir d'espaces
            .FILL "A"        ;remplir de A ($41)
```

Une autre forme de la directive DS.X est la directive DC.X. Cette directive réserve une zone mémoire et la remplit en même temps avec des constantes (tableaux). Le programmeur peut indiquer à la suite de la directive une liste d'expressions alphanumériques, séparées par des virgules, qui seront ensuite générées en fonction de la largeur d'écriture utilisée (.X).

Exemples:

```
TAB .DC.B 1,"A"      ;produit $0141
    .DC.W 1,"A"      ;produit $0001,$4100
    .DC.L 1          ;produit $0000,$0001
    .DC.L "AB"       ;produit $4142,$0000
```

Les exemples ci-dessus montrent aussi le traitement spécifique des chaînes de caractères. Les chaînes de caractères sont remplies de $00 en fonction de la largeur de traitement choisie.

En général, le compteur d'adresse de l'assembleur n'est pas automatiquement dirigé sur la prochaine adresse paire (directive EVEN) après une directive DC.X, de façon à ce que plusieurs DC.X puissent former un tableau cohérent. Les assembleurs intelligents reconnaissent automatiquement la fin d'un groupe d'instructions DC.X et corrigent le compteur d'adresse pour qu'il contienne une adresse paire, dès qu'une instruction DC.X est suivie d'une instruction non-DC.X.

Une autre tâche importante est la définition de symboles et de labels auxquels l'assembleur ne peut pas affecter une valeur automatiquement (buts de saut externes, constantes). Ces définitions sont réalisées avec la directive EQU. On ne distingue pas à cet égard s'il s'agit de symboles (données) ou de labels (adresses). Un symbole peut recevoir n'importe quelle valeur, représentée par une expression numérique, à condition qu'elle ne dépasse pas 32 bits. Un symbole ne peut être défini qu'une fois. Si un symbole doit recevoir une nouvelle valeur, le programmeur peut, sur certains assembleurs, utiliser une directive REDEF pour cette redéfinition de la valeur du symbole.

Exemples:

```
ADRESSE     .EQU  $1234     ;correspond à $00001234
CARACTERE   .EQU  "A"       ;correspond à $00000041
TEXTE       .EQU  "ABC      ;correspond à $00414243

TEXTE       .REDEF CR       ;correspond à $0000000D
```

Un autre groupe de directives assembleur concerne la gestion de plusieurs textes source, réalisés indépendamment les uns des autres, qui doivent aboutir à un programme assembleur unique.

On utilise en général à cet effet les directives INCLUDE et FILE. La directive file permet de "chaîner" plusieurs textes source. On écrit pour cela à la fin du premier texte source la directive file avec le nom du texte source suivant.

Lorsque l'assembleur assemblera le premier texte source et qu'il interprétera la directive file, l'assemblage se poursuivra avec le prochain texte source. La directive include se comporte de façon semblable mais à la fin du second texte source, celui indiqué par la directive include, l'assemblage se poursuit dans le premier texte source, à la suite de la directive include.

Exemples:

```
.FILE A:PARTIE2.SRC        ;continuer avec partie2
.INCLUDE A:LABELS.SRC      ;insérer les définitions
```

Le programmeur dispose d'une série de directives pour influer sur l'ordre de l'assemblage. La directive la plus simple est ".END". Cette directive met fin à l'assemblage dans la ligne de texte source qui contient la directive END.

```
Exemple:    .END           ;fin de l'assemblage
```

Tout un groupe de directives concerne l'assemblage conditionnel. A cet effet, avant l'assemblage d'une section marquée spécialement du texte source, une condition posée par le programmeur est testée et l'assemblage de cette section du texte source n'est effectué que si la condition peut être considérée comme remplie. Si la condition n'est pas remplie, cette section du texte source est exclue de l'assemblage.

Avant que nous n'en venions aux règles de syntaxe de ces directives, nous souhaitons illustrer brièvement l'application de l'assemblage conditionnel.

Supposons que vous développiez un programme qui doive être utilisé en français et en anglais. Vous pourriez développer le programme d'abord en français et entreprendre ensuite la traduction en anglais une fois le développement achevé. Vous vous rendrez alors compte qu'il ne suffit pas de traduire des mots français en anglais.
Il arrive en effet souvent que la structure générale d'un masque écran soit modifiée simplement parce qu'un seul mot compte un caractère de trop et qu'il n'est pas possible de lui trouver une abréviation ou un autre nom sensé. Des modifications de masque aussi importantes entraînent souvent dans leur sillage toute une série d'autres modifications. Une fois qu'un programme est terminé, même un programmeur expérimenté aura beaucoup de mal à se souvenir de la structure de toutes les sections du programme. Dans une modification après coup, il est souvent fatal de négliger certains endroits où des modifications devraient également être apportées.

La solution est l'assemblage conditionnel! Le programmeur pourra ainsi programmer et tester chaque solution partielle en deux versions française et anglaise. Au moyen de l'assemblage conditionnel, il pourra alors, lors de la suite du développement, prescrire que ne soit chaque fois assemblées avec les parties neutres que les parties françaises ou que les parties anglaises. Il n'existe pas en assemblage conditionnel de condition dans laquelle deux opérandes puissent être comparés entre eux. On calcule simplement une expression arithmétique dont le résultat est alors évalué. Si le résultat est vrai (non nul), la condition est remplie. Si le résultat est nul, la condition est considérée comme non remplie (fausse). Dans notre exemple, le programmeur pourrait fixer un symbole "LANGUE" sur 0 pour le français et sur 1 pour l'anglais. Il n'aurait alors qu'à utiliser ce symbole pour l'assemblage conditionnel.

Un section de texte source qui doit être assemblée de façon conditionnelle est introduite par une des directives suivantes:

Exemples:

```
.IFE LANGUE          ;assemblage si LANGUE=0
.IFN LANGUE          ;assemblage si LANGUE<>0
```

La section de texte source qui doit être assemblée de façon conditionnelle est terminée par la directive "ENDIF".
Si la condition n'était pas remplie, l'assemblage se poursuit à la ligne suivante.

Exemple: `.ENDIF ;fin de l'assemblage conditionnel`

A propos de l'assemblage conditionnel, c'est ici le lieu de traiter de la possibilité de commande de l'assembleur pendant le déroulement de l'assemblage. Les bons assembleurs autorisent la conduite d'un dialogue avec le programmeur pendant l'assemblage. Deux directives sont utilisées à cet effet. La directive PRINT permet de sortir des messages sur l'écran. Avec la directive "INPUT", on peut demander au programmeur d'effectuer une entrée au clavier. L'assembleur affecte alors, comme une directive EQU, l'entrée au clavier à un symbole. La directive INPUT est particulièrement utile lors de la phase de développement et de test, lorsqu'il s'agit de modifier des paramètres variables souvent utilisés. Avec cette forme de programmation, on évite d'avoir à utiliser un éditeur pour ne modifier que quelques paramètres. Avec l'utilisation d'un éditeur, le programmeur est contraint à des accès inutiles à la mémoire de masse. Par rapport à notre précédent exemple, on pourrait sortir une question sur la langue à employer et affecter par une entrée au clavier la valeur de commande correspondante (0 ou 1) au symbole LANGUE.

Exemple:

```
.PRINT "Francais (0) ou anglais (1) ?",CR
.INPUT LANGUE
```

Un autre groupe de directives concernent la COMMANDE DU FORMAT.

Ces directives permettent de déterminer le format du listing d'assemblage. Le programmeur peut en outre utiliser certaines options pour adapter le listing à ses besoins ou pour l'organiser de façon plus claire (documentation du programme). La présentation qui suit montre les possibilités très diverses et explique les règles de syntaxe:

Exemples:

```
.NOLIST          ;ne pas produire de listing
.LIST            ;produire à nouveau un listing
.PAGE            ;passer à la page suivante
.NOFORMAT        ;pas de sortie formatée
.FORMAT          ;sortir un listing formaté
.SPC 3           ;imprimer trois lignes vides
.LLEN 80         ;format 80 caractères par ligne
.LINE 72         ;fixer longueur de page à 72 lignes
.TOP 6           ;6 lignes vides entre les pages
.TITLE "Texte"   ;définir le titre de page
.XPUNCH          ;pas de hex dump
.PUNCH           ;à nouveau hex dump
.NOCROSSREF      ;interdire références croisées
.CROSSREF        ;imprimer références croisées
```

Certains assembleurs manipulent les différentes possibilités de la commande de l'assemblage au moyen d'une directive OPTION. On utilise donc non pas une directive pour chaque possibilité de commande, mais une directive unique pour toutes les possibilités. Une liste d'options placée à la suite de la directive OPTION permet d'indiquer quelles commandes doivent être activées ou désactivées.

Exemples:

```
.OPTION NOLIST, NOPUNCH, NOFORMAT
.OPTION LIST, PUNCH, FORMAT
```

Les directives assembleur sont en général parmi les éléments les moins normalisés de la programmation en assembleur. Nous avons essayé de vous faire comprendre l'utilité et les possibilités des directives. Il est cependant conseillé de bien étudier le manuel de votre assembleur pour vous assurer des règles de syntaxe qu'il utilise.

Traitement de macros avec l'assembleur

Les possibilités du traitement des macros vous ont déjà été présentées. Nous souhaitons vous présenter ici les règles de syntaxe qui les régissent. La définition et l'appel des MACROS s'effectuent avec des instructions semblables aux directives.

Exemple:

```
.MACRO TABLOCTT_ADR (XIND,YIND,ADR)
;
MOVEM D0,-(SP)          ;sauver les registres
MOVE.W YIND,D0          ;index Y pour multiplier
MULU.W #100,D0          ;*octets par ligne
ADD.L XIND,D0           ;additionner index X
ADDI.L #BASE,D0         ;adresse de dégut du tableau
MOVE.L D0,ADR           ;stocker adresse
MOVEM (SP)+,D0          ;registres sur ancienne valeur
;
.ENDM                   ;fin de la définition

...TABLOCTT_ADR (XPOI,YPOI,ERG)
```

Il s'agit dans notre exemple de calculer l'adresse d'un octet dans un tableau bi-dimensionnel. Rappelons qu'un macro n'est pas un appel de sous-programme mais que l'assembleur se comporte comme si, à la place de l'appel de macro, le texte de la définition de la

macro-instruction avait été écrit. L'intérêt est pour le programmeur de pouvoir utiliser plusieurs fois dans un programme un macro, une fois qu'il a été défini. L'assembleur doit pour cela savoir à quelles données (symboles ou labels) le macro doit être appliqué. Lors de la définition du macro, le programmeur indique à l'assembleur quels symboles doivent être remplacés par d'autres lors de l'appel du macro. Tous les symboles utilisés dans le macro ne valent qu'à l'intérieur du macro de façon à éviter des doubles définitions lors d'appels répétés de macros.

Le programmeur peut cependant se référer à l'intérieur d'un macro à tout symbole existant par ailleurs.

Nous allons vous montrer le traitement des macros en vous montrant ci-dessous ce que l'assembleur réalise lors de l'assemblage si notre exemple est appelé par deux fois.

```
:.TABLOCTT_ADR (XINDEX,YINDEX,ADRESSE)


MOVEM D0,-(SP)        ;sauver les registres
MOVE.W YINDEX,D0      ;index Y pour multiplier
MULU.W #100,D0        ;*octets par ligne
ADD.L XINDEX,D0       ;additionner index X
ADDI.L #BASE,D0       ;adresse de dégut du tableau
MOVE.L D0,ADRESSE     ;stocker adresse
MOVEM (SP)+,D0        ;registres sur ancienne valeur


..TABLOCTT_ADR (XPOI,YPOI,ERG)
```

```
MOVEM D0,-(SP)      ;sauver les registres
MOVE.W YPOI,D0      ;index Y pour multiplier
MULU.W #100,D0      ;*octets par ligne
ADD.L XPOI,D0       ;additionner index X
ADDI.L #BASE,D0     ;adresse de dégut du tableau
MOVE.L D0,ERG       ;stocker adresse
MOVEM (SP)+,D0      ;registres sur ancienne valeur
```

Indiquons simplement brièvement qu'il existe encore des FONCTIONS MACRO qui permettent de doter les macros d'une sorte "d'intelligence propre".

Il existe encore sur les assembleurs les plus complets des STRUCTURED CONTROL STATEMENTS que l'on peut considérer comme des macros pré-définis. La programmation en assembleur offre encore de nombreuses autres possibilités sur lesquelles nous ne voulons cependant pas nous étendre dans le présent ouvrage. Il s'agissait pour nous de décrire le principe des macros et nous espérons qu'avec ces possibilités très puissantes, qui se rapprochent de la programmation en langage évolué, nous vous avons "mis l'eau à la bouche".

Le calcul d'adresse

Pour améliorer notre compréhension de la programmation en assembleur et du mode de fonctionnement d'un assembleur, nous allons nous intéresser maintenant au calcul d'adresse. Nous pouvons comprendre comme calcul d'adresse toutes les activités de l'assembleur qui concernent la gestion des symboles, le calcul des adresses relatives ou des décalages et la gestion de la zone d'adresses.

Le calcul d'adresse constitue ainsi, avec la traduction des mnémoniques, le plus important allègement du travail du programmeur. Par ailleurs, c'est justement à propos du calcul d'adresse qu'apparaissent certaines "erreurs de débutant" caractéristiques.

La plupart des assembleurs autorisent une écriture pleinement symbolique du texte source. Dans un cas extrême, il peut arriver que le programmeur ait défini toutes les constantes et adresses externes d'un programme comme des symboles ou labels (par des déclarations). Ces symboles ne posent pas de problème à l'assembleur puisque leur valeur réelle a été définie. Les choses se compliquent en ce qui concerne les labels qui sont définis à l'intérieur d'un programme, c'est-à-dire les buts de saut pour les branchements, appels de sous-programmes et accès aux tableaux.

Pour comprendre cette difficulté, souvenons-nous de la façon dont se déroule l'assemblage.

Le texte source est traité par l'assembleur ligne par ligne, en commençant par la première ligne. Si une définition de symbole apparaît dans une ligne, le symbole est entré dans une table, avec la valeur correspondante. Cette table des symboles contient tous les symboles connus jusqu'ici avec leurs valeurs respectives. Si un symbole est utilisé comme opérande lors de l'assemblage, par exemple comme but de saut, l'assembleur cherchera la valeur du symbole correspondant dans la table des symboles et produira l'instruction de saut avec cette valeur comme opérande. C'est ici qu'apparaît déjà la première difficulté pour l'assembleur. Le symbole recherché figure-t-il bien dans la table ou la définition n'intervient-elle que dans une des lignes suivantes?

Si le symbole est contenu dans la table, c'est évidemment qu'il a été défini avant la ligne actuellement traitée. C'est pourquoi ce cas est qualifié de référence retrospective.

Si le symbole recherché ne figure pas encore dans la table, c'est certainement qu'il sera défini quelque part dans la suite du texte. Ce cas est appelé référence prospective.

Ces références prospectives constituent le problème principal du calcul d'adresse. Pour traduire une instruction, l'assembleur a également besoin de la valeur d'un référence prospective. L'assembleur utilise donc une astuce élémentaire pour maîtriser cette difficulté. Le texte source est simplement assemblé à deux reprises consécutives mais aucun code machine n'est produit lors du premier passage (PASS). A la fin du premier passage, tous les symboles utilisés devraient normalement figurer dans la table des symboles, avec leurs valeurs. Si ce n'est pas le cas, un message d'erreur est produit et l'assemblage est interrompu. Cela explique donc pourquoi un assembleur entièrement symbolique peut être qualifié également d'assembleur à deux passages.

Malgré cette astuce, un programmeur peut "désarçonner" un assembleur avec des constructions tout à fait élémentaires bien qu'il utilise un texte source correct sur le plan de la syntaxe et de la logique. Lorsque ces situations d'erreur peuvent être maîtrisées, cela plaide bien sûr pour la qualité d'un assembleur moderne. Mais comme il existe de nombreux assembleurs qui se font "piéger" par de telles constructions, nous allons vous présenter quelques-unes de ces erreurs.

Indiquons en premier lieu les "définitions cycliques". Il s'agit là d'une erreur du programmeur que certains assembleurs ont du mal à déceler.

Exemple:

```
SYMBOL1 .EQU SYMBOL2      ;première définition
SYMBOL2 .EQU SYMBOL3      ;seconde définition
SYMBOL3 .EQU SYMBOL1      ;la valeur demeure inconnue
```

Il est évident que notre exemple conduit à une erreur, puisque finalement aucun des symboles n'a été défini. Il en va autrement dans le cas suivant qui ressemble beaucoup au premier.

Exemple:

```
SYMBOL1 .EQU SYMBOL2      ;première définition
SYMBOL2 .EQU SYMBOL3      ;seconde définition
SYMBOL3 .EQU 1234567      ;la valeur est inconnue
```

Il s'agit dans cet exemple d'une multiple référence prospective. Il est aisé de comprendre pourquoi un assembleur à deux passages ordinaire échouera dans notre exemple. Il suffit pour cela d'effectuer l'assemblage "manuellement". Après le passage 1, les symboles 1 et 2 sont indéfinis. Au second passage, le symbol1 ne peut pas être défini puisque le symbol2 n'a toujours pas été défini. Ce n'est qu'au cours du second passage que le symbol2 pourrait être défini puisque le symbol3 est connu après le passage 1.

Mais cette définition ne pourra pas s'effectuer car lors du second passage l'assemblage sera déjà interrompu avec un message d'erreur lors de la définition de symbol1.

Pour être complet, il convient d'indiquer ici une autre source d'erreur bien que la plupart des assembleurs soient à même de rectifier cette erreur. Il s'agit de ce qu'on appelle l'erreur de phase. On désigne ainsi un écart entre le calcul d'adresse du premier passage et celui du second passage; cet écart est en général reconnu par l'assembleur et corrigé sans que le programmeur s'en aperçoive. Ces erreurs de phase sont produites par des instructions machine qui ont une longueur variable, en fonction de la taille de l'opérande.

Si dans une telle instruction l'opérande est défini par une référence prospective, l'assembleur réservera au premier passage la longueur maximum pour cette instruction et calculera les adresses symboliques en fonction de cette longueur maximum. Si un petit opérande entraîne au second passage une réduction de la taille de l'instruction par rapport à celle qui avait été supposée au premier passage, toutes les références aux labels suivants doivent être également modifiées en fonction de cette réduction.

Le listing assembleur

Nous avons déjà évoqué plusieurs fois le listing assembleur et les possibilités de formatage de la sortie du listing. Nous souhaitons ici souligner certains détails et particularités concernant le traitement des erreurs.

La page suivante vous présente un listing assembleur typique. Dans la présentation suivante, nous ne nous attacherons pas au contenu du programme d'exemple mais à une explication des différents éléments du listing.

1. Titre pouvant être modifié par le programmeur
2. Nom de fichier du texte source traité
3. Numérotage des pages, numérotage continu
4. Numéro de ligne du texte source, numérotage continu
5. Adresse mémoire de l'instruction ou du tableau
6. HEX dump de l'instruction ou du tableau
7. Champ du label, contient le nom du label de la ligne
8. Champ des mnémoniques, contient les mnémoniques ou directives
9. Champ des opérandes, contient les opérandes ou le mode d'adressage
10. Champ de commentaire, contient les commentaires

11. Ligne de commentaire, contient un commentaire sur une ligne entière

12. Table des symboles, liste de tous les symboles
13. Valeur de symbole, valeur (adresses, données) des symboles

On distingue dans l'analyse des erreurs assembleur des errors, fatal errors et des warnings. Ces derniers indiquent une source d'erreur possible sans provoquer l'interruption de l'assemblage car il se peut que le programmeur souhaite explicitement arriver à la situation indiquée. L'assembleur indiquera par exemple un symbole inutilisé ou une possibilité de réduire la largeur d'un mot lors du traitement.

```
1
2
3  ********************************************
4  *** Clignotage de l'ecran    Listing demo ***
5  ********************************************
6
7  00000000 7E09          move.l  #9,d7       * nombre de clignot.
8
9  00000002 2C7C00078000  flash:  movea.l  ##78000,a6  * pointeur sur ecran
10
11 00000008 469E          loop:   not.l   (a6)+       * inverser
12
13 0000000A BDFC00080000          cmpa.l  #$78000+$8000,a6 * fin ecran ?
14 00000010 65F6                  blo     loop        * N: continuer
15
16 00000012 2C3C0004FFFF          move.l  ##$4ffff,d6 * temporisation
17
18 00000018 5386          wait:   subq.l  #1,d6       * jusqu'a ce que d6=0
19
20 0000001A 0C8600000000          cmpi.l  #0,d6       * tester
21 00000020 66F6                  bne     wait        * N: wait
22
23 00000022 51CFFFDE              dbf.w   d7,flash    * repeter eventuell.
24
25 00000026 3F3C0000              move.w  #0,-(sp)    * code: WARMSTART
26 0000002A 4E41                  trap    #1          * appel de GEMDOS
27
28
29 0000002C                       .end
```

Les fatal errors sont des erreurs qui entraînent l'interruption de l'assemblage parce que le traitement de la suite du texte serait sans intérêt. Des déclarations de symboles manquantes constituent par exemple des fatal errors.

Certains assembleurs traitent les erreurs normales telles par exemple que les distances de saut trop importantes comme des fatal errors. Bien que cela puisse entraîner des erreurs dérivées, il peut cependant être intéressant de poursuivre l'assemblage jusqu'à la première fatal error, ne serait-ce que pour afficher les erreurs rencontrées. Cette méthode a l'avantage de permettre de déceler de nombreuses erreurs élémentaires (par exemple de syntaxe) dès le premier assemblage. Le programmeur pourra ainsi, dès après le premier parcours de correction, corriger bien plus que la première erreur constatée. Cela permet un gain de temps considérable pendant le développement car il n'est pas nécessaire d'éditer le texte source et de lancer à nouveau l'assembleur pour chaque erreur constatée.

Le traitement des erreurs et la structure du listing assembleur sont sur certains assembleurs très différents de notre exemple, de sorte que nous devons encore une fois vous inviter à vous reporter au manuel de votre assembleur. Vous y trouverez une description de tous les messages d'erreur et warnings.

Le mode d'emploi de l'assembleur

Vous avez édité un texte source et vous voulez maintenant le traduire en langage machine avec l'assembleur. Nous partons du principe que vous avez produit un texte source avec un éditeur, que le texte source a été stocké sur la disquette et que vous avez mis fin au programme d'éditeur. Vous vous trouvez maintenant au niveau des commandes de tout le package assembleur. Ce mode commande est le plus souvent identique aux modes TOS ou GEM de l'Atari ST.

Vous vous trouvez donc au niveau du système d'exploitation. C'est à partir de ce niveau que l'assembleur peut être lancé.

Avant que l'assembleur ne puisse commencer la traduction, il a encore besoin du nom de fichier du texte source qui doit être assemblé. Il est ici possible de faire en sorte que l'assembleur demande lui-même, après avoir été appelé, le nom du fichier ou que le programmeur transmette le nom de fichier en même temps que le nom de programme de l'assembleur, directement au niveau commande (système d'exploitation). Le système d'exploitation recevra le nom de fichier comme paramètre et le transmettra à l'assembleur.

Exemples:

```
A:ASSEM                 ...Appel
File? TEST.SRC          ...Demande du paramètre

A:ASSEM TEST.SRC        ...Appel avec paramètre
```

Outre le nom de fichier qui doit toujours être indiqué, le programmeur peut également indiquer lors de l'appel de l'assembleur des options d'assemblage. Il s'agit de fonctions de commande pour le fichier de code objet et le listing assembleur. Au moyen d'un paramètre de commande ou d'une interrogation de l'assembleur, le programmeur peut ainsi interdire la réalisation d'un fichier de code objet ou diriger la sortie du listing vers une imprimante ou vers l'écran.

Exemples:

```
A:ASSEM                     ...Appel
File? TEST.SRC              ...Texte source
Object? ABS.OBJ             ...Nom du fichier objet
Listing? P                  ...Sortie sur imprimante
A:ASSEM TEST.SRC/ABS.OBJ/P  ...Paramètres
```

Ces paramètres de commande doivent absolument être indiqués. L'assembleur supposera des valeurs par défaut en cas d'indications manquantes. Notre exemple avait simplement pour but de vous expliquer le principe. Nous vous invitons à vous reporter à votre manuel pour savoir quelles valeurs par défaut votre assembleur suppose et comment il doit être appelé.

Le débugger

Nous avons déjà parlé du débugger/moniteur. Nous allons vous présenter l'utilisation et les possibilités de cet "utilitaire" de test.

Le débugger est en général lancé comme l'assembleur. On peut en général employer comme paramètre le nom de programme du programme machine à tester.

Exemples:

DDT	...Appel du débugger
DDT TEST.OBJ	...Appel et chargement de TEST

DDT est dans notre exemple le nom de programme du "Dynamic Debugging Tool" du système d'exploitation CP/M 68K. Sur l'Atari ST, le débugger s'appelle SID (Symbolic Instruction Debugger). Le SID est une version perfectionnée du DDT qui autorise le traitement symbolique d'un programme.
Nous allons vous présenter ici les principes d'utilisation de cet outil.

Les fonctions d'un programme de débugger/moniteur peuvent être réparties en quatre groupes principaux:

- lecture/écriture (disquette) du programme (zone mémoire)
- examen et modification du contenu des registres et de la mémoire
- test du comportement du programme (trace et break points)
- fonctions auxiliaires (conversion hex/déc, arithmétique, etc.)

Après avoir été appelé, le débugger se trouve à un niveau de commande propre. Cela signifie que le débugger met à la disposition du programmeur une ligne d'entrée et qu'il attend l'entrée d'une instruction qui devra être terminée par l'emploi de la touche Return. Remarquons ici que le débugger est le plus souvent un outil très primitif et inintelligent. La structure des commandes est très rigide et très sensible aux entrées incorrectes. Les erreurs de syntaxe ou de logique, pour autant qu'un contrôle précis des erreurs soit bien effectué, entraînent des messages très succincts. Le programmeur doit par ailleurs avoir bien conscience de la très grande "proximité" de la machine car des modifications irréfléchies dans la zone d'adresses peuvent conduire au "plantage" tant redouté de la machine.

Nous n'allons pas décrire dans ce qui suit chaque instruction du débugger. Nous souhaitons plutôt vous présenter des détails intéressants qui doivent vous rendre plus facile l'entrée dans la phase de test. Le débugger est finalement le seul outil qui permette au programmeur de suivre à la trace une erreur de programmation.

Le débugger que vous utilisez peut avoir une structure d'instruction quelque peu différente. Il est donc en tout cas recommandé de bien étudier le manuel de votre débugger. La liste suivante vous montre, à titre d'exemple, de quelles instructions dispose un débugger typique tel que le DDT 68K de Digital Research.

Enom	chargement d'un programme pour le tester
V	afficher les paramètres du programme chargé
Inom	générer le bloc de contrôle du fichier (FCB)
Rnom	charger une zone mémoire à partir de la disquette
Wnom,s,f	écrire une zone mémoire de l'adresse s à l'adresse f
Ds	affichage HEX/ASCII d'octets, à partir de l'adresse s
Ds,f	affichage HEX/ASCII d'octets, adresses s à f
DWs	affichage HEX/ASCII de mots, à partir de l'adresse s
DLs	affichage HEX/ASCII de longs mots, à partir de s
Ls	désassembler à partir de l'adresse s
Ls,f	désassembler, adresses s à f
X	affichage des registres 68000 Rn,PC,USP,SSP,ST
Xr	modifier un registre (r=Rn,PC,USP,SSP,ST)
Ss b...b	écrit des octets (b) dans la mémoire à partir de s
SWs w...w	écrit des mots (w) dans la mémoire à partir de s
SLs l...l	écrit des longs mots (l) dans la mémoire à partir de s
Fs,f,x	remplit la mémoire de s à f avec un octet (b)
FWs,f,x	remplit la mémoire de s à f avec un mot (w)
FLs,f,x	remplit la mémoire de s à f avec un long mot (l)
Ms,f,d	copie mémoire de s à f dans d (b)
G	début du programme à partir de l'état actuel du PC
Gs	début du programme à partir de l'adresse s
Gs,b1..b2	début programme à partir de l'adr. s avec break points
T	trace programme à partir d'état actuel du PC
Tn	trace n instructions machine à partir état PC
U	exécuter une instruction machine à partir état PC
Un	exécution programme, trace après n instructions
Hx1,x2	former somme et différence de x1 et x2

Légende:

nom = nom de fichier	s = adresse de début	f = adr. finale
r = registre	b = octet	w = mot
l = long mot	x = b/w/l	d = adr.objet
bn = break point	n = nombre (1...n)	... = série

Une des fonctions les plus intéressantes du débugger est le désassemblage. Il s'agit exactement du contraire de l'assemblage. Une section de programme désassemblée est montrée au programmeur non seulement comme hex-dump mais aussi sous la forme d'un texte en clair, d'après l'écriture assembleur. L'affichage comprend les mnémoniques, les opérandes et les modes d'adressage. Cette forme de représentation est très lisible et facilite au programmeur la recherche des erreurs.

La fonction d'aide pour les tests la plus importante est certainement la fonction trace. Elle permet le traitement du programme machine en mode pas à pas, le programme machine étant traité instruction pour instruction. Si l'on excepte certaines limites concernant les routines du système d'exploitation et les sections de programme pour lesquelles la rapidité d'exécution joue un rôle fondamental (interruptions), le programme à tester peut être traité exactement comme il le serait s'il était exécuté en temps réel sans programme d'aide. En mode pas à pas, chaque instruction exécutée est affichée ainsi que le jeu de registres actuel. Cet utilitaire permet au programmeur de simuler pas à pas le mode de fonctionnement de son programme.

Le traitement des break points constitue le complément idéal du mode trace. Le programme à tester n'est pas traité ici pas à pas mais exactement comme s'il n'y avait aucun utilitaire de test. Le programmeur a cependant la possibilité d'interrompre de façon sélective, à des endroits déterminés, le traitement du programme.

Il définit à cet effet un break point (point d'interruption). Le débugger surveille le programme machine et interrompt le traitement lorsque le processeur arrive à l'adresse d'un break point. Le contenu actuel des registres est indiqué au programmeur. Avec les autres instructions de manipulation du débugger, le programmeur est ainsi à même de vérifier le résultat de l'exécution de son programme.

Conventions de procédure

Vous serez confronté à plus d'un titre, en programmation assembleur, aux conventions de procédure. En général les programmes machine ne peuvent pas tourner aussi facilement sur un ordinateur. Il y a un système d'exploitation qui soutient les fonctions essentielles d'un ordinateur. Même en programmation machine, le programmeur ne programmera pas lui-même toutes les fonctions. Il peut utiliser des routines du système d'exploitation (par exemple pour sortir un caractère à l'écran) sans devoir réfléchir aux conditions, dépendant de l'électronique de la machine, de la réalisation de ces fonctions.

Ces fonctions du système d'exploitation sont standardisées. C'est pourquoi des programmes qui accèdent uniquement à des routines du système d'exploitation peuvent tourner sur n'importe quel ordinateur utilisant le même microprocesseur et le même système d'exploitation. Lors de l'utilisation des fonctions du système d'exploitation, un ou plusieurs paramètres sont transmis à la fonction avant qu'elle ne soit appelée. Après exécution de la fonction, un résultat est transmis en retour au programme d'appel.

Les conventions de procédure ont donc pour but de définir comment les paramètres doivent être transmis, comment la fonction doit être appelée et où les résultats doivent être placés.

Une autre forme de conventions de procédure concerne le programme machine. Le programme objet doit être doté d'une marque déterminée et d'informations concernant le programme à exécuter (adresse de début, longueur du programme, etc.), de façon à ce qu'il puisse être exécuté par le système d'exploitation.

Les conventions de procédure sont fixées de façon définitive en ce qui concerne le système d'exploitation. Il est cependant conseillé de se conformer à certaines conventions de procédure également en ce qui concerne la modularisation de vos propres programmes machine. Nous allons vous présenter maintenant un certain nombre de possibilités à cet égard.

La forme la plus simple est la transmission par registres. Tous les paramètres sont alors transmis à travers les registres d'adresse ou de données. Les résultats seront à leur tour placés dans des registres déterminés. On veille en général à ce que tous les registres inutilisés restent inchangés. Si une fonction nécessite des registres supplémentaires, ils sont sauvés au début de la fonction et restaurés à la fin de celle-ci. La sauvegarde des registres s'effectue sur la pile, comme d'habitude.

Nous userons dans nos exemples de préférence de la transmission par les registres. Lorsque nous utiliserons d'autres possibilités, par exemple pour la programmation des récursions, nous expliquerons ces possibilités de façon spécifique.

Une autre forme de transmission de paramètres consiste à définir certaines zones mémoire pour la transmission des valeurs. Après exécution de la fonction, les résultats sont également placés dans un endroit déterminé du bloc de paramètres. Un bloc de paramètres peut figurer à une adresse fixe de la mémoire mais en général le programmeur transmettra à la fonction, dans un registre d'adresse, l'adresse de début du bloc de paramètres.

La forme la plus élégante de transmission de paramètres est l'utilisation de la pile utilisateur ou de piles définies par le programmeur au moyen des registres d'adresse. Cette forme est particulièrement soutenue par les instructions LINK et UNLINK du 68000. Cette transmission de paramètres ressemble au bloc de contrôle. Les zones de paramètres ne sont cependant pas ici placées à une adresse fixe de la mémoire mais gérées de façon dynamique à travers la pile. Nous expliquerons précisément cette forme de transmission de paramètres avec un exemple illustrant les récursions.

Dans l'utilisation de procédures et de routines du système d'exploitation, les conventions de procédure ne définissent pas uniquement les paramètres en entrée et en sortie mais aussi la forme de l'appel de la routine elle-même. On distingue essentiellement les modes d'appel suivants:

- appel de sous-programme avec l'adresse de début de la routine

- appel de sous-programme à travers une table de saut. Ici, toute une série d'instructions de saut consécutives sont définies dans la mémoire. Les différentes instructions de saut branchent sur les véritables routines qui exécutent les fonctions. Le programmeur utilise lors de l'appel du sous-programme uniquement l'adresse de l'instruction de saut dans la table de saut.

- appel de sous-programme d'une routine définie, la fonction étant transmise au moyen d'un numéro de fonction. Dans la fonction appelée, la véritable fonction sera appelée d'après le numéro de fonction.

- appel de fonction à travers des traps. Un numéro de fonction est en général transmis. La véritable routine exécutant les fonctions est définie dans la table des vecteurs du 68000. Le système d'exploitation de l'Atari ST fait un grand usage de cette forme d'appels de fonction.

INTEGRATION DE ROUTINES ASSEMBLEUR DANS UN LANGAGE EVOLUE

Dans la programmation avec un langage évolué, il y a souvent des problèmes qui ne peuvent être résolus que par des routines assembleur ou tout au moins qui ne peuvent être résolus de façon satisfaisante que par des routines assembleur. La plupart des compilers produisent par exemple pour les programmes graphiques un code qui est trop lent. Pour accélérer de tels programmes, on écrit des routines en assembleur qui se chargeront des parties devant tourner beaucoup plus vite.

Se pose alors le problème de l'intégration de la partie assembleur dans le programme en langage évolué.

Même (ou surtout) avec des langages interprétés "classiques" tels que le BASIC, on résoudra en assembleur des sections de programme pour lesquelles la vitesse nécessaire est très élevée.

Il faut respecter, dans les programmes assembleur, la CONVENTION DE PROCEDURE du langage évolué de façon très "pointilleuse". En voici quelques exemples:

- est-ce que la réception des paramètres est correcte?
- le résultat est-il renvoyé correctement?
- est-ce que la pile a été sauvée?
- est-ce que les registres utilisés (pour autant que cela soit nécessaire d'après la convention) ont été sauvés puis restaurés?
- est-ce que la pile dépasse dans le programme la limite prescrite?
- est-ce que les données sont transférées vers des zones interdites (par exemple vers des mémoires provisoires) ?
- est-ce que le programme assembleur est interrompu par des interruptions?
- est-ce que la zone mémoire traitée par l'assembleur est modifiée par une procédure DMA?

On utilise généralement un linker pour intégrer des routines assembleur dans des programmes compilés. La routine assembleur est appelé par son nom dans le programme en langage évolué. Le linker veille à ce que le saut se fasse effectivement à la bonne adresse.

Dans les programmes BASIC on appelle presque toujours la routine assembleur avec l'instruction "CALL adresse" (ou "SYS adresse", "USR ..."). L'utilisateur doit donc veiller lui-même dans le programme BASIC à ce que la routine assembleur soit chargée à la bonne adresse.

LA PROGRAMMATION ETAPE PAR ETAPE

1) Introduction

2) Exemple de "Conversion décimal/binaire"

Nous allons dans ce chapitre vous initier à la programmation assembleur pratique. Nous supposons connues les notions expliquées dans les chapitres IV et VII du présent ouvrage. A moins que vous n'ayez acquis par ailleurs les connaissances nécessaires sur le processeur 68000 et sur l'emploi de l'assembleur, nous vous conseillons donc, pour une meilleure compréhension de ce chapitre, de ne le lire qu'après avoir lu les chapitres IV et VII.

Dans notre premier exemple, nous allons vous montrer le développement "étape par étape" d'une petite routine de conversion décimal/binaire. Nous avons ici renoncé volontairement à l'emploi de quelconques "astuces" ou fonctions du système d'exploitation. Nous souhaitons en effet vous faire découvrir progressivement les possibilités du 68000 et, d'autre part, nous souhaitons que vous puissiez par la suite "apprécier" toute la puissance du système d'exploitation. Notre expérience nous permet de dire que cette façon de travailler dans l'abord d'un nouveau système informatique conduit à des connaissances reposant sur une base très solide.

Conversion décimal/binaire

Description du problème

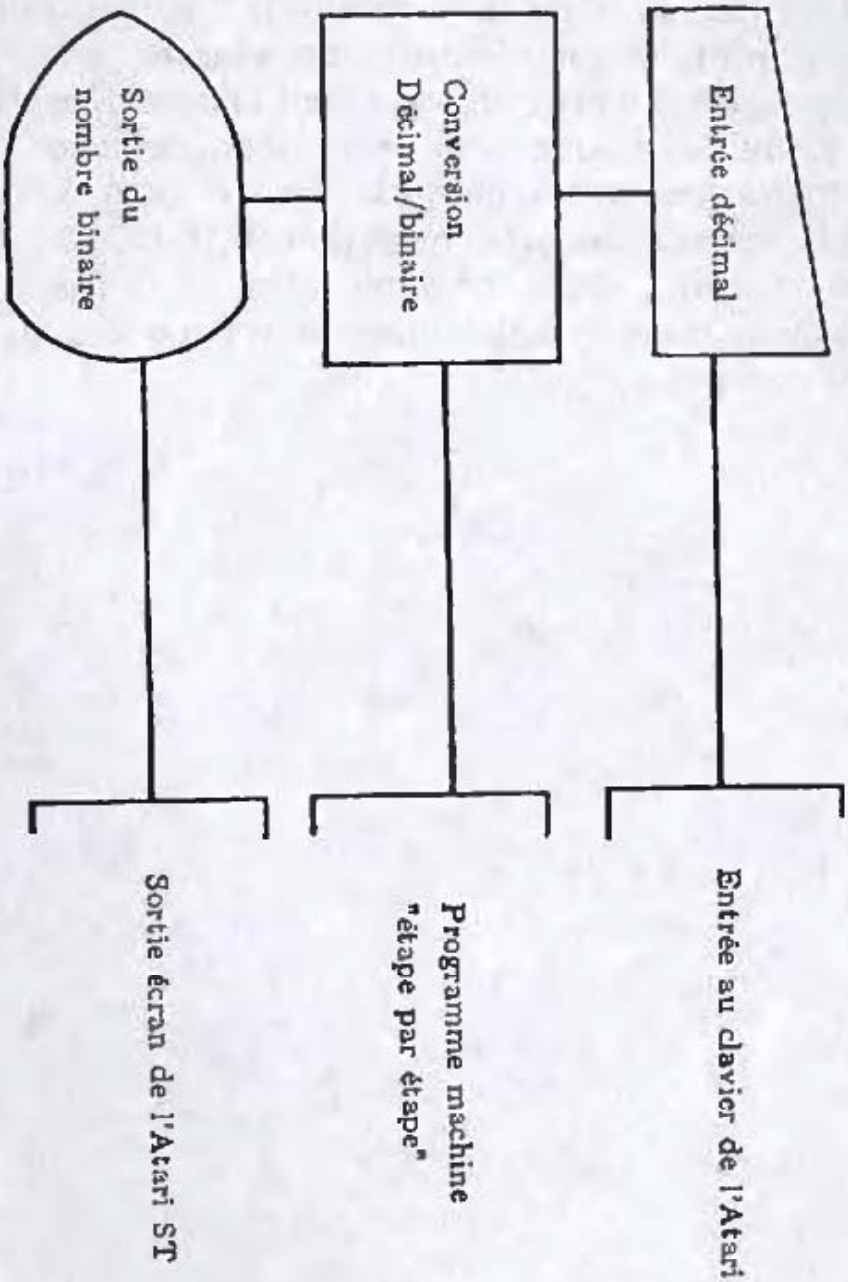
Nous voulons convertir un nombre décimal en un nombre binaire, au moyen d'un programme machine approprié. Comme vous vous en souvenez certainement encore, un nombre décimal est identique à un nombre binaire de même valeur; seule la représentation des deux nombres (la base numérique) est différente.

Nous expliquerons d'abord avec quelles DONNEES le programme travaille. Nous distinguerons entre les données en ENTREE et les données en SORTIE. Dans notre exemple, le nombre décimal sera la donnée en entrée et le nombre binaire la donnée en sortie. Notre programme nécessite encore la définition du TRAITEMENT. Nous nous contenterons pour le moment de l'appeler CONVERSION sans entrer dans le FONCTIONNEMENT de la conversion. Nous avons maintenant toutes les informations nécessaires pour créer un PLAN DE FLUX DE DONNEES.

Plan de flux de données

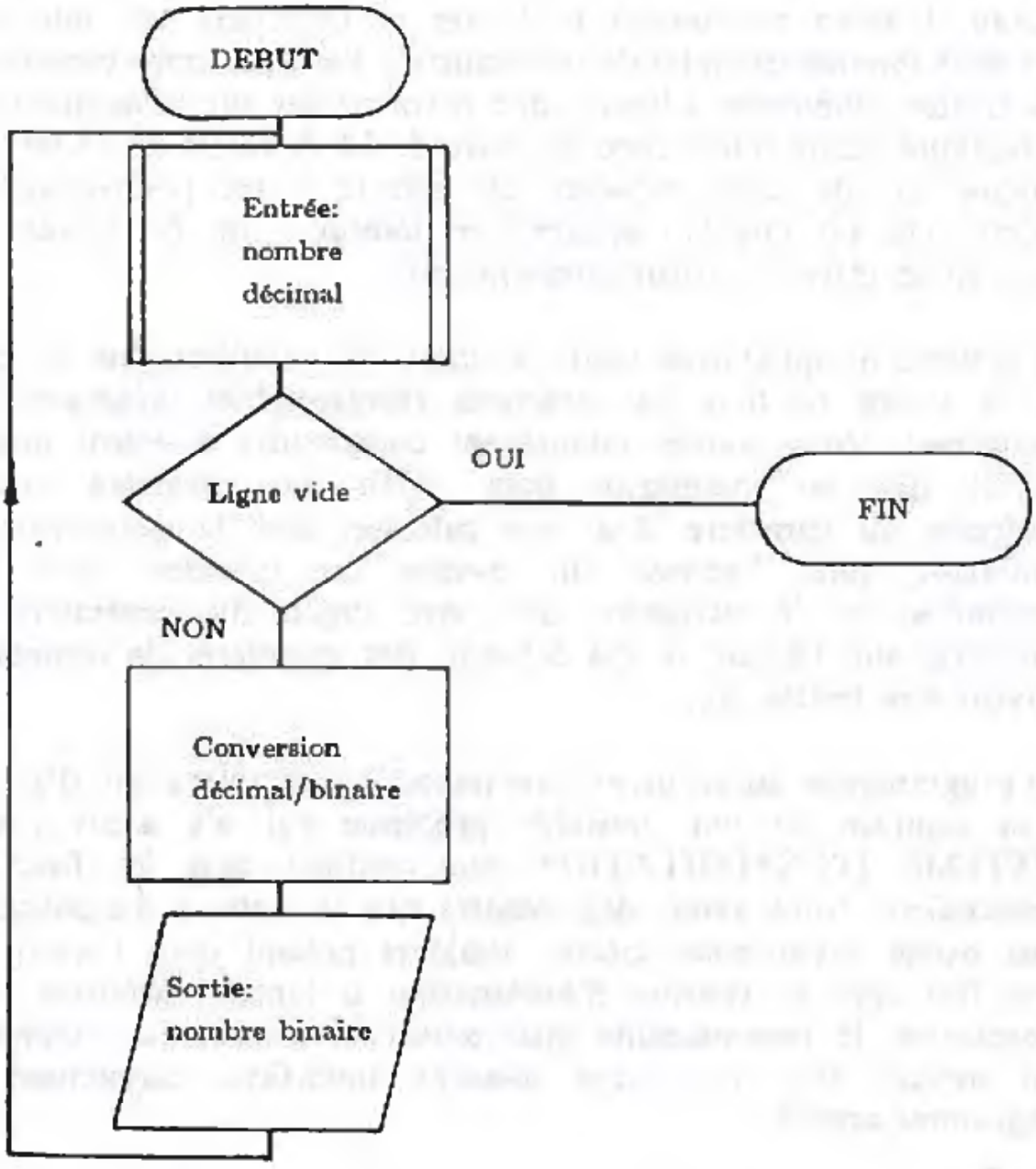
Vous voyez à la page suivante le plan de flux de données de notre programme d'exemple. Le flux de données indique le CHEMIN et le TYPE des données ainsi que CE QUE deviennent les données et QUAND elles le deviennent.

Du fait de la simplicité de notre problème, il n'est pas nécessaire d'expliquer notre plan de flux de données plus en détail. Par souci d'exhaustivité, nous indiquerons encore que les symboles du plan de flux de données se conforment à la norme DIN 66001. Il en va de même pour les symboles de l'organigramme.


Entrée décimal
Conversion
Décimal/binaire
Sortie du
nombre binaire
Entrée au clavier de l'Atari ST
Programme machine
"étape par étape"
Sortie écran de l'Atari ST

Organigramme

L'organigramme doit nous permettre d'indiquer COMMENT les données doivent être traitées. Lors du développement d'un programme il n'y a que deux possibilités principales pour avancer vers la solution finale d'un problème. Le programmeur peut affiner le problème étape par étape jusqu'à ce que soit ainsi réalisée une description précise de toutes les instructions de travail pour le processeur. Cette méthode est appelée programmation TOP-DOWN. Pour cette méthode de travail, vous trouverez dans les pages suivantes quelques organigrammes qui décomposent comme il convient notre programme d'exemple.


DEBUT
Entrée:
nombre
décimal
Ligne vide
OUI
FIN
NON
Conversion
décimal/binaire
Sortie:
nombre binaire

Première étape: la sortie de caractères

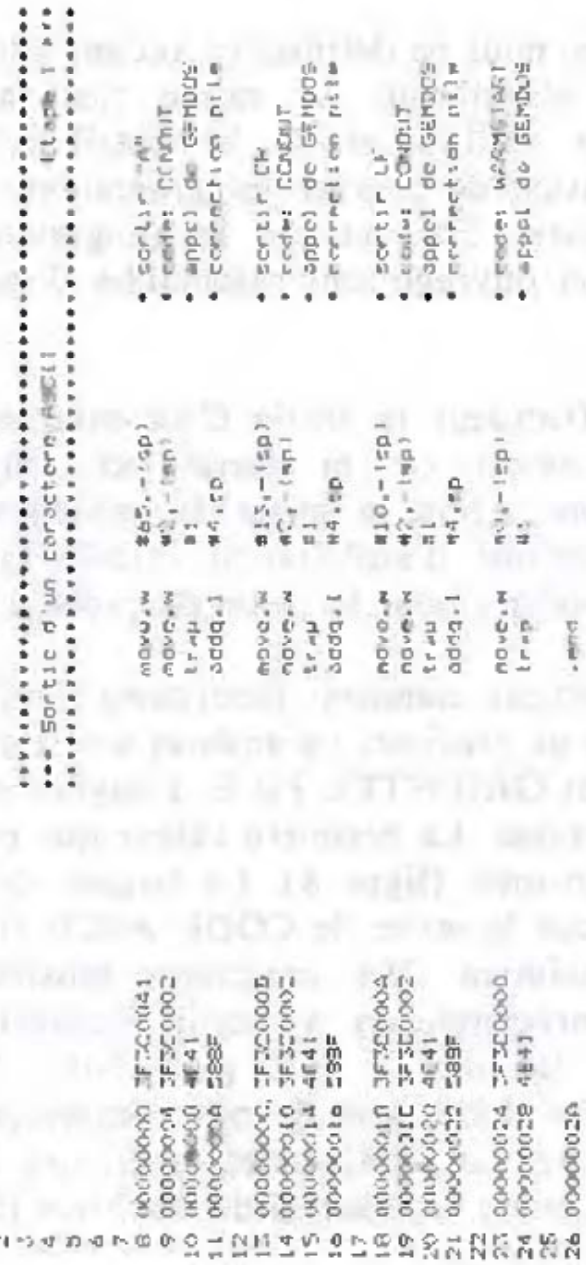
Passons à la première étape, qui consistera à sortir un caractère à l'écran. L'écran ne connaît ni lettres ni caractères tels que ceux que nous sommes capables de reconnaître à l'écran comme constituant des entités cohérentes. L'image que nous voyons sur le moniteur est la réplique exacte d'une zone mémoire de 32 K octets de l'Atari ST. Chaque bit de cette mémoire est affecté à un point-image de l'écran. Un bit mis (1) apparaît en sombre - un bit annulé (0) apparaît en clair (moniteur monochrome).

Le système d'exploitation contient une table qui détermine bit pour bit la forme de tous les caractères représentables (générateur de caractères). Vous pouvez maintenant comprendre aisément tout le travail qui est nécessaire pour sortir un caractère ASCII. L'adresse du caractère doit être calculée dans le générateur de caractères, puis l'adresse du modèle du caractère doit être déterminée et le caractère doit être copié du générateur de caractères sur l'écran; le cas échéant, des caractères de commande doivent être traités, etc.

Un programmeur aurait un énorme travail à accomplir avant d'arriver à la solution de son véritable problème s'il n'y avait pas le SYSTEME D'EXPLOITATION qui contient déjà les fonctions élémentaires. Nous avons déjà montré que le système d'exploitation n'est qu'un programme spécial, toujours présent dans l'Atari ST. Une fois que le système d'exploitation a lancé l'exécution d'un programme, la responsabilité pour toutes les actions du programme (du moins, dans une large mesure) appartient maintenant au programme appelé.

Pour le programmeur en langage machine se pose donc le problème de savoir comment son programme peut volontairement rendre le contrôle au système d'exploitation, et comment il pourra ensuite le

reprendre, pour faire exécuter certaines tâches (par exemple l'entrée et la sortie de caractères) par le système d'exploitation, à travers ce qu'on appelle les routines du système d'exploitation. Les règles que le programmeur doit observer dans l'utilisation des différentes routines du système d'exploitation sont définies par le constructeur (conventions de procédure).



```
                    Etape 1 ere
*** Sortie d'un caractere ASCII
*****************************************

* sortir "A"
* code: CONOUT
* appel de GEMDOS
* correction pile
move.w  #65,-(sp)
move.w  #2,-(sp)
trap    #1
addq.l  #4,sp

* sortir CR
* code: CONOUT
* appel de GEMDOS
* correction pile
move.w  #13,-(sp)
move.w  #2,-(sp)
trap    #1
addq.l  #4,sp

* sortir LF
* code: CONOUT
* appel de GEMDOS
* correction pile
move.w  #10,-(sp)
move.w  #2,-(sp)
trap    #1
addq.l  #4,sp

* code: W/SYSTART
* appel de GEMDOS
move.w  #0,-(sp)
trap    #1
.end
```

Nous vous présentons ici un premier petit programme machine. Il montre la sortie d'un caractère ASCII sur l'écran et l'utilisation d'une routine du système d'exploitation. Nous utilisons systématiquement l'interface GEMDOS de l'Atari ST. D'après nos informations (documentation fournie aux développeurs), Atari ne garantit une totale compatibilité avec les générations ultérieures de machines que pour l'interface GEMDOS.

Dans nos exemples, nous ne définissons aucune adresse de début pour nos programmes assembleur. La raison tient aux possibilités de l'assembleur/linker utilisé et à la possibilité offerte par le système d'exploitation de charger les programmes de façon relative avant de les exécuter. En tout cas, les programmes que nous vous présentons dans cet ouvrage sont exécutables directement sous GEM ou TOS.

Les lignes 7-9 effectuent la sortie d'un caractère (A ASCII) sur l'écran. L'emplacement de la sortie est toujours la position actuelle du curseur. Après la sortie du caractère (ou caractère de commande), le système d'exploitation calcule la nouvelle position du curseur et la prépare pour la sortie du prochain caractère.

Examinons précisément comment fonctionne l'utilisation du système d'exploitation. Un ou plusieurs paramètres sont transmis à GEMDOS. La transmission est ORIENTEE PILE. Essayons de suivre un peu le déroulement des choses. La première valeur que nous fournissons est le caractère à imprimer (ligne 8). La largeur de traitement est de 1 mot. Comme vous le savez, le CODE ASCII (ici ASCII ATARI) comporte au maximum 256 caractères possibles. Pour pouvoir représenter ces caractères, on a besoin exactement d'1 octet. La transmission sur la pile se fait par contre toujours par mot entier, pour que les données de la pile commencent toujours à une adresse d'octet paire. La moitié-octet supérieure du mot ASCII n'a aucune fonction dans le type actuel de machines (520 ST).

Pour maintenir toutefois une compatibilité avec d'éventuels nouveaux jeux de caractères, nous vous conseillons d'ignorer la partie de plus grande valeur et de toujours la remplir de zéros binaires.

A la ligne 9, un autre paramètre est placé sur la pile. Il s'agit d'un numéro de fonction au moyen duquel le système d'exploitation saura ce qu'il doit faire des données sur la pile. Le numéro de fonction définit également de façon implicite le nombre de paramètres. Il est ainsi assuré que le système d'exploitation traite toujours tous les paramètres. L'ensemble paramètres et numéro de fonction est appelé "BLOC DE PARAMETRES". Dans notre exemple, le bloc de paramètres se compose de deux mots (caractère et code de fonction de la sortie de caractères).

L'instruction TRAP de la ligne 10 appelle le système d'exploitation (GEMDOS). Nous ne nous intéresserons pas ici plus précisément au travail qui est alors pratiquement effectué. Nous souhaitons simplement vous montrer avec le schéma ci-dessous ce qui doit se passer sur la pile lors de la préparation de l'appel du système d'exploitation et après. Notez que le système d'exploitation travaille toujours, de façon interne, avec la pile superviseur (puisque le 68000 se trouve après un TRAP, -donc une exception-, en mode superviseur) et qu'il va chercher les données sur la pile qui était activée lors de l'appel de la fonction. Ce sera en général la pile utilisateur car les programmes utilisateur sont exécutés en mode utilisateur.

Après l'appel d'une fonction du système d'exploitation, le programmeur doit veiller à ce que tous les paramètres soient bien retirés de la pile. Cela peut se faire par exemple avec l'instruction "ADDQ.L" sur le pointeur de pile.

L'instruction TRAP travaille ici comme un appel de sous-programme. L'adresse de ce sous-programme est définie par le numéro TRAP.

Dans la table des vecteurs du système 68000, le processeur trouvera l'adresse de la routine du système d'exploitation qui doit être appelée par l'instruction trap. Une fois la fonction du système d'exploitation exécutée, le programme machine se poursuivra à la prochaine instruction (ligne 13).

```
Adresse:
n+2        ancien          ancien          ancien
n          ancien  <-SP    ancien          ancien
n-2          .             $0041   <-SP    $0041
n-4          .               .             $0002   <-SP
n-6          .               .               .

Situation de départ     MOVE.W #$41,-(sp)   MOVE.W #2,-(sp)

Adresse:
n+2        ancien          ancien
n          ancien          ancien  <-SP
n-2        $0041           $0041
n-4        $0002   ->SP    $0002
n-6          .               .

           TRAP #1         ADDQ.L #4,sp
```

Les lignes 13 à 21 constituent une répétition de la fonction que nous venons d'évoquer. Mais ici, aucun caractère n'est sorti. Il s'agit des caractères de commande Carriage Return (CR) et Line Feed (LF). Ces appellations rappellent bien sûr l'époque ou les télex étaient utilisés pour dialoguer avec un ordinateur. CR (retour de chariot) fixe dans notre exemple le curseur (emplacement d'écriture) sur la première colonne de la ligne d'écran actuelle.

LF effectue un "passage à la ligne" suivante. Le curseur est donc placé à l'écran sur la ligne suivante, en restant toujours dans sa colonne actuelle. Si le curseur se trouvait dans la dernière ligne, un "scrolling" serait exécuté. On désigne ainsi le fait de décaler d'une ligne vers le haut toutes les lignes de l'écran. La ligne supérieure de l'écran disparaît et une ligne vide est insérée dans l'emplacement de la dernière ligne de l'écran. L'écran se comporte ainsi exactement comme une feuille de papier sur une machine à écrire.

Les instructions des lignes 23 et 24 de notre exemple rendent le contrôle au système d'exploitation. Cet appel du système d'exploitation ne nécessite pas d'autre paramètre que le numéro de fonction.

Dans ce petit exemple, nous vous montrons déjà trois principes importants de la programmation en langage machine. Nous vous avons expliqué le principe des appels du système d'exploitation, la sortie de caractères et le retour au système d'exploitation. Pour la solution de notre programme d'exemple, il nous faut encore comprendre comment le système d'exploitation peut nous aider à transmettre des caractères au programme à travers le clavier (entrée de caractères). Examinons pour cela un autre programme d'exemple. Notez par ailleurs que la méthode utilisée ici, consistant à résoudre un problème étape par étape, est appelée programmation "button up".

```
1
2
3
4
5
6
7
8  00000000 3F3C0001          move.w  #1,-(sp)        * code: CONIN
9  00000004 4E41              trap    #1              * appel de GEMDOS
10 00000006 548F              addq.l  #2,sp           * correction pile
11
12 00000008 3E00              move.w  d0,d7           * sauver caractere
13
14 0000000A 3F3C000D          move.w  #13,-(sp)       * sortir CR
15 0000000E 3F3C0002          move.w  #2,-(sp)        * code: CONOUT
16 00000012 4E41              trap    #1              * appel de GEMDOS
17 00000014 588F              addq.l  #4,sp           * correction pile
18
19 00000016 3F3C000A          move.w  #10,-(sp)       * sortir LF
20 0000001A 3F3C0002          move.w  #2,-(sp)        * code: CONOUT
21 0000001E 4E41              trap    #1              * appel de GEMDOS
22 00000020 588F              addq.l  #4,sp           * correction pile
23
24 00000022 CE7C00FF          and.w   #$ff,d7         * masquer caracteres
25
26 00000026 3F07              move.w  d7,-(sp)        * sortir caractere
27 00000028 3F3C0002          move.w  #2,-(sp)        * code: CONOUT
28 0000002C 4E41              trap    #1              * appel de GEMDOS
29 0000002E 588F              addq.l  #4,sp           * correction pile
30
31 00000030 3F3C0000          move.w  #0,-(sp)        * code: WARMSTART
32 00000034 4E41              trap    #1              * appel de GEMDOS
33
34 00000036                   .end
```

****
*** Entree d'un caractere ASCII   Etape 2 ***
****

Deuxième étape, l'entrée de caractères

L'exemple ci-dessus vous montre l'entrée d'un caractère au clavier. Pour contrôler si l'entrée fonctionne, nous faisons immédiatement sortir le caractère entré.

Dans les lignes 8-9, la fonction du système d'exploitation CONIN (dont le code de fonction est 1) est appelée. Cette fonction n'a pas d'autre paramètre. Après appel de la routine, le système d'exploitation attend, jusqu'à ce qu'une touche du clavier soit actionnée. Lorsqu'une touche du clavier est appuyée, le code ASCII correspondant est déterminé et transmis au programme d'appel à travers le registre D0. Pour la suite du travail, nous corrigeons la pile (ligne 10) et nous copions avec l'instruction MOVE le contenu du registre D0 dans le registre D7 (ligne 12). Les lignes 14 à 22 vous sont déjà connues depuis notre premier exemple. Elles veillent à ce que le curseur soit fixé sur la première colonne d'une nouvelle ligne.

Lors de l'appel de la routine de sortie de caractères, le contenu du registre D0 est modifié. C'est pourquoi nous avons effectué à la ligne 12 une copie du caractère entré dans le registre D7. Avant de sortir le caractère entré en guise de démonstration, nous nous assurons que l'octet de plus grande valeur du caractère soit fixé sur zéro en binaire. Nous utilisons à cet effet l'opération AND (ligne 24). Avec cette fonction logique, tous les bits de la constante $FF sont reliés aux bits correspondants du registre D7.

31	30	29	. . .	9	8	7	6	5	4	3	2	1	0	31 T	
0	0	0	. . .	0	0	1	1	1	1	1	1	1	1		Constante
x	x	x	. . .	x	x	A	A	A	A	A	A	A	A		Caractère
0	0	0	. . .	0	0	A	A	A	A	A	A	A	A		Résultat

Le tableau ci-dessus montre encore une fois clairement ce qui se passe avec l'opération logique AND. Les bits qui sont dans la constante fixés à zéro seront toujours à zéro dans le résultat. Les bits qui sont fixés sur un dans la constante seront dans le résultat identiques au caractère originel. Cet emploi de la fonction AND s'appelle aussi MASQUAGE.

C'est maintenant le lieu d'expliquer quelles conventions nous utilisons dans nos exemples. Nous avons expliqué au chapitre précédent les possibilités existantes de conventions de procédure. Dans le cadre de nos exemples, nous ne définirons qu'une convention simple de registres. Tous les paramètres sont transmis à travers les registres de données ou d'adresse. Les registres seront pour cela utilisés dans l'ordre décroissant de leur première utilisation (D7, D6,D5, ... D3 et A5) à moins qu'ils n'aient été sauvés par le programmeur lui-même puis restaurés après utilisation. Toutes collisions avec les sous-programmes sont ainsi évitées. Nous souhaitons cependant souligner qu'il s'agit là d'une convention très élémentaire qu'il est déconseillé d'utiliser également pour des programmes d'envergure.

Revenons à notre second exemple: les lignes 26-29 sortent à l'écran le caractère masqué en appelant la fonction CONOUT (sortie console) du système d'exploitation. Les lignes 31 et 32 terminent le programme comme nous l'avons déjà vu.

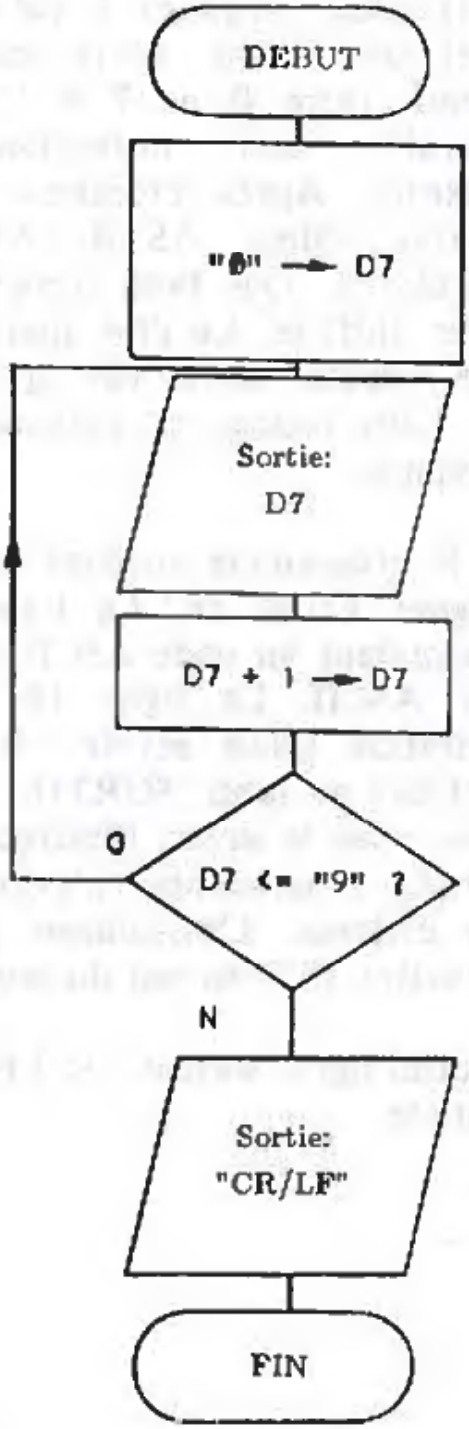
Ces deux exemples nous fournissent déjà les informations essentielles sur le système d'exploitation pour nous permettre de réaliser notre but, la conversion décimal/binaire. Nous ne traiterons pas d'autres fonctions du système d'exploitation dans ce chapitre.

L'exemple suivant doit nous permettre de franchir une nouvelle étape vers la solution de notre problème. Nous souhaitons vous montrer comment travailler avec les chaînes de caractères.

```
1
2
3  ********************************************************
4  *** Sortie d'une ligne ASCII:          Etape 3 ***
5  ********************************************************
6
7
8  00000000 3E3C0030          move.w  #$30,d7     * zero ASCII
9
10 00000004 3F07       sortie: move.w  d7,-(sp)    * caractere a sortir
11 00000006 3F3C0002           move.w  #2,-(sp)    * code: CONOUT
12 0000000A 4E41               trap    #1          * appel de GEMDOS
13 0000000C 588F               addq.l  #4,sp       * correction pile
14
15 0000000E 5207               addq.b  #1,d7       * nouveau caractere ASCII
16
17 00000010 0C070039           cmpi.b  #$39,d7     * = "9" ASCII ?
18 00000014 63EE               bls     sortie      * OUI: caractere suivant
19
20 00000016 3F3C000D           move.w  #13,-(sp)   * sortir CR
21 0000001A 3F3C0002           move.w  #2,-(sp)    * code: CONOUT
22 0000001E 4E41               trap    #1          * appel de GEMDOS
23 00000020 588F               addq.l  #4,sp       * correction pile
24
25 00000022 3F3C000A           move.w  #10,-(sp)   * sortir LF
26 00000026 3F3C0002           move.w  #2,-(sp)    * code: CONOUT
27 0000002A 4E41               trap    #1          * appel de GEMDOS
28 0000002C 588F               addq.l  #4,sp       * correction pile
29
30 0000002E 3F3C0000           move.w  #0,-(sp)    * code: WARMSTART
31 00000032 4E41               trap    #1          * appel de GEMDOS
32
33 00000034                    .end
```

Troisième étape, le travail en boucle

Dans notre troisième exemple, nous sortons plusieurs caractères ASCII au moyen d'une boucle. Nous découvrons ainsi la structure d'une boucle à l'intérieur d'un programme machine.



Nous voulons sortir les chiffres 0 à 9 sous la forme d'une chaîne de caractères. Nous créons pour cela d'abord en ligne 8 la constante $30 (zéro en ASCII) dans le registre de données D7. En lignes 10-13, le caractère en D7 (encore zéro) est sorti.

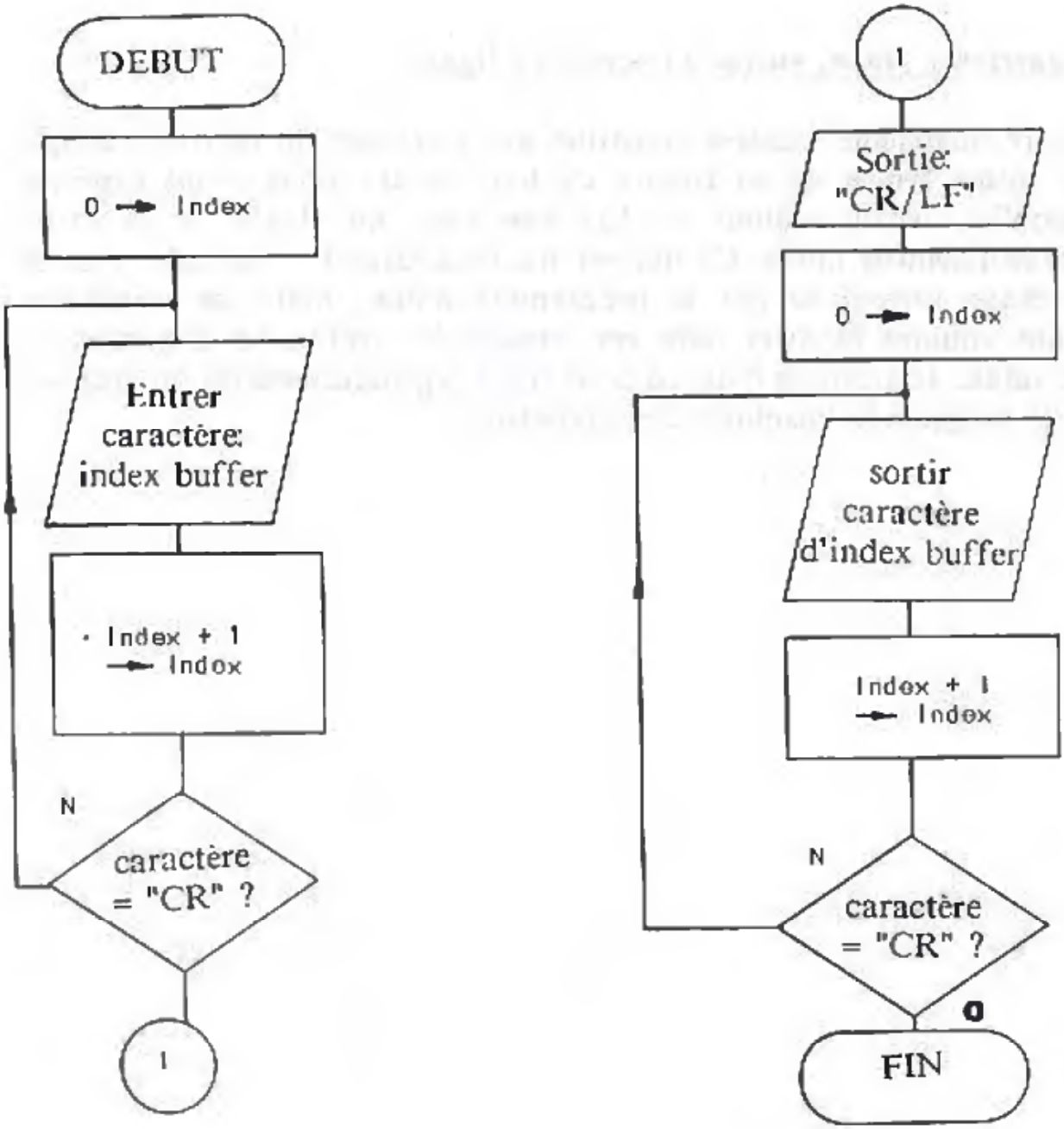
Il nous faut maintenant produire les caractères suivants (un à neuf ASCII). Nous utilisons à cet effet l'instruction ADDQ (addition rapide) du 68000. Cette instruction permet d'additionner une constante entre 0 et 7 à l'objet indiqué. Cette instruction est comparable aux instructions d'incrémentation des autres processeurs. Après exécution de la ligne 15, D7 contient la prochaine valeur ASCII. Avant de sortir les chiffres ainsi constitués, il nous faut encore contrôler si nous avons déjà sorti tous les chiffres. Le plus simple est de formuler une condition de boucle "répéter sortie tant que le chiffre est inférieur ou égal à neuf". Cette logique est également présentée dans l'organigramme de cet exemple.

Dans le programme machine, la condition de boucle est réalisée par les lignes 17 et 18. La ligne 17 effectue la comparaison, $39 correspondant au code ASCII du neuf et D7 contenant la nouvelle valeur ASCII. La ligne 18 teste le résultat de l'opération de comparaison (pour savoir s'il est inférieur ou égal) et saute le cas échéant au label "SORTIE" où un caractère sera à nouveau sorti. Comme vous le savez, l'instruction BLS est une instruction de saut relatif. Le programmeur n'a toutefois pas à se préoccuper du calcul de la distance. L'assembleur calcule lui-même le saut relatif de l'instruction BLS au but du saut "SORTIE".

Les autres lignes sortent CR/LF et terminent le programme, comme à l'habitude.

Quatrième étape, entrée et sortie de lignes

Notre quatrième exemple constitue une extension du dernier exemple en même temps qu'un résumé de tout ce que nous avons expliqué jusqu'ici. Nous voulons ici lire une ligne au clavier et la sortir immédiatement après. Ce qui est nouveau dans cet exemple, c'est le stockage provisoire par le programme d'une chaîne de caractères. Nous voulons montrer dans cet exemple la création et la gestion de variables. Examinons d'abord pour cela l'organigramme du programme et le programme machine correspondant.


DEBUT
0 → Index
Entrer caractère: index buffer
Index + 1 → Index
N
caractère = "CR" ?
1
1
Sortie: "CR/LF"
0 → Index
sortir caractère d'index buffer
Index + 1 → Index
N
caractère = "CR" ?
0
FIN

```
                ********************************************************
                *** Entree et sortie d'une ligne ASCII      Etape 4 ***
                ********************************************************

                        .
00000000 2A7C0000004C           movea.l  #ligne,a5  * creer pointeur

00000006 3F3C0001       entree: move.w   #1,-(sp)   * code: CONIN
0000000A 4E41                   trap     #1         * appel de GEMDOS
0000000C 548F                   addq.l   #2,sp      * correction pile

0000000E 1AC0                   move.b   d0,(a5)+   * sauvegarder caractere

00000010 0C00000D               cmpi.b   #13,d0     * caractere etait un CR ?
00000014 66F0                   bne      entree     * NON: caractere suivant

00000016 3F3C000D               move.w   #13,-(sp)  * sortir CR
0000001A 3F3C0002               move.w   #2,-(sp)   * code: CONOUT
0000001E 4E41                   trap     #1         * appel de GEMDOS
00000020 588F                   addq.l   #4,sp      * correction pile

00000022 3F3C000A               move.w   #10,-(sp)  * sortir LF
00000026 3F3C0002               move.w   #2,-(sp)   * code: CONOUT
0000002A 4E41                   trap     #1         * appel de GEMDOS
0000002C 588F                   addq.l   #4,sp      * correction pile

0000002E 2A7C0000004C           movea.l  #ligne,a5  * restaurer pointeur

00000034 1E1D           sortie: move.b   (a5)+,d7   * 1 caractere du buffer

00000036 3F07                   move.w   d7,-(sp)   * sortir
00000038 3F3C0002               move.w   #2,-(sp)   * code: CONOUT
0000003C 4E41                   trap     #1         * appel de GEMDOS
0000003E 588F                   addq.l   #4,sp      * correction pile

00000040 0C00000D               cmpi.b   #13,d0     * caractere etait un CR ?
00000044 66EE                   bne      sortie     * NON: continuer sortie

00000046 3F3C0000               move.w   #0,-(sp)   * code: WARMSTART
0000004A 4E41                   trap     #1         * appel de GEMDOS

0000004C                ligne:  .ds.b    80         * buffer 80 caracteres

                        .end
0000009C
```

Examinons cet exemple en commençant par la ligne 44. Une zone mémoire de 80 octets est ici réservée avec la directive DS.B. Dans cette zone mémoire, nous stockerons provisoirement tous les caractères entrés avant de les sortir à nouveau. En même temps, la ligne 44 affecte au symbole "LIGNE" l'adresse de début de la mémoire provisoire. Une telle mémoire provisoire est également appelée BUFFER.

La ligne 8 crée un POINTEUR sur le début du buffer. L'adresse du buffer est à cet effet chargée dans le registre d'adresse A5. D'autres instructions que nous expliquerons par la suite permettront d'accéder aux divers éléments du buffer grâce au pointeur figurant dans le registre d'adresse.

Intéressons-nous cependant d'abord à l'entrée de caractères au clavier. Un caractère est lu (lignes 10 à 12) au moyen déjà connu de la fonction CONIN (entrée console) du système d'exploitation. L'instruction MOVE en ligne 14 transfère le caractère lu dans le buffer. Notez le mode d'adressage utilisé: "registre d'adresse indirect avec postincrément". Le premier caractère entré est ainsi placé à l'adresse mémoire indiquée indirectement par le contenu du registre d'adresse A5. Après ce transfert, le contenu du registre d'adresse est augmenté de 1 puisque nous avons choisi le format "octet" comme largeur de traitement. Le contenu du registre d'adresse indique ainsi, après transfert du premier caractère, le prochain emplacement libre dans le buffer de caractères.

Comme nous voulons entrer plusieurs caractères, il nous faut à nouveau créer une boucle pour l'entrée au clavier. Notez cependant qu'il n'est pas nécessaire d'effectuer de sortie de l'entrée au clavier à l'intérieur de cette boucle pour que l'entrée soit immédiatement visible. La fonction du système d'exploitation utilisée sort automatiquement à l'écran un caractère correspondant à la touche appuyée. Il nous faut encore trouver un critère approprié pour fermer notre boucle.

Une entrée est en général terminée en appuyant sur la touche RETURN. Cette touche fournit lorsqu'elle est appuyée le code ASCII non-représentable CR que nous avons déjà rencontré pour la sortie. Cela correspond d'ailleurs à la fonction véritable de cette touche. Dans notre exemple, c'est donc cette touche qui sera interprétée comme fin de la ligne d'entrée. Nous avons donc programmé en lignes 16 et 17 la condition d'interruption de la boucle avec cette touche. Tant donc que le caractère entré n'est pas la touche RETURN (code 13), le caractère entré est placé dans le buffer et on attend une nouvelle entrée.

Le lecteur attentif aura certainement remarqué que nous ne demandons pas expréssément si le buffer est déjà plein. Nous avons volontairement renoncé à cette interrogation pour ne pas compliquer le programme. Pour éviter des situations d'erreur produites par un débordement du buffer, nous avons simplement défini le buffer plus grand que nécessaire. Si vous disposez d'un assembleur, vous pouvez essayer de résoudre ce problème à titre d'exercice.

Les lignes 19 à 27 se chargent de la sortie de CR/LF pour que le curseur soit fixé sur le début de la ligne suivante avant que le contenu du buffer ne soit sorti.

Avant que nous ne puissions sortir à nouveau le buffer caractère par caractère, nous devons replacer (en ligne 29) le pointeur du buffer sur le premier emplacement. Nous utilisons à nouveau une instruction MOVE qui charge dans le registre d'adresse A5 l'adresse du buffer.

Les instructions des lignes 31 à 36 vont chercher un caractère dans le buffer et le représentent sur l'écran grâce à la fonction CONOUT (sortie console) du système d'exploitation.

Les lignes 31 et 33 méritent une attention particulière. L'instruction MOVE de la ligne 31 transfère un caractère, indiqué

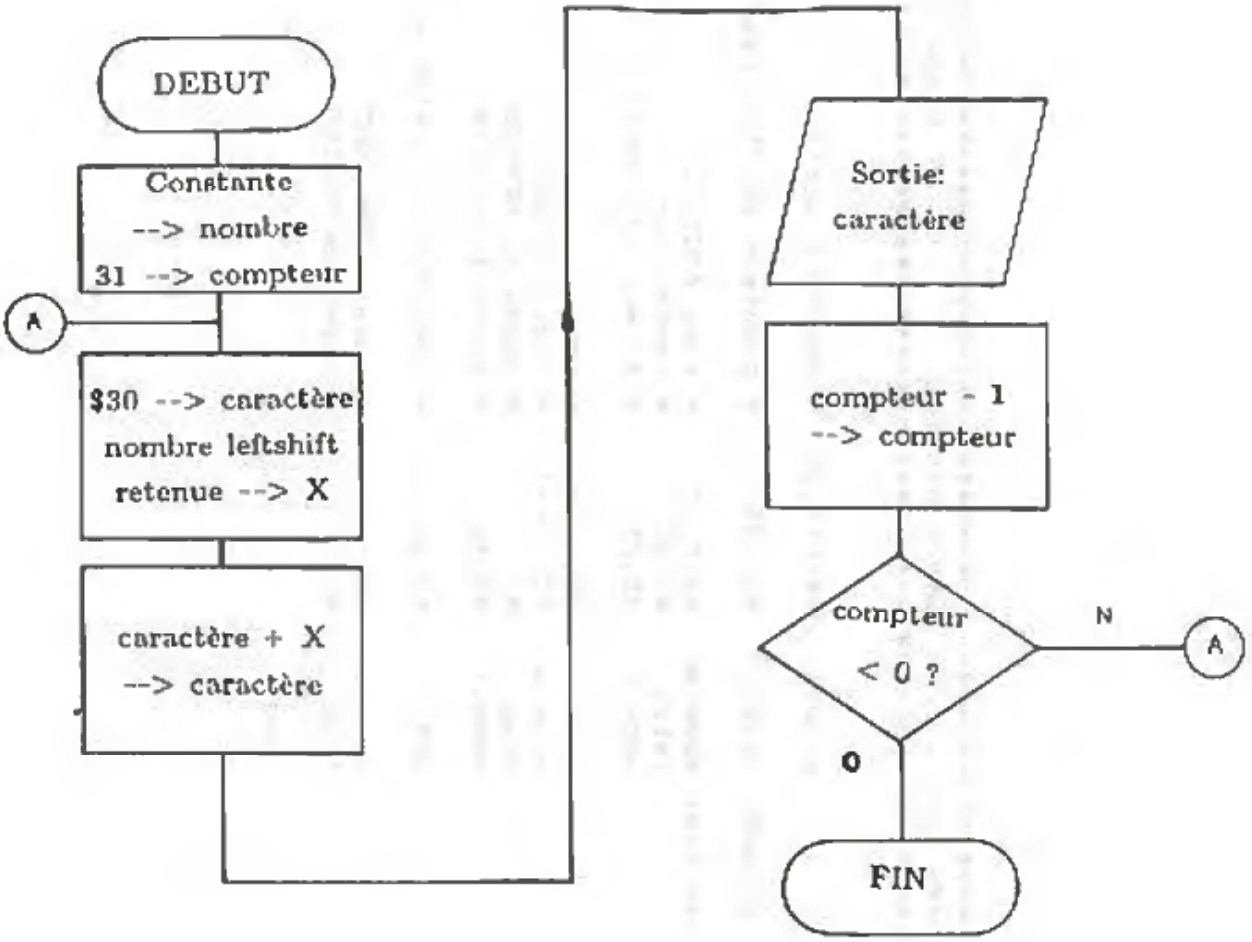
indirectement par l'adresse dans le registre d'adresse A5, du buffer dans le registre de données D7. L'adresse dans le registre d'adresse est immédiatement augmentée de 1 pour indiquer le prochain caractère du buffer. L'instruction MOVE de la ligne 33 pousse le caractère sur la pile pour qu'il puisse ensuite (lignes 34 et 36) être sorti par un appel du système d'exploitation. Il n'est pas possible de réaliser, avec une seule instruction, la transmission du caractère du buffer à la pile. Comme vous vous en souvenez certainement, seuls des mots ou longs mots peuvent être placés sur la pile. Mais, dans notre buffer, nous lisons les caractères octet par octet. Comme le 68000 n'autorise la sélection de la largeur de traitement que simultanément pour les adresses source et objet, le programmeur doit gérer lui-même le traitement de largeurs de données différentes.

La sortie de la mémoire buffer est à nouveau programmée sous forme d'une boucle. Le critère de fin de boucle est à nouveau le caractère CR qui a été stocké lui aussi dans le buffer lors de l'entrée (lignes 38 et 39). Les instructions suivantes terminent le programme comme à l'habitude.

Cinquième étape, la sortie binaire

Notre prochain exemple représente une solution partielle de notre problème de programmation. Nous voulons convertir un nombre binaire en une chaîne de caractères ASCII. Nous partons du principe que le nombre à sortir se trouve dans le registre D7. La sortie de ce nombre s'effectuera bit par bit avec des zéros et des uns ASCII. Examinons l'organigramme pour ce problème ainsi que le programme correspondant en langage machine.

```
2
3                ********************************************
4                *** Sortie d'un nombre binaire  Etape 5 ***
5                ********************************************
6
7
8  00000000 2E3C0000F0F0         move.l  #$f0f0,d7   * nombre a sortir
9
10 00000006 7C1F         binout: move.l  #31,d6      * compteur de chiffres
11
12 00000008 3A3C0018     sortie: move.w  #$18,d5     * zero ASCII/2
13 0000000C E38F                 lsl.l   #1,d7       * isoler bit
14 0000000E DA05                 addx.b  d5,d5       * former 0/1 ASCII
15
16 00000010 3F05                 move.w  d5,-(sp)    * sortir
17 00000012 3F3C0002             move.w  #2,-(sp)    * code: CONOUT
18 00000016 4E41                 trap    #1          * appel de GEMDOS
19 00000018 588F                 addq.l  #4,sp       * correction pile
20
21 0000001A 51CEFFEC             dbf     d6,sortie   * compteur-1, tester si -1
22
23 0000001E 3F3C0000             move.w  #0,-(sp)    * code: WARMSTART
24 00000022 4E41                 trap    #1          * appel de GEMDOS
25
26 00000024                      .end
```


DEBUT
Constante
--> nombre
31 --> compteur
A
$30 --> caractère
nombre leftshift
retenue --> X
caractère + X
--> caractère
Sortie:
caractère
compteur - 1
--> compteur
compteur
< 0 ?
N
A
O
FIN

A la ligne 8, nous affectons au registre de données D7 la valeur qu'il doit représenter en binaire. Nous devons maintenant définir encore un compteur de chiffres pour pouvoir traiter tous les bits dans une boucle. Cela est nécessaire car nous ne pouvons déduire d'un nombre binaire quelconque quand il a été entièrement sorti. Il est usuel en représentation binaire d'afficher également les zéros initiaux. C'est pourquoi nous formons notre compteur de chiffres avec le registre de données D6 et la constante 31, à la ligne 10 du programme.

La constante 31 est formée pour que puisse être utilisée une forme particulière de la structure de boucle. Il semble naturel au premier abord de compter avec un compteur de 0 à 32. Il existe cependant une instruction machine qui permet de décrémenter un registre de données (dx = dx-1) et de comparer le résultat. Tant que le résultat n'est pas égal à -1, un saut au début de la boucle est déclenché (lignes 12 et 21). Une autre particularité de l'instruction DBcc est l'exécution conditionnelle de l'instruction. Avant exécution de l'instruction DBcc, un code de condition (comme pour l'instruction conditionnelle de saut Bcc) est testé. L'instruction DBcc n'est exécutée que tant que la condition définie par cc n'est PAS remplie. Comme nous n'avons pas recours à cette possibilité dans notre exemple, nous devons indiquer "F" (FALSE=jamais) comme code de condition.

A l'intérieur de la boucle ainsi créée, qui sera parcourue exactement 32 fois, nous allons maintenant sortir chaque fois un bit. La sortie du caractère ASCII en lignes 16-19 s'effectuera de la façon habituelle, par un appel du système d'exploitation. Le caractère ASCII (0 ou 1) sorti dépend des différents bits du registre de données D7. Le caractère ASCII est produit aux lignes 12-14 en suivant pour le calcul la procédure que nous allons expliquer maintenant.

La constante $18 est formée dans le registre de données D5. Cela correspond à la moitié du zéro ASCII ($30/2=$18). Les instructions suivantes expliquent pourquoi nous avons choisi cette constante. Pour sortir chaque fois le bit le plus élevé n'ayant pas encore été sorti, nous utilisons l'instruction LSL. Cette instruction permet de décaler les bits à l'intérieur du registre, d'un nombre d'emplacements donné. A un bout du registre, des bits zéros sont insérés et à l'autre bout les bits en trop sont expulsés. Le dernier bit expulsé est, ce faisant, toujours stocké provisoirement dans les flags X et C. Comme nous voulons sortir dans notre boucle toujours le prochain bit le plus élevé, nous avons choisi la direction de décalage gauche. Notre instruction indique enfin également que nous voulons effectuer chaque fois un décalage d'un emplacement.

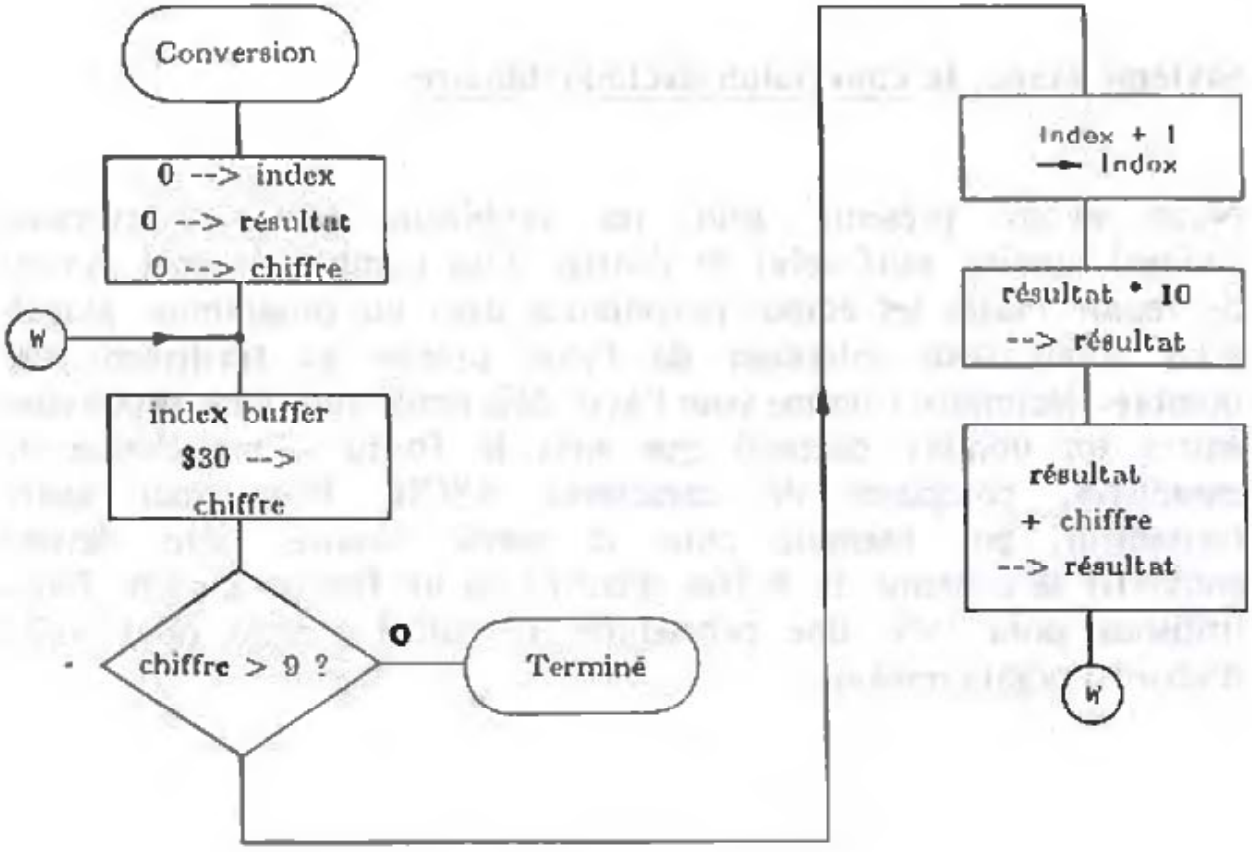
Après exécution de l'instruction, notre bit (0 ou 1) se trouve donc notamment dans le registre X du processeur. Nous utiliserons ce fait pour former notre caractère ASCII. Le code ASCII pour zéro est $30 et $31 pour un. Nous pourrions donc former le code ASCII en additionnant le flag X à la constante $31. Il n'y a malheureusement pas d'instruction exécutant explicitement cette opération. Il existe cependant une instruction qui additionne une source, un objet et le flag X. Le flag X est dans ce cas utilisé comme bit de retenue pour les additions.

Nous devrions donc normalement former en ligne 12 la constante $30 et exécuter en ligne 14, avec l'instruction ADDX, l'addition de $30 (dans le registre D5), de la constante $00 et du contenu du flag X. Il nous faudrait pour cela produire encore la constante $00 dans l'instruction d'addition. Il est cependant beaucoup plus pratique d'indiquer le même registre comme source et objet. Ainsi serait effectuée une addition de $18, d'encore une fois $18 (=$30) et du flag X. L'avantage est ici essentiellement d'arriver à un programme un peu plus court et une rapidité d'exécution un peu supérieure. Bien sûr, si on n'a pas besoin d'une rapidité maximum d'exécution,

on peut bien résoudre ce problème autrement, d'une façon qui serait plus claire mais aussi un peu plus longue.

Sixième étape, la conversion décimal/binaire

Nous avons présenté tous les problèmes d'une conversion décimal/binaire, sauf celui de l'entrée d'un nombre décimal. Avant de réunir toutes les étapes parcellaires dans un programme global, nous allons nous intéresser de façon précise au traitement des nombres décimaux. Comme vous l'avez déjà remarqué, nous ne pouvons entrer un nombre décimal que sous la forme d'une chaîne de caractères, composée de caractères ASCII. Pour tout autre traitement, par exemple pour la sortie binaire, nous devons convertir le contenu du buffer d'entrée en un format binaire. Nous utilisons pour cela une procédure de calcul simple dont voici d'abord l'organigramme.


Conversion
0 --> index
0 --> résultat
0 --> chiffre
W
index buffer
$30 -->
chiffre
chiffre > 9 ?
O
Terminé
Index + 1
--> Index
résultat * 10
--> résultat
résultat
+ chiffre
--> résultat
W

Avant que nous n'examinions le programme correspondant, encore une petite explication concernant la conversion décimal/binaire. Considérons tout d'abord un chiffre décimal isolé. Il peut avoir une valeur entre 0 et 9. Cela correspond d'ailleurs au code BCD. La conversion d'un chiffre ASCII en un chiffre BCD se fait par simple soustraction de la constante $30.

Chiffres 0-9 = ASCII $30-$39 = binaire BCD $00-$09

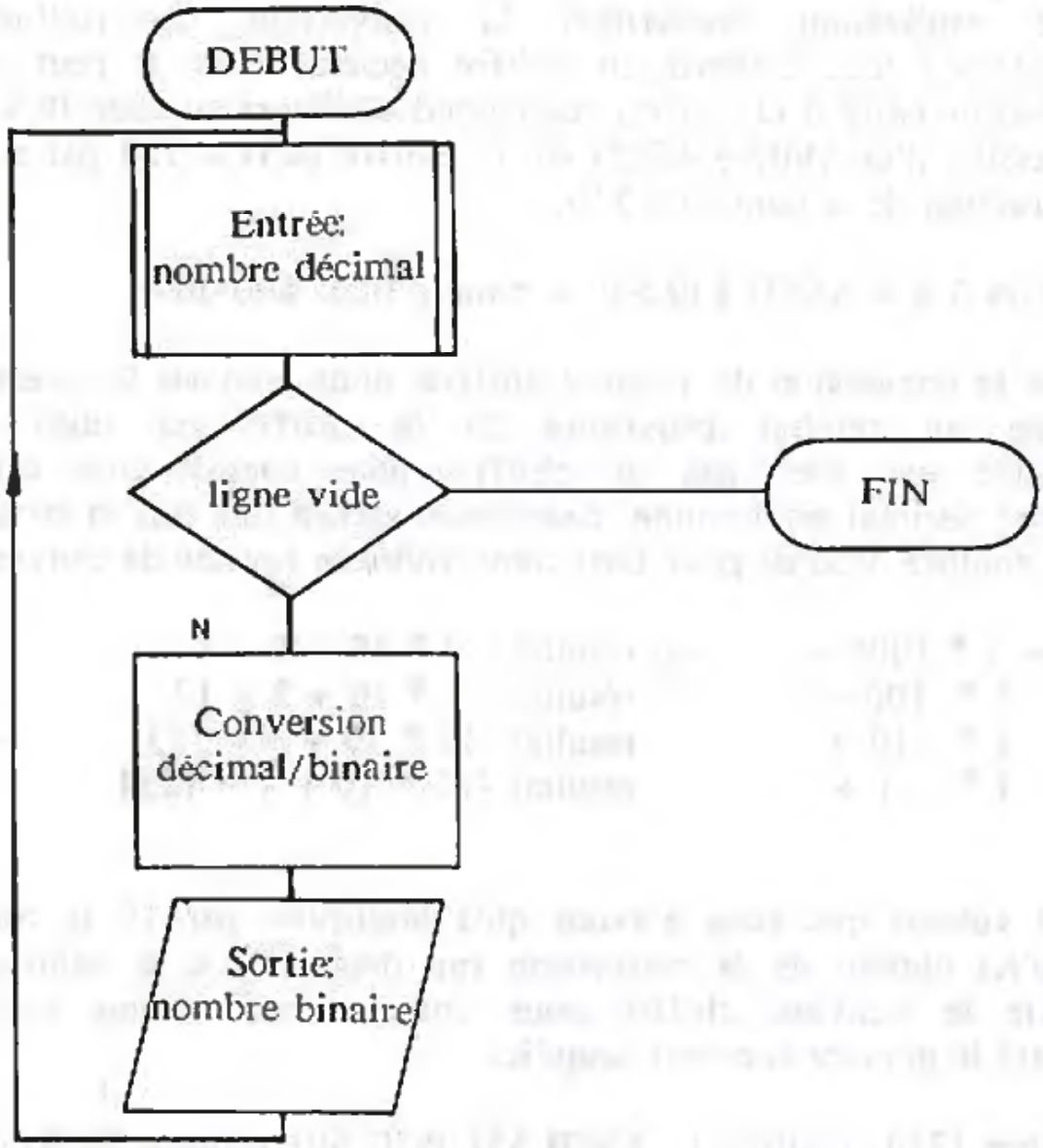
Après la conversion du premier chiffre, nous pouvons le considérer comme un résultat provisoire. Si le chiffre est suivi d'un caractère qui n'est pas un chiffre, nous considérerons que le nombre décimal est terminé. Examinons encore une fois la structure d'un nombre décimal pour bien comprendre la routine de conversion.

```
123 = 1 * 1000 +      ou résultat   0 * 10 + 1 = 1
      2 *  100 +          résultat   1 * 10 + 2 = 12
      3 *   10 +          résultat  12 * 10 + 3 = 123
      4 *    1 +          résultat 123 * 10 + 4 = 1234
```

Nous voyons que nous n'avons qu'à multiplier par 10 le résultat jusqu'ici obtenu de la conversion (au début: 0) et à additionner ensuite le nouveau chiffre pour obtenir ainsi comme nouveau résultat le nombre converti jusqu'ici.

```
Nombre 1234  Chiffre 1  ASCII $31 BCD $01 résultat $0001
Reste   234  Chiffre 2  ASCII $32 BCD $02 résultat $000C
Reste    34  Chiffre 3  ASCII $33 BCD $03 résultat $007B
Reste     4  Chiffre 4  ASCII $34 BCD $04 résultat $04D2
```

Cette procédure est utilisée aux lignes 28 à 42 de notre exemple. Nous expliquerons la réalisation en langage machine à la suite du listing qui est maintenant plus complet.


DEBUT
Entrée:
nombre décimal
ligne vide
FIN
N
Conversion
décimal/binaire
Sortie:
nombre binaire

```
                                ********************************************
                                *** Conversion decimal/binaire      Etape 6 ***
                                ********************************************

00000000 2A7C00000C6A           move.l  #ligne,a5   * creer pointeur
00000006 3F3C0001   entree: move.w  #1,-(sp)    * code: CONIN
0000000A 4E41               trap    #1          * appel de GEMDOS
0000000C 548F               addq.l  #2,sp       * correction pile
0000000E 1AC0               move.b  d0,(a5)+    * sauvegarder caractere
00000010 0C00000D           cmpi.b  #13,d0      * caractere etait un CR ?
00000014 66F0               bne     entree      * NON: caractere suivant
00000016 3F3C000D           move.w  #13,-(sp)   * sortir CR
0000001A 3F3C0002           move.w  #2,-(sp)    * code: CONOUT
0000001E 4E41               trap    #1          * appel de GEMDOS
00000020 588F               addq.l  #4,sp       * correction pile
00000022 3F3C000A           move.w  #10,-(sp)   * sortir LF
00000026 3F3C0002           move.w  #2,-(sp)    * code: CONOUT
0000002A 4E41               trap    #1          * appel de GEMDOS
0000002C 588F               addq.l  #4,sp       * correction pile
0000002E 2A7C00000C6A           movea.l #ligne,a5   * restaurer pointeur
00000034 4287               clr.l   d7          * annuler cheap resultat
00000036 4286               clr.l   d6          * annuler chiffre
00000038 1C1D       conver: move.b  (a5)+,d6    * traiter chiffre
0000003A 0406003D           subi.b  #'0',d6     * de ASCII a BCD
0000003E 0C060009           cmpi.b  #9,d6       * chiffre BCD trop grand
00000042 6200               bhi     binout      * OUI: plus de chiffre
00000044 CEFC000A           mulu.w  #10,d7      * decalage d'un chiffre
00000048 DE06               add.l   d6,d7       * additionner chiffre
0000004A 60EC               bra     conver      * nouveau chiffre
0000004C 7C1F       binout: move.l  #31,d6      * compteur de chiffres
0000004E 3A3C001D   sortie: move.w  #$1D,d5     * zero ASCII/2
00000052 E38F               lsl.l   #1,d7       * isoler bit
00000054 DA05               addx.b  d5,d5       * former 0/1 ASCII
00000056 3F05               move.w  d5,-(sp)    * sortir
00000058 3F3C0002           move.w  #2,-(sp)    * code: CONOUT
0000005C 4E41               trap    #1          * appel de GEMDOS
0000005E 588F               addq.l  #4,sp       * correction pile
00000060 51CEFFEC           dbf     d6,sortie   * compteur-1, tester si -1

00000064 3F3C0000           move.w  #0,-(sp)    * code: WARMSTART
00000068 4E41               trap    #1          * appel de GEMDOS

0000006A            ligne:  ds.b    80          * buffer 80 caracteres

000000BA                    .end
```

Le listing précédent présente maintenant la solution complète de la tâche que nous nous étions fixé. Notre programme réunit trois groupes de fonction:

- entrée d'une ligne (lignes 8 à 26)
- conversion ASCII-binaire (lignes 28 à 42)
- sortie d'un nombre binaire (lignes 44 à 55)

Examinons encore une fois les instructions dans leur ensemble. La ligne 8 crée un pointeur (A5) sur le buffer d'entrée. Les lignes 10 à 16 constituent la boucle d'entrée. En lignes 10 à 12, un caractère est lu au clavier. Avec l'instruction MOVE, le code de fonction (1) de la fonction CONIN (entrée console) du système d'exploitation est placé sur la pile. Le système d'exploitation est appelé avec l'instruction TRAP de la ligne suivante. Le caractère entré est transmis à travers le registre D0 et transféré par l'instruction MOVE suivante dans la zone buffer, tant qu'aucun CR n'est entré. Notez que le pointeur sur le buffer est dirigé sur l'octet suivant du buffer par l'instruction MOVE.

Les lignes 18 à 26 fixent le curseur sur la ligne suivante de l'écran. A cet effet, des caractères de commande CR et LF sont sortis grâce à la fonction CONOUT (sortie console) du système d'exploitation. Le caractère de commande est d'abord transmis sur la pile, suivi du code de fonction pour CONOUT (2). Le système d'exploitation est appelé avec l'instruction TRAP. Les paramètres sont ensuite retirés de la pile par manipulation du pointeur de pile.

Les lignes 28 à 42 convertissent le nombre contenu dans le buffer en un nombre binaire (résultat en D7), suivant la procédure de conversion que nous avons décrite. Le pointeur sur le buffer est d'abord dirigé à nouveau en ligne 28 sur le premier élément. Les lignes 30 et 31 annulent les registres D7 et D6. La véritable boucle de conversion commence à partir de la ligne 33.

Un caractère est transféré du buffer d'entrée dans un registre auxiliaire (D6), le pointeur de buffer (A5) étant simultanément augmenté d'un octet. Par soustraction en ligne 34 de la constante $30, le chiffre ASCII est converti en un nombre BCD. On teste la validité du résultat (ligne 36). Si le résultat est supérieur à 9, c'est qu'il ne s'agissait pas d'un chiffre et la conversion est terminée (ligne 37). La ligne 39 multiplie le résultat actuel par la constante 10. Le chiffre qui vient d'être calculé est ensuite additionné au résultat actuel en D7 (ligne 40). L'instruction BRA de la ligne 42 déclenche alors un saut "inconditionnel" au début de la boucle.

L'instruction de saut de la ligne 37 marque la fin de la conversion décimal/binaire. Le programme se poursuit à la ligne 44, le nombre décimal se trouvant dans le registre D7. La section de programme des lignes 44 à 55 sort maintenant ce nombre sous la forme d'un nombre binaire. Comme notre routine de conversion décimale ne peut travailler sans problème qu'à l'intérieur de la zone 0 - 65535 ($0 - $FFFF), à cause de l'instruction de multiplication qui ne traite que des mots, nous ne sortons que 16 chiffres du résultat. Nous vous avons déjà expliqué comment fonctionne la sortie d'un nombre binaire. Nous allons cependant y revenir à propos du dernier listing assembleur.

En ligne 44, le compteur de chiffres dans le registre de données D6 est fixé sur 15. Le compteur de chiffres est ensuite diminué chaque fois de 1, jusqu'à ce qu'il devienne inférieur à 0 (instruction DBF en ligne 55). Cela correspond exactement à 16 parcours de la boucle. A l'intérieur de cette boucle, le bit le plus élevé du mot de moindre valeur est transféré en ligne 47, au moyen de l'instruction LSL, de D7 dans le flag X. En ligne 46 est formée la constante $18 qui correspond à la moitié du zéro ASCII ($30/2 = $18). Avec l'addition en ligne 48, cette constante est additionnée à elle-même ce qui correspond à une multiplication par deux. Lors de cette addition, le contenu du flag X est également additionné.

Comme le bit à sortir est contenu dans le flag X, cette addition fournit soit un $30 soit un $31 dans le registre D5. Ces valeurs correspondent aux caractères ASCII pour zéro et un.

Le résultat de cette conversion dans le registre D5 est sorti à l'écran par les lignes 50 à 53. L'instruction de la ligne 50 pousse sur la pile le contenu du registre D5. Le code de fonction 2, pour la fonction CONOUT du système d'exploitation est alors formé. Le système d'exploitation est appelé par l'instruction TRAP et le caractère est sorti. Le pointeur de pile est alors corrigé.

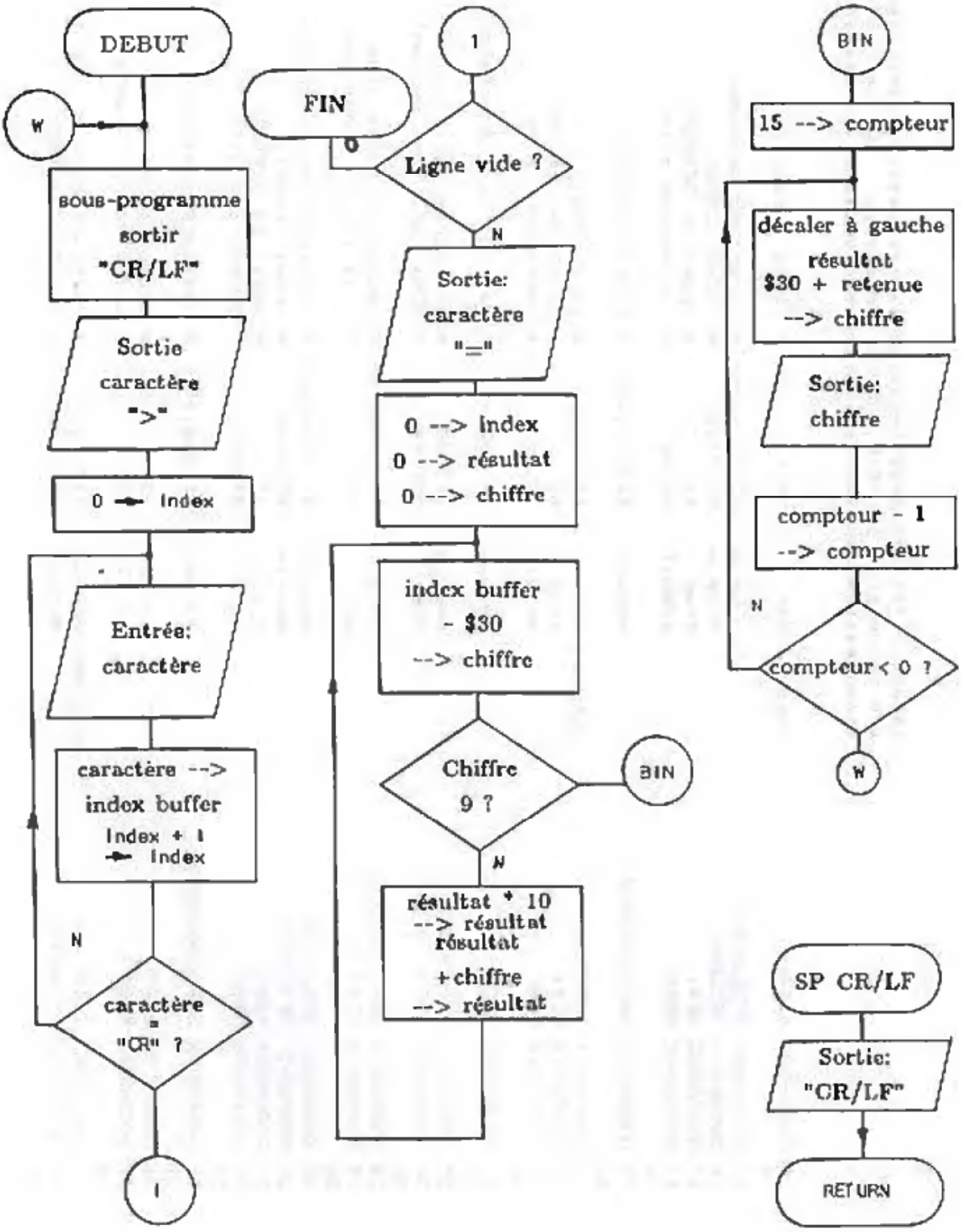
Une fois la boucle terminée, le programme se poursuit en ligne 57. Ici est formé le code de fonction pour Warm Start après quoi le système d'exploitation est appelé (ligne 58). L'exécution du programme est alors terminée.

La ligne 60 du listing assembleur contient la définition de la zone buffer pour la boucle d'entrée.

Septième étape, la boucle d'entrée

En conclusion de ce chapitre, nous allons encore affiner un peu notre programme. Un message initial (caractère d'interrogation) demandera d'une part l'entrée d'un nombre décimal et d'autre part la routine tout entière tournera dans une boucle d'entrée. Cela signifie que lorsqu'une conversion décimal/binaire aura été traitée, on demandera à nouveau l'entrée d'un nombre. Le programme ne sera terminé que lorsqu'aucun nouveau nombre n'aura été entré.

Vous trouverez dans les pages suivantes un organigramme étendu en conséquence ainsi que le listing assembleur du programme d'exemple étendu. Nous décrirons ensuite ce listing assembleur.


DEBUT
W
sous-programme
sortir
"CR/LF"
Sortie
caractère
">"
0 → Index
Entrée:
caractère
caractère -->
index buffer
Index + 1
→ Index
N
caractère
=
"CR" ?
I
FIN
I
O
Ligne vide ?
N
Sortie:
caractère
"="
0 --> Index
0 --> résultat
0 --> chiffre
index buffer
- $30
--> chiffre
Chiffre
9 ?
BIN
N
résultat * 10
--> résultat
résultat
+chiffre
--> résultat
BIN
15 --> compteur
décaler à gauche
résultat
$30 + retenue
--> chiffre
Sortie:
chiffre
compteur - 1
--> compteur
N
compteur < 0 ?
W
SP CR/LF
Sortie:
"CR/LF"
RETURN

```
                                ********************************************************
                                *** Conversion DEC/BIN & boucle d'entree    Etape 7 ***
                                ********************************************************


00000000 6174                   repet:  bsr     crlf            * ligne suivante
                                
00000002 3F3C203F                       move.w  #" ?",-(sp)     * caractere message
00000006 3F3C0002                       move.w  #2,-(sp)        * code: CONOUT
0000000A 4E41                           trap    #1              * appel de GEMDOS
0000000C 588F                           addq.l  #4,sp           * correction pile
                                
0000000E 2A7C00000090                   movea.l #ligne,a5       * creer pointeur
                                
00000014 3F3C0001               entree: move.w  #1,-(sp)        * code: CONIN
00000018 4E41                           trap    #1              * appel de GEMDOS
0000001A 548F                           addq.l  #2,sp           * correction pile
                                
0000001C 1AC0                           move.b  d0,(a5)+        * sauvegarder caractere
                                
0000001E 0C00000D                       cmpi.b  #13,d0          * caractere etait un CR ?
00000022 66F0                           bne     entree          * NON: caractere suivant
                                
00000024 B8FC00000091                   cmpa.l  #ligne+1,a5     * tester si ligne vide
0000002A 6744                           beq     fin             * OUI: fin du programme
                                
0000002C 3F3C203D                       move.w  #" =",-(sp)     * caractere de separation
00000030 3F3C0002                       move.w  #2,-(sp)        * code: CONOUT
00000034 4E41                           trap    #1              * appel de GEMDOS
00000036 588F                           addq.l  #4,sp           * correction pile
                                
0000003B 2A7C00000090                   movea.l #ligne,a5       * restaurer pointeur
                                
0000003E 4207                           clr.l   d7              * champ resultat
00000040 4286                           clr.l   d6              * champ calcul (reste)
                                
00000042 1C1D                   conver: move.b  (a5)+,d6        * traiter chiffre
00000044 04060030                       subi.b  #$30,d6         * de ASCII a BCD
                                
```

```
42 00000048 0C060009                    cmpi.b   #9,d6          * chiffre BCD trop grand
43 0000004C 6208                        bhi      binout         * OUI: plus de chiffre
44
45 0000004E CEFC000A                    mulu.w   #10,d7         * decalage d'un chiffre
46 00000052 DE86                        add.l    d6,d7          * additionner chiffre
47
48 00000054 60EC                        bra      conver         * nouveau chiffre
49
50 00000056 7C0F                binout: move.l   #15,d6         * compteur de chiffres
51
52 00000058 3A3C0018            sortie: move.w   #$18,d5        * zero ASCII/2
53 0000005C E38F                        lsl.l    #1,d7          * isoler bit
54 0000005E DB05                        addx.b   d5,d5          * former 0/1 ASCII
55

56 00000060 3F05                        move.w   d5,-(sp)       * sortir
57 00000062 3F3C0002                    move.w   #2,-(sp)       * code: CONOUT
58 00000066 4E41                        trap     #1             * appel de GEMDOS
59 00000068 588F                        addq.l   #4,sp          * correction pile
60 0000006A 51CEFFEC                    dbf      d6,sortie      * compteur-1, tester si -1
61
62 0000006E 6090                        bra      repet          * nouvelle entree
63
64 00000070 3F3C0000            fin:    move.w   #0,-(sp)       * code: WARMSTART
65 00000074 4E41                        trap     #1             * appel de GEMDOS
66
67 00000076 3F3C000D            crlf:   move.w   #13,-(sp)      * sortir CR
68 0000007A 3F3C0002                    move.w   #2,-(sp)       * code: CONOUT
69 0000007E 4E41                        trap     #1             * appel de GEMDOS
70 00000080 588F                        addq.l   #4,sp          * correction pile
71
72 00000082 3F3C000A                    move.w   #10,-(sp)      * sortir LF
73 00000086 3F3C0002                    move.w   #2,-(sp)       * code: CONOUT
74 0000008A 4E41                        trap     #1             * appel de GEMDOS
75 0000008C 588F                        addq.l   #4,sp          * correction pile
76
77 0000008E 4E75                        rts                     * retour
78
79 00000090                     ligne:  .ds.b    80             * buffer 80 caracteres
80
81 000000E0                             .end
```

La première modification évidente apportée à notre programme concerne la sortie de CR/LF. Nous avons réalisé ici cette fonction sous la forme d'un sous-programme. Le sous-programme est défini aux lignes 68 à 78. Par sa fonction il est identique à nos exemples précédents.

Ce sous-programme est appelé en ligne 8. Il n'est utilisé en aucun autre endroit du programme. Nous n'y avons donc eu recours qu'à titre de démonstration des instructions BSR et RTS. Les lignes 10 à 13 sortent un caractère de message (?), la fonction CONOUT du système d'exploitation étant à nouveau utilisée. La ligne 10 présente une particularité: nous définissons le caractère à sortir au moyen d'une constante de texte. Nous remédions ainsi par une astuce à une insuffisance de l'assembleur. L'instruction MOVE définit le "mot" comme largeur de traitement. Si nous n'indiquions qu'un caractère ASCII, l'assembleur le compléterait pour atteindre la largeur d'un mot. Il ajouterait pour cela un $00 à droite de notre caractère ASCII. Le caractère à sortir ne se trouverait plus alors dans la partie de moindre valeur du mot, ce qui entraînerait que le caractère sorti ne serait pas un caractère visible. Cette sortie serait correcte en soi mais $00 n'est pas un caractère imprimable. Nous pouvons remédier à cette insuffisance de l'assembleur en indiquant deux caractères comme constante de texte, ESPACE et un caractère ASCII. L'assembleur formera maintenant un mot dont l'octet de moindre valeur contiendra le caractère ASCII. Cette astuce est cependant programmée d'une façon qui n'est pas très orthodoxe car la partie de plus grande valeur du mot devrait toujours valoir zéro pour rester compatible avec les systèmes d'exploitation futurs. Comme nous avons toutefois formulé nos exemples concrètement pour l'Atari 520 ST, nous nous permettrons de négliger ce petit défaut.

Les lignes 15 à 24 traitent l'entrée d'une ligne. La réalisation de cette fonction ne se distingue pas de notre exemple antérieur.

A cet endroit ont été ajoutées les lignes 26 et 27. On teste ici si un nombre décimal a bien été entré. On interroge pour cela le pointeur (A5) sur le buffer pour voir s'il est dirigé sur le second élément du buffer. Si c'est le cas, un seul caractère a pu être entré dans le buffer. Comme le dernier caractère dans le buffer est toujours un CR, nous pouvons être certain ici que, si le buffer ne contient qu'un caractère, c'est qu'aucun chiffre n'a été entré. Une fois qu'une ligne vide a été reconnue, au saute directement au traitement final (lignes 65 et 66). Le programme sera alors terminé de façon habituelle, avec retour au système d'exploitation.

Si ce n'est pas une ligne vide qui a été entrée, les lignes 29 à 32 produisent un signe de séparation qui sépare le nombre en entrée de la sortie. Nous utilisons à nouveau à cet effet la fonction CONOUT du système d'exploitation.

Les lignes 34 à 48 exécutent à nouveau la conversion décimal/binaire. Comme nous avons déjà décrit cette fonction dans les exemples précédents, nous ne nous répèterons pas ici. Il en va de même pour la sortie binaire des lignes 50 à 61.

En ligne 63 figure une instruction de saut inconditionnelle "retour à l'entrée" qui ferme notre boucle d'entrée. Lors de l'exécution du programme, un message initial est sorti à la ligne suivante et on attend une entrée. Ce n'est qu'en entrant un Return ou en effectuant un reset du processeur que le programme peut être quitté.

Nous arrivons maintenant à la conclusion de notre introduction à la programmation assembleur "étape par étape". Dans le chapitre suivant, nous vous présenterons des programmes un peu plus longs mais nous n'entrerons plus dans une description aussi détaillée du développement des programmes.

SOLUTION
DE PROBLEMES CARACTERISTIQUES

1) Introduction

2) Conversion hexadécimal/décimal

3) Conversion décimal/hexadécimal

4) Calcul de la moyenne

5) Tri simple

6) Sortie: chaînes de caractères

7) Entrée: chaîne de caractères avec contrôle

8) Sortie: date

9) Calcul de factorielles

Nous allons vous présenter dans ce chapitre des programmes d'exemple allant un peu plus loin. Nous en profiterons pour vous expliquer encore quelques techniques de programmation et quelques fonctions du système d'exploitation ainsi que pour vous présenter des algorithmes caractéristiques.

Nous aurions certainement pu utiliser à certains endroits des fonctions plus puissantes du système d'exploitation ce qui nous aurait permis d'abréger nos programmes d'exemple. Le but essentiel de cet ouvrage est cependant, comme nous l'avons déjà indiqué, de vous permettre d'assimiler la méthode de programmation en assembleur et de vous exercer avec quelques exemples.

Chaque exemple est organisé en plusieurs parties. En introduction, nous vous présenterons le problème et vous indiquerons des solutions possibles. Suivront ensuite chaque fois un organigramme et un listing assembleur complet qui sera ensuite expliqué, ainsi que les algorithmes utilisés.

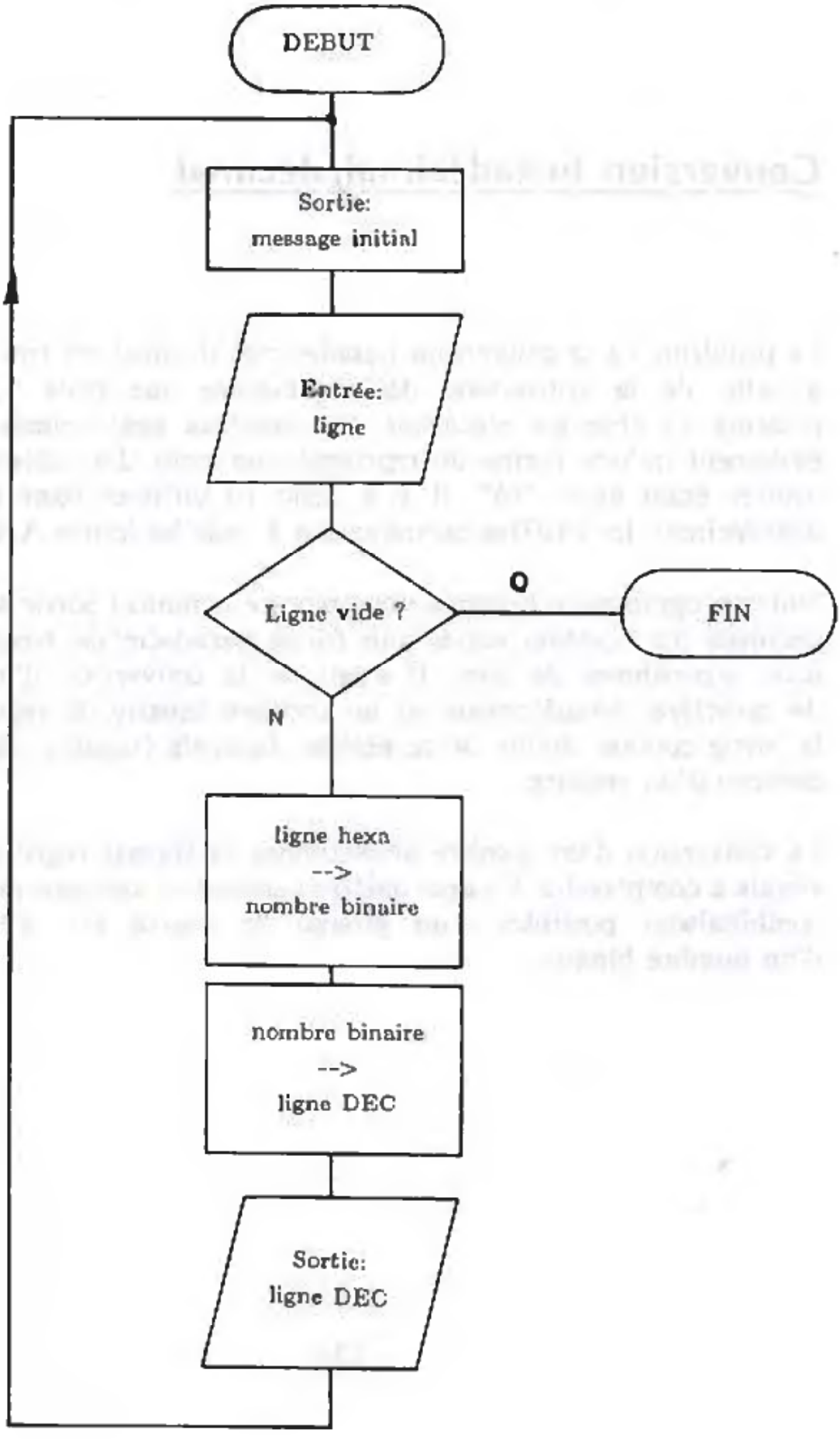
Nous vous recommandons de reproduire entièrement tous les exemples. Si vous possédez déjà un assembleur, vous pouvez essayer tous les exemples sur l'Atari ST.

Conversion hexadécimal/décimal

Le problème de la conversion hexadécimal/décimal est très semblable à celui de la conversion décimal/binaire que nous vous avons présenté au chapitre précédent. Les nombres hexadécimaux ne sont également qu'une forme de représentation pour des valeurs, la base utilisée étant alors "16". Il y a donc 16 chiffres dans le système hexadécimal, les chiffres normaux 0 à 9 puis les lettres A à F.

Notre programme d'exemple vous montre comment sortir sous forme décimale des nombres entrés sous forme hexadécimale. Nous utilisons deux algorithmes de base. Il s'agit de la conversion d'une chaîne de caractères hexadécimale en un contenu binaire de registre et de la sortie comme chaîne de caractères décimale (nombre décimal) du contenu d'un registre.

La conversion d'un nombre hexadécimal en format registre est assez simple à comprendre. Chaque chiffre correspond exactement à une des combinaisons possibles d'un groupe de quatre bits à l'intérieur d'un nombre binaire.


DEBUT
Sortie:
message initial
Entrée:
ligne
Ligne vide ?
O
FIN
N
ligne hexa
-->
nombre binaire
nombre binaire
-->
ligne DEC
Sortie:
ligne DEC

```
1
2
3                          ********************************************************
4                          *** Conversion hexadecimal/decimal          Exemple 1 ***
5                          ********************************************************
6
7
8  00000000 A11C000A       repet:  bsr     cr14            * ligne suivante
9
10 00000004 3F3C203F               move.w  #' ?',-(sp)     * caractere message
11 00000008 3F3C0002               move.w  #2,-(sp)        * code: CONOUT
12 0000000C 4E41                   trap    #1              * appel de BDOS
13 0000000E 588F                   addq.l  #4,sp           * correction pile
14
15 00000010 2A7C00000E2            movea.l #ligne,a5       * creer pointeur
16
17 00000016 3F3C0001    entree: move.w  #1,-(sp)        * code: CONIN
18 0000001A 4E41                   trap    #1              * appel de BDOS
19 0000001C 548F                   addq.l  #2,sp           * correction pile
20
21 0000001E 1A00                   move.b  d0,(a5)+        * sauvegarder caractere
22
23 00000020 0D000D0                cmpi.b  #13,d0          * caractere etait un CR ?
24 00000024 66F0                   bne     entree          * NON: caractere suivant
25
26 00000026 B9FC00000E3            cmpa.l  #ligne+1,a5     * tester si ligne vide
27 0000002C 67000074               beq     fin             * OUI: fin du programme
28
29 00000030 2A7C00000E2            movea.l #ligne,a5       * restaurer pointeur
30
31 00000036 4287                   clr.l   d7              * chang resultat
32 00000038 4206                   clr.l   d6              * chang calcul (reste)
33
34 0000003A 1C18        convers: move.b  (a5)+,d6       * traiter chiffre
35 0000003C 0C060030               cmpi.b  #'0',d6         * caractere de commande ?
36 00000040 6534                   blo     decout          * OUI: fin conversion
37
38 00000042 04060030               subi.b  #'0',d6         * de ASCII a BCD
39 00000046 0C060009               cmpi.b  #9,d6           * chiffre BCD ou ?
40 0000004A 6720                   beq     fin             * OUI: c'etait un chiffre
41
42 0000004C 04060027               subi.b  #39,d6          * correction chiffre BCD
43
44 00000050 0C06000A               cmpi.b  #'a',d6         * lettre HEX ?
45 00000054 6520                   blo     decout          * NON: fin conversion
46
47 00000058 0E06000F               cmpi.b  #'f',d6         * lettre HEX ?
48 0000005C 621A                   bhi     decout          * NON: fin conversion
49
50 0000005E E98F        ecale:  lsl.l   #4,d7           * decalage d'un chiffre
51 00000060 DF86                   add.l   d6,d7           * additionner chiffre
52
53 00000062 0C07FFF0               cmpi.b  #$fff0,d7       * tester depassement
54 00000066 6D04                   blt     conver          * NON: nouveau chiffre
55
56 00000068 6160                   bsr     crlf            * erreur sur ligne suivante
57 0000006A 3F3C0071               move.w  #'q',-(sp)      * message d'erreur
58 0000006E 3F3C0002               move.w  #2,-(sp)        * code: CONOUT
59 00000072 4E41                   trap    #1              * appel de BDOS
60 00000074 588F                   addq.l  #4,sp           * correction pile
61
62 00000076 6000FFB4               bra     repet           * entrer nouveau nombre
63
64
65 0000007A 6150        decout: bsr     crlf            * curseur sur ligne suivante
66
```

```
                        move.w  #" =",-(sp)  * message de resultat
                        move.w  #2,-(sp)     * code: CCONOUT
                        trap    #1           * appel de GEMDOS
                        addq.l  #4,sp        * correction pile

                        andi.l  #$ffff,d7    * limiter chiffres

                        movea.l #ligne,a5    * viser pointeur

dubec1                  move.l  d7,d6        * traiter chiffre

                        divu.w  #10,d6       * calcul valeur/10
                        move.w  d6,d7        * sauver resultat
                        swap    d6           * former reste
                        addi.w  #$30,d6      * former ASCII

                        move.b  d6,(a5)+     * chiffre dans buffer

                        cmpi.w  #0,d7        * tous les chiffres?
                        bne     dubec        * NON:chiffre suivant

sortie1                 cmpa.l  #ligne,a5    * tester buffer
                        beq     repet        * OUI: fini, suivant

                        move.b  -(a5),d7     * retirer caractere
                        andi.w  #$ff,d7      * formatiser caractere

                        move.w  d7,-(sp)     * sortir caractere
                        move.w  #2,-(sp)     * code: CONOUT
                        trap    #1           * appel de GEMDOS
                        addq.l  #4,sp        * correction pile

                        bra     sortie       * tester si fini

fini                    move.w  #0,-(sp)     * code WARMSTART
                        trap    #1           * appel de GEMDOS

repet                   move.w  #13,-(sp)    * sortir CR
                        move.w  #2,-(sp)     * code CONOUT
                        trap    #1           * appel de GEMDOS
                        addq.l  #4,sp        * correction pile

                        move.w  #10,-(sp)    * sortir LF
                        move.w  #2,-(sp)     * code CONOUT
                        trap    #1           * appel de GEMDOS
                        addq.l  #4,sp        * correction pile

                        rts                  * retour


ligne:                  .ds.b   80           * buffer 80 caracteres

                        .end
```

Nous avons donc simplement à convertir le chiffre représenté par un code ASCII en un nombre binaire correspondant. Nous sommes aidé en cela par le caractère systématique du code ASCII.

Binaire 0000 à 1001 (0 - 9 hexadécimal) = ASCII $30 - $39
Binaire 1000 à 1111 (A - F hexadécimal) = ASCII $41 - $46

De ce caractère systématique, nous pouvons déduire une procédure de calcul:

> Si le chiffre hexadécimal est compris entre ASCII 0 et 9, nous soustrayons $30 pour obtenir la valeur binaire du chiffre. Si le chiffre hexadécimal est compris entre ASCII A et F, nous soustrayons $37 pour obtenir la valeur binaire du chiffre. Si le chiffre ne se trouve pas compris dans l'un de ces deux intervales, nous supposons que le nombre hexadécimal est terminé. S'il y a déjà un chiffre, le résultat antérieur est multiplié par 16 (décalage de 4 bits vers la gauche) et le nouveau chiffre y est additionné.

L'algorithme pour la conversion binaire/décimal est plus complexe en théorie mais il est très aisé à réaliser sur le 68000. La règle de calcul utilisée se base sur le schéma de HORNER. Un nombre à convertir est dans ce schéma divisé par la nouvelle base (10 en l'occurrence). Le reste de la division correspond à un chiffre du système numérique correspondant (ici 0 à 9). Cette division se poursuit jusqu'à ce que le résultat devienne nul. Un exemple illustrera ce principe:

$04D2 (déc 1234) / $A (déc 10) = $7B (déc 123) reste 4
$007B (déc 123) / $A (déc 10) = $0C (déc 12) reste 3
$000C (déc 12) / $A (déc 10) = $01 (déc 1) reste 2
$0001 (déc 1) / $A (déc 10) = $00 (déc 0) reste 1

Vous voyez que nous obtenons avec ce schéma tous les chiffres du nombre décimal, mais dans un ordre inversé. C'est pourquoi tous les chiffres doivent être entre-stockés pour être sortis dans l'ordre inverse après la conversion.

Passons maintenant à la description du programme machine:

En ligne 8 est appelé le sous-programme produisant CR/LF. Ce dernier a été programmé en lignes 106 à 116. Il est de fonction identique au sous-programme du dernier exemple. Les instructions en lignes 106 à 109 sortent un CR sur l'écran, avec emploi de la routine GEMDOS CONOUT. Les lignes 111 à 114 sortent un LF avec le même procédé. Le sous-programme se termine par l'instruction RTS en ligne 116.

Les lignes 10 à 13 sortent sur l'écran un "?" comme message d'entrée. C'est à nouveau la fonction de sortie console de GEMDOS qui est utilisée. En ligne 6, le pointeur sur le buffer d'entrée est initialisé. Il est maintenant dirigé sur le premier octet de cette zone buffer. La zone d'entrée elle-même est définie en ligne 119. Elle peut recevoir dans notre exemple un maximum de 80 caractères.

Les lignes 15 à 24 constituent la boucle d'entrée. Les lignes 17 à 19 appellent le système d'exploitation. La fonction CONIN de GEMDOS, que vous connaissez depuis le chapitre précédent, est ici utilisée. La ligne 21 transfère un caractère entré dans le buffer d'entrée. Le code ASCII de la touche est alors transféré à travers l'octet inférieur du registre D0. En lignes 23 et 24, la condition pour quitter la boucle est interrogée. Tant qu'aucun CR ne figure dans le registre D0, la boucle, c'est-à-dire l'entrée, se poursuit avec l'instruction de la ligne 17 qui permet l'entrée du caractère suivant.

Si le dernier caractère était un CR (touche RETURN), le programme se poursuit par l'instruction de la ligne 26. On teste ici si seulement un caractère a été placé dans le buffer. Si c'est le cas, le programme saute à l'instruction de la ligne 102. Le programme se termine alors par l'appel bien connu du système d'exploitation.

S'il ne s'agissait pas d'une ligne vide, on continue en ligne 29. Ici commence la routine de conversion qui convertit le contenu du buffer d'entrée en un nombre binaire. Un pointeur est d'abord dirigé à nouveau, en ligne 29, sur le premier octet du buffer. Conformément à notre procédure de calcul pour la conversion HEX/BIN et à notre organigramme, les instructions des lignes 31 et 32 initialisent deux registres de données (en les fixant sur zéro). D7 sera utilisé comme champ de résultat et D6 comme champ de calcul pour un chiffre hexadécimal.

En ligne 34 commence le traitement d'un chiffre hexadécimal (à l'intérieur de la boucle). L'instruction en ligne 34 transfère un chiffre hexadécimal du buffer dans le champ de calcul (D6) et fixe le pointeur (A5) sur le prochain champ. L'instruction de comparaison en ligne 35 filtre, avant le traitement proprement dit, certains des nombres NON HEXA possibles. Si un caractère est inférieur à $30, il ne peut s'agir que d'un caractère de commande ou d'un caractère spécial. Dans ce cas, l'instruction en ligne 36 saute à la routine de sortie décimale (à partir de la ligne 65). Si le caractère ASCII est supérieur à $30, on en soustrait $30, sans tenir compte du fait qu'il peut encore s'agir d'un caractère NON HEXA. L'instruction de comparaison de la ligne 79 détermine si le caractère peut être considéré comme un caractère hexadécimal du groupe 0 à 9. Si c'est le cas, on saute à l'instruction de la ligne 50 où commence le traitement de ce chiffre. Si le caractère traité n'est pas dans le groupe 0 à 9, la conversion se poursuit en ligne 43. On soustrait ici la constante $27. N'oubliez pas que $30 avait déjà été ôté du caractère ASCII originel.

Le résultat doit maintenant être compris entre $A et $F, ce qui correspond au second groupe de chiffres hexadécimaux. Cette condition est vérifiée en lignes 44 à 48. Si le caractère n'appartient pas non plus à ce groupe, la sortie décimale commence. Dans le cas contraire, le traitement d'un chiffre correctement converti commence à partir de la ligne 50.

L'instruction LSL.L en ligne 50 décale le champ de résultat actuel de quatre bits vers la gauche. Le nouveau chiffre hexadécimal est ensuite ajouté au champ de résultat (ligne 51).

En conclusion de la boucle de conversion, le résultat est encore contrôlé (ligne 53). Si le champ de résultat est inférieur à $FFFF, commence la conversion du chiffre suivant. Cela se produit par un retour au début de la boucle (ligne 34). S'il y a dépassement, le curseur est fixé sur le début de la ligne suivante par un appel du sous-programme CRLF en ligne 56 et un caractère "!" est sorti comme message d'erreur par les instructions des lignes 57 à 60. La conversion est alors interrompue. Le programme saute en cas d'erreur au début du programme (ligne 62).

En ligne 65 commence la routine de sortie décimale. Nous utilisons ici aussi exactement l'algorithme que nous avons décrit auparavant et qui est défini dans l'organigramme. Le curseur est d'abord fixé sur la ligne suivante par un appel en ligne 65 du sous-programme CRLF et le signe égale (=) est sorti en lignes 67 à 70. L'appel du système d'exploitation a déjà été expliqué et nous n'y reviendrons pas.

Les instructions des lignes 72 et 74 préparent la sortie d'après notre algorithme. La valeur sortie est limitée aux nombres binaires dans l'intervale 0 - 65535 et le pointeur est fixé sur le début de la zone buffer. Celle-ci ne sert plus maintenant à l'entrée d'un chaîne de caractères mais au stockage intermédiaire des résultats d'après le schéma de Horner.

A partir de l'instruction en ligne 76 commence le calcul des chiffres décimaux. En ligne 78, la valeur obtenue est divisée par 10. Le résultat entier est sauvegardé comme nouvelle valeur dans le registre D7 où il est conservé pour un nouveau parcours de la boucle et où il est utilisé comme valeur obtenue pour le chiffre suivant.

L'instruction SWAP de la ligne 80 échange ensuite entre elles les parties supérieure et inférieure du registre. La partie de plus grande valeur contient le reste de la division qui correspond déjà à un chiffre "prêt". Le code ASCII est calculé par addition de la constante $30 (ligne 81). Le résultat est placé dans le buffer (ligne 83), le compteur d'adresse étant avancé d'un octet.

En ligne 86, on saute au début de la routine de conversion. La sortie de tous les chiffres décimaux doit être réalisée par la routine des lignes 88 à 99. On teste d'abord si tous les caractères ont bien été sortis. Si le buffer a été vidé, on saute à nouveau au début du programme global. Dans le cas contraire, un caractère est retiré du buffer, le pointeur est augmenté, le mot est masqué et le caractère est sorti en conséquence (CONOUT). La boucle se poursuit par un saut à son début (ligne 99). Elle ne peut être quittée que par la condition en ligne 89.

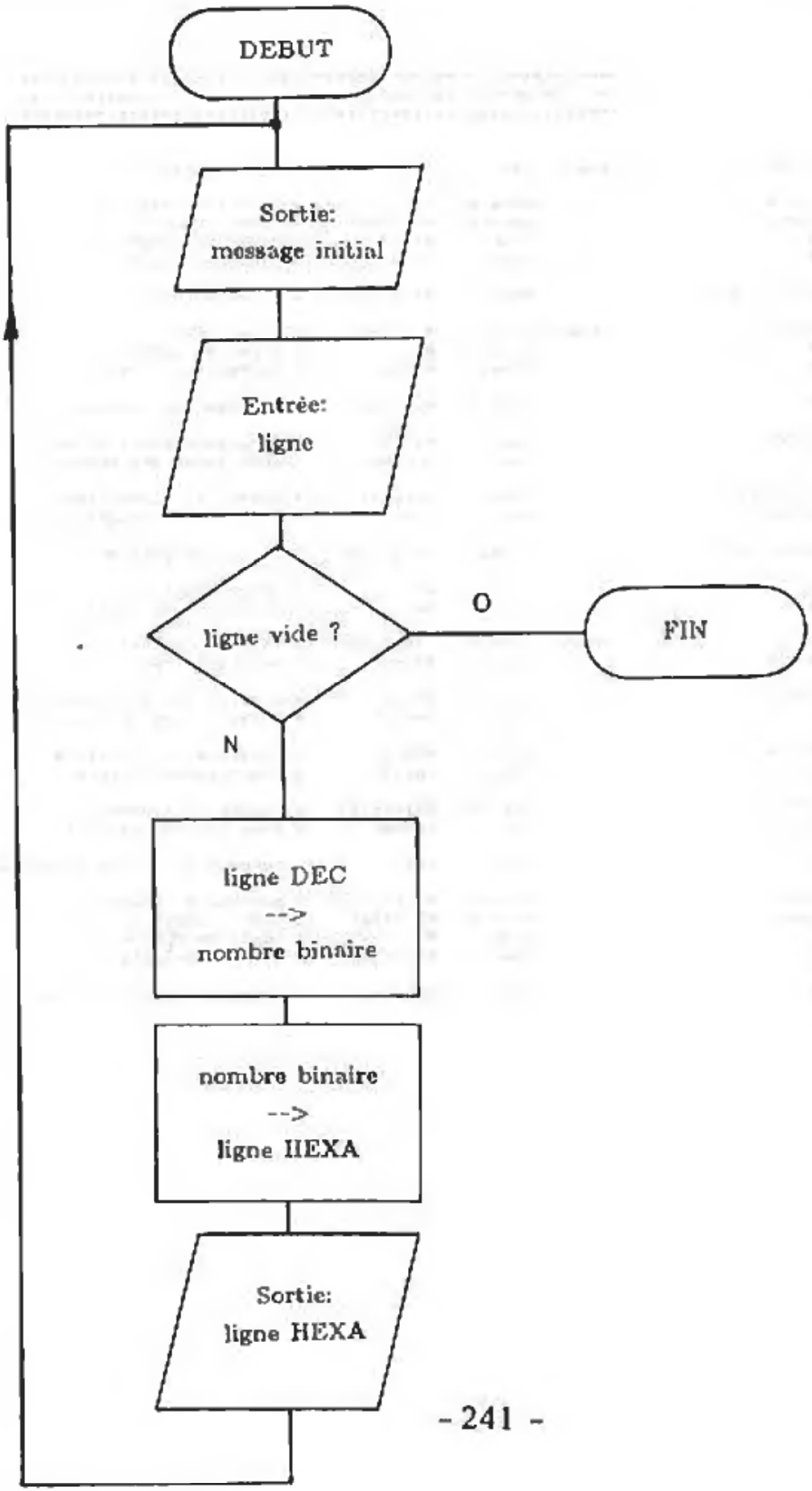
Vous retrouverez certaines parties de ce programme dans nos exemples. Nous ne décrirons bien sûr pas chaque fois les sections de programme identiques.

Conversion décimal/hexadécimal

Le problème de la conversion décimal/hexadécimal est l'inverse de la conversion hexadécimal/décimal présentée dans l'exemple précédent. Nous allons à nouveau utiliser deux algorithmes de base. Il s'agit de la conversion d'une chaîne de caractères décimale en un contenu binaire de registre et de la sortie d'un contenu de registre sous la forme d'une chaîne de caractères hexadécimale.

Nous avons déjà expliqué au chapitre précédent le traitement de nombres décimaux entrés au clavier (conversion décimal/binaire). La sortie d'un nombre hexadécimal pose par contre un problème nouveau. L'algorithme utilisé est facile à comprendre si l'on se souvient de l'exemple précédent (conversion hexadécimal/décimal). Pour sortir un nombre hexadécimal, il suffit de subdiviser le contenu binaire d'un registre en groupes de 4 bits. Cette subdivision est obtenue avec des opérations logiques SHIFT. Chaque groupe de 4 bits correspond à un chiffre hexadécimal. Ce chiffre hexadécimal doit être converti en un caractère ASCII pour pouvoir être sorti. Cela est simplement obtenu en additionnant des constantes. On additionne d'abord $30 pour former les chiffres 0 à 9 (ASCII $30 - $39). Si le résultat est supérieur à $39, une seconde addition, de $27, permet de former maintenant les chiffres hexadécimaux A - F (ASCII $61-$66).

Examinons l'organigramme et le listing assembleur de la conversion décimal/hexadécimal:


DEBUT
Sortie:
message initial
Entrée:
ligne
ligne vide ?
O
FIN
N
ligne DEC
-->
nombre binaire
nombre binaire
-->
ligne HEXA
Sortie:
ligne HEXA

```
1
2
3                                  ********************************************************
4                                  *** Conversion decimal/hexadecimal          Exemple 2 ***
5                                  ********************************************************
6
7
8  00000000 61000008               repet:  bsr     crlf            * ligne suivante
9
10 00000004 3F3C0209                       move.w  #" ",-(sp)      * caractere message
11 00000008 3F3C0002                       move.w  #2,-(sp)        * code: CONOUT
12 0000000C 4E41                           trap    #1              * appel de GEMDOS
13 0000000E 584F                           addq.l  #4,sp           * correction pile
14
15 00000010 2A7C00000006                   movea.l #ligne,a5       * creer pointeur
16
17 00000016 3F3C0001               entree: move.w  #1,-(sp)        * code: CONIN
18 0000001A 4E41                           trap    #1              * appel de GEMDOS
19 0000001C 584F                           addq.l  #2,sp           * correction pile
20
21 0000001E 1A00                           move.b  d0,(a5)+        * sauvegarder caractere
22
23 00000020 0C000030                       cmpi.b  #"0",d0         * caractere etait un CR ?
24 00000024 66F0                           bne     entree          * NON: caractere suivant
25
26 00000026 BBFD00000007                   cmpa.l  #ligne+1,a5     * tester si ligne vide
27 0000002C 67000088                       beq     fin             * OUI: fin du programme
28
29 00000030 2A7D00000006                   movea.l #ligne,a5       * restaurer pointeur
30
31 00000036 4207                           clr.l   d7              * champ resultat
32 00000038 4206                           clr.l   d6              * champ calcul (reste)
33
34 0000003A 1C18                   convers: move.b  (a5)+,d6       * traiter chiffre
35 0000003C 04060030                       subi.b  #$30,d6         * de ASCII a BCD
36
37 00000040 0C060009                       cmpi.b  #9,d6           * chiffre BCD trop grand?
38 00000044 6A1C                           bhi     hexcnt          * OUI: plus de chiffre
39
40 00000046 CEFC000A                       mulu.w  #10,d7          * decalage d'un chiffre
41 0000004A DE86                           add.l   d6,d7           * additionner chiffre
42
43 0000004C 0C07FFFF                       cmpi.b  #$ffff,d7       * tester depassement
44 00000050 63E8                           bls     convers         * NON: nouveau chiffre
45
46 00000052 6100                           bsr     crlf            * curseur sur ligne suivante
47
48 00000056 3F3C2021                       move.w  #" !",-(sp)     * message d'erreur
49 0000005A 3F3C0002                       move.w  #2,-(sp)        * code: CONOUT
50 0000005E 4E41                           trap    #1              * appel de GEMDOS
51 00000060 584F                           addq.l  #4,sp           * correction pile
52
53 00000062 60AE                           bra     repet           * entrer nouveau chiffre
54
55
```

```
56
57 00000062 6150                  hexout: bsr     crlf        * curseur sur ligne suivante
58
59 00000064 3F3C2038              move.w  #" =",-(sp) * message de resultat
60 00000068 3F3C0002              move.w  #2,-(sp)    * code: CONOUT
61 0000006C 4E41                  trap    #1          * appel de GEMDOS
62 0000006E 588F                  addq.l  #4,sp       * correction pile
63
64 00000070 0280FFFFFFFF          andi.l  #$ffff,d7   * limiter chiffres
65
66 00000076 207C00000DB6          movea.l #ligne,a5   * index pointeur
67
68 0000007C 2E07                  dodec:  move.l  d7,d6       * traiter chiffre
69 0000007E 024A000F              andi.w  #$f,d6      * masquer valeur
70 00000082 E84F                  lsr.w   #4,d7       * former reste
71 00000084 06460030              addi.w  #$30,d6     * former ASCII
72
73 00000088 0C4A0039              cmpi.w  #$39,d6     * lettre ?
74 0000008C 6304                  bls     ires        * NON: chiffre OK
75
76 0000008E 06460027              addi.w  #$27,d6     * corriger chiffre
77
78 00000092 1ACa                  ires:   move.b  d6,(a5)+    * chiffre dans buffer
79
80 00000094 0C47000F              cmpi.w  #0,d7       * tous les chiffres ?
81 00000098 66E2                  bne     dodec       * NON: chiffre suivant
82
83 0000009A B8FC00000DB6          sortie: cmpa.l  #ligne,a5   * tester buffer
84 000000A0 67000F5E              beq     rmet        * OUI: fini, suivant
85
86 000000A4 1E25                  move.b  -(a5),d7    * retirer caractere
87 000000A6 024700FF              andi.w  #$ff,d7     * normaliser caractere
88
89 000000AA 3F07                  move.w  d7,-(sp)    * sortir caractere
90 000000AC 3F3C0002              move.w  #2,-(sp)    * code: CONOUT
91 000000B0 4E41                  trap    #1          * appel de GEMDOS
92 000000B2 588F                  addq.l  #4,sp       * correction pile
93
94 000000B4 60E4                  bra     sortie      * tester si fini
95
96
97 000000B6 3F3C0000              fin:    move.w  #0,-(sp)    * code: WARMBOOT
98 000000BA 4E41                  trap    #1          * appel de GEMDOS
99
100
101 000000BC 3F3D000D             crlf:   move.w  #13,-(sp)   * sortir CR
102 000000C0 3F3C0002             move.w  #2,-(sp)    * code: CONOUT
103 000000C4 4E41                 trap    #1          * appel de GEMDOS
104 000000C6 588F                 addq.l  #4,sp       * correction pile
105
106 000000C8 3F3C000A             move.w  #10,-(sp)   * sortir LF
107 000000CC 3F3C0002             move.w  #2,-(sp)    * code: CONOUT
108 000000D0 4E41                 trap    #1          * appel de GEMDOS
109 000000D2 588F                 addq.l  #4,sp       * correction pile
110
111 000000D4 4E75                 rts                 * retour
112
113
114 000000D6                      ligne:  .ds.b   80          * buffer 80 caracteres
115
116
117 00000126                      .end
```

La conversion décimal/hexadécimal commence à nouveau en lignes 8 à 13 par la sortie de "CR/LF" et d'un point d'interrogation comme demande d'entrée. La sortie de "CR/LF" s'effectue dans un sous-programme défini aux lignes 101 à 111. Les lignes 15 à 24 retirent une ligne d'entrée au clavier pour la placer dans le buffer. Celui-ci est défini en ligne 114. On contrôle ensuite (lignes 26 et 27) le buffer pour voir s'il contient une ligne vide, auquel cas le programme s'achève en lignes 97 et 98.

Si des caractères figurent dans le buffer, il est converti en une valeur de registre (D7) binaire par la routine en lignes 29 à 53.

En ligne 57 commence la sortie du nombre hexadécimal qui se trouve maintenant dans le registre D7. Le curseur est d'abord fixé sur le début de la ligne suivante, par la sortie d'un "CR/LF" par notre sous-programme en ligne 101. Un caractère "?" est alors sorti (lignes 59 à 62) pour marquer le début de la sortie hexadécimale et le contenu de D7 est masqué pour ne garder que le mot de moindre valeur (ligne 64).

En ligne 66 commence le traitement d'un chiffre hexadécimal. Une copie du nombre à sortir (en D7) est d'abord transférée dans un registre auxiliaire (D6). Dans le registre auxiliaire, les 4 bits de moindre valeur sont masqués (ligne 69). Le nombre à traiter, dans le registre D7, est décalé de 4 positions de bit vers la droite (ligne 70), puisque ces bits se trouvent maintenant dans le registre auxiliaire D6. Ces bits sont convertis en un chiffre hexadécimal conformément à notre procédure de calcul.

En ligne 71 est additionnée la constante $30 et on teste ensuite s'il s'agit d'un chiffre de 0 à 9 (ligne 73). Si c'est le cas, le registre D6 peut être sorti comme chiffre hexadécimal (à partir de la ligne 78). Dans le cas contraire, la constante $27 est encore additionnée (ligne 76).

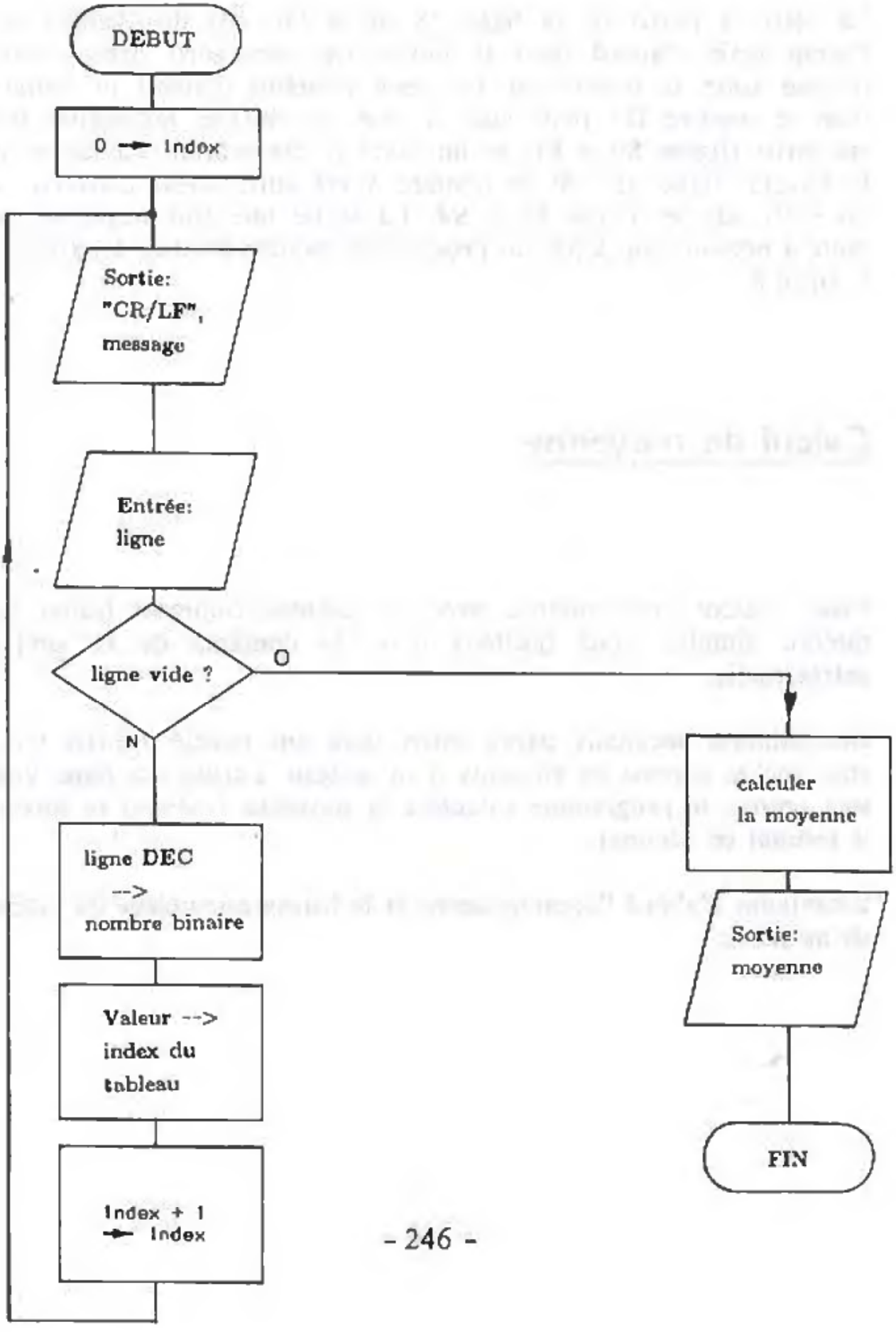
La sortie à partir de la ligne 78 ne se fait pas directement sur l'écran mais d'abord dans le buffer qui sera sorti dans l'ordre inverse après la conversion. On teste toutefois d'abord le nombre dans le registre D7 pour voir si tous les chiffres nécessaires ont été sortis (lignes 80 à 81) et on saute le cas échéant au début de la boucle (ligne 68). Si le nombre a été entièrement converti, il est sorti par les lignes 83 à 94. La sortie une fois terminée, on saute à nouveau au début du programme (boucle d'entrée à partir de la ligne 8).

Calcul de moyenne

Nous voulons vous montrer avec cet exemple comment traiter un tableau simple. Nous quittons donc le domaine de la simple entrée/sortie.

Des nombres décimaux seront entrés dans une boucle d'entrée pour être stockés comme les éléments d'un tableau. Lorsqu'une ligne vide sera entrée, le programme calculera la moyenne (entière) et sortira le résultat en décimal.

Examinons d'abord l'organigramme et le listing assembleur du calcul de moyenne:


DEBUT
0 → Index
Sortie:
"CR/LF",
message
Entrée:
ligne
ligne vide ?
O
N
calculer
la moyenne
Sortie:
moyenne
FIN
ligne DEC
-->
nombre binaire
Valeur -->
index du
tableau
Index + 1
→ Index

```
1
2
3                                  ********************************************
4                                  *** Calcul de moyenne                Exemple 3 ***
5                                  ********************************************
6
7
8  00000000 2B7C00000142           move.l  #tab,a4     * intervale nombres
9
10 00000006 613000B4       repet:  bsr     crlf        * ligne suivante
11
12 0000000A 3F3C002F               move.w  #'/',-(sp)  * caractere message
13 0000000E 3F3C0002               move.w  #2,-(sp)    * code: CONOUT
14 00000012 4E41                   trap    #1          * Appel de BEMDOS
15 00000014 588F                   addq.l  #4,sp       * correction pile
16
17 00000016 2A7C00000072           move.l  #ligne,a5   * creer pointeur
18
19 0000001C 3F3C0001       entrer: move.w  #1,-(sp)    * code: CONIN
20 00000020 4E41                   trap    #1          * appel de BEMDOS
21 00000022 548F                   addq.l  #2,sp       * correction pile
22
23 00000024 1AC0                   move.b  d0,(a5)+    * sauvegarder caractere
24
25 00000026 0C00000D               cmpi.b  #13,d0      * caractere etait un CR ?
26 0000002A 66F0                   bne     entrer      * NON: caractere suivant
27
28 0000002C B8FC00000000           cmpa.l  #ligne+1,a5 * tester si ligne vide
29 00000032 67000062               beq     reponse     * OUI: calculer somme
30
31 00000036 2A7C00000072           movea.l #ligne,a5   * restaurer pointeur
32
33 0000003C 4287                   clr.l   d7          * Champ resultat
34 0000003E 4286                   clr.l   d6          * champ calcul reste
35
36 00000040 1C1B           convert move.b  (a5)+,d6    * traiter chiffre
37 00000042 0406FFD0               subi.b  #$30,d6     * de ASCII a BCD
38
39 00000046 0C060009               cmpi.b  #9,d6       * chiffre BCD trop grand?
40 0000004A 621C                   bhi     traite      * OUI: plus de chiffre
41
42 0000004C E18F0006               asl.w   #8,d7       * decalage d'un chiffre
43 00000050 DE86                   add.l   d6,d7       * additionner chiffre
44
45 00000052 0C87FFFF               cmpi.b  #$ffff,d7   * tester depassement
46 00000058 63E8                   bls     convert     * NON: nouveau chiffre
47
48 0000005A 613A                   bsr     crlf        * curseur sur ligne suivante
49
50 0000005C 3F3C0021               move.w  #'!',-(sp)  * message d'erreur
51 00000060 3F3C0002               move.w  #2,-(sp)    * code: CONOUT
52 00000064 4E41                   trap    #1          * appel de BEMDOS
53 00000066 588F                   addq.l  #4,sp       * correction pile
54
55 00000068 60CE                   bra     repet       * entrer nouveau chiffre
56
57
58 0000006A 3E07           traite: move.w  d7,(a4)+    * valeur dans tableau
59
60 0000006C 60B8                   bra     repet       * entrer nouveau chiffre
61
62
63 0000006E 3F3C0020       reponse move.w  #' ',-(sp)  * message de resultat
64 00000072 3F3C0002               move.w  #2,-(sp)    * code: CONOUT
65 00000076 4E41                   trap    #1          * appel de BEMDOS
66 00000078 588F                   addq.l  #4,sp       * correction pile
67
68 0000007A 0287000FFFFF           andi.l  #$fffff,d7  * limiter chiffres
69
```

```
70 0000007E 2A7C<<0000F2                move.l  #ligne,a5       * lire pointeur
71
72 00000084 2E07                dudec:  move.l  d7,d6           * traiter chiffre
73 00000086 8CFC000A                    divu.w  #10,d6          * calcul valeur/10
74 0000008A 3E06                        move.w  d6,d7           * sauver resultat
75 0000008C 4846                        swap    d6              * former reste
76 0000008E 0646003C                    addi.w  #'0',d6         * former ASCII
77 00000092 1ACA                        move.b  d6,(a5)+        * chiffre dans buffer
78
79 00000094 0C470000                    cmpi.w  #0,d7           * tous les chiffres?
80 00000098 66EA                        bne     dudec           * NON:chiffre suivant
81
82 0000009A B0FC00000000F2              cmpa.l  #ligne,a5       * tester buffer
83 000000A0 6602                        bne     nchif           * NON, tous chiffres!
84
85 000000A2 4E75                        rts                     * fini, retour
86
87 000000A4 1E25                nchif:  move.b  -(a5),d7        * retirer caractere
88 000000A6 0247003F                    andi.w  #$3f,d7         * normaliser caractere
89
90 000000AA 3F07                        move.w  d7,-(sp)        * sortir caractere
91 000000AC 3F3C0002                    move.w  #2,-(sp)        * code: CONOUT
92 000000B0 4E41                        trap    #1              * appel de GEMDOS
93 000000B2 588F                        addq.l  #4,sp           * correction pile
94
95 000000B4 60E4                        bra     sortie          * tester si fini
96
97
98 000000B6 3F3C0001            fini:   move.w  #1,-(sp)        * code: MAUSTAST
99 000000BA 4E41                        trap    #1              * appel de BEMDOS
100
101
102 000000BE 3F3C000B                   move.w  #11,-(sp)       * sortir CS
103 000000C2 3F3C0002                   move.w  #2,-(sp)        * code: CONOUT
104 000000C6 4E41                       trap    #1              * appel de GEMDOS
105 000000C8 588F                       addq.l  #4,sp           * correction pile
106
107 000000CA 3F3C000A                   move.w  #10,-(sp)       * sortir LF
108 000000CE 3F3C0002                   move.w  #2,-(sp)        * code: CONOUT
109 000000D2 4E41                       trap    #1              * appel de BEMDOS
110 000000D4 588F                       addq.l  #4,sp           * correction pile

111
112 000000D6 4E75                       rts                     * retour
113
114
115 000000D8 42B7               annee:  clr.l   (d7)            * annuler annee
116 000000DA 4286                       clr.l   d6              * supprimer nombre
117
118 000000DC B0FC00000142               cmpa.l  #tab,a4         * fini ?
119 000000E2 6306                       bls     moyen           * OUI: calcul moyenne
120 000000E4 5286                       addq.l  #1,d6           * augmenter compteur
121 000000E6 DE64                       add.w   -(a4),d7        * valeur table
122
123 000000E8 60F2                       bra     ann             * nombre suivant
124
125 000000EA D6C6               moyen:  mulu.w  d6,d7           * valeur moyenne
126
127 000000EC 6100000A                   bsr     cr14            * curseur sur ligne suivante
128 000000F0 6100FF7E                   bsr     decout          * sortir resultat
129
130 000000F4 60C4                       bra     fin             * fin du programme
131
132
133 000000F6                    ligne:  .ds.b   80              * buffer 80 caracteres
134
135 00000146                    tab:    .dc.w   100             * 100 valeurs
136
137
138 00000206                            .end
```

Dans cet exemple, seule la section de programme pour déterminer la moyenne arithmétique est nouvelle. La moyenne est calculée par addition de tous les éléments et division de la somme par le nombre d'éléments.

Pour un calcul de moyenne, il n'est pas nécessaire de stocker provisoirement la valeur de chaque élément. Une sommation suffit, dans une boucle d'entrée. Nous allons cependant choisir une méthode un peu plus compliquée, de façon à expliquer la manipulation d'un tableau.

En ligne 8 est mis en place un pointeur sur la zone de données qui est définie en ligne 136. C'est là que les valeurs entrées devront être stockées par mots entiers. La boucle d'entrée commence à partir de la ligne 10. Ici est appelé un sous-programme de sortie de "CR/LF" qui est défini en lignes 102-112. Chaque ligne d'entrée commence par un message initial ("?") qui est produit par les lignes 12 à 15. Les lignes 17 à 26 permettent d'entrer un nombre décimal. Si ce n'est pas une ligne vide qui a été entrée (lignes 28 et 29), l'entrée est convertie en format de registre (vers D7). La routine de conversion des lignes 31 à 55 est bien connue. Le nombre converti est transféré dans le tableau par l'instruction en ligne 58 et le pointeur est dirigé sur le prochain élément. Après que la valeur ait été stockée, l'entrée d'un nouveau nombre commence (ligne 60).

Si une ligne vide est alors entrée, le programme saute en ligne 29 à la ligne 115. C'est ici qu'est véritablement effectué le calcul de moyenne. Les lignes 115 et 116 initialisent des registres pour la somme des éléments et pour le nombre d'éléments. Le tableau est traité du dernier au premier élément. C'est pourquoi les lignes 118-119 comparent, comme critère de fin du travail, le pointeur (A4) au début du tableau. En cas de fin du travail, on saute à la ligne 126.

Dans le cas contraire, le compteur est augmenté de 1 en ligne 121. Notez que le registre d'adresse A4 indique toujours le prochain élément du tableau.

Pour le calcul de la somme des éléments, nous accédons donc en ligne 122 à l'élément avant le pointeur. La boucle se termine par un saut inconditionnel au début de la boucle (ligne 124) où on teste alors si tous les éléments ont été traités.

Une fois tous les éléments additionnés (en D7) et leur nombre déterminé (en D6), la moyenne peut être calculée par division en ligne 126. Le résultat figure à nouveau en D7.

La sortie d'un nombre décimal est dans cet exemple réalisée sous la forme d'un sous-programme qui figure en lignes 63 à 95. L'algorithme utilisé nous est connu depuis les exemples précédents.

La sortie de la moyenne en ligne 128 est précédée de celle de "CR/LF". Le résultat (D7) est ensuite sorti par appel du sous-programme "decout" en ligne 129 et le programme se termine de la manière habituelle (lignes 131, 98 et 99).

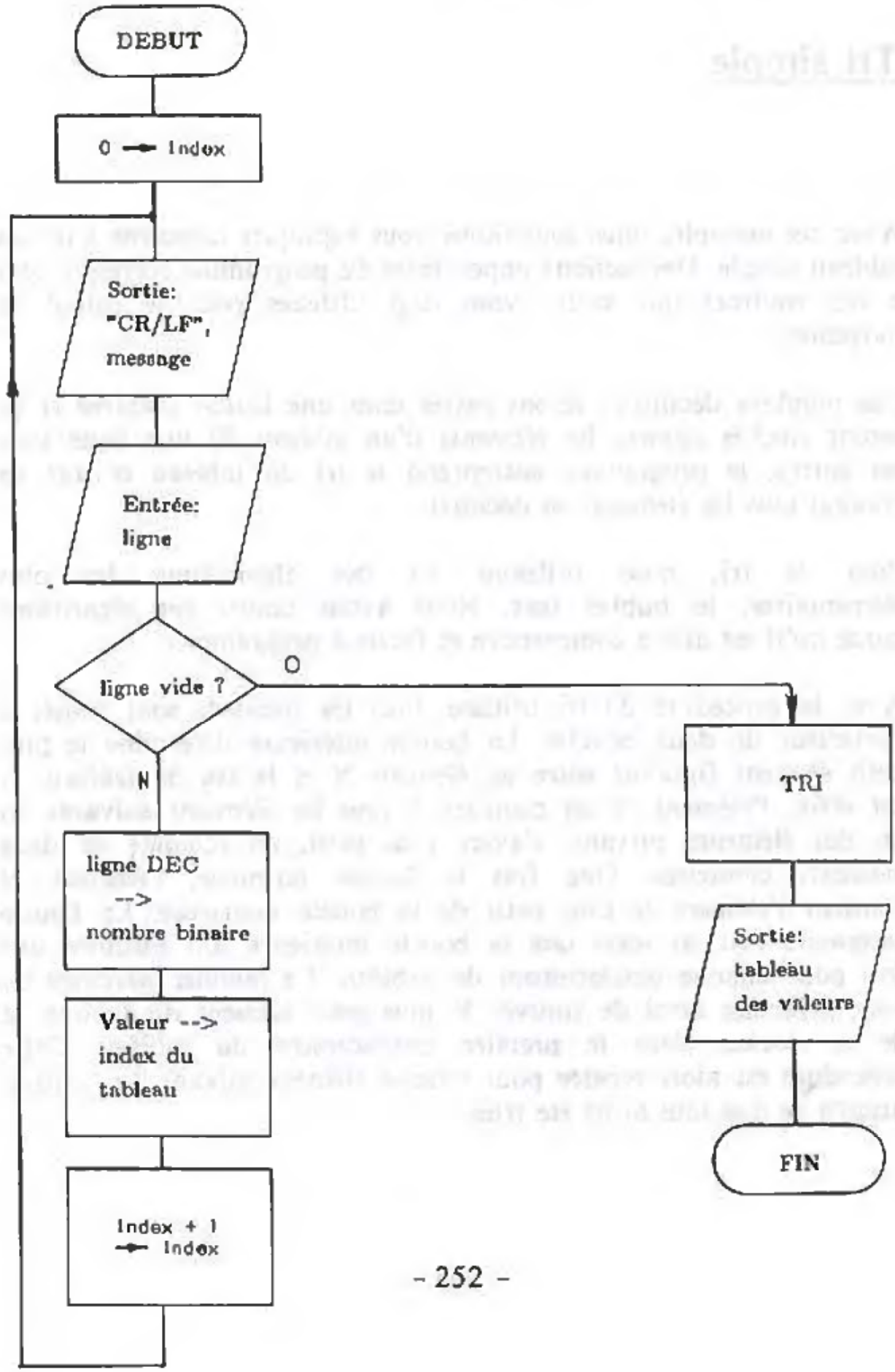
Tri simple

Avec cet exemple, nous souhaitons vous expliquer comment trier un tableau simple. Des sections importantes du programme correspondent à des routines que nous avons déjà utilisées pour le calcul de moyenne.

Des nombres décimaux seront entrés dans une boucle d'entrée et ils seront stockés comme les éléments d'un tableau. Si une ligne vide est entrée, le programme entreprend le tri du tableau et sort en résultat tous les éléments en décimal.

Pour le tri, nous utilisons un des algorithmes les plus élémentaires, le bubbel sort. Nous avons choisi cet algorithme parce qu'il est aisé à comprendre et facile à programmer.

Avec la procédure de tri utilisée, tous les éléments sont traités à l'intérieur de deux boucles. La boucle intérieure détermine le plus petit élément figurant entre un élément N et la fin du tableau. A cet effet, l'élément N est comparé à tous les éléments suivants. Si un des éléments suivants s'avère plus petit, on échange les deux éléments comparés. Une fois la boucle terminée, l'élément N contient l'élément le plus petit de la boucle intérieure. La boucle extérieure fait en sorte que la boucle intérieure soit exécutée une fois pour chaque emplacement du tableau. Le premier parcours de boucle permet ainsi de trouver le plus petit élément du tableau et de le stocker dans le premier emplacement du tableau. Cette procédure est alors répétée pour chaque élément suivant du tableau, jusqu'à ce que tous aient été triés.


DEBUT
0 → Index
Sortie:
"CR/LF",
message
Entrée:
ligne
ligne vide ?
O
N
TRI
ligne DEC
-->
nombre binaire
Valeur -->
index du
tableau
Index + 1
→ Index
Sortie:
tableau
des valeurs
FIN

```
1
2
3                                  ********************************************************
4                                  *** Tri simple de nombres                    Exemple 4 ***
5                                  ********************************************************
6
7
8  00000000 207C00000160                   movea.l  #tab,a4      * intervale nombres
9
10 00000006 61000004       repet:  bsr      cr14          * ligne suivante
11
12 0000000A 3F3C203F                       move.w   #' ?',-(sp)  * caractere message
13 0000000E 3F3C0002                       move.w   #2,-(sp)     * code: CONOUT
14 00000012 4E41                           trap     #1           * appel de GEMDOS
15 00000014 588F                           addq.l   #4,sp        * correction pile
16
17 00000016 2A7C00000110                   movea.l  #ligne,a5    * creer pointeur
18
19 0000001C 3F3C0001       entree: move.w   #1,-(sp)     * code: CONIN
20 00000020 4E41                           trap     #1           * appel de GEMDOS
21 00000022 548F                           addq.l   #2,sp        * correction pile
22
23 00000024 1A00                           move.b   d0,(a5)+     * sauvegarder caractere
24
25 00000026 0C00000D                       cmpi.b   #13,d0       * caractere etait un CR ?
26 0000002A 6F00                           ble      entree       * NON: caractere suivant
27
28 0000002C B8FC00000111                   cmpa.l   #ligne+1,a5  * tester si ligne vide
29 00000032 67000052                       beq      tri          * OUI: trier
30
31 00000036 2A7C00000110                   movea.l  #ligne,a5    * restaurer pointeur
32
33 0000003C 4287                           clr.l    d7           * champ resultat
34 0000003E 4286                           clr.l    d6           * champ calcul (reste)
35
36 00000040 1C1B           convert: move.b   (a5)+,d6     * traiter chiffre
37 00000042 0406003O                       subi.b   #$30,d6      * de ASCII a BCD
38
39 00000046 0C060009                       cmpi.b   #9,d6        * chiffre BCD trop grand?
40 0000004A 621C                           bhi      traite       * OUI: plus de chiffre
41
42 0000004C C3FC000A                       mulu.w   #10,d7       * decalage d'un chiffre
43 00000050 DE86                           add.l    d6,d7        * additionner chiffre
44
45 00000052 0C07FFFF                       cmpi.b   #$ffff,d7    * tester depassement
46 00000056 63E0                           bls      convert      * NON: nouveau chiffre
47
48 00000058 6162                           bsr      cr14         * curseur sur ligne suivante
49
50 0000005A 3F3C2021                       move.w   #' !',-(sp)  * message d'erreur
51 0000005E 3F3C0002                       move.w   #2,-(sp)     * code: CONOUT
52 00000062 4E41                           trap     #1           * appel de GEMDOS
53 00000064 588F                           addq.l   #4,sp        * correction pile
54
55 00000066 609E                           bra      repet        * entrer nouveau
```

```
56
57 00000080 3B07          traite: move.w   d7,(a4)-    * valeur dans tableau
58
59 00000082 60DA                  bra      repet       * entrer nouveau chiffre
60
61
62
63 00000084 3F3C0009      decout: move.w   #9,-(sp)    * message de resultat
64 00000088 3F3C0002              move.w   #2,-(sp)    * code: CONOUT
65 0000008C 4E41                  trap     #1          * appel de BDOS
66 0000008E 588F                  addq.l   #4,sp       * correction pile
67
68 00000090 C2BC0000FFFF          and.l    #$ffff,d7   * limiter chiffres
69
70 00000096 2A7C00000110          movea.l  #ligne,a5   * init pointeur
71
72 0000009C 2C07          dodec:  move.l   d7,d6       * traiter chiffre
73 0000009E 8CFC000A              divu.w   #10,d6      * calcul valeur/10
74 000000A2 3E06                  move.w   d6,d7       * sauver resultat
75 000000A4 4846                  swap     d6          * former reste
76 000000A6 0606 0030             addi.w   #$30,d6     * former ASCII
77 000000AA 1B86                  move.b   d6,(a5)+    * chiffre dans buffer
78
79 000000AC 0C470000              cmpi.w   #0,d7       * tous les chiffres?
80 000000B0 66EA                  bne      dodec       * NON:chiffre suivant
81
82 000000B2 BBFC00000110  sortie: cmpa.l   #ligne,a5   * tester buffer
83 000000B8 6602                  bne      nchif       * NON: tous chiffres!
84
85 000000BA 4E75                  rts                  * fin, retour
86
87 000000BC 1A25          nchif:  move.b   -(a5),d7    * retirer caractere
88 000000BE 02470FFF              andi.w   #$ff,d7     * normaliser caractere
89 000000C2 3F07                  move.w   d7,-(sp)    * sortir caractere
90 000000C4 3F3C0002              move.w   #2,-(sp)    * code: CONOUT
91 000000C8 4E41                  trap     #1          * appel de BDOS
92 000000CA 588F                  addq.l   #4,sp       * correction pile
93
94 000000CC 60E4                  bra      sortie      * tester si fini
95
96
97 000000CE 3F3C0000      fin:    move.w   #0,-(sp)    * code: WARMSTART
98 000000D2 4E41                  trap     #1          * Appel de BDOS
99
100
101 000000D4 3F3C0009     crlf:   move.w   #13,-(sp)   * sortir CR
102 000000D8 3F3C0002             move.w   #2,-(sp)    * code: CONOUT
103 000000DC 4E41                 trap     #1          * Appel de BDOS
104 000000DE 588F                 addq.l   #4,sp       * correction pile
105
106 000000E0 3F3C000A             move.w   #10,-(sp)   * sortir LF
107 000000E4 3F3C0002             move.w   #2,-(sp)    * code: CONOUT
108 000000E8 4E41                 trap     #1          * Appel de BDOS
109 000000EA 588F                 addq.l   #4,sp       * correction pile
110
```

```
111 000000D4 4E75                       rts                 * retour
112
113
114 000000D6 267C00000160       tri:    movea.l #tab,a3     * 1er index
115
116 000000DC 244B               dotri:  movea.l a3,a2       * 2d index
117
118 000000DE 3E13               next:   move.w  (a3),d7     * registre auxiliaire
119 000000E0 3C12                       move.w  (a2),d6     * registre auxiliaire
120 000000E2 BC47                       cmp.w   d7,d6       * test
121 000000E4 6504                       blo     nochan      * OUI: pas d'echange
122
123 000000E6 3487                       move.w  d7,(a2)     * echanger
124 000000E8 3686                       move.w  d6,(a3)
125
126 000000EA D5FC00000002       nochan: adda.l  #2,a2       * augmenter 2d index
127 000000F0 B5CC                       cmpa.l  a4,a2       * fin du tableau ?
128 000000F2 65EA                       blo     next        * OUI: continuer test
129
130 000000F4 D7FC00000002               adda.l  #2,a3       * augmenter 1er index
131 000000FA B5CC                       cmpa.l  a4,a3       * fin du tableau ?
132 000000FC 65DE                       blo     dotri       * OUI: continuer tri
133
134 000000FE B9FC00000160       montre: cmpa.l  #tab,a4     * fini ?
135 00000104 6380                       ble     fin         * OUI: fin du programme
136
137 00000106 3E24                       move.w  -(a4),d7    * valeur du tableau
138
139 00000108 61B2                       bsr     crlf        * ligne suivante
140 0000010A 6100FF60                   bsr     decout      * sortir valeur
141
142 0000010E 60EE                       bra     montre      * nombre suivant
143
144
145
146 00000110                    ligne:  .ds.b   80          * buffer 80 caracteres
147
148 00000160                    tab:    .ds.w   100         * 100 valeurs
149
150 000002C8                    hlp:    .ds.l   1           * memoire auxiliaire
151
152
153 000002CC                            .end
```

En ligne 8 est mis en place un pointeur sur la zone de données qui est définie en ligne 149. C'est là que les valeurs entrées devront être stockées par mots entiers. La boucle d'entrée commence à partir de la ligne 10. Ici est appelé un sous-programme de sortie de "CR/LF" qui est défini en lignes 101-111. Chaque ligne d'entrée commence par un message initial ("?") qui est produit par les lignes 12 à 15. Les lignes 17 à 29 permettent d'entrer un nombre décimal. Si ce n'est pas une ligne vide qui a été entrée (lignes 28 et 29), l'entrée est convertie en format de registre (vers D7). La routine de conversion des lignes 31 à 55 est bien connue. Le nombre converti est transféré dans le tableau par l'instruction en ligne 57 et le pointeur est dirigé sur le prochain élément. Après que la valeur ait été stockée, l'entrée d'un nouveau nombre commence (ligne 59).

Si une ligne vide est alors entrée, le programme saute en ligne 29 à la ligne 114. C'est ici qu'est effectuée l'opération de tri.

Le tri s'étend de la ligne 114 à la ligne 133. La boucle intérieure est constituée par les lignes 114 à 129. La ligne 114 crée un pointeur pour les boucles de tri et le dirige sur le premier élément du tableau. En ligne 116, ce pointeur est copié pour former le pointeur pour la boucle intérieure. Les lignes 124 à 132 comparent l'élément de la boucle extérieure à celui de la boucle intérieure en procédant à un échange le cas échéant (lignes 124 et 125). Le compteur de la boucle intérieure est augmenté par les lignes 127 à 129 et la boucle est à nouveau parcourue à partir de la ligne 118 tant que le dernier élément n'a pas été atteint. Une fois la boucle intérieure terminée, l'élément indiqué par le pointeur en A3 contient la plus petite valeur. La boucle extérieure est répétée par les lignes 131 à 133 jusqu'à ce que tous les éléments aient été traités.

```
4 3 2 5 1  Les deux pointeurs sont dirigés sur le même élément
^
4 3 2 5 1  Les éléments sont échangés
↑↑
3 4 2 5 1  Les éléments sont échangés
↑  ↑
2 4 3 5 1  OK, pas d'échange
↑    ↑
2 4 3 5 1  Les éléments sont échangés
↑      ↑
1 4 3 5 2  Les deux pointeurs sont dirigés sur le même élément
  ^
1 4 3 5 2  Les éléments sont échangés
  ↑↑
1 3 4 5 2  OK, pas d'échange
  ↑  ↑
1 3 4 5 2  Les éléments sont échangés
  ↑    ↑
1 2 4 5 3  Les deux pointeurs sont dirigés sur le même élément
    ^
1 2 4 5 3  OK, pas d'échange
    ↑↑
1 2 4 5 3  Les éléments sont échangés
    ↑  ↑
1 2 3 5 4  Les deux pointeurs sont dirigés sur le même élément
      ^
1 2 3 5 4  Les éléments sont échangés
      ↑↑
1 2 3 4 5  Tous les éléments ont été triés
```

L'illustration ci-dessus devrait vous aider à bien comprendre le déroulement du tri.

Une fois tous les éléments triés, ceux-ci sont sortis dans une boucle (lignes 135 à 143). La sortie d'un nombre décimal est dans cet exemple réalisée sous la forme d'un sous-programme qui figure en lignes 63 à 94. L'algorithme utilisé nous est connu depuis les exemples précédents.

La sortie d'un élément en ligne 140 est précédée de celle de "CR/LF". L'élément est ensuite sorti par appel du sous-programme "decout" en ligne 141. Du fait de ce mode de sortie, c'est d'abord l'élément le plus grand qui est sorti. La boucle se termine par l'instruction de saut inconditionnelle de la ligne 143. Une comparaison en lignes 135 et 136 permet de déterminer si tous les éléments ont été sortis, auquel cas le programme se termine de la manière habituelle (lignes 136, 97 et 98).

Sortie: chaînes de caractères

Dans cet exemple et les exemples suivants, nous nous attacherons à vous montrer l'utilisation d'autres routines du système d'exploitation.

Avec la fonction 9 de GEMDOS, nous allons sortir à l'écran toute une chaîne de caractères qui sera produite comme constante dans le programme assembleur. Après la sortie, le programme devra attendre que soit actionnée une touche quelconque. Nous utiliserons pour cela la fonction 7 de GEMDOS. Celle-ci est exactement identique à la fonction 2 de GEMDOS, si ce n'est que le caractère entré ne sera pas représenté à l'écran. Il est ainsi possible de recevoir avec la fonction 7 de GEMDOS même des caractères "non imprimables" (tels que CTRL-C etc.).

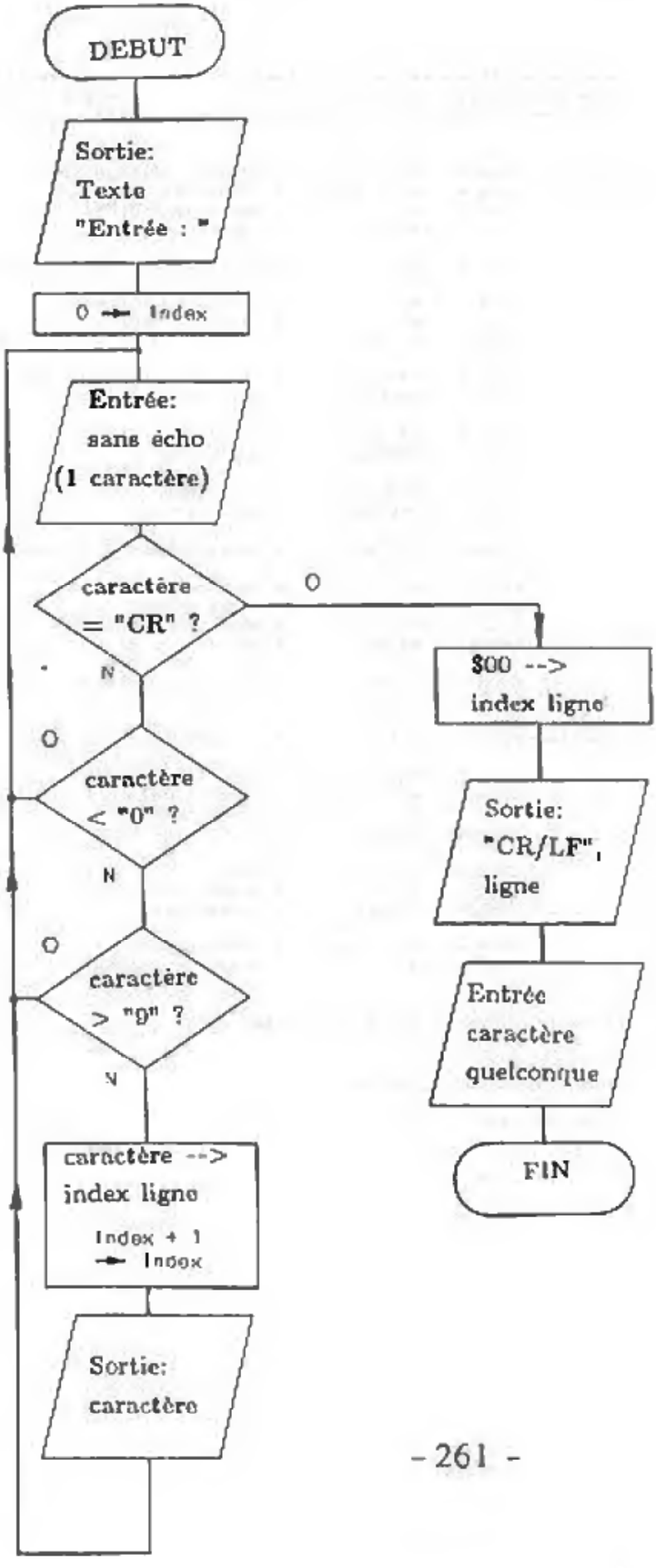
En ligne 8, l'adresse de la chaîne de caractères à sortir est placée, en format de long mot, sur la pile. En ligne 9 vient le code de fonction. L'appel de GEMDOS se fait comme d'habitude. La routine GEMDOS sort les caractères à partir de l'adresse transmise. Elle s'arrête lorsqu'elle rencontre un caractère $00. A part cela, tous les codes, y compris par exemple CR ou LF sont autorisés. La pile doit être corrigée de 6 octets en ligne 11 (à cause du format long mot). Les lignes 13 à 15 appellent une forme spéciale de l'entrée console avec laquelle le caractère entré n'apparaît pas à l'écran mais est transmis à travers le registre D0. Les lignes 17 et 18 terminent le programme. En lignes 21 et 23 est produit le texte à sortir.

```
1
2
3                ********************************************
4                *** Sortie d'une chaine de caracteres   Exemple 5 ***
5                ********************************************
6
7
8  00000000 2F3C0000001C debut:  move.l   #txt,-(sp)  * adresse de la chaine
9  00000006 3F3C0009             move.w   #9,-(sp)    * code: PRTLINE
10 0000000A 4E41                 trap     #1          * appel de GEMDOS
11 0000000C 508F                 addq.l   #8,sp       * correction pile
12
13 0000000E 3F3C0007             move.w   #7,-(sp)    * code: CONIN direct
14 00000012 4E41                 trap     #1          * appel de GEMDOS
15 00000014 548F                 addq.l   #2,sp       * correction pile
16
17 00000016 3F3C0000             move.w   #0,-(sp)    * code: WARMSTART
18 0000001A 4E41                 trap     #1          * appel de GEMDOS
19
20
21 0000001C 426F6E6A6F757220 txt:  .dc.b "Bonjour, chers amis !!!"
21 00000024 2063686572732061
21 0000002C 6D69732021212121
22
23 00000033 0D0A00            .dc.b 13,10,0        * CR/LF et marque de fin
24
25 00000036                   .end
```

Entrée: chaîne de caractères avec contrôle

Avec la fonction 9 de GEMDOS, nous allons sortir sur l'écran un texte entré au clavier. La chaîne de caractères à sortir sera définie comme constante dans le programme assembleur. A la suite de la sortie, le programme lira en entrée un nombre décimal, seules les touches numériques et la touche RETURN (CR) étant autorisées. Nous utiliserons à cet effet la fonction 7 de GEMDOS car avec cette fonction d'entrée la touche appuyée n'est pas sortie automatiquement à l'écran. C'est pourquoi nous pouvons d'abord tester la validité de la touche entrée et l'ignorer en cas d'erreur. Cette méthode nous oblige cependant à nous charger nous même de la sortie d'une touche valable. Pour contrôler si une entrée a été correctement effectuée, nous sortirons le nombre décimal après qu'il ait été entré, terminant ainsi le programme.

Vous trouverez sur les pages suivantes l'organigramme et le listing assembleur du programme.


DEBUT
Sortie:
Texte
"Entrée : "
0 → Index
Entrée:
sans écho
(1 caractère)
caractère
= "CR" ?
O
N
$00 -->
index ligne
O
caractère
< "0" ?
N
Sortie:
"CR/LF",
ligne
O
caractère
> "9" ?
N
Entrée
caractère
quelconque
caractère -->
index ligne
Index + 1
→ Index
FIN
Sortie:
caractère

```
1
2
3                                  ********************************************
4                                  *** Entree avec controle          Exemple 6 ***
5                                  ********************************************
6
7
8  00000000 2F3C00000056   debut:  move.l  #txt,-(sp)  * adresse message debut
9  00000006 3F3C0009               move.w  #9,-(sp)    * code PRTLINE
10 0000000A 4E41                   trap    #1          * appel de GEMDOS
11 0000000C 5C8F                   addq.l  #6,sp       * correction pile
12
13 0000000E 2A7C00000067           move.l  #tmpbuf,a5  * pointeur sur buffer texte
14
15 00000014 3F3C0007    entree: move.w  #7,-(sp)    * code CONIN direct
16 00000018 4E41                   trap    #1          * appel de GEMDOS
17 0000001A 548F                   addq.l  #2,sp       * correction pile
18
19 0000001C 0C000020               cmpi.b  #$20,d0     * caractere etait un CR ?
20 00000020 671A                   beq     sortie      * OUI: sortie ligne
21
22 00000022 0C000039               cmpi.b  #$39,d0     * CARACTERE > 9 ?
23 00000026 6CEC                   bht     entree      * OUI: ignorer
24
25 00000028 0C000030               cmpi.b  #$30,d0     * caractere < 0 ?
26 0000002C 65E6                   blo     entree      * OUI: ignorer
27
28 0000002E 1AC0                   move.b  d0,(a5)+    * sauvegarde caractere
29
30 00000030 3F00                   move.w  d0,-(sp)    * sortir caractere
31 00000032 3F3C0002               move.w  #2,-(sp)    * code CONOUT
32 00000036 4E41                   trap    #1          * appel de GEMDOS
33 00000038 588F                   addq.l  #4,sp       * correction pile
34
35 0000003A 60D8                   bra     entree      * caractere suivant
36
37
38 0000003C 4215        sortie: clr.b   (a5)        * marque fin de ligne
39
40 0000003E 2F3C00000049           move.l  #outbuf,-(sp) * adresse buffer
41 00000044 3F3C0009               move.w  #9,-(sp)    * code PRTLINE
42 00000048 4E41                   trap    #1          * appel de GEMDOS
43 0000004A 5C8F                   addq.l  #6,sp       * correction pile
44
45 0000004C 3F3C0007               move.w  #7,-(sp)    * code CONIN direct
46 00000050 4E41                   trap    #1          * appel de GEMDOS
47 00000052 548F                   addq.l  #2,sp       * correction pile
48
49 00000054 3F3C0000               move.w  #0,-(sp)    * code WARMSTART
50 00000058 4E41                   trap    #1          * appel de GEMDOS
51
52
53 0000005A 0D0A45AE747F45A5 txt:  .dc.b   $0d,$0a,'Entrez ',$0
54 00000062 3A2000
55
56 00000065 0D0A        outbuf: .dc.b   $0d,$0A
57
58 00000067             tmpbuf: .ds.b   40
59
60 0000008F                     .end
```

Les lignes 8 à 11 sortent le texte initial à l'écran et la ligne 13 met en place le pointeur sur le buffer d'entrée. Les lignes 15 à 17 effectuent l'entrée d'un caractère mais sans sortie à l'écran. Si c'était un "CR", l'entrée est terminée et le buffer d'entrée est sortie à partir de la ligne 38.

Les lignes 22 à 26 testent s'il s'agit d'un caractère valable. Si ce n'est pas le cas, on saute au début de la boucle d'entrée (ligne 15). Ce n'est que s'il s'agit d'un caractère valable qu'il est transféré dans le buffer d'entrée et que le pointeur est augmenté (ligne 28). Le caractère valable doit ensuite encore être sorti à l'écran pour que l'utilisateur voit que son entrée a été acceptée (lignes 30 à 33). Ceci fait, l'instruction de saut inconditionnelle nous ramène au début de la boucle d'entrée (ligne 35).

La sortie du nombre décimal dans le buffer d'entrée commence en ligne 38. Nous utilisons ici la fonction de sortie d'une chaîne de caractères entière. Nous devons auparavant marquer la fin du buffer d'entrée avec un $00 (ligne 38). Notez que pour la sortie (lignes 40 à 43), nous n'indiquons pas l'adresse du buffer d'entrée mais celle du buffer de sortie (lignes 55 et 57). Il s'agit d'une astuce simple pour pouvoir sortir encore un CR/LF avant la sortie proprement dite. Si vous examinez les déclarations à partir de la ligne 53, vous constaterez que le buffer de sortie renferme le buffer d'entrée puisque le buffer de sortie n'a pas été marqué par un $00 comme critère de fin.

Les lignes 45 à 47 attendent encore une entrée quelconque au clavier avant que le programme ne se termine par les instructions des lignes 49 et 50.

Sortie: date

Nous souhaitons vous montrer dans cet exemple l'utilisation de la fonction 42 de GEMDOS qui vous permet d'utiliser la date du jour dans vos programmes.

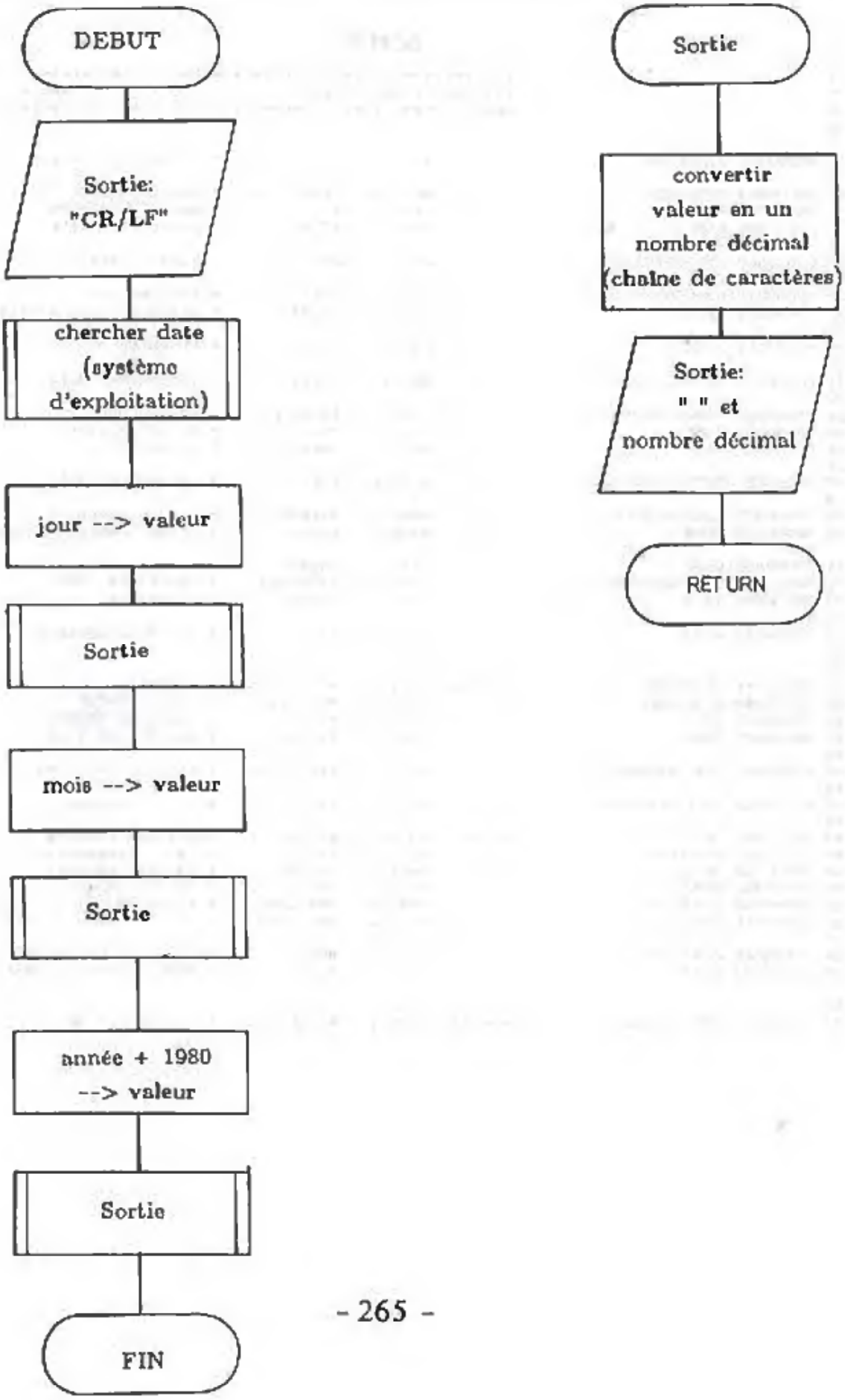
Le programme de notre exemple lira simplement la date actuelle et la sortira à l'écran, en décimal, sous la forme jour-mois-année. Nous utilisons ici les sous-programmes déjà bien connus de sortie d'un nombre décimal et de production d'un passage à la ligne.

Nous allons auparavant expliquer brièvement le fonctionnement de l'interrogation de la date. Après appel de la fonction DATE, GEMDOS fournit la date du jour, codée en binaire, dans le registre D0. La signification des bits de ce code est la suivante:

bits 0 à 4 jour, compris entre 1 et 31 en binaire
bits 5 à 8 mois, compris entre 1 et 12 en binaire
bits 9 à 15 année, compris entre 0 et 119 en binaire

L'année se réfère aux années postérieures à 1980. Pour obtenir l'année réelle, il faut donc additionner la constante "1980" ou "80" au champ de l'année.

Vous trouverez sur les pages suivantes l'organigramme et le listing assembleur du programme.


DEBUT
Sortie: "CR/LF"
chercher date (système d'exploitation)
jour --> valeur
Sortie
mois --> valeur
Sortie
année + 1980 --> valeur
Sortie
FIN
Sortie
convertir valeur en un nombre décimal (chaine de caractères)
Sortie: " " et nombre décimal
RETURN

Sortie

```
                                ********************************************
                                *** Sortie du le date              Exemple 7 ***
                                ********************************************


00000000 A1000074               jsr     crlf        * curseur sur ligne suivante

00000004 3F3C0020               move.w  #$20,-(sp)  * code1 SETDATE
00000008 4E41                   trap    #1          * appel de GEMDOS
0000000A 548F                   addq.l  #2,sp       * correction pile

0000000C 33D000000100           move.w  d0,hlp      * sauver date

00000012 02B000000000           andi.l  #$1f,d0     * traiter jour
00000018 2200                   move.l  d0,d7       * preparer pour sortie

0000001A 6128                   bsr     decout      * et sortir

0000001C 3E3900000100           move.w  hlp,d7      * rechercher date

00000022 02B700000IE0           andi.l  #$1e0,d7    * isoler mois
0000002B EA8F                   lsr.l   #5,d7       * et normaliser
0000002A 6118                   bsr     decout      * et sortir

0000002C 3E3900000100           move.w  hlp,d7      * rechercher date

00000032 02B70000FE00           andi.l  #$fe00,d7   * isoler annee
00000038 7E09                   move.l  #9,d6       * fixer nombre decalages

0000003A EC6F                   lsr.l   d6,d7       * et normaliser (9x)
0000003C 06B700000?BC           addi.l  #1980,d7    * constante 1980
00000042 6102                   bsr     decout      * et sortir

00000044 6046                   bra     fin         * fin du programme


0000004e 3F3C2020               decout: move.w  #"  ",-(sp) * espaces
00000052 3F3C0002                       move.w  #2,-(sp)    * code1 CONOUT
00000056 4E41                           trap    #1          * appel de GEMDOS
00000058 588F                           addq.l  #4,sp       * correction pile

0000005A 02870000FFFF                   andi.l  #$ffff,d7   * limiter chiffres

00000060 2A7C00000000R0                 movea.l #ligne,a5   * fixer pointeur

00000066 2207                   dedec:  move.l  d7,d6       * traiter chiffre
00000068 BCFC000A                       divu.w  #10,d6      * calcul valeur/10
0000006C 3E06                           move.w  d6,d7       * sauver resultat
0000006E 4846                           swap    d6          * former reste
00000070 0646003O                       addi.w  #$30,d6     * former ASCII
00000074 1AC6                           move.b  d6,(a5)+    * chiffre dans buffer

00000076 0C470000                       cmpi.w  #0,d7       * tous les chiffres?
0000007A 66EA                           bne     dedec       * NON:chiffre suivant


0000007C BBFC00000080           sortie: cmpa.l  #ligne,a5   * tester buffer
```

```
58 0000007A 6602                        bne     nchif           * NON: tous chiffres!
59
60 0000007C 4E75                        rts                     * fini, retour
61
62 0000007E 1E25        nchif:  move.b  -(a5),d7        * retirer caractere
63 00000080 024700FF            andi.w  #$ff,d7         * normaliser caractere
64
65 00000084 3F07                        move.w  d7,-(sp)        * sortir caractere
66 00000086 3F3C0002            move.w  #2,-(sp)        * code: CONOUT
67 0000008A 4E41                        trap    #1              * appel de GEMDOS
68 0000008C 588F                        addq.l  #4,sp           * correction pile
69
70 0000008E 60E4                        bra     sortie          * tester si fini
71
72
73 00000090 3F3C0000    fin:    move.w  #0,-(sp)        * code: WARMSTART
74 00000094 4E41                        trap    #1              * appel de GEMDOS
75
76
77 00000096 3F3C000D    crlf:   move.w  #13,-(sp)       * sortir CR
78 0000009A 3F3C0002            move.w  #2,-(sp)        * code: CONOUT
79 0000009E 4E41                        trap    #1              * appel de GEMDOS
80 000000A0 588F                        addq.l  #4,sp           * correction pile
81
82 000000A2 3F3C000A            move.w  #10,-(sp)       * sortir LF
83 000000A6 3F3C0002            move.w  #2,-(sp)        * code: CONOUT
84 000000AA 4E41                        trap    #1              * appel de GEMDOS
85 000000AC 588F                        addq.l  #4,sp           * correction pile
86
87 000000AE 4E75                        rts                     * retour
88
89
90 000000B0             ligne:  .ds.b   80              * buffer 80 caracteres
91
92 00000100             hlp:    .ds.l   1               * memoire auxiliaire
93
94
95 00000104                             .end
```

Les routines de sortie d'un nombre décimal (lignes 38 à 70) et le sous-programme pour "CR/LF" (lignes 37 à 87) sont connus. La sortie de la date occupe les lignes 8 à 35.

CR/LF est d'abord sorti et la fonction GEMDOS, pour déterminer la date du jour, est appelée (lignes 8 à 12). La date du jour en D0 est sauvegardée avant poursuite du traitement (ligne 14). Les lignes 16 à 19 sortent le jour qui est formé simplement par masquage (ligne 16).

La sortie du mois est tout aussi simple (lignes 21 à 25), si ce n'est qu'après le masquage en ligne 23, le résultat doit être étendu à un mot, après quoi le mois est sorti en ligne 25.

Le calcul de l'année est un peu plus compliqué. Après le masquage en ligne 29, la valeur doit être normalisée par plusieurs décalages vers la droite (lignes 30 et 31) car un décalage en une seule fois avec cette instruction n'est possible que sur 7 bits maximum. Nous additionnons ensuite en ligne 32 la constante 1980 pour obtenir l'année correcte.

Les lignes 35, 73 et 74 terminent le programme comme à l'habitude.

Calcul de factorielles

La factorielle de n est définie comme le produit des n premiers nombres naturels:

n! = 1*2*3* ... *n, où 0! est défini comme égal à 1

Exemples:

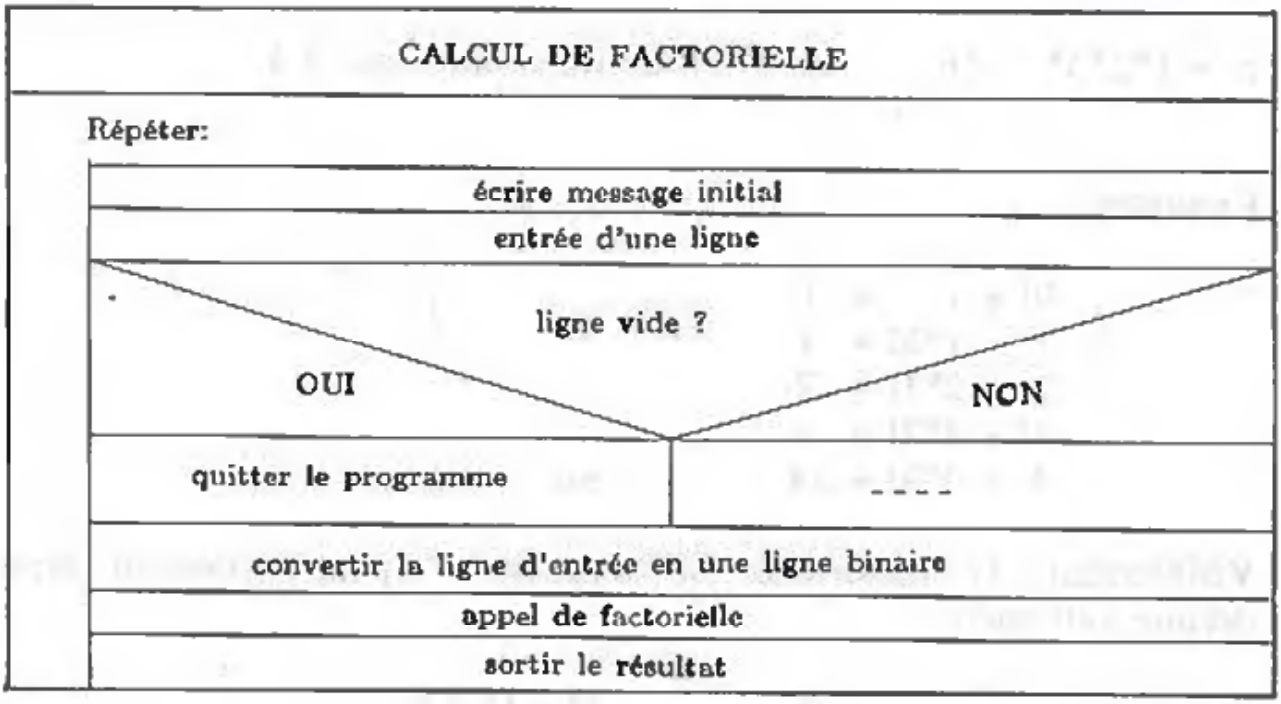
0! = 1 = 1
1! = 1*0! = 1
2! = 2*1! = 2
3! = 3*2! = 6
4! = 4*3! = 24 etc.

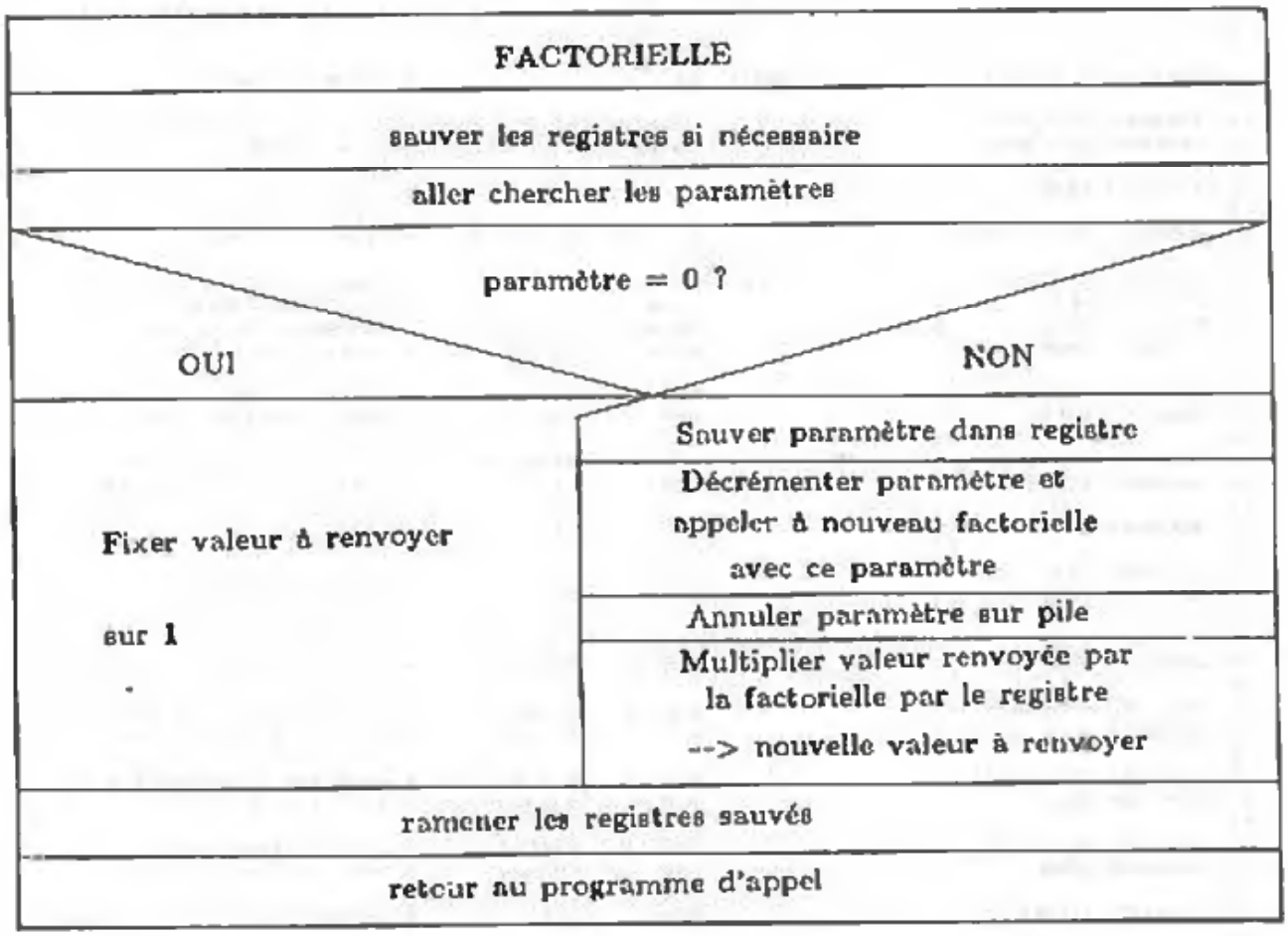
Visiblement, la factorielle du nombre "n" peut également être définie autrement:

0! = 1 et n! = (n-1)! *n

Cette forme de la définition est appelée RECURSIVE. Chaque élément de la séquence est déterminé à travers ses prédécesseurs, on se "raccroche" en quelque sorte au résultat précédent. La récursion est un procédé également utilisé en informatique. On parle d'un programme récursif (par opposition à un programme itératif) lorsqu'une routine s'appelle elle-même directement ou indirectement.

On doit bien entendu veiller à ce que la boucle de ré-appel ne soit pas infinie. Les représentations récursives ont l'avantage d'être relativement faciles à démontrer. De nombreux problèmes n'admettent d'ailleurs que des solutions récursives. Nous ne calculerons pas la factorielle de façon itérative car il s'agit d'un exemple relativement simple de récursion, même si cela constitue la plus grande difficulté du présent ouvrage.




FACTORIELLE
sauver les registres si nécessaire
aller chercher les paramètres
paramètre = 0 ?
OUI
NON
Fixer valeur à renvoyer
sur 1
Sauver paramètre dans registre
Décrémenter paramètre et
appeler à nouveau factorielle
avec ce paramètre
Annuler paramètre sur pile
Multiplier valeur renvoyée par
la factorielle par le registre
--> nouvelle valeur à renvoyer
ramener les registres sauvés
retour au programme d'appel

```
1
2
3                               ********************************************************
4                               *** Calcul de factorielle                    Exemple 8 ***
5                               ********************************************************
6
7
8  00000000 A1000CEA            repet:  bsr     crlf        * ligne suivante
9
10 00000004 3F3C0020                    move.w  #' ',-(sp)  * caractere message
11 00000008 3F3C0002                    move.w  #2,-(sp)    * code CONOUT
12 0000000C 4E41                        trap    #1          * appel de BEMDOS
13 0000000E 588F                        addq.l  #4,sp       * correction pile
14
15 00000010 267C00000100                move.l  #ligne,a3   * creer pointeur
16
17 00000016 3F3C0001            entrer: move.w  #1,-(sp)    * code CONIN
18 0000001A 4E41                        trap    #1          * appel de BEMDOS
19 0000001C 1680                        move.b  d0,(a3)+    * sauvegarder caractere
20 0000001E 548F                        addq.l  #2,sp       * correction pile
21
22 00000020 0B000000D                   cmpi.b  #13,d0      * caractere etait un CR ?
23 00000024 66F0                        bne     entrer      * NON: caractere suivant
24
25 00000026 B6FC00000101                cmpa.l  #ligne+1,a3 * tester si ligne vide
26 0000002C 67000082                    beq     fin         * OUI: fin du programme
27
28 00000030 267C00000100                movea.l #ligne,a3   * restaurer pointeur
29
30 00000036 4287                        clr.l   d7          * champ resultat
31 00000038 4286                        clr.l   d6          * champ calcul (reste)
32
33 0000003A 1C1B            convert: move.b  (a3)+,d6   * traiter chiffre
34 0000003C 0406FFD0                    subi.b  #$30,d6     * de ASCII a DCD
35    *
36 00000040 0C060009                    cmpi.b  #9,d6       * chiffre DCD trop grand?
37 00000044 621E                        bhi     traite      * OUI: plus de chiffre
38
39 00000046 CEFC000A                    mulu.w  #10,d7      * decalage d'un chiffre
40 0000004A DE86                        add.l   d6,d7       * additionner chiffre
41
42 0000004C 0C07FFFF                    cmpi.b  #$ffff,d7   * tester depassement
43 00000050 63E8                        bls     convert     * NON: nouveau chiffre
44
45 00000052 61000092                    bsr     crlf        * curseur sur ligne suivante
46
47 00000056 3F3C2021                    move.w  #'! ',-(sp) * message d'erreur
48 0000005A 3F3C0002                    move.w  #2,-(sp)    * code CONOUT
49 0000005E 4E41                        trap    #1          * appel de BEMDOS
50 00000060 588F                        addq.l  #4,sp       * correction pile
51
52 00000062 60A0                        bra     repet       * entrer nouveau chiffre
53
54
55 00000064 2F07            traite: move.l  d7,-(sp)    * factorielle
```

```
56 00000066 A104                bsr     fact            * calculer
57 00000068 5C8F                addq.l  #6,sp           * paramètre de pile
58 0000006A 6020                bra     dvbout          * et sortir
59
60 0000006C 48E40000    fact:   movm.l  a0,-(sp)        * pile locale
61 00000070 2F05                move.l  d5,-(sp)        * sauver registres
62 00000072 2A7C0000    [illegible]     move.l  4(a4),a5        * retirer paramètre
63 00000076 0C850000    [illegible]     cmpi.l  #0,d5           * tester fin
64 0000007C 6712                beq     dvbfac          * OUI: fini
65 0000007E 2005                move.l  d5,d0           * copie pour décrément
66 00000080 5380                subq.l  #1,d0           * décrémenter compteur
67 00000082 2F00                move.l  d0,-(sp)        * nouveau paramètre
68 00000084 A1EA                bsr     fact            * récursion
69 00000086 5C8F                addq.l  #6,sp           * paramètre de pile
70 00000088 CEC5                mulu.w  d5,d7           * calculer n-1!
71
72 0000008A 301F        fdecrets: move.l  (sp)+,d5      * retirer de la pile
73 0000008C 4E5E                unlk    a6              * libérer pile
74 0000008E 4E75                rts                     * nouveau cas: fini
75
76 00000090 7001        endfact:  move.l  #1,d7         * fini; récursion
77 00000092 60F6                bra     fdecret         * traitement final
78
79
80
81 00000094 6150        decout:   bsr     crlf            * curseur ligne suivante
82
83 00000096 3F3C0008            move.w  #8,-(sp)        * message de résultat
84 0000009A 3F3C0002            move.w  #2,-(sp)        * code CONOUT
85 0000009E 4E41                trap    #1              * appel de GEMDOS
86 000000A0 508F                addq.l  #8,sp           * correction pile
87
88 000000A2 0287000001FF        andi.l  #$1ff,d7        * limite chiffres
89 000000A8 207C00000100        move.l  #ligne,a0       * fixer pointeur
90
91 000000AE 2007        dodec:  move.l  d7,d0           * traiter chiffre
92 000000B0 [illegible]         divu.w  #10,d0          * calcul valeur/10
93 000000B4 3E06                move.w  d6,d7           * sauver resultat
94 000000B6 4840                swap    d0              * prendre reste
95 000000B8 0640[illegible]     addi.w  #$30,d6         * ajouter ASCII
96 000000BC 1D86                move.b  d6,-(a6)        * chiffre dans buffer
97 000000BE 0E47[illegible]     cmpi.w  #0,d7           * tous les chiffres?
98 000000C2 66EA                bne     dodec           * NON:chiffre suivant
99
100 000000C4 [illegible]        cmpa.l  #ligne,a6       * tester buffer
101 000000C8 67000F34           beq     repet           * OUI: tous chiffres!
102
103 000000CC 1E25               move.b  -(a5),d7        * retirer caractere
104 000000D0 024700FF           andi.w  #$ff,d7         * annuler caracteres
105
106 000000D4 3F07               move.w  d7,-(sp)        * sortir caractere
107 000000D6 3F3C0002           move.w  #2,-(sp)        * code CONOUT
108 000000DA 4E41               trap    #1              * appel de GEMDOS
109 000000DC 508F               addq.l  #8,sp           * correction pile
110
111 000000DE A0E4               bra     sortie          * tester si fini
112
113
114 000000E0 3F3C0009   finit:  move.w  #9,-(sp)        * code: WORMSTART
115 000000E4 4E41               trap    #1              * appel de GEMDOS
116
117
118 000000E6 3F3C000B   [illegible]     move.w  #11,-(sp)       * sortir CR
119 000000EA 3F3C0002           move.w  #7,-(sp)        * code CONOUT
120 000000EE 4E41               trap    #1              * appel de GEMDOS
121 000000F0 508F               addq.l  #8,sp           * correction pile
122
123 000000F2 3F3C000A           move.w  #10,-(sp)       * sortir LF
124 000000F6 3F3C0002           move.w  #2,-(sp)        * code CONOUT
125 000000FA 4E41               trap    #1              * appel de GEMDOS
126 000000FC 508F               addq.l  #8,sp           * correction pile
127
128 000000FE 4E75               rts                     * retour
129
130
131 00000100            ligne:  .ds.b   9               * buffer 8 caracteres
132
133
134 00000150                    .end
```

Cet exemple comprend beaucoup d'éléments connus. Seul le calcul de la factorielle est nouveau.

En ligne 8 est appelé un sous-programme de sortie de "CR/LF" qui est défini en lignes 119-129. Chaque ligne d'entrée commence par un message initial ("?") qui est produit par les lignes 10 à 13. Les lignes 15 à 23 permettent d'entrer un nombre décimal. Si ce n'est pas une ligne vide qui a été entrée (lignes 25 et 26), l'entrée est convertie en un nombre binaire. Le nombre converti (ligne 55) est transmis à la routine de factorielle à travers la pile. Après calcul de la factorielle, la pile est corrigée et le résultat est sorti en décimal (lignes 82 à 112). Après quoi l'entrée d'un nouveau nombre commence (ligne 8).

Si une ligne vide est alors entrée, le programme saute en ligne 26 à la ligne 115. C'est ici que se termine le programme, par un retour à GEM.

Le calcul de factorielle occupe les lignes 61 à 78.

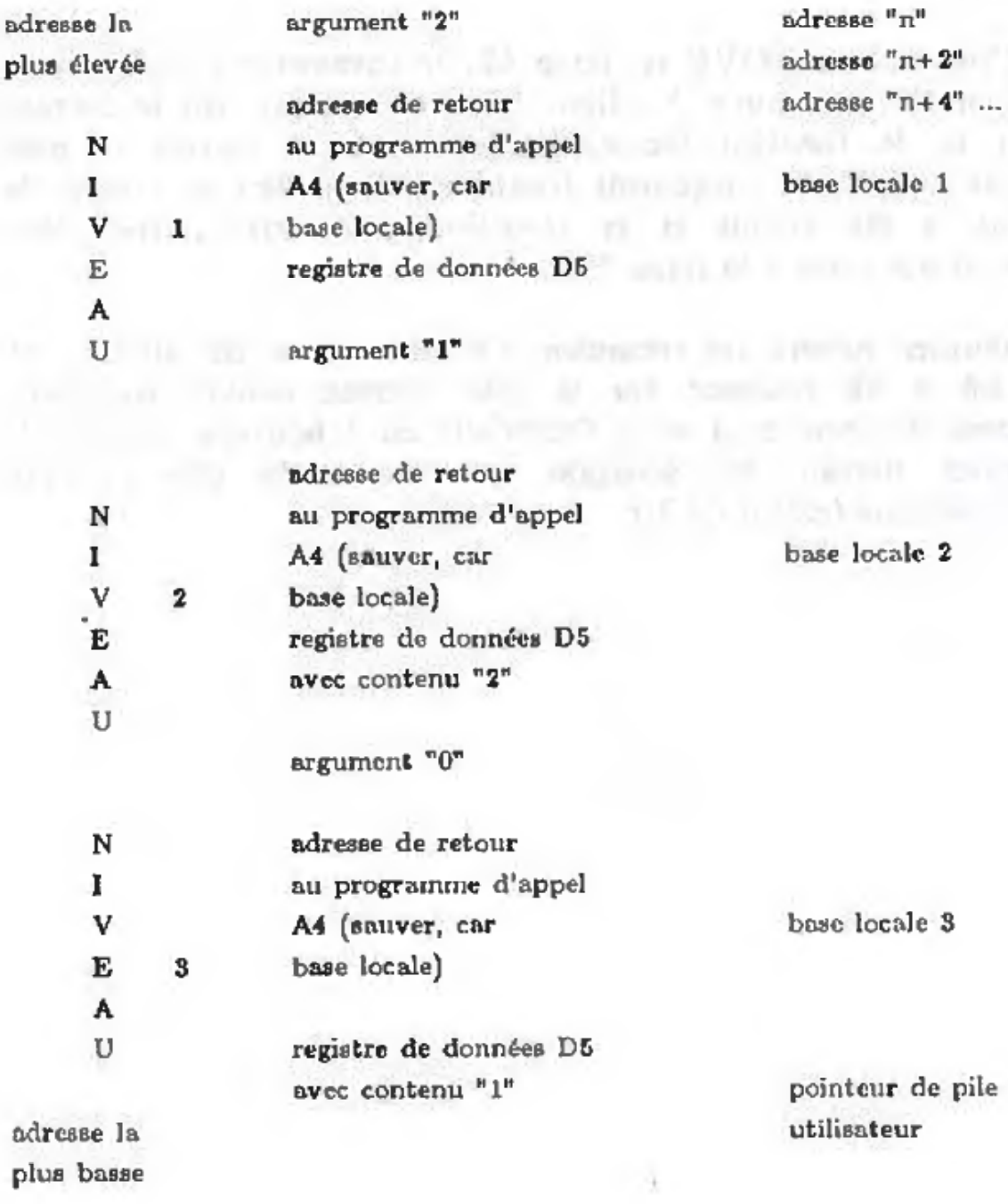
L'instruction LINK permet d'abord de créer ce qu'on appelle une base locale. Lors de l'exécution de cette instruction, un certain nombre d'opérations sont effectuées:
Le contenu du registre A4 est d'abord placé sur la pile. Après cela, la valeur actuelle du pointeur de pile est copiée dans le registre d'adresse A4 qui vient juste d'être sauvé et le pointeur de pile est modifié à raison de la valeur indiquée comme opérande objet.

Lorsqu'on indique une distance négative, le pointeur de pile est décalé vers le bas. Cela crée une zone d'adresses "locale" au milieu de la zone de la pile. Nous n'avons pas besoin de zone locale de pile pour le calcul de la factorielle. C'est pourquoi c'est un #0 qui est indiqué dans l'instruction LINK.

Nous utilisons ici le mécanisme LINK/UNLK pour faciliter la gestion de la pile.

Avec l'instruction MOVE en ligne 62, le paramètre actuel, qui se trouve en D5, est sauvé. En ligne 63, c'est chaque fois le dernier élément de la fonction factorielle qui est lu à travers la base locale de la pile. Si l'argument fourni est 0, le dernier niveau de récursion a été atteint et la récursion peut être quittée dans l'ordre inverse (saut à la ligne 77).

Si le dernier niveau de récursion n'a pas encore été atteint, les lignes 66 à 68 poussent sur la pile, comme nouvel argument, l'argument diminué de 1 et la factorielle est à nouveau appelée. Si le dernier niveau de récursion est atteint, la pile présente l'image suivante (calcul de 2!):


adresse la plus élevée
argument "2"
adresse "n"
adresse "n+2"
adresse de retour
adresse "n+4" ...
N
I
V
E
A
U
1
au programme d'appel
A4 (sauver, car
base locale)
base locale 1
registre de données D5
argument "1"
adresse de retour
au programme d'appel
A4 (sauver, car
base locale)
base locale 2
registre de données D5
avec contenu "2"
argument "0"
adresse de retour
au programme d'appel
A4 (sauver, car
base locale)
base locale 3
registre de données D5
avec contenu "1"
pointeur de pile utilisateur
adresse la plus basse

La récursion est maintenant abandonnée et la pile est détruite par transmission, comme résultat du niveau de récursion, d'un "1" (0! = 1), comme valeur de fonction dans D7, au programme d'appel (lignes 77 et 78). La pile est défaite à partir de la ligne 73 en restaurant l'argument du programme d'appel dans D5 et la pile locale est à nouveau libérée par UNLK. La fonction se termine par un RTS.

Pour autant que la fonction se soit appelée elle-même, le résultat en D7 est multiplié par l'argument et la pile est corrigée (lignes 70 et 71).

En conclusion de ce chapitre, nous vous conseillons de vous livrer à un petit exercice: jouez vous-même au processeur! Vous pouvez exécuter l'exemple du calcul de factorielle sur le papier. Dessinez l'image d'une pile montrant comment le calcul de factorielle défait à nouveau la pile et montrez dans une liste comment les valeurs des registres se modifient. Cela est très intéressant à observer et cela vous aidera en outre à mieux comprendre cet exemple.

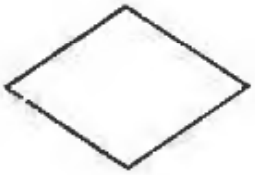
ANNEXE

SYMBOLES D'APRES DIN 66001 (EXTRAITS)

Organigramme:

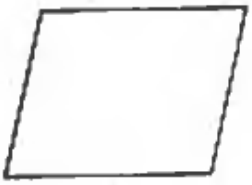
traitement général:

branchement:



sous-programme:

entrée/sortie:



début/fin:
(d'un sous-programme
ou d'un programme)

continuation:

LES CODES DE CONDITION

Pour le test général des flags:

CC if carry clear	si C=0
CS if carry set	si C=1
PL if plus	si N=0
MI if minus	si N=1
VC if overflow clear	si V=0
VS if overflow set	si V=1
NE if not equal	si Z=0
EQ if equal	si Z=1

Après les comparaisons:

EQ if equal	pour OP1 = OP2
NE if not equal	pour OP1 <> OP2

Après les comparaisons de valeurs non-signées:

LO if lower	pour OP2 < OP1
LS if lower same	pour OP2 <= OP1
HI if higher	pour OP2 > OP1
HS if higher same	pour OP2 >= OP1

Après les comparaisons de valeurs signées:

LT less than	pour OP2 < OP1
LE less/equal	pour OP2 <= OP1
GT greater	pour OP2 > OP1
GE greater/equal	pour OP2 >= OP1

En outre:

T	True: la condition est toujours remplie
F	False: la condition n'est jamais remplie

LES MODES D'ADRESSAGE DU 68000

No	Description	Syntaxe	Exemple
1)	registre de données direct	Dn	D3
2)	registre d'adresse direct	An	A3
3)	registre d'adresse indirect	(An)	(A3)
4)	registre d'adresse indirect avec postincrément	(An)+	(A5)+ (SP)+
5)	registre d'adresse indirect avec prédécrément	-(An)	-(A5) -(SP)
6)	registre d'adresse indirect avec distance sur 16 bits	d (An) 16	$1234(A5)
7)	registre d'adresse indirect avec distance sur 8 bits	d (An,Rn) 8	$C0(A1,D1)
8)	absolu court	$xxxx.W	$3000
9)	absolu long	$x..x.L	$12345678
10)	immédiat	#"données"	#$0D
11)	relatif au compteur de programme avec distance sur 16 bits	d (PC) 16	$1000(PC)

12)	relatif au compteur de programme avec distance sur 8 bits et index (registre)	d (PC,Rn) 8	$30(PC,D5)

Rn: un registre de données ou d'adresse quelconque
Dn: un registre de données quelconque
An: un registre d'adresse quelconque

LISTE D'INSTRUCTIONS

Mnémonique	Anglais	Français	Nbre OP	Modes d'adressage autorisés 1	2	3	4	5	6	7	8	9	10	11	12	FLAGS XNZVC
ABCD.X	Add BCD with Extend	Additionner nombres BCD OP2 = OP1 + OP2 + X	2	x				x								*u*u*
ADD.X	Add binary	Additionner nombres binaires OP2 = OP1 + OP2	2	s	s	x	x	x	x	x	x	x	s	s	s	*****
ADDA.W/L	Add binary to Adressreg.	Additionner nombre binaire à registre d'adresse	2	x	x	x	x	x	x	x	x	x	x	x	x	-----
ADDI.X	Add Immediate	Additionner valeur binaire immediate	2	x		x	x	x	x	x	x	x				*****
ADDQ.X	Add Immediate quick	Additionner immédiatement 0..7	2	x	x	x	x	x	x	x	x	x				*****
ADDX.X	Add binary with Extend	Additionner nombres binaires et flag X	2	x				x								*****
AND.X	Logical AND	ET logique	2	s		x	x	x	x	x	x	x	s	s	s	-**00
ANDI.X	Logical AND Immediate	ET logique avec valeur immediate	2	x		x	x	x	x	x	x	x				-**00
ASL,ASR.X	Arithmetic Shift Left/Right	Décalage arithmétique gauche/droite	1/2	x		x	x	x	x	x	x	x				*****
Bcc	Branch conditional	Saut relatif, si condition "cc" vraie	1													-----
BCHG.X	Test Bit and change	Tester bit et écrire son complément	2	x		x	x	x	x	x	x	x				--*---
BCLR.X	Clear Bit	Supprimer bit OP1 dans OP2	2	x		x	x	x	x	x	x	x				--*---
BRA	Branch always	Saut relatif inconditionnel	1													-----
BSET.X	Bit TEST and SET	Tester un bit et le mettre	2	x		x	x	x	x	x	x	x				--*---
BSR	Branch to Subroutine	Saut relatif au sous-programme	1													-----
BTST.X	Bit TEST	Tester un bit	2	x		x	x	x	x	x	x	x				--*---
CHK.W	Check reg. against Bounds	Tester si valeur registre entre deux limites	2	x		x	x	x	x	x	x	x	x	x		*-UUU
CLR.X	Clear	Supprimer bits de l'opérande	1	x		x	x	x	x	x	x	x				-0100
CMP.X	Compare	Comparer operandes	2	x	x	x	x	x	x	x	x	x	x	x	x	-****
CMPA.X	Compare Adressregister	Comparer registres d'adresse	2	x	x	x	x	x	x	x	x	x	x	x	x	-****
CMPI.X	Compare Immediate	Comparer avec valeur immédiate	2	x		x	x	x	x	x	x	x				-****
CMPM.X	Compare with Memory	Comparer à l'intérieur de la mémoire	2			x			x	x	x	x		x	x	-****
DBcc.W	Decrement and Branch	Décrémenter et saut relatif si "cc"	2													-----
DIVS.W	Divide signed	Division avec signe	2	x		x	x	x	x	x	x	x	x	x	x	-***0
DIVU.W	Divide unsigned	Division sans signe	2	x		x	x	x	x	x	x	x	x	x	x	-***0
EOR.X	Logical exclusive OR	OU exclusif logique	2	x		x	x	x	x	x	x	x				-**00
EORI.X	EOR Immediate	OU exclusif logique avec valeur immédiate	2	x		x	x	x	x	x	x	x				-**00
EXG.L	Exchange register	Echanger contenus de registres	2	x	x											-----
EXT.X	Sign Extend	Etendre l'operande avec signe	2	x												-**00
JMP	Jump absolute	Saut à adresse dans l'opérande	1			x			x	x	x	x		x	x	-----
JSR	Jump to Subroutine	Saut au sous-programme	1			x			x	x	x	x		x	x	-----
LEA.L	Load Effective Adress	Charger adresse effective	2			x			x	x	x	x		x	x	-----
LINK	Link local Basepointer	Reserver place pour var. loc. dans pile	2		x											-----
LSL,LSR.X	Logical Shift Left/Right	Decalage logique gauche/droite	1/2	x		x	x	x	x	x	x	x				***0*

Mnemonique	Anglais	Français	Nbre OP	Modes d'adressage autorisés 1	2	3	4	5	6	7	8	9	10	11	12	FLAGS XNZVC
MOVE.X	Move	Transfert de données	2	x	s	x	x	x	x	x	x	x	s	s	s	-**00
MOVEA.W/L	Move to Adreßregister	Charger dans registre d'adresse	2	x	x	x	x	x	x	x	x	x	x	x	x	----
MOVE.X	Move to CCR	Charger dans octet flag du registre d'états	1	x		x	x	x	x	x	x	x	x	x	x	*****
MOVE.X	Move from SR	Retirer registre d'états (ex MOVE SR,D5)	1	x		x	x	x	x	x	x	x				----
MOVE.X	Move to SR	Charger dans registre d'états (ex MOVE #$00,SR)	1	x		x	x	x	x	x	x	x	x	x	x	*****
MOVE.X	Move Userstackpointer	Charger dans/retirer point.pile utilis. (ex MOVE USP,A3)	1		x											----
MOVEM.X	Move multiple registers	Charger dans ou sauver plusieurs registres	2			x	s	d	x	x	x	x		s	s	----
MOVEP.W/L	Move peripheral	Envoyer/retirer données de la périphérie	2	x					x							----
MOVEQ.L	Move Immediate quick	Charger vite valeur immédiate 8 bits dans Dn	1	d												-**00
MULS.W	Multiply signed	Multiplication avec signe	2	x		x	x	x	x	x	x	x	x	x	x	-**00
MULU.W	Multiply unsigned	Multiplication sans signe	2	x		x	x	x	x	x	x	x	x	x	x	-**00
NBCD.B	Negate BCD-Byte	Objet = 0 - objet - X - flag	1	x		x	x	x	x	x	x	x				*U*U*
NEG.X	Negate	Objet = 0 - objet	1	x		x	x	x	x	x	x	x				*****
NEGX.X	Negate with Extend	Objet = 0 - objet - X - flag	1	x		x	x	x	x	x	x	x				*****
NOP	No Operation	Pas d'operation (rien ne se passe)	-													----
NOT.X	Logical NOT	Negation bit par bit (inversion)	1	x		x	x	x	x	x	x	x				-**00
OR.X	Logical OR	OU logique bit par bit	2	x		x	x	x	x	x	x	x	s	s	s	-**00
ORI.X	Logical OR Immediate	OU logique bit par bit avec valeur immédiate	2	x		x	x	x	x	x	x	x				-**00
PEA.L	Push effective Adress	Placer operande sur la pile	1			x			x	x	x	x		x	x	-----
RESET	Reset	Activer canal RESET pour 124 cycles d'horloge	-													----
ROL,ROR.X	Rotate Left/Right	Rotation bit par bit (gauche ou droite)	1/2	x		x	x	x	x	x	x	x				-**0*
ROXL,ROXR.X	Rotate L/R with Extend	Rotation bit par bit avec flag X	1/2	x		x	x	x	x	x	x	x				***0*
RTE	Return from Exception	Retour de la routine d'exception	-													*****
RTR	Return and restore CCR	Retour de sous programme, octet flag de la pile	-													*****
RTS	Return from Subroutine	Retour de sous-programme	-													----
SBCD.B	Sutract BCD with Extend	Soustraire nombres BCD. OP2 = OP2-OP1-X	2	x			x									*U*U*
Scc.B	Set Byte according "cc"	#$FF dans objet si cc, sinon #$00	1	x		x	x	x	x	x	x	x				----
STOP.X	Stop with CCR loaded	Charger dans CCR puis stop	1										x			*****
SUB.X	Subtract binary	Soustraction binaire	2	s	s	x	x	x	x	x	x	x	s	s	s	*****
SUBA.W/L	Subtract bin. from Adressreg.	Soustraction binaire d'un registre d'adresse	2	x	x	x	x	x	x	x	x	x	x	x	x	----
SUBI.X	Subtract Immediate	Soustraction binaire d'une valeur immédiate	2	x		x	x	x	x	x	x	x				*****
SUBQ.X	Subtract quick	Soustraction binaire rapide de #0 _ 7	2	x	x	x	x	x	x	x	x	x				*****
SUBX.X	Subtract with Extend	Objet = objet - source - X - flag	2	x				x								*****
SWAP	Swap register halves	Echanger mots supérieur et inférieur du reg.	1	x												-**00
TAS.B	Test Byte and SET	Tester octet et mettre bit 7	1	x		x	x	x	x	x	x	x				-**00
TRAP	Trap	Declencher exception	1										x			----
TRAPV	Trap on Overflow	Exception si flag depassement mis	-													----
TST.X	Test byte	Tester octet, mot ou long mot	1	x		x	x	x	x	x	x	x				-**00
UNLK	Unlink local area	Libérer place de LINK dans pile	1		x											----

MICRO APPLICATION MICRO APPLICATION

ENTRETIEN ET REPARATION DU VC 1541
Plus de 250 pages.

Ce livre vous permettra de réparer et d'entretenir le lecteur de disquettes Commodore VC 1541. Vous y trouverez amplement décrits ses principes mécaniques et électroniques de ce périphérique. Pour ceux qui ne connaissent pas l'électronique cet ouvrage en est aussi un parfait apprentissage.
Courant Octobre.

Réf. : ME154
Prix : 145 FF

LIVRES DIVERS

ATARI ST TRUCS et ASTUCES
Plus de 250 pages.

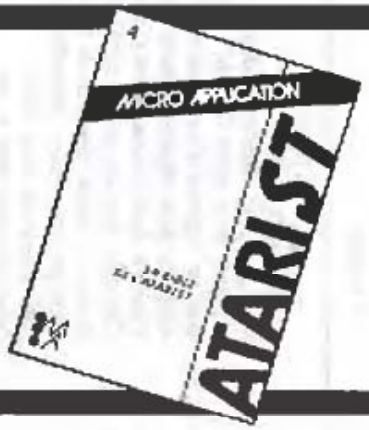
Un recueil complet de trucs et d'astuces que votre tout nouveau ATARI ST va beaucoup apprécier ! Des graphismes fantastiques à partir de programmes en BASIC, des exemples et des conseils pour programmer en langage C et en Assembleur...
Disponible courant Novembre.

Réf. : ME143
Prix : 145 FF

La BIBLE de l'ATARI ST
Plus de 500 pages.

Ce livre contient un ensemble complet d'informations sur l'ATARI ST, la description HARDWARE de la machine ainsi que des schémas détaillés et amplement expliqués notamment des interfaces V24, du port d'extension, de l'interface midi, la structure des graphiques, du BIOS, de GEM, les adresses systèmes importantes, le fonctionnement de la souris...
Disponible courant Novembre.

Réf. : ME142
Prix : 249 FF



LE LIVRE DU LANGAGE MACHINE DE l'ATARI ST
Plus de 250 pages.

Tout ce qu'il faut savoir pour tirer au mieux parti de votre ATARI ST : système de calcul et de bit manipulation du 68000, utilisation des registres, structure des commandes, programmation structurée : récursion, piles, procédures et fonctions, listings sources de programmes assembleurs, routines systèmes... Un super livre !
Disponible Octobre.

Réf. : ME141
Prix : 145 FF

MICRO APPLICATION MICRO APPLICATION

Achevé d'imprimer en novembre 1985
sur les presses de l'imprimerie Laballery et Cie
58500 Clamecy
Dépôt légal : novembre 1985
Numéro d'imprimeur : 511025