ASSEMBLAGE

modélisation, programmation (80x86)

M. MARGENSTERN

Préface de L. NOLIN

MASSON

MANUELS INFORMATIQUES MASSON

MÉTHODES DE PROGRAMMATION ET ALGORITHMIQUE

- INTRODUCTION A LA PROGRAMMATION
  1 Algorithmique et langages J Biondi et G Clavel
  2 Structures de données G Clavel et J Biondi
  3 Exercices corrigés G Clavel et F B Jorgensen
- SCHÉMAS ALGORITHMIQUES FONDAMENTAUX P C Scholl et J P Peyrin
- MATÉRIEL ET LOGICIEL L Nolin
  1 Cours de licence avec exercices
  2 Cours de maîtrise avec exercices

INTRODUCTION A LA PROGRAMMATION SYSTÉMATIQUE N Wirth

- ALGORITHMIQUE ET REPRÉSENTATION DES DONNÉES
  1 Files, automates d'états finis M Lucas, J P Peyrin et P C Scholl
  2 Évaluations, arbres, graphes, analyse de texte M Lucas
  3 Récursivité et arbres P C Scholl
- ALGORITHMIQUE Conception et analyse G Brassard et P Bratley

GÉNIE LOGICIEL M Thorin
MAINTENANCE DU LOGICIEL R L Glass et R A Noiseux
LA TRANSPORTABILITÉ DU LOGICIEL O Lecarme et M Pellissier
ANALYSE FORMELLE D'ALGORITHMES Raisonnements et erreurs dans des algorithmes R Lesuisse
PROCESSUS CONCURRENTS Introduction à la programmation parallèle M Ben-Ari
PROCESSUS SÉQUENTIELS COMMUNICANTS C A R Hoare
LA COMPRESSION DES DONNÉES Méthodes et applications G Held
PRINCIPES DE PROGRAMMATION FONCTIONNELLE H Glaser, C Hankin et D Till
PROGRAMMATION EN LOGIQUE C J Hogger
MISE EN ŒUVRE DES LANGAGES FONCTIONNELS DE PROGRAMMATION S L Peyton Jones
CONSTRUCTION ET VÉRIFICATION DE PROGRAMMES R Backhouse
OUTILS LOGICIELS POUR LA PROGRAMMATION SYSTÈME T J Biggerstaff

LES LANGAGES ET LEUR TRAITEMENT

- LES LANGAGES DE PROGRAMMATION Concepts essentiels, évolution et classification J Lonchamp

INTRODUCTION AU LANGAGE ADA D Price
MANUEL ADA Langage normalisé complet M Thorin
ADA Une introduction H Ledgard
APPRENDRE ET APPLIQUER LE LANGAGE APL B Legrand
APL et GDDM Travail en plein écran B Legrand
PROGRAMMATION EN ASSEMBLEUR. Initiation à partir du Fortran J F Phelizon
LE LANGAGE C B W Kernighan et D M Ritchie
LE LANGAGE C Solutions C L Tondo et S E Gimpel

- LANGAGE C norme ANSI Vers une approche orientée objet Ph Drix

LANGAGE C Problèmes et exercices A R Feuer
PROGRAMMER EN C++ S C Dewhurst et K T Stark
CONSTRUCTION LOGIQUE DE PROGRAMMES COBOL Mise à jour COBOL 85 M Koutchouk
COBOL Perfectionnement et pratique M Koutchouk

- (COMMON) LISP Une introduction à la programmation H Wertz

OCCAM 2 Manuel de référence INMOS
MODULA-2 A B Fontaine
LES LANGAGES DE PROGRAMMATION Pascal, Modula, Chill, Ada Ch Smedema, P Medema, M Boasson
LE LANGAGE PASCAL J M Crozet et D Serain
LANGAGE PL/1 Initiation Perfectionnement R P Balme
POP 11 Un langage adapté à l'intelligence artificielle R Barrett, A Ramsay et A Sloman

- TRAITEMENT DES LANGAGES ÉVOLUÉS Compilation Interprétation Support d'exécution Y Noyelle
- APPRENDRE PASCAL ET LA RÉCURSIVITÉ Avec exemples en TURBO PASCAL R Romanetti

---

• Cours rédigé et enseigné par un professeur francophone

Suite page 3 de couverture

ASSEMBLAGE

modélisation, programmation (80x86)

CHEZ LE MÊME ÉDITEUR

Du même auteur :

LANGAGE PASCAL ET LOGIQUE DU PREMIER ORDRE. *Collection Logique Mathématiques Informatique (n° 3 et 4)* :

Tome 1. — Programmation en Pascal, Prédicats, Systèmes formels, Fonctions récursives. 1989, 304 pages.

Tome 2. — Récursivité et dérécursivication, Preuves et complexité d'algorithme. 1990, 304 pages.

Autres ouvrages :

MATÉRIEL ET LOGICIEL. Par L. NOLIN. *Collection MIM-Algorithmique, Programmation.*

Tome 1. — Cours de licence avec exercices. 1998, 208 pages.

Tome 2. — Cours de maîtrise avec exercices. 1988, 200 pages.

LES MICROPROCESSEURS 80286/80386. Nouvelles architectures PC. Par A.-B. FONTAINE et F. BARRAND. *Collection MIM-Algorithmique, Programmation.* 1987, 320 pages.

PROGRAMMATION STRUCTURÉE EN ASSEMBLEUR 8086 ET 80286. Par J.-P. MALENGÉ, A. HADDAD, L. ANDRÉANI et Ph. COLLARD. 1988, 152 pages.

PROGRAMMATION STRUCTURÉE EN ASSEMBLEUR 6502. Par J.-P. MALENGÉ, L. ANDRÉANI et Ph. COLLARD. 1987, 152 pages.

PROGRAMMATION STRUCTURÉE EN ASSEMBLEUR 68000. Par J.-P. MALENGÉ, S. ALBERTSEN, Ph. COLLARD et L. ANDRÉANI. 1987, 184 pages.

IMPLANTATION DE FONCTIONS USUELLES EN 68000. Par F. BRET. *Collection MIM-Matériel.* 1990, 232 pages.

TRAITEMENT DES LANGAGES ÉVOLUÉS. Compilation, interprétation, support d'exécution. Par Y. NOYELLE. Préface de J. HEBENSTREIT. *Collection MIM-Algorithmique, Programmation.* 1987, 360 pages.

OUTILS LOGICIELS POUR LA PROGRAMMATION SYSTÈME. Par T.-J. BIGGERSTAFF. Traduit de l'anglais par A. KERMARREC. *Collection MIM-Algorithmique, Programmation.*

Voir liste complète de la collection MIM en pages 2 et 3 de couverture.

MANUELS INFORMATIQUES MASSON

ASSEMBLAGE

modélisation, programmation (80x86)

Maurice MARGENSTERN

Maître de conférences
à l'université Paris-Sud
Membre du LITP (Institut Blaise Pascal)

Préface du professeur Louis NOLIN

MASSON Paris Milan Barcelone Bonn **1991**

ISBN : 2-225-**82500-9**

ISSN : 0249-6992

MASSON S.A. 120, bd Saint-Germain, 75280 Paris Cedex 06
MASSON S.p.A. Via Statuto 2/4, 20121 Milano
MASSON S.A. Balmes 151, 08008 Barcelona
DÜRR und KESSLEI Maarweg, 30, 5342 Rheinbreitbach b. Bonn

PREFACE

L'informatique souffre d'un mal que peu de sciences — ou si le mot vous paraît osé, de techniques — ont connu : n'importe quel beau parleur, jugeant d'un coup d'oeil qu'il croit sûr d'après quelques démonstrations à lui habilement présentées, croit pouvoir exprimer hautement sa pensée profonde sur le sujet. Et on l'écoute, surtout s'il sait placer dans son discours deux ou trois termes du jargon que toute activité spécialisée sécrète.

Il faut un peu plus de modestie, à défaut d'honnêteté, pour juger sainement.

S'il y a du chômage en informatique, c'est dans la population des manoeuvres — pourrait-on dire — tous gens recrutés pour un modeste savoir-faire qui leur suffisait tout juste à maîtriser quelque technique simple ou à manipuler tel produit du moment : coder une information explicitement fournie pour des machines à ruban, transcrire en COBOL ce qu'on leur dictait en bon français... pour s'en tenir à des exemples anciens. Certains ont pensé que la situation tiendrait aussi longtemps qu'eux : ils se sont perdus ; sauvés ceux qui se sont recyclés... et qui persévèrent. Car, quoi qu'ils disent parfois, les employeurs ont besoin d'esprits capables de s'adapter au nouveau quotidien ; mieux, de deviner ce que sera demain. Faute de quoi l'entreprise végète si elle ne se ruine pas.

Pour ne vexer personne en place, je rappellerai seulement la mésaventure survenue jadis à un constructeur. Sa machine était plus rapide que celle de son concurrent mais son programme de tri sur bandes bâclé. Le client voyait dans le tri l'essentiel de ce qu'il aurait à faire. Pour choisir, il proposa quelques essais, montre en main. Vous devinez la suite.

On a pu croire parfois la programmation moribonde. Erreur : il faut encore du monde pour rédiger des systèmes experts efficaces et, en deçà, un logiciel de base solide, d'emploi aisé ; en bout de chaîne, des esprits cultivés et audacieux pour en concevoir et réaliser le noyau — tâche réservée aux constructeurs, direz-vous — ou encore l'adapter vraiment aux usages ultérieurs ; domaine beaucoup plus ouvert, inutile de le nier : vous prouveriez seulement que vous n'avez jamais scruté quelques-uns des produits courants de ce genre ;

car vous y auriez vu sans peine les couches que des équipes successives ont superposées ; chaque apport visant à faciliter l'emploi de la couche précédente... et à combler certaines de ses lacunes, quitte à en reprendre des pans entiers. Avec, toujours, une vision très courte du but à atteindre. Tout cela doit être refondu.

Dans beaucoup d'ateliers de mécanique, il reste encore un tour à métaux qui n'est pas muni d'une boîte automatique de filetage. Le tourneur doit donc parfois calculer des trains d'engrenages pour lier la translation de l'outil à la rotation de la broche. C'est la bête noire des candidats au CAP. Leur peine serait moindre s'ils avaient appris plus tôt comment calculer des fractions continues — ce qui ne met en jeu que la division des nombres entiers — et connu de celles-ci une ou deux propriétés essentielles.

Même problème pour les fraiseurs — à ceci près qu'ils ont à coordonner deux rotations — aggravé toutefois par le mystère de la "division différentielle" ; les fractions continues, toujours, avec la connaissance d'une propriété supplémentaire pour pouvoir résoudre l'équation en nombres entiers $ax - by = c$.

La plupart des mécaniciens quelque peu doués parviennent à se tirer d'affaire par l'étude réfléchie ou la pratique continue : c'est que leur art reste, pour l'essentiel, ce que l'*Encyclopédie* de Diderot nous montre dans des planches magnifiques.

L'informaticien praticien ne peut pas espérer s'en tirer à si bon compte. S'il veut pouvoir maîtriser tout ce qui semble chaque jour nouveauté à un esprit inculte, il lui faut remonter aux sources. Aux sources, c'est-à-dire à la notion mathématique de calcul — d'où découlent la gamme des opérations — et à celle, logique, de substitution, dont dérivent les modes d'adressage. Puis, prendre clairement conscience, sans concession, du but à atteindre par ces moyens. En bref, lire cet ouvrage : c'est pour lui, en quelque sorte, une assurance-vie.

Louis NOLIN,
Professeur émérite
à l'université Paris VII.

TABLE DES MATIÈRES

REMERCIEMENTS

Ce livre n'aurait pas vu le jour sans le concours de trois éminents collègues que je tiens à remercier tout particulièrement :

Le préfacier, le professeur Louis NOLIN, Serge GRIGORIEFF et André WARUSFEL.

Lors de la rédaction de l'ouvrage, j'ai bénéficié des encouragements de nombreux collègues du LITP et du Centre d'Orsay. Qu'ils trouvent ici le témoignage de ma reconnaissance.

Bernard GOOSSENS et Jean-Yves MARION, tous deux du LITP, ont bien voulu relire le manuscrit. Le présent texte a bénéficié de leurs suggestions et il serait assurément très imparfait sans le travail d'examen minutieux dont Bernard GOOSSENS s'est aussi chargé et grâce auquel non seulement de nombreuses erreurs ont été éliminées mais aussi plusieurs 'codes' ont été simplifiés et améliorés. Ces quelques mots ne suffisent pas à exprimer ma gratitude ni tout ce que je dois, et s'il reste des erreurs et des imperfections, j'en suis le seul responsable.

COMMENT LIRE CE LIVRE

Ce livre a été conçu pour apprendre le langage d'assemblage de la famille des processeurs 80*x*86. Mais il ne se contente pas de fournir un certain nombre de savoir faire qui permettent de 's'en sortir' dans de nombreuses situations concrètes. Sans être obligé d'entrer dans toutes les 'ficelles du métier', il vise à vous donner des repères, des principes de base, une vision d'ensemble de l'assembleur qui vous permettront non seulement de ne pas vous 'noyer' dans les 'docs' les plus volumineuses, au premier abord excessivement verbeuses, mais de retrouver par vous mêmes nombre d'informations que vous aurez déduites des principes généraux, car ces principes s'appliquent partout de la même façon, qu'il s'agisse des processeurs *INTEL* ou *MOTOROLA* pour ne citer que les plus connus.

La raison en est que tous ces processeurs sont la réalisation physique d'un même modèle, disons le mot, théorique.

L'ouvrage s'articule en deux parties : tout d'abord les moyens et les ressources dont disposent les programmeurs, des concepts théoriques aux outils ; puis la mise en oeuvre de ces moyens dans des situations où le modèle théorique le plus simple débouche sur tous les problèmes concrets de l'informatique.

Après avoir rappelé au chapitre 1 le sens de mots fréquemment utilisés mais pas très souvent définis, l'ouvrage expose au chapitre 2 le modèle théorique. Puis, au chapitre 3, on commence l'apprentissage concret de l'assembleur : il vous est recommandé de vous trouver à proximité d'une machine compatible avec qui vous savez, équipée du système *MS-DOS* (version 2.00 à 3.30) pour vérifier les exemples qui vous sont donnés.

Si vous êtes pressé, vous pouvez commencer la lecture de l'ouvrage par ce chapitre 3. Vous y trouverez une description des instructions du processeur du point de vue de leurs fonctions, présentées dans l'ordre qui découle naturellement des principes exposés au chapitre précédent. Vous pouvez alors avoir envie de vous y reporter, mais vous pouvez aussi continuer la lecture au niveau pratique, laissant à plus tard une seconde lecture, que vos propres réflexions ne manqueront pas d'enrichir. A la fin du chapitre 3, un dictionnaire vous permet de retrouver aisément la syntaxe des instructions. A ce stade, vous pourrez écrire sans problème des programmes en assembleur relativement courts.

Le chapitre 4 vous donne, à partir de quelques exemples simples les principes de la programmation structurée qui constituent le volet *logique* des moyens et ressources dont disposent les programmeurs.

La seconde partie traite du problème fondamental de l'implantation à travers des exemples s'appuyant sur une modélisation concrète du modèle théorique. Cette modélisation, faite au chapitre 5, suppose la lecture de la première partie du chapitre 2.

Le chapitre 6 en fournit une application à la théorie des nombres, occasion de visiter les problèmes posés par l'implantation des opérations élémentaires. Le chapitre 7 fournit l'occasion de revoir les instructions du processeur et certaines fonctions du système d'exploitation, en particulier les mécanismes d'exécution. Vous vous trouverez alors, sans y prendre garde, en pleine 'programmation avancée'. Le chapitre 8 aborde cette autre catégorie d'outils que sont les macro-assembleurs et dont on ne peut véritablement tirer profit que si l'on sait exactement démêler ce qui s'adresse au processeur et ce qui s'adresse au logiciel d'assemblage.

Le chapitre 9 constitue une courte introduction aux nouveaux développements : protection et architecture à jeu d'instruction restreint. Il vous en expose les principes, ce qui vous permettra, ultérieurement, de vous familiariser avec les techniques correspondantes, beaucoup mieux que si vous deviez commencer par 'éplucher' les documentations des fabricants, au demeurant fort précieuses.

La programmation est entièrement tournée vers l'action et toute action conséquente passe par un moment de réflexion. Ceci consiste en premier lieu à prendre du recul par rapport à ce qu'on fait. C'est pourquoi, dans l'ouvrage, ce temps de la réflexion se concrétise par trois chapitres à caractère moins immédiatement concret :

Le chapitre 2 expose d'abord le modèle théorique général : un modèle encore plus simple que la familière calculette programmable à x pas (x pas très élevé). Puis il traite de l'équivalence entre ce modèle et un concept mathématique fondamental et incontournable puisqu'il est dans la nature des choses. Enfin, on aborde un nouveau modèle théorique : la machine de von Neumann au dernier paragraphe de ce chapitre. La relation entre les modèles théoriques a des 'retombées' pratiques très importantes. On déduit d'un des théorèmes que nous énoncerons, la possibilité d'écrire en assembleur un programme d'assemblage. Il vaut la peine de préciser que le théorème date de 1936, à une époque où le mot compilateur n'existait pas encore, et pour cause, et que le premier livre de programmation au sens informatique date de 1951[1].

Il paraît souhaitable de prendre connaissance, le plus tôt possible, des paragraphes 1 et 3 du chapitre 2.

Le chapitre 4 est aussi un moment de réflexion que l'on pourrait qualifier d'*appliquée*. Il est évidemment indispensable d'en lire les deux premiers paragraphes. Le dernier, qui traite de l'implantation de la récursivité en assembleur, peut être remis à une seconde lecture.

Le chapitre 5 s'appuie sur le modèle de machine à registre exposé au chapitre 2. Il constitue un exercice d'implantation pratique, programme à l'appui, de ce modèle théorique. Il représente le point capital du retour à l'action *après* la réflexion.

Dernière illustration : le chapitre 7 expose la construction du codage des instructions machine. Après sa lecture, plus aucune documentation sur un jeu d'instructions n'aura de secret pour vous.

---

[1] Dans cet ouvrage, la machine considérée, certes électronique, fonctionnait avec des lampes, ce qui n'empêchait nullement les programmes qu'il contient de suivre sans toujours le dire, les principes généraux exposés ici.

CHAPITRE 1.

ASSEMBLAGE.

Que signifie le mot *assemblage*? Pourquoi le distinguer du terme plus courant *assembleur*?

Il nous paraît utile de répondre à ces deux questions et de saisir cette occasion pour rappeler la définition de plusieurs termes fréquemment utilisés en informatique. Le lecteur pourra ensuite s'y référer autant que de besoin.

1. Langages.

Un ***programme*** est constitué par des ***instructions*** qui sont elles-mêmes des *mots* dans un certain ***langage***. Un langage est, quant à lui, défini par deux données : d'une part, un ***alphabet*** A, d'autre part, un ensemble de ***règles*** S. L'alphabet est un ensemble fini de *symboles* et on appelle ***mot sur*** A, toute juxtaposition, on dit encore *assemblage*, de symboles appartenant à A.

Les 256 caractères ASCII que tout le monde connaît bien, constituent un exemple d'alphabet et les mots, au sens usuel, qui composent le texte d'un programme contenu dans un fichier ASCII sont des mots au sens ainsi défini.

> Formellement, on définit les mots comme suit : l'absence de symbole est un mot, appelé le mot vide et noté ε ; si m est un mot et s un symbole, ms est aussi un mot. Comme on ne connaît au départ que le mot vide, ou encore mot à zéro symboles, cette définition permet d'obtenir, par récurrence, les mots à un symbole, deux symboles, etc...
>
> On remarquera que le terme *symbole* n'a pas été défini. On ne le fait jamais. Dans les exemples concrets, on donne explicitement la liste complète des symboles.

Tous les mots que l'on peut former sur A n'appartiennent pas nécessairement au langage. Ainsi, en *PASCAL*, *SixZeros* est un mot admissible du langage tandis que *6Zeros* ne l'est pas, car un mot commençant par un chiffre doit être un nombre. Si on peut donner la liste complète des symboles d'un alphabet, il est en général impossible de le faire pour les mots d'un langage, car même si leur nombre est fini, ce nombre peut être très grand. Il faut donc définir des critères permettant de distinguer les mots qui appartiennent au langage de ceux qui n'en font pas partie. C'est la raison d'être des règles. Leur ensemble est encore appelé ***syntaxe du langage***.

Si l'on regarde les mots du langage à leur tour, comme des symboles d'un nouvel alphabet (éventuellement infini), les ***instructions*** d'un programme apparaîtront comme des mots sur cet alphabet étendu, et une partie des règles de S ont pour objet de dire quels *assemblages* de mots de A sont des instructions. En répétant encore ce procédé, on obtient les programmes, mots sur l'alphabet des instructions obéissant à des règles particulières de S. Les règles attachées à la définition d'une instruction constituent sa ***syntaxe***.

Un langage de programmation est donc un langage au sens précédent dont les assemblages ultimes sont les programmes.

Tous les langages de programmation relèvent de cette définition, qu'ils soient dits de haut niveau ou qu'ils soient des langages d'assemblage. Ce qui distingue ces

deux catégories de langage, est leur place dans le dialogue homme-machine que constitue l'exécution, par la machine, d'un programme. En première approximation, les langages de haut-niveau sont les plus proches de l'homme et les langages d'assemblage sont les plus proches de la machine.

Codes.

Comme on le sait, un ordinateur exécute des *instructions*, en général, les unes après les autres. Dans le cas des 8086 d'INTEL, ces instructions parviennent au processeur par groupes de deux fois huit bits dans une file d'attente de 48 bits (six octets). L'exemple ci-dessous donne une idée de ce que reçoit un tel processeur, en considérant que chaque ligne, sauf la dernière, représente une file d'attente pleine :

```
10111001 00010100 00000000 10111110 00000000 00000001
10001010 00010100 10110100 00000010 11001101 00100001
01000110 11100010 11110111 10111000 00000000 01001100
11001101 00100001.
```

Ces instructions sont un exemple de ce que l'on appelle habituellement ***code binaire*** et le langage, dans lequel sont écrites ces instructions, est appelé ***langage machine***. Un code binaire n'est pas très parlant, aussi transcrit-on habituellement ses octets dans un autre *alphabet* comportant 256 symboles. Comme $256 = 2^8$, cet alphabet donne la liste de toutes les valeurs possibles d'un octet. On lui a donné deux formes : la forme d'un ensemble de symboles graphiques dit, jeu des caractères ASCII et la forme de nombres à deux chiffres en base 16 puisque $256 = 16^2$. Les chiffres de la base seize, dits ***hexadécimaux*** sont, par convention : **0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.** Comme $16 = 2^4$, on partage un octet en ses deux quartets, chacun représentant en base deux un chiffre hexadécimal unique. Cette correspondance est fournie par le tableau :

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0000	0001	0010	0011	0100	0101	0110	0111
1000	1001	1010	1011	1100	1101	1110	1111.
8	**9**	**A**	**B**	**C**	**D**	**E**	**F**

Les instructions ci-dessus sont alors représentées par la suite d'octets ci-après, échantillon de ce qui est appelé ***code hexadécimal*** :

```
B9 14 00 BE 00 01 8A 14 B4 02 CD 21 46 E2 F7 B8 00 4C CD 21
```

Ceci est beaucoup plus compact, mais guère plus 'parlant'. C'est pourquoi, pour développer des programmes en langage machine, on a recours à un autre langage dit **symbolique** dont la seule fonction est de traduire exactement sous une forme *humainement acceptable* les instructions représentées sous la forme hexadécimale ou binaire. Ainsi, au code hexadécimal de l'exemple ci-dessus correspond le **code symbolique** suivant (voir au chapitre 4 ce qu'il produit) :

```
0100 B91400          MOV  CX,0014
0103 BE0001          MOV  SI,0100
0106 8A14            MOV  DL,[SI]
0108 B402            MOV  AH,02
010A CD21            INT  21
010C 46              INC  SI
010D E2F7            LOOP 0106
010F B8004C          MOV  AX,4C00
0112 CD21            INT  21
```

Dans cette représentation, on dit souvent en informatique : *sous ce format*, nous donnons le code hexadécimal et le code symbolique en regard l'un de l'autre. Voir au chapitre 3 la raison de la numérotation figurant dans la colonne de gauche.

On a coutume d'utiliser le mot ***code*** pour désigner des programmes sous leur forme hexadécimale ou symbolique. Nous verrons dans les chapitres suivants, notamment aux chapitres 5 et 7 un autre sens de ce mot.

Assembleurs.

Le passage de l'une à l'autre des formes d'un code est, depuis le début de l'informatique, réalisé automatiquement par la machine qui exécute, pour ce faire, un programme particulier nommé ***assembleur***. L'assembleur, en fait, a deux fonctions : il *assemble du code* et il le *désassemble*. L'***assemblage*** consiste à passer du code symbolique au code binaire (ou hexadécimal), et le désassemblage consiste à passer du code hexadécimal (ou binaire) au code symbolique. Le ***langage d'assemblage*** est le langage dans lequel s'écrit le code symbolique et, traditionnellement, l'assembleur vérifie, pendant l'assemblage, la correction syntaxique du code symbolique proposé. En français, on appelle très souvent *assembleur* le langage d'assemblage, ce qui crée une confusion regrettable avec le sens premier que devrait avoir ce mot en informatique, à savoir un *outil* : un *programme* qui assemble. Mais l'abus de langage est tel que nous utiliserons aussi dans cet ouvrage le mot *assembleur* pour le terme *langage d'assemblage*. Nous utiliserons aussi l'expression *langage machine symbolique* de façon synonyme. Quant à l'outil, nous l'appellerons ***constructeur***, à l'image de certains programmes d'exploitation utilisés par les fabricants de processeurs.

> Pour les langages de haut niveau la transformation du code source symbolique, écrit dans le langage de programmation considéré, en code machine binaire est effectuée par un programme généralement appelé ***compilateur***. Un ***interpréteur*** fonctionne de façon différente : il se contente de traduire pas à pas chaque instruction du code source symbolique qu'il transforme en code machine et qu'il exécute de la même manière. Ce mode d'exécution, en général beaucoup plus lent que par un compilateur, se prête mieux au mode conversationnel.

L'essor des langages de haut niveau a conduit à améliorer sans cesse les *constructeurs*, ce qui a fait apparaître un nouveau type de langage : les ***macro-assembleurs***, du nom des programmes qui les mettent en oeuvre. Dans un tel langage, le code symbolique de la page précédente prend l'aspect ci-après :

```
Code      SEGMENT   PARA
          ORG       100h
          ASSUME    CS : Code, DS : Code, ES: Code, SS: Code
Repère    EQU       100h
Quanta    EQU       014h
Début :
          mov       cx,Quanta
          mov       si,Repère
Boucle :
          mov       dl,[si]
          mov       ah,02h
          int       21h
          inc       si
          loop      Boucle
          mov       ax,4C00h
          int       21h
Code      ENDS
          END       Début
```

Dans sa forme, le macro-assembleur se présente comme un mélange d'assembleur et de langage de haut niveau. Nous y reviendrons de façon plus approfondie au chapitre 8. Examinons, à présent les différences entre langages d'assemblage et langages de haut niveau.

Pour ce faire, fixons un instant notre attention sur deux termes courants situés bien en amont du dialogue homme-machine : le mot programme et le mot algorithme.

2. Programmes et algorithmes.

Qu'est-ce qu'un algorithme?

Comme nous en exposerons plusieurs dans cet ouvrage, il nous paraît préférable de dire ce dont il s'agit. Mais la chose est difficile, car la notion d'algorithme ne se laisse pas formaliser, du moins en l'état actuel de la science. Bien que nécessairement approximative, l'idée que nous pouvons en donner est assez précise pour comprendre les rapports entre cette notion et les programmes.

Un algorithme est la description aussi précise que possible de la marche à suivre pour résoudre un problème, conduisant effectivement des données au résultat.

Cette expression : *des données au résultat*, sous-entend aussi qu'un algorithme ne se rapporte pas en fait à un seul *problème*[1], mais à un *ensemble de problèmes* dont les données sont susceptibles de varier dans des limites fixées à l'avance. Le but de l'algorithme est de donner une solution ***uniforme***, valable dans *tous les cas*.

Prenons un exemple élémentaire : l'addition de deux nombres.

Tout le monde connaît l'algorithme de l'addition en base 10. Rappelons-le dans le cadre à peine plus général d'une base b quelconque avec $b \geq 2$. Soient m et n les deux nombres à additionner. On les écrit, respectivement $m = a_k b^k + ... + a_1 b + a_0$, et $n = c_h b^h + ... + c_1 b + c_0$. On peut, sans perte de généralité, supposer que $h \leq k$. Pour chaque i avec $0 \leq i \leq h$, posons $a_i + c_i = u_i + b r_i$, où $r_i \in \{0,1\}$ est appelé la ***retenue***, et soit $m+n = s_{k+1} b^{k+1} + ... + s_1 b + s_0$. L'algorithme classique est le suivant :

1. Poser $i = 0$ et $r_{i-1} = r_{-1} = 0$.
2. Calculer u_i et poser $s_i = u_i + r_{i-1}$.
3. Poser $i = i+1$. Si $i > h$, poser $c_i = 0$. Si $i \leq k$, retourner au point 2.
4. Si $r_{i-1} = 0$, l'opération est terminée. Sinon, poser $s_{k+1} = r_k$.

Justification mathématique de cet algorithme : $a_i + c_i \leq b-2$, de sorte que l'addition de la retenue au plus un n'engendre qu'une retenue au plus égale à un. Noter la propagation de la retenue dans le cas 9...9+1 = 10...0 en base 10.

Dans les langages de haut niveau, comme en assembleur, on pourrait se croire dispensé de connaître cet algorithme puisque le langage fournit une instruction 'réalisant' cette opération. Ceci signifie, dans le cas de l'assembleur, que le processeur considéré 'sait' faire cette opération ou, en d'autres termes, qu'elle est déjà programmée dans ses circuits. De ce fait, il suffit d'une seule instruction en langage machine pour commander au processeur l'exécution de cette opération. Dans le cas du langage de haut niveau, cela signifie que le programmeur n'a pas à se préoccuper de la façon dont l'algorithme de l'addition est mis en oeuvre, car le compilateur du langage s'en charge : il sait que le processeur auquel il s'adresse 'sait' faire ou non cette opération. Si le processeur sait faire, le compilateur produit l'instruction correspondante, s'il ne sait pas faire, le compilateur produit un programme, dans l'assembleur de ce processeur, qui exécute l'addition.

Mais il suffit de modifier très peu les *conditions* du problème pour se trouver dans l'obligation de programmer, et, dans ce cas, notre connaissance d'un algorithme n'est pas inutile.

---

[1] Tout comme le terme symbole, faute de pouvoir faire mieux, nous nous contenterons de définir celui-ci *par exhibition*.

Le problème du chronomètre.

Pour comparer les performances entre algorithmes résolvant un même problème ou entre programmes implantant un même algorithme, il est utile de savoir mesurer le temps d'exécution d'un programme. D'autant que peu de langages fournissent des instructions à cet effet.

Habituellement, si les ordinateurs ne disposent pas de chronomètre, ils possèdent, par contre, une horloge. La possibilité de repérer des instants nous suffit pour l'utiliser comme chronomètre. Le temps est en général fourni en heures, minutes, secondes et centièmes de secondes, selon une procédure que nous signalons au chapitre 4. Si un programme commence au temps (h0,m0,s0,c0) et se termine au temps (h1,m1,s1,c1), sa durée est donnée par (h1,m1,s1,c1) – (h0,m0,s0,c0). Voici un algorithme de ce calcul, la durée étant recueillie dans (h,m,s,c) :

```
1. Si c1 < c0, ajouter 100 à c1 et poser r = 1, sinon poser
r = 0 ; poser c = c1-c0.
2. Ajouter r à s0 ; si s1 < s0, ajouter 60 à s1 et poser r = 1,
sinon poser r = 0 ; poser s = s1-s0 ;
3. Ajouter r à m0 ; si m1 < m0, ajouter 60 à m1 et poser r = 1,
sinon poser r = 0 ; poser m = m1-m0 ;
4. Ajouter r à h0 ; si h1 < h0, ajouter 24 à h1 ; poser h = h1-h0.
```

Nous observons à nouveau la production d'une retenue et sa propagation vers la gauche. Observons aussi que les quatre étapes de l'algorithmes sont identiques si avant l'étape 1 on pose r = 0, si on ajoute r à c0 avant le test de l'étape 1 et si à l'étape 4 on définit r comme dans les étapes précédentes, ce qui ne modifie pas le résultat. Comparons la programmation d'une telle étape en *PASCAL* et en assembleur :

```
IF  m < (n+r)
  THEN                                0100 80D400     ADC  AH,00
    BEGIN                             0103 38E0       CMP  AL,AH
      Difference :=                   0105 7307       JNB  010E
        m + correctif - (n+r) ;       0107 00C8       ADD  AL,CL
        r := 1                        0109 28E0       SUB  AL,AH
    END                               010B F9         STC
  ELSE                                010C EB03       JMP  0111
    BEGIN                             010E 28E0       SUB  AL,AH
      Difference := m - (n+r) ;       0110 F8         CLC
      r := 0                          0111
    END
```

Ces instructions d'assembleur sont extraites du programme mentionné au chapitre 4.

La différence nous paraît manifeste. Le langage PASCAL utilise abondamment l'expression de *variables* dont le nom est laissé au choix du programmeur. Ceci permet d'écrire les instructions, exception faite des mots-clés du langage, en termes du problème posé. Par contraste, l'assembleur ne permet rien de tel. Son *discours* ne concerne que des emplacements de la mémoire ou des registres du processeur et il se borne à prescrire des opérations très élémentaires : transfert du contenu d'un endroit dans un autre, opérations arithmétiques élémentaires pour lesquelles la place des arguments et celle du résultat sont explicitement indiqués lorsque l'instruction elle-même ne contient pas, implicitement, une telle indication. Comme nous le verrons plus loin, le langage d'assemblage accepte l'insertion de commentaires. Ils sont fort précieux pour le programmeur mais n'interviennent en aucune façon dans le travail du *constructeur*. En particulier, ils ne modifient en rien le code produit. De la même manière, les programmes de haut niveau admettent des commentaires que le compilateur, en général, ne regarde pas[1]. Mais les commentaires d'un programme en langage de haut niveau peuvent être plus concis et plus *algorithmiques*. On peut dire, de

[1] Il y a, naturellement, des exceptions, fâcheuses, au demeurant.

façon imagée, que les langages de haut niveau reviènnent à intégrer des commentaires dans le code symbolique sous une forme très règlementée.

Les macro-assembleurs ont, en apparence un statut intermédiaire entre le langage d'assemblage et le langage de haut niveau. Cependant, dans la mesure où leur objet premier reste la production du code machine, nous aurons l'occasion de le voir en détail au chapitre 8, les programmes rédigés dans ces langages n'en restent pas moins des discours sur les emplacements de la mémoire, les registres, et les opérations qu'il convient d'y effectuer. Et tout comme l'assembleur, les macros-assembleurs sont, du fait même de ce discours, des langages symboliques machine attachés à une machine concrète ou, plus exactement, une famille de machines chez un même fabricant[1].

Nous pourrions donner d'autres exemples d'algorithmes sur le thème de l'addition, de même que nous pourrions donner d'autres programmes PASCAL ou en assembleur pour implanter l'algorithme considéré ci-dessus. Nous y reviendrons suffisamment dans les chapitres suivants pour qu'il soit utile de le faire ici. Il nous paraît plus opportun d'apporter des précisions sur la manière de programmer.

Styles de programmation.

La façon, ou plutôt les diverses façons de considérer les objets de la programmation ont conduit à la constitution de familles de langages de programmation. Ainsi parle-t-on de programmation structurée, de programmation impérative, de programmation fonctionnelle, de programmation logique.

> Le terme *impératif* s'applique à tout langage de programmation basé sur la notion d'*affectation* (transfert d'information entre variables). A l'opposé, la programmation *fonctionnelle* est conçue comme le calcul ininterrompu de *valeurs*, appelé *évaluation*.

On pourrait dire, mais ce ne serait qu'une boutade, qu'un programme donne toujours des ordres au processeur, directement, dans le cas de l'assembleur, ou indirectement, par le truchement d'un compilateur, qu'il doit être de préférence cohérent et bien construit, et qu'il doit s'attacher à bien délimiter les différentes fonctions, au sens le plus général, qu'il met en oeuvre. Chaque famille de langages s'est développée en mettant davantage l'accent sur une de ces préoccupations. Ainsi, *PASCAL* est-il un prototype des langages impératifs structurés, *LISP* un prototype des langages fonctionnels.

Quelle est la place de l'assembleur, de ce point de vue?

Au premier abord, c'est un langage de programmation impératif non structuré. Le lecteur pourra s'en convaincre largement avec le chapitre 3. Nous y regarderons de plus près cependant, de deux façons : par l'amont, au chapitre 2 et par l'aval, au chapitre 4. On constatera alors que, précisément parce qu'il est peu structuré, on ce langage permet de faire tout ce qu'il est possible de faire en programmation. Personne n'en doute, certes, mais il est bon de savoir pourquoi. Et du coup, il devient plus simple d'indiquer comment. Puis, la pratique aidant, on se rend compte que des styles de programmation différents peuvent être développés en assembleur, voire que tous les aspects évoqués plus haut peuvent s'y combiner. C'est exactement selon ce que le programmeur en aura décidé. Et, à côté de sa puissance incontestée, c'est peut-être aussi cela qui fait l'attrait de l'assembleur.

---

[1] En fait, deux fabricants se partagent pour l'instant la quasi totalité du marché de la micro-informatique : *INTEL* et *MOTOROLA*. Donc, fondamentalement, deux assembleurs.

PREMIERE PARTIE : MOYENS ET RESSOURCES.

CHAPITRE 2

LES CONCEPTS.

De la calculette programmable la plus simple aux grands systèmes les plus sophistiqués, la plupart des ordinateurs actuels relèvent de la machine de von Neumann qui, depuis le début des années cinquante, en définit le fonctionnement sur un plan théorique.

> Exceptions, dans une certaine mesure, à cette règle : les machines connexionnistes constituées par un grand nombre de petits processeurs identiques fonctionnant en parallèle. Cependant, chacun de ces processeurs-composants est une machine de von Neumann rudimentaire et le contrôle global d'une machine connexionniste est réalisé par un séquenceur de sorte qu'en définitive, on ne sort pas du cadre de la semi-récursivité.

Décrire en détail une machine théorique reviendrait à décrire de façon globale tous les processeurs actuels quitte à signaler, à chaque fois, ce qui est propre à chaque famille. Nous procéderons inversement : nous donnerons les grandes lignes de la structure théorique que nous illustrerons en détail et concrètement sur une famille, celle des processeurs 80x86 d'INTEL.

Mais pour mieux comprendre la machine de von Neumann et ses réalisations concrètes, nous examinerons un modèle théorique plus simple mais plus puissant : les machines à registres que nous étudierons avec un des outils de base de l'informatique théorique, véritable étalon de la séquentialité, les fonctions récursives.

2.1. Machines à registres.

Parmi les nombreuses formes de la notion de machines à registres, nous choisissons, pour sa simplicité, la machine de Minsky[1]. Elle fournit un modèle très proche, sous de nombreux rapports, avec les machines réelles sans, cependant, perdre en généralité, car, comme nous l'expliquerons, on obtient, en principe, ni plus ni moins que toutes les fonctions semi-récursives.

Une telle machine dispose de registres $r_1,...,r_k$ en nombre fini (variable d'une machine à une autre) et effectue des actions sur ses registres selon les instructions d'un *programme*[2] exécutées une à une et qui prennent l'une des formes suivantes :

E_n renvoie à l'instruction suivante ;
Zr_i consiste à placer 0 dans le registre r_i : $r_i := 0$.
Sr_i consiste à ajouter 1 au contenu du registre r_i : $r_i := r_i+1$.
Dr_i,E_n consiste en deux actions :
si le contenu de r_i est positif, on le diminue de 1 : $r_i := r_i-1$;

---

1 Voir notamment [3].

2 La présentation que nous donnons ici est assez proche de celle de [3]. Voir également [1] pour une approche voisine avec les fonctions récursives.

si le contenu de r_i est nul, on se rend à l'instruction E_n ;
H consiste à arrêter l'exécution du programme.

Une instruction de la forme E_i est appelée **étiquette** ; une instruction de la forme Zr est appelée **annulation** ; une instruction de la forme Sr est appelée **incrémentation**, une instruction Dr,E_n, **branchement conditionnel** et l'instruction H, **halte** ou instruction **d'arrêt**.

On peut donner des représentations très différentes de ces instructions. On peut leur donner la forme d'instructions d'un langage évolué de style *PASCAL*, mais aussi, ce que nous faisons à présent, la forme d'instructions en assembleur. Nous noterons **R0, R1**, ..., **Rk** les registres dont on dispose ou **RA, RB**, ... lorsque leur nombre est petit et que cela s'avère plus commode. On remplacera les étiquettes par une numérotation des instructions (ce qui revient à faire précéder toute instruction des quatre autres types par une étiquette) et les quatre types d'instructions seront notés comme suit lorsqu'ils seront présentés sous forme de listage[1] :

`NUL`	`Ri`	pour Zr_i
`INC`	`Ri`	pour Sr_i
`DSZ`	`Ri,n`	pour Dr_i,E_n
`HLT`		pour H

(`DSZ` : Décrémenter ou Saut si registre à zéro et `HLT` : HaLTe) et en caractères gras (**nul Ri, inc Ri, dsz Ri,*n*, hlt**) dans le corps du texte.

Un programme d'une machine de Minsky est une suite finie d'instructions appartenant à un des cinq types indiqués ci-dessus. Les instructions sont exécutées l'une après l'autre, dans l'ordre que définissent les branchements conditionnels qu'elles contiennent éventuellement. Nous utiliserons conjointement les deux notations : la notation proche des programmes de calculette et la notation de style assembleur.

Exemple I :

Le programme suivant, sur une machine à trois registres **a**, **b** et **c**, fournit dans **a** la somme de deux entiers donnés initialement dans **a** et dans **b** :

$Zc; E1; Db,E2; Sa; Dc,E1; E2; H$

ou encore, en notation assembleur (les instructions sont implicitement numérotées depuis 0 selon leur ordre d'apparition) :

```
            NUL  RC
; 1 :
            DSZ  RB,4
            INC  RA
            DSZ  RC,1
; 4 :
            HLT
```

où le numéro de l'instruction est mis en commentaire (annoncé par ;) lorsque cela facilite la compréhension.

Le lecteur aura deviné que l'algorithme implanté est le suivant : on décrémente RB en même temps que l'on incrémente RA et ce, jusqu'à ce qu'on ne puisse plus décrémenter le registre RB.

Sur cet exemple, on observe que **c**, après avoir été initialisé par 0, n'est plus jamais affecté par le programme : ceci réalise un **saut inconditionnel** (le **goto** de

[1] On prononce souvent 'listing', mais il faut écrire ***listage***.

nombreux langages évolués). Dans notre exemple, ce saut inconditionnel permet de construire une boucle dont on ne sort que si **b** est nul. Comme on n'entre dans la boucle que si **b** > 0, on a réalisé un *WHILE* (au sens de *PASCAL*). L'utilisation d'un registre mis à zéro pour effectuer un **goto** peut être introduite dans le langage. On s'autorise donc une nouvelle instruction :

```
ST  n
```

(ST pour SAUT) dans la version 'assembleur' de la représentation des machines de Minsky.

L'instruction ainsi ajoutée est dite **admissible** car on peut aussi bien la remplacer par une suite adéquate d'instructions au sens strict. Cette adjonction d'opérations admissibles est une pratique fréquente en logique mathématique.

L'introduction du saut inconditionnel nous permet de faire l'économie de l'instruction d'arrêt **hlt** : il suffit de convenir que l'instruction qui suit la dernière instruction écrite est l'instruction d'arrêt. L'instruction **hlt** sera dite alors ***implicite***. Si le programme doit s'arrêter avant, il suffit alors d'utiliser un saut à l'instruction qui suit la dernière. On peut le voir sur l'exemple suivant, à partir duquel les programmes ne seront plus donnés qu'en version 'assembleur' seulement.

Exemple II :

Le programme effectue la multiplication de deux entiers.

Le principe de l'algorithme est le suivant ; chaque fois que l'on décrémente RA, on ajoute le contenu de RB au registre RR dans lequel on place le résultat. Pour ce faire, au cours de l'addition de RB à RR selon l'algorithme précédent, on 'sauvegarde' la valeur de RB en incrémentant le registre RU, nul au départ, chaque fois que l'on décrémente RB : ceci permet, en inversant les rôles de RU et RB (instructions 11 à 13) de restituer RB quand on revient à la décrémentation de RA (instruction 6).

Cette version 'assembleur' du programme, est commentée selon la convention suivie dans ce type de langage : les commentaires sont annoncés par le symbole ; dont l'effet perdure jusqu'à la fin de la ligne.

```
            NUL  RR       ;
            DSZ  RA,14    ; si RA ou RB est nul,
            DSZ  RB,14    ; le résultat l'est aussi...
;                         ; on rétablit RA et RB :
            INC  RA       ;
            INC  RB       ;
;                         ; on initialise RU :
            NUL  RU       ;
; 6 :                     ;
            DSZ  RA,14    ; si RA = 0, c'est fini
; 7 :                     ;
            DSZ  RB,11    ; si RB = 0, le renouveler
            INC  RR       ; on augmente RR
            INC  RU       ; et RU
            ST   7        ; de la valeur initiale de RB
; 11 :                    ; renouvellement de RB
            DSZ  RU,6     ; si RU   0, c'est fini
            INC  RB       ; on augmente RB
            ST   11       ;
; 14 :                    ; suivi de l'arrêt du programme
```

Exemple III :

Il s'agit à présent de simuler la division euclidienne de deux entiers. On suppose les données placées dans deux registres RA et RB, le diviseur, avec RB ≠ 0. La machine doit donner le quotient et le reste, respectivement, dans les registres RQ et RR.

Le programme consiste à retrancher RB de RA autant de fois qu'il le faut pour que le nombre obtenu soit plus petit que RB. Le registre RQ compte le nombre des soustractions effectuées. Il nous faut à priori réaliser aussi la comparaison de deux nombres. En fait, il suffira de 'surveiller' le déroulement de la soustraction pour statuer sur la comparaison des deux nombres. Avant de soustraire RB, on sauve son contenu sur deux registres dont l'un sera utilisé pour effectuer la soustraction :

```
            NUL  RV       ; initialisations
            NUL  RW       ;
            NUL  RR       ;
            NUL  RQ       ;
; 4 :                      ; RR := RA
            DSZ  RA,7     ;
            INC  RR       ;
            ST   4        ;
; 7 :                      ; RV := RB, RW := RB
            DSZ  RB,11    ;
            INC  RV       ;
            INC  RW       ;
            ST   7        ;
; 11 :                     ; on restaure RB depuis RV
            DSZ  RV,14    ;
            INC  RB       ;
            ST   11       ;
; 14 :                     ; RR := RR - RB
;                          ; (RW = RB au départ)
            DSZ  RW,17    ;
            DSZ  RR,21    ; -> calcul du reste
            ST   14       ;
; 17 :                     ; RQ := RQ + 1
            INC  RQ       ;
            DSZ  RR,28    ; si ici, RR = 0, reste nul
            INC  RR       ;
            ST   7        ; -> soustraire RB tant que RR > 0
; 21 :                     ; on a 'trop' retranché,
;                          ; on rétablit le reste
            INC  RW       ;
; 22 :                     ; RB := RB - RW
            DSZ  RW,25
            DSZ  RB,28    ; ce saut ne peut se produire
            ST   22       ;
; 25 :                     ; fin de calcul du reste :
            DSZ  RB,28    ; RR := RW
            INC  RR       ;
            ST   25       ;
; 28 :                     ; résultat dans RQ et RR
```

Observons que cette machine utilise six registres (non compris le registre utilisé par les instructions st). En fait, elle pourrait n'en utiliser que cinq puisque RA n'est jamais utilisé.

2.2. Machines à registres et fonctions récursives.

Les fonctions semi-récursives[1] permettent d'établir l'équivalence entre deux concepts pratiquement aux antipodes l'un de l'autre. L'un, en un sens, ce qu'il y a de plus *fonctionnel*, l'autre, sous le même angle, ce qu'il y a de plus *impératif*[2]. Cette équivalence a une portée considérable, à la fois théorique et pratique.

Les fonctions récursives.

Partons d'un exemple. L'addition de deux entiers peut être définie de la façon suivante :

$$+(x,0) = x$$
$$+(x,Sy) = S(+(x,y)),$$

où Sy (noté aussi S(y)) est le suivant de y et où x et y parcourent N. Appliquons ceci à 2 et 2 :

$$\begin{aligned} +(2,2) &= S(+(2,1)) \ (2^{\text{ème}} \text{ équation}) \\ &= S(S(+(2,0)) \ (2^{\text{ème}} \text{ équation}) \\ &= S(S(2)) \ (1^{\text{ère}} \text{ équation}) \\ &= S(3) = 4. \end{aligned}$$

On définit la multiplication de la même façon, à partir de l'addition :

$$*(x,0) = 0$$
$$*(x,Sy) = +(x,*(x,y)).$$

D'une façon générale, pour obtenir les fonctions semi-récursives, on procède par assemblages successifs à partir de 'briques', à la manière d'un *Lego*. Ces éléments premiers, appelés ***fonctions de base***, comprennent ainsi la fonction O qui à x associe 0 et la fonction S qui à x associe son ***successeur*** $x+1$. Elles comprennent également toutes les fonctions de la forme

$$U^k_i(x_1,...,x_k) = x_i,$$

où $k \geq 1$ et $1 \leq i \leq k$. Ce sont les fonctions ***projecteurs***. Les assemblages se font par application de trois règles appelées ***schémas***.

Premier schéma : la **composition**.

Si $h(x_1,...,x_n)$ et $g_1(x_1,...,x_k),...,g_n(x_1,...,x_k)$ sont des fonctions semi-récursives, alors nous décidons que la fonction f, *composée* de $g_1,...,g_n$ par h, définie par :

$$f(x_1,...,x_k) = h(g_1(x_1,...,x_k),...,g_n(x_1,...,x_k))$$

est également semi-récursive.

Second schéma : la **minimisation**.

Soit $g(x_1,...,x_k,y)$ une fonction. La *minimisée* de g est la fonction f qui, pour chaque $x_1,...,x_k$, fournit le plus petit entier naturel w tel que $g(x_1,...,x_k,w) = 0$ et n'est évidemment pas définie si un tel w n'existe pas. On note cette relation entre f et g par :

$$f(x_1,...,x_k) = \mu y \ (g(x_1,...,x_k,y) = 0)$$

---

1 Le terme mathématique est semi-récursif. Dans un contexte euristique, nous l'abrègerons en récursif. Pour la différence entre ces deux termes sur un plan strictement mathématique, voir [2].

2 *cf.* l'explication donnée page 6.

où, par convention, $\mu y\ (A(y) = 0)$ est le plus petit entier naturel w pour lequel $A(w) = 0$ si un tel entier existe et n'est pas défini sinon. Le symbole $\simeq$ au lieu du symbole $=$ indique que la fonction n'est pas toujours définie. C'est le symbole de l'***égalité conditionnelle*** : l'égalité n'a lieu que si les deux membres de la relation sont définis et si un des termes est défini, l'autre doit l'être également.

Le second schéma stipule que si g est une fonction semi-récursive, sa minimisée f l'est aussi.

Troisième schéma : la **récursion.**

Cette règle permet de définir une fonction $f(x_1,...,x_k,y)$ par récurrence sur les valeurs de y à l'aide d'une fonction pilote h et d'une fonction initiale g à l'aide des équations :

$$f(x_1,...,x_k,0) \simeq g(x_1,...,x_k),$$
$$f(x_1,...,x_k,Sy) \simeq h(x_1,...,x_k,y,f(x_1,...,x_k,y))$$

Si h et g sont des fonctions semi-récursives, f l'est aussi.

On remarque que l'addition et la multiplication ont été définies ci-dessus par application du schéma de récursion. Ainsi dans l'exemple de l'addition, la fonction pilote est la fonction $h(x,y,z) = S(z)$. En toute rigueur, le membre de droite de cette équation doit faire apparaître toutes les variables de la fonction. C'est à cela qu'on utilise les fonctions *projecteurs*. Ce qui permet d'écrire la fonction pilote de l'addition :

$$h(x,y,z) = S(U^3_3(x,y,z))$$

puisque $U^3_3(x,y,z) = z$. La fonction initiale de l'addition est la fonction 0.

Exercice :

Donner la fonction pilote et la fonction initiale de la multiplication.

De ce mode de construction des fonctions récursives, on déduit que leur ensemble est au plus dénombrable. Il existe donc beaucoup plus de fonctions non récursives que de fonctions semi-récursives. Cependant, l'ensemble des fonctions récursives contient la plupart des fonctions usuelles de N dans N. Mieux (ou pire), les fonctions usuelles sont contenues dans un ensemble plus petit : l'ensemble des fonctions *primitives* récursives qui joue un très grand rôle en programmation. Ces dernières fonctions s'obtiennent en n'utilisant, dans leur construction, que deux des trois schémas : le schéma de composition et le schéma de récursion. Les fonctions primitives récursives constituent un cas particulier des fonctions *récursives totales* qui sont, par définition, les fonctions semi-récursives définies partout.

Il y a d'autres schémas qui, appliqués à des fonctions semi-récursives, ont pour résultat des fonctions semi-récursives : on le démontre en montrant, pour chacun des nouveaux schémas, qu'on peut reconstruire la fonction trouvée à partir des fonctions

La division euclidienne

Le quotient de a par b, $b > 0$, se définit sans trop de difficulté par un schéma de récursion :

$$div(0,b) = 0$$
$$div(Sa,b) = div(a,b) + zr(a*S(div(a,b))\backslash Sa)$$

où zr est définie par :

$$zr(0) = 1$$
$$zr(Sx) = O(x)$$

et \, soustraction positive, est définie par :

$$\backslash(a,b) = a - b \text{ si } a \geq b$$
$$\backslash(a,b) = 0 \text{ si } a < b,$$

ce qui s'exprime par une suite finie de schémas de composition et de récursion.

On a une définition plus simple (et de calcul plus rapide) avec le schéma de minimisation :

$$div(a,b) \simeq \mu q\ (Sa\backslash b * Sq = 0)$$

données en utilisant exclusivement les trois schémas donnés ci-dessus. Ce sont des schémas **admissibles**. On peut en donner un exemple avec le **schéma de définition par cas** (d'où dérive, par exemple, l'instruction CASE du *PASCAL*) :

$h(x_1,...,x_k)$ étant une fonction à valeurs dans $\{1,...,p\}$, et $g_1(x_1,...,x_k),...,g_p(x_1,...,x_k)$ des fonctions, on pose

$$f(x_1,...,x_k) \simeq g_1(x_1,...,x_k) \text{ si } h(x_1,...,x_k) = 1,$$

...

$$f(x_1,...,x_k) \simeq g_p(x_1,...,x_k) \text{ si } h(x_1,...,x_k) = p.$$

Le schéma de définition par cas est admissible : si les fonctions h et $g_1,...,g_p$ sont semi-récursives, la fonction f l'est aussi.

On peut le voir aisément pour une fonction d'une variable lorsque $p = 2$. On peut supposer que la fonction h a pour valeurs 0 ou 1. La fonction suivante :

$$f(x) = g_1(x).zr(h(x)) + g_2(x).(1 \dot{-} zr(h(x))),$$

où la fonction zr est définie dans l'encadré ci-dessus, répond à la question. La généralisation à p choix et k variables procède de là.

Codage

Les fonctions semi-récursives permettent de rendre compte d'une propriété importante des nombres naturels aux conséquences insoupçonnables : le *codage*.

Le codage est partout : le système de numération à base dix est le codage le plus utilisé, mais il y en a beaucoup d'autres. Ainsi des bases deux et 256 utilisées intensivement par les ordinateurs réels. La notion de codage fait aussitôt intervenir celle de code, l'objet qui sert de support au codage d'une donnée.

On peut formaliser la construction des fonctions semi-récursives à partir des fonctions de base et des schémas utilisés, ce que nous ne ferons pas dans le cadre de cet ouvrage[1]. La représentation formelle d'une fonction semi-récursive peut être *codée* par un *nombre* qui en constitue en quelque sorte la *clé* puisque décodé, il fournit les étapes et les composants de la construction de la fonction.

On peut aller plus loin : les codes ne sont pas tous les entiers naturels, mais seulement certains d'entre eux qu'on peut ranger en une suite croissante. Pour obtenir un *code*, donc la fonction semi-récursive qu'il décrit, il suffit de connaître son numéro d'ordre dans la suite des codes. On définit ainsi la **numérotation** des fonctions semi-récursives : φ_n désigne la fonction codée par le code de numéro n ; on dit, en abrégé, la **fonction de numéro** n.

Or, on peut définir le codage des fonctions semi-récursives de façon récursive. On tire de là, ce qui n'est pas immédiat, le *théorème de la fonction universelle*. Ce théorème démontre l'existence d'une fonction semi-récursive U telle que pour tout *numéro* de fonction e et tout code $\mathbf{x}$ de données $x_1,...,x_k$ (dans un codage fixé),

$$U(e,\mathbf{x}) \simeq \varphi_e(\mathbf{x})$$

si et seulement si la donnée $x_1,...,x_k$ est dans le domaine de définition de φ_e (on remarque que dans le schéma de minimisation, le plus petit entier cherché peut ne pas exister, de sorte qu'une fonction semi-récursive n'est pas nécessairement définie partout).

Signalons, pour le lecteur intéressé, que l'on déduit du théorème de la fonction universelle des résultats fondamentaux dans deux directions.

---

[1] Nous renvoyons en toute modestie à [2] où cette propriété nous paraît suffisamment mise en lumière.

La première est illustrée par le *théorème du point fixe* : étant donné une fonction récursive f définie sur N, on peut construire un numéro de fonction n tel que $\varphi_n = \varphi_{f(n)}$, c'est-à-dire, tel que n et $f(n)$ numérotent deux fonctions égales, quant à leur graphe (elles calculent les mêmes valeurs aux mêmes points). L'encadré ci-contre donne un exemple d'application du théorème du point fixe qui joue un rôle important dans la théorie. On l'utilise, en particulier, pour établir le caractère admissible d'un schéma.

Les fonctions qui se reproduisent

Le théorème de la fonction universelle permet d'écrire la fonction

$$f(x,y) = x$$

sous la forme $\varphi_{s(x)}$ où s est une fonction récursive *totale* sur N. Le théorème du point fixe fournit un numéro e tel que

$$\varphi_{s(e)} = \varphi_e$$

et donc $\varphi_e(y) = e$ pour tout y. La fonction φ_e renvoie toujours son propre numéro.

La seconde direction est définie par les théorèmes d'indécidabilité récursive. L'encadré ci-après illustre la traduction récursive du paradoxe du menteur sur laquelle repose ce type de résultats. De là découle, en particulier, l'*indécidabilité du problème de la halte.*

Ce nom vient de la considération suivante : comme on le sait déjà, seul le schéma de minimisation peut être cause de la non définition de la valeur d'une fonction en un point lorsque les fonctions 'constitutives' sont, quant à elles, définies partout. Lorsqu'on cherche le plus petit entier w tel que $g(x_1,...,x_k,w) = 0$, on effectue le calcul pas à pas. On calcule $z_i = g(x_1,...,x_k,i)$ en commençant par $i = 0$, et, si $z_i \neq 0$, on calcule z_{i+1} et ainsi de suite, tant que $z_w \neq 0$. Si l'entier cherché par la minimisation n'existe pas, ce calcul ne s'arrête pas.

Suis-je ou ne suis-je pas récursif?

Le paradoxe du menteur énonce :

Un menteur ne peut pas prononcer la phrase : 'je mens'.

Traduction récursive :

W_e désigne le domaine de définition de φ_e la fonction récursive de numéro e (on le note également *dom* φ_e). On démontre qu'il existe une fonction récursive f telle que $x \in dom f$ si et seulement si $x \in W_x$. Par contre, il n'existe pas de fonction récursive *totale* g telle que $x \in dom g$ si et seulement si $x \notin W_x$.

En effet, sinon, si e était le numéro d'une telle fonction, on aurait $x \in We$ si et seulement si $x \notin Wx$; en particulier, on aurait $e \in W_e$ si et seulement si $e \notin W_e$.

Par une technique de codage, on déduit du théorème indiqué dans l'encadré ci-dessus qu'il n'existe pas de fonction récursive *totale* qui, appliquée à e et $\mathbf{x} = (x_1,...,x_k)$ permettrait de savoir si $\mathbf{x}$ est ou n'est pas dans le domaine de définition de φ_e.

Ce théorème a une grande signification pratique : des algorithmes de correction de programmes donnés a priori ont nécessairement une portée limitée (bien qu'ils puissent être fort utiles en pratique) ; une autre démarche est à envisager : celle qui consiste, dans chaque cas, à construire un programme suivant un schéma démonstratif à partir d'éléments prouvés.

Machines universelles.

Nous démontrerons succinctement ici que les machines de Minsky ont la puissance des fonctions semi-récursives[1].

Compte tenu des schémas donnés pour les fonctions semi-récursives et le théorème d'élimination du schéma de récursion (cf. [2] pour une démonstration complète), il suffit de démontrer que l'ensemble des fonctions représentables par les machines de Minsky contiennent les fonctions semi-récursives, puis que le fonctionnement d'une machine de Minsky peut être simulé par les fonctions semi-récursives.

[1] Voir également [2], corrigé de l'exercice 4 du chapitre 4.

Représentation des machines à registres par les fonctions récursives.

Soit donc M une machine de Minsky donnée par un programme de la forme que nous avons définies précédemment. On peut numéroter ses instructions de 0 à n, où n est le numéro de l'instruction d'arrêt, 0 celui de l'instruction initiale. On peut désigner les registres de M par $R1, ..., Rk$.

Désignons par $x_1,...,x_k$ les valeurs des $R1,...,Rk$, par i le numéro de l'instruction à exécuter et par p le nombre d'instructions exécutées jusque là (avec répétition, naturellement). Soit I l'instruction numérotée i lorsque $0 < i < n$, I non définie si $i > n$. Supposons donc I définie, différente de l'instruction d'arrêt, et désignons par $x'_1,...,x'_k$ les nouvelles valeurs des $R1,...,Rk$ après exécution de I et par i' le numéro de la prochaine instruction. Observons que p devient $p+1$. Les valeurs de i' et $x'_1,...,x'_k$ se déduisent aisément :

si I = NUL Rj, on a :
$i' = Si, x'_j = 0$ et $x'_h = x_h$ pour tout $h \neq j$;
si I = INC Rj, on a :
$i' = Si, x'_j = Sx_j$ et $x'_h = x_h$ pour tout $h \neq j$;
si I = DSZ Rj, m, on a :
$i' = Si, x'_j = x_j$-1 et $x'_h = x_h$ pour tout $h \neq j$ si $Rj \neq 0$,
$i' = m, x'_h = x_h$ pour tout h si $Rj = 0$;
si I = ST m, on a :
$i' = m, x'_h = x_h$ pour tout h ;
si I est l'instruction d'arrêt, on a :
$i' = n, x'_h = x_h$ pour tout h.

On appellera **fonctions de transfert** de I, les fonctions $f_1,...f_k,j$ définies de la façon suivante :

$$f_h(x_1,...,x_k,p) = x'_h \text{ où } 1 \leq h \leq k \text{ et } j(x_1,...,x_k,p) = i',$$

les $x'_1,...,x'_k,i'$ étant associés aux $x_1,...,x_k,i$ comme ci-dessus.

Il est clair que les fonctions de transfert d'une instruction donnée sont des fonctions semi-récursives (en fait récursives totales) des $x_1,...,x_k,i$. On peut associer à M des fonctions de transfert *globales* d'exécution de la machine après p pas : si $x_1,...,x_k$ est un ensemble de valeurs initiales des registres de M, $F_h(x_1,...,x_k,p)$ désigne, par définition, la valeur x'_h du registre Rh après p pas d'exécution. De même, on définit J par $J(x_1,...,x_k,p) = i'$, numéro de l'instruction à exécuter pour le pas $p+1$.

Fixons-nous un codage des suites de $k+1$ entiers naturels que nous noterons $<x_1,...,x_{k+1}>$. Soit alors $G = <F_1,...,F_k,J>$. On a clairement :

$$G(x,0) = <x_1,...,x_k,0> (= <U1_k(x),...,Uk_k(x),0(x)>)$$
$$G(x,Sp) = <f_1(G(x,p)),...f_k(G(x,p)),j(x,p))>,$$

où $f_1,...f_k$ sont les fonctions de transfert de l'instruction de numéro $J(x,p)$ et $x = x_1,...,x_k$. Or, pour le programme de la machine M fixée, les fonctions $f_1,...f_k$ et j associées à l'instruction de numéro i de ce programme sont définissables *récursivement* en fonction de i : il suffit de reprendre les définitions données ci-dessus par type d'instructions et de les replacer dans un schéma de définition par cas qui, à chaque numéro d'instruction, associera les fonctions de transfert qui correspondent à son type, au registre et/ou à la référence qu'elle définit.

Le résultat de l'exécution de M pour les données initiales $x_1,...,x_k = x$ des registres, est donné par $G(x,p)$, où :

$$p = \mu y \left(\left| J(x,y) - n \right| = 0 \right).$$

Représentation des fonctions récursives par les machines à registres.

On définit au préalable la notion de fonction représentable par une machine de Minsky. Si $f(x_1,...,x_k)$ est une fonction à valeurs dans **N**, on dit que f est **représentable par les machines de Minsky**, s'il en existe une, soit M, comportant au moins $k+1$ registres $R1,...,Rk$, RY telle que pour tous entiers naturels $x_1,...,x_k$, si les x_i sont les valeurs initiales des Ri, la machine achève son travail si et seulement si $x_1,...,x_k \in domf$ auquel cas la valeur de RY quand M arrête son travail est y. Comme nous savons que l'addition et la multiplication sont représentables par les machines de Minsky ainsi que le successeur et les projecteurs, ce qui est évident, il nous reste à démontrer que cette propriété n'est pas modifiée quand on applique les schémas de composition et de minimisation à des fonctions qui la possèdent déjà.

Montrons le d'abord pour la minimisation. Soit donc

$$f(x_1,...,x_k) = \mu y\ (g(x_1,...,x_k,y) = 0),$$

où g est représentée par une machine N. Parmis les registres de N, désignons par $R1,..Rk,RY$ ceux qui correspondent à $x_1,...,x_k,y$ et par RZ le registre recevant la valeur de la fonction g à la fin du calcul. Supposons, pour simplifier, qu'en fin de calcul, les registres $R1,..Rk$ et RY retrouvent leurs valeurs initiales. Si les registres $R1,...,Rk$ sont initialisés par $x_1,...,x_k$. Si les instructions de N sont numérotées de 0 à M, renumérotons les de 1 à M+1 (M+1 est donc le numéro de la dernière instruction qui, par notre convention, n'est pas représentée). On définit une machine M par le programme Minsky suivant :

```
          NUL  RY              ; initialisation
; B :                         ; haut de boucle (ici B = 1)
    programme de la machine N
; M+B :                       ; test de sortie de boucle :
          DSZ  RZ,H            ; si RZ = 0, terminé
          INC  RY              ; sinon, on incrémente RY
          ST   B               ; et on retourne au calcul de RZ
; H :                         ; dernière instruction
```

Si $g(x_1,...,x_k,0) = 0$, la suite des instructions exécutées est la suivante :

```
0,B,...,M+B,H
```

puisque à l'instruction M+1, on constate que $RZ = 0$.

Sinon, en supposant que $x_1,...x_k \in domf$, soit $y = f(x_1,...,x_k)$ avec $y > 0$. Soit alors $z_i = g(x_1,...,x_k,i)$ pour $i = 0,...,y$. On a $z_y = 0$ et $z_i > 0$ pour $0 < i < y$. La suite des instructions exécutée est donc (on note en regard la valeur de y) :

```
0   0,B,...,M+B,M+B+1
1   M+B+2,B,...,M+B,M+B+1
2   M+B+2,B,...,M+B,M+B+1
      ...
y   M+B+2,1,...,M+B,H
```

Cette séquence va se répéter, tant que $z_i > 0$. Il résulte de l'étude du cas $g(x_1,...,x_k,0) = 0$ que lorsque l'on a calculé y, $RZ = 0$ et donc l'instruction M+B est suivie de l'exécution de H, c'est-à-dire l'arrêt de la machine. On constate que RY a la valeur requise, par récurrence sur y. Le même raisonnement montre que M ne s'arrête pas si $x_1,...x_k \notin domf$. La machine M représente donc la fonction f.

Si en fin de calcul la machine M ne restitue pas aux registres Ri et RY leurs valeurs initiales, on modifie le programme ci-dessus de la façon suivante. On introduit des registres $U1,V1,...,Uk,Vk,RU$ et RV pour sauvegarder les valeurs des registres Ri et RY. On étend la phase d'initialisation précédente par un annulation des registres Ui, Vi, RU et RV. Soit alors S_1 le numéro de l'instruction suivante ($S_1 = 2k+3$). En S_1 commence la sauvegarde des registres Ri et RY représentée par les instructions de gauche ci-après :

```
; Si :                                                    Di :
      DSZ   Ri,Ti          ; si Ri = 0,                          NUL   Ri
      INC   Ui             ; terminé                      Ei :
      INC   Vi             ;                                     DSZ   Ui,Di+1
      ST    Si             ;                                     INC   Ri
; Ti :                     ; à présent, Ri = 0                   ST    Ei
      DSZ   Vi,Si+1        ; si Vi = 0,                   Di+1 :
      INC   Ri             ; terminé
      ST    Ti             ;
; Si+1 :                   ;  à présent, Ri = Ui et Vi = 0
```

La séquence S_i-S_{i+1} est répétée de i = 1 à k+1 (*RU* et *RV* remplacent, respectivement, *U*i et *V*i pour i = k+1), de sorte que l'instruction S_{k+2} est la première de la machine M. L'instruction HLT implicite de la machine *M* est remplacée par l'instruction $D1$ de la séquence de droite ci-dessus. L'instruction D_{k+2} est celle du test **dsz RZ,H**. On pose ici $B = S_1$, de sorte que la suite des exécutions de la page précédente se reproduit telle quelle.

La même propriété se démontre plus aisément dans le cas du schéma de composition : nous le laissons en exercice au lecteur.

D'où l'équivalence annoncée.

Il en résulte donc que la propriété de fonction universelle des fonctions semi-récursives a également lieu pour les machines de Minsky :

Théorème

Il existe une machine de Minsky U, un codage c des programmes de machines de Minsky et un codage g des suites finies d'entiers tels que pour toute machine de Minsky M et toute donnée x1,...xk de la machine M, la machine U appliquée aux données c(M) et g(x1,...xk) a exactement le même résultat que la machine M appliquée aux données x1,...xk (en particulier n'a pas de résultat si M appliquée à x1,...xk n'en a pas).

REMARQUE : Ce théorème a une conséquence pratique de grande portée : c'est sur lui que repose la possibilité de construire des compilateurs d'assembleur en assembleur.

Cependant, la démonstration que nous en avons donné ne donne pas explicitement une machine de Minsky universelle. Nous n'en donnerons pas ici de construction détaillée en raison de la complexité d'un codage récursif en termes de machines de Minsky, mais voici quelles peuvent en être les grandes lignes.

Schéma de machine à registre universelle.

Supposons que l'on dispose de 'sous-machines' permettant de coder et décoder les suites finies d'entiers naturels. Soit *M* la machine à simuler. On fixe un registre *RC* qui contiendra le code du programme de *M*. Un autre registre *RR* contiendra le code de la suite des valeurs des registres de *M*, un troisième registre *RI* contiendra le numéro de l'instruction à exécuter (0 au départ), un quatrième, *RJ*, contiendra le code l'instruction de numéro *RI* et un cinquième, *RH*, contiendra le numéro de l'instruction d'arrêt. Le programme de la machine universelle comprend une boucle fondamentale de la forme suivante :

```
            NUL   RI
; 1 :
            ST    10        ; -> exécution de I, de n° RI
; 2 :
            ST    20        ; -> RA := |RI - RH|
            DSZ   RA,5
            ST    1
; 5 :
            ST    1000      ; renvoi à la dernière instruction
```

où **ST 10** renvoie à une partie de la machine où l'on exécute sur *RR* l'instruction codée dans *RJ* et **ST 20** calcule, dans *RA*, la différence absolue entre *RI* et *RH*. Si *RA* = 0, l'instruction d'arrêt est la prochaine exécution : on a donc terminé. Sinon, on a dans *RI* le numéro de la prochaine instruction à exécuter, d'où le renvoi à l'instruction n°1.

REMARQUE : Dans l'instruction n°5, le numéro référencé a été choisi assez grand pour qu'on puisse insérer les sous-programmes décrits plus loin. Pour que la numérotation des instructions ne présente pas de lacunes, on insère, autant que de besoin, des instructions à effet 'nul', par exemple le couple d'instructions :

```
INC  RA
DSZ  RA,5.
```

On ne détaille pas le calcul effectué par **st 20**, (on l'a pratiquement déjà fait). Détaillons le sous-programme auquel renvoie **st 10** et qui dispose de registres auxiliaires *RB*, *RD*, *RE*, *RF* et *R*1, *R*2, *R*3, *R*4, ces quatre derniers contenant, respectivement, les nombres 1, 2, 3, 4[1] :

```
                ST   110      ; RB reçoit le type de l'instruction J
                              ; codée par RJ, RD le n° de registre
                              ; éventuel (0 si J est ST) et RD,
                              ; l'étiquette éventuelle
                ST   120      ; RF := |RB - R1|
                DSZ  RF,160   ;
                ST   130      ; RF := |RB - R2|
                DSZ  RF,170   ;
                ST   140      ; RF := -RB - R3|
                DSZ  RF,180   ;
                ST   150      ; RF := |RB - R4|
                DSZ  RF,190   ;
                ST   200      ; saut dans une boucle infinie
; 160 :                       ;
                ST   210      ; RF = (C)h, h = RD
                NUL  RF       ;
                ST   215      ; (C)h = RF
                INC  RI       ;
                ST   2        ;
; 170 :                       ;
                ST   210      ;
                INC  RF       ;
                ST   215      ;
                INC  RI       ;
                ST   2        ;
; 180 :                       ;
                ST   210      ;
                DSZ  RF,190   ;
                ST   215      ;
                INC  RI       ;
                ST   2        ;
; 190 :                       ;
                ST   220      ; RI := RE
                ST   2        ;
```

Les sous-programmes qui ne sont pas détaillés ci-dessus, font appel aux sous-programmes de décomposition du code d'une suite et de réintégration d'un élément de la suite codée dans le code.

---

[1] Ces quatre derniers registres font partie des registres initialisés avant l'exécution de la première instruction.

On peut donner une idée succincte de la façon de procéder :

On peut coder la suite $x_1,...,x_k$ par $2^{y_1}\cdot 3^{y_2}\cdot...\cdot q^{y_k}$ où q est le $k$ème nombre premier et $y_h = x_h+1$ (codage imaginé par GÖDEL en 1931). Une modification sur x_i consiste soit à remplacer p^{y_i} par p, p $i$ème nombre premier (annulation), soit à multiplier ce nombre par p (incrémentation), soit à le diviser par p (incrémentation). La lecture de x_i se fait également par une suite de divisions et l'insertion de p dans la suite (p a 'disparu' de la suite par l'extraction effectué par **st 210**) se fait par une suite de multiplications, toutes ces opérations portant sur le registre *RR* et le registre *RC* avec cette différence, pour ce dernier, que seules des 'lectures' sont effectuées : on prend d'abord une 'copie' de *RC* pour extraire l'instruction qui nous intéresse, on décode à son tour cette information que l'on peut supposer organisée sous la forme $<t,r,i>$ où t désigne le type de l'instruction (numéroté de 1 à 4), **r** désigne le registre sur lequel agit l'instruction (numéroté de 1 à k si $t \neq 4$ et 0 quand $t = 4$, puisque un saut n'agit sur aucun registre), et i est l'instruction à laquelle renvoie $<t,r,i>$ lorsque $t \geq 3$ (types **dsz** et **st**).

On voit dès à présent le rôle fondamental joué par le codage dans la théorie. Observons qu'un raffinement du processus de codage permet de réduire encore le nombre des registres de la machine universelle. On peut démontrer qu'ils peuvent être réduits à deux (puisque tout registre peut être en fait considéré comme une réunion finie de registres et qu'il faut au moins un registre pour des opérations de sauvegarde).

Le codage des suites finies d'entiers a une fonction essentiellement théorique : l'hypothèse que tout registre peut contenir des nombres aussi grands que l'on veut permet donc de 'coder' n'importe quelle configuration 'à la Turing' par un seul nombre. C'est là que se situe la différence fondamentale entre les machines de Minsky et la machine de von Neumann.

Le schéma de récursion.

Il résulte du théorème dont nous venons d'esquisser la démonstration, que le schéma de récursion peut être simulé par les machines de Minsky. Comme ce schéma fonde, sur un plan théorique, la notion de programme récursif et que nous aborderons cette question au chapitre 4, nous indiquons, ci-après, comment simuler directement ce schéma.

Le problème est, évidemment, celui de la simulation de l'équation :

$$f(x_1,...,x_k,Sy) = h(x_1,...,x_k,y,f(x_1,...,x_k))$$

Mettons-nous à la place d'une machine de Minsky : au moment d'exécuter la fonction h pour les arguments $x_1,...,x_k$ et y-1 dont le programme est supposé connu, on se rencontre qu'il manque une valeur ; celle de $f(x_1,...,x_k,y\text{-}1)$. Que faire? On appelle à nouveau le programme de calcul de f et, si on ne fait rien d'autre, on va retrouver la même difficulté. Or les différents appels de f se distinguent les uns des autres par la valeur des arguments de f. Regardons de plus près : les $x_1,...,x_k$ restent toujours identiques, mais par contre, y est modifié d'un appel au suivant. On va donc conserver la valeur de y sur un registre avant d'appeler f à nouveau. Comme le nombre de registres de sauvegarde risquerait d'être non borné, il y a là une difficulté. Le codage nous permet de trouver une solution : on réserve un registre, soit *RP* pour sauvegarder les valeurs successives de y. Au premier appel, de f, *RP* va contenir $<y1>$, où $y1$ est la valeur de l'argument y de f à cet appel ; au $k$ème appel, si *RP* contient $<y_1,...,y_h>$, on va conserver $<y_1,...,y_h,y_k>$, où y_k est la valeur de y à cet appel. Lorsque l'on dispose enfin d'une valeur de $f(x_1,...,x_k,y)$, soit w que l'on va mettre dans un registre *RW*, on va pouvoir revenir au calcul de la valeur de h, ce qui fournira alors la valeur de $f(x_1,...,x_k,y)$ permettant à son tour de calculer la valeur de $h(x_1,...,x_k,y,f(x_1,...,x_k,y))$. La valeur de y nécessaire est à prendre, à chaque fois, dans le registre *RP* : on aura $<y_1,...,y_h,y_k>$ pour le calcul de f au moment où l'on a besoin de y_k, et on laissera donc $<y_1,...,y_h>$ dans *RP*, de sorte qu'après le calcul de f, puis de h, la machine puisse retrouver y_h pour le calcul suivant de la valeur de h.

Cette façon de sauver les valeurs successives de y puis de les retrouver revient à ***empiler*** les valeurs $y_1,...,y_h$ l'une après l'autre, la dernière valeur fournie se trouvant, par construction, au ***sommet de la pile***. Pour retrouver la valeur de y au moment opportun, on

dépile en prenant la valeur qui se trouve au sommet de la pile. Lorsque la pile est vide ($RP = 0$), on fini de calculer toutes les valeurs de $f(x_1,...,x_k,z)$ intermédiaires. D'où le programme suivant, dans lequel les registres $R1,...Rk$ contiennent les valeurs initiales de $x_1,...,x_k$, RY celle de y et RW contient la valeur de f, et où les programmes de calcul des fonctions h et g sont censé restituer aux registres $R1,...,Rk$ et RY leurs valeurs initiales en fin de calcul :

```
            NUL  RP        ; RP (la pile) et RW sont initialisés
            NUL  RW        ; à zéro
; 2 :                      ;
            DSZ  RY,8      ; y = 0?
                           ; sinon, on calcule h: x1,...,xk, y
                           ; sont prêts mais pas f(x1,...,xk,y) :
            ST   E         ; donc, on empile y dans RP
; 4 :                      ;
            ST   2         ; on appelle le calcul de f
                           ; à présent, RW := f(x1,...,xk,y-1)
; 5 :                      ;
            DSZ  RP,400    ; la pile est-elle vide?
            INC  RP        ; on restaure RP
            ST   D         ; on dépile la bonne valeur de y
; 6 :                      ;
            ST   M2        ; RW := h(x1,...,xk,y-1,w)
                           ; où w := f(x1,...,xk,y-1)
; 7 :                      ;
            ST   5         ; calcul de f terminé
; 8 :                      ; cas où y = 0
            ST   M1        ; on calcule g :
                           ; à l'instruction suivante, 9,
; 9 :                      ; RW := g(x1,...,xk) = f(x1,...,xk,0)
            ST   5         ; calcul de f terminé
; E :                      ;
            programme d'empilement
            ST   4         ; retour à la suite du calcul
; D :                      ;
            programme de dépilement
            ST   6         ; retour à la suite du calcul
; M1 :                     ;
            programme du calcul de g
            ST   9         ;
; M2 :                     ;
            programme du calcul de h
            ST   7         ;
; 400 :                    ; dernière instruction
```

Au chapitre 4, nous verrons une implantation différente du mécanisme de la pile, qui conduira à une modification de la condition d'arrêt du programme (ici pile vide), que nous indiquerons alors.

En attendant, observons que nous n'avons pas vraiment besoin d'empiler les valeurs de y : en effet, la valeur de f est w_y où $w_0 = g(x_1,...,x_k)$ et $w_{i+1} = h(x_1,...,x_k,i,w_i)$. Il suffit, pour s'en convaincre de faire le calcul depuis $f(x_1,...,x_k,y) = w_{y-1}$, les w_i étant alors inconnus, puis de le reprendre, à l'envers, de w_0 à w_{y-1}. Ceci nous conduit à un programme beaucoup plus simple, page ci-contre.

2.3. Machine de von Neumann.

Classiquement, cette machine se compose d'une ***unité centrale*** (ou ***de contrôle***) qui supervise le fonctionnement de la machine, d'une ***unité de calculs***, d'une ***unité d'échanges*** que nous appellerons plus simplement ***entrées/sorties***, et d'une ***mémoire*** finie. L'unité centrale et l'unité de calcul tendent actuellement à être intégrées au sein d'un processeur unique. Ces deux unités, comportent des registres, comme les machines de Minsky, mais à la différence de ces dernières, les registres sont ici en nombre fini

Ce programme utilise un registre *RI* initialisé à zéro sur lequel on recopie *RY*, tout au début du programme. SI *RI* est alors nul, on a $y = 0$ et donc la valeur cherchée, w, est donnée par $w = w_0$. Sinon, on décrémente *RI* et on incrémente *RY* chaque fois qu'on a calculé w_i par le biais de h. Ainsi, à la fin du programme, *RY* retrouve la valeur de y donnée initialement.

```
            NUL  RW
            NUL  RI
; 2 :                    ;
            DSZ  RY,5      ;
            INC  RI        ;
            ST   2         ;
; 5 :                    ; à présent, RI = y et RY = 0
            ST   M1        ; RW := g(x1,...,xk)
; 6 :                    ;
            DSZ  RI,400    ; RI = 0? si oui, on a calculé f
            ST   M2        ; sinon : RW := h(x1,...,xk,y,wy)
; 8 :                    ;
            INC  RY        ; y := i+1
            ST   6         ;
; M1 :                   ;
            programme du calcul de g
            ST   6         ;
; M2 :                   ;
            programme du calcul de h
            ST   8         ;
; 400 :                  ; dernière instruction
```

fixe, et surtout, ils contiennent des nombres positifs ou nuls *majorés* par un nombre également fixé N. La limite ainsi imposée aux registres est compensée en partie par la présence de la *mémoire*. En partie seulement, car la mémoire est *finie*.

REMARQUES : Si on adjoint une mémoire infinie à la machine de von Neumann, on retrouve les machines de Minsky : il suffit d'utiliser la mémoire pour stocker des nombres naturels arbitrairement grands afin de simuler les registres d'une machine de Minsky. La mémoire infinie permet, au demeurant, de simuler autant de registres qu'on le désire.

Le fait que le nombre des registres soit fixe n'est pas une grande restriction comme le montre l'existence de machines de Minsky *universelles*.

La présence de cette mémoire est une des raisons de la mise en évidence de l'unité centrale dont un des rôles est d'exécuter les instructions permettant de gérer la mémoire. Comme cette mémoire a pour première raison d'être de fournir des registres supplémentaires, elle ne contient, de façon réelle, que des *nombres*. De ce fait, tout ce que traite la machine de von Neumann doit revêtir un format numérique, qu'il s'agisse de programmes ou de données. Il n'y a donc pas de distinction *intrinsèque* possible dans une telle machine entre instructions de programmes et données. Une telle distinction est nécessairement du niveau de l'interprétation et ne peut se faire, par conséquent, qu'en décidant d'un critère *extérieur*. Nous en verrons plus loin les modalités concrètes.

La machine de von Neumann est un modèle théorique de machine réelle, suffisamment proche de la réalité pour que son étude détaillée soit faite sur une machine réelle. Nous nous contenterons ici de relever les différences de ce modèle avec les machines de Minsky.

La différence essentielle, après la finitude fondamentale de la machine de von Neumann, est le fait que cette dernière est une machine de *dialogue* avec l'homme. Une machine de Minsky, à la limite, calcule 'pour soi' : on suppose que les registres ont des valeurs données *au départ* et on s'intéresse, en général, à la valeur d'*un* registre, distingué à l'avance, dans lequel se trouvera le résultat du calcul. Pour une *utilisation concrète* de cette machine il manque deux mécanismes : l'un qui indique comment

affecter, *initialement*, les registres de la machine; l'autre qui permette de prendre connaissance *commodément* d'un résultat.

Cette caractéristique va entraîner deux modifications fondamentales de la machine de von Neumann : la présence de dispositifs d'entrée/sortie dans l'architecture de la machine, et l'organisation des actions d'entrée/sortie dans le fonctionnement de la machine. Ces actions sont des instructions particulières qui interrompent le fonctionnement automatique du processeur et sont appelées de ce fait **interruptions**. C'est grâce à elles que le *dialogue* homme-machine est possible.

Il importe de voir à présent, que du point de vue de la machine, les interruptions jouent le rôle moteur : en effet, au sens strict, la machine de von Neumann n'a pas d'instruction d'arrêt. Son fonctionnement repose sur l'exécution de la boucle infinie suivante :

```
                Démarrage
; 1 :
                AttenteCommande
                EnregistrementCommande
                ExécutionCommande
                st 1
```

qui se termine, en pratique, lorsque l'utilisateur coupe l'alimentation électrique de la machine. Les deux programmes *AttenteCommande* et *EnregistrementCommande* constituent les deux aspects de l'*interruption*.

> Naturellement, le schéma ci-dessus est un schéma très général. Sur les machines réelles le programme *EnregistrementCommande* se décompose en une analyse de la commande entrée par l'utilisateur puis un enregistrement proprement dit : la machine détermine d'abord si la commande entrée fait partie des instructions qu'elle 'connaît', puis, dans un certain nombre de cas, elle effectue une vérification de conditions permettant d'assurer que la commande pourra être exécutée. L'enregistrement lui-même peut être suivi, avant l'exécution, lorsque la machine fonctionne avec plusieurs utilisateurs ou en mode multitâche, de la détermination, en fonction de critères de priorité, de la commande qui va être réellement exécutée à cette étape.

La mémoire de la machine est habituellement décomposée en mémoire morte (anglais ROM : Read Only Memory) et en mémoire vive (anglais RAM : Random Access Memory).

La mémoire morte contient le programme de base de l'unité centrale : celui qui définit toutes les instructions réelles de la boucle ci-dessus. En particulier, il contient les instructions nécessaires à l'exécution des interruptions.

La mémoire vive est un espace directement accessible à l'unité centrale et au processeur de calcul. Elle est utilisée par la machine pour exécuter le programme de base. Une partie de cette mémoire est à la disposition de l'utilisateur afin de faire exécuter par la machine les programmes qu'il aura rédigés. L'organisation de cette mémoire est un aspect propre à la machine de von Neumann : puisque les machines de Minsky n'ont pas de mémoire.

La mémoire est divisée en unités de mémoires considérées comme contiguës. Du point de vue de la programmation, chaque unité de mémoire vive est un registre en puissance. L'organisation de la mémoire repose sur une numérotation de toutes les unités qui la compose : c'est le système de l'adressage que nous décrirons en détail, au chapitre suivant, pour les processeurs 80x86.

Les instructions données à la machine sont, a priori, écrites dans le même langage, qu'il s'agisse du programme qu'elle contient en mémoire morte ou des programmes écrits par l'utilisateur. Bien entendu, rien n'interdit à ce dernier de définir un langage qui lui soit propre dans le cadre d'un programme écrit en instructions machine. C'est ce que font couramment les logiciels de compilation d'un langage de haut niveau, principalement en raison de la multiplicité des machines réelles,

suffisamment différentes entre elles d'un fabricant à l'autre (et souvent, pour un même fabricant, d'un modèle à un autre). Il y a souvent au moins deux intermédiaires entre les instructions machines et le programme auquel accède habituellement l'utilisateur.

Enfin, les entrées/sorties des machines réelles se sont considérablement développées. En particulier, elles permettent l'adjonction de mémoire à la machine. Cette mémoire ajoutée, a le plus souvent une fonction de *stockage* de l'information. Elle est alors appelée *mémoire de masse*. Elle se distingue des extensions mémoire qui ont pour objet d'augmenter la mémoire vive de la machine et donc, pour simplifier, qui ont pour but d'ajouter des registres. La différence essentielle entre ces deux types de mémoire est le caractère *temporaire* de la première (jsuqu'à la coupure de l'alimentation électrique) et le caractère *permanent* de la seconde. En outre, la vitesse d'accès du processeur à la mémoire est beaucoup plus rapide avec la mémoire vive qu'avec les mémoires de masse. D'autres entrées/sorties permettent de recueillir des informations diverses. Chaque appareil affecté à une telle utilisation est un ***périphérique***. Il en est deux privilégiés : le clavier et l'écran (qui aujourd'hui, remplace les cartes perforées qui ont subsisté jusqu'à la fin des années soixante-dix).

Nous verrons au chapitre suivant, dans le cas du processeur 80x86, les principales interruptions qui permettent de gérer les périphériques.

A présent que nous avons précisé les outils théoriques sur lesquels s'appuient le reste de l'ouvrage, nous allons passer au point suivant de notre démarche : le recensement des moyens matériels et logiciels afin de les mettre en oeuvre dans une programmation que nous chercherons la plus claire possible.

Cependant, comme nous l'avons indiqué au début de ce chapitre, nous avons choisi un modèle parmi ceux, nombreux, qui existent. Signalons, au lecteur qui souhaiterait compléter son information dans ce domaine, les deux autres modèles de base que sont la machine de Turing et le λ-calcul. La machine de Turing, qui date de 1936, outre son intérêt théorique considérable, a eu l'immense mérite de démontrer la possibilité d'implanter *concrètement* les concepts de la récursivité. On consultera, par exemple [2], [3] ou [5] pour s'initier à cette notion. Le λ-calcul, qui date de la même époque, a inspiré, *a posteriori* semble-t-il, un des langages de programmation les plus anciens et des plus prisés, *LISP*, prototype de la famille des langages de programmation fonctionnelle. Les développements du λ-calcul jouent également un rôle très important dans un domaine capital : celui de la preuve des programmes, dont nous aborderons quelques aspects pratiques aux chapitres 4 et 5.

CHAPITRE 3

LES PROCESSUS.

Les concepts que nous avons décrits au chapitre précédent, en particulier, celui de la machine de von Neumann sont mis en oeuvre dans un objet matériel : le processeur de tout ordinateur. Pourquoi *processeur?* Tout simplement parce qu'il met en action des ***processus*** que nous pouvons qualifier d'***élémentaires*** et qui englobent tout à la fois des actions *élémentaires* et les moyens de les déclencher.

Nous ne décrivons pas, dans cet ouvrage, le fonctionnement des circuits électroniques permettant l'implantation de ces processus. Nous renvoyons le lecteur intéressé par cette importante question à l'ouvrage clé [4]. Nous décrivons par contre le langage d'assemblage du processeur 8086. D'une part, tous les programmes écrits dans ce langage fonctionnent sans problèmes sur les appareils dotés des processeurs 80x86 avec $x \geq 2$. D'autre part, le principe de fonctionnement de ces mêmes processeurs est le même que le 8086 (à l'addition près de nouveaux registres pour les 80386 et 80486) tant que l'on se trouve dans le mode dit *réel* de ces processeurs. En mode protégé, les mêmes instructions fonctionnent sur la base d'une *interprétation* différente : nous en donnerons un aperçu succinct en fin d'ouvrage.

Mais avant de donner les instructions du langage, nous allons préciser le mode de représentation de la mémoire utilisée par l'ordinateur et l'outil que nous utiliserons pour écrire des programmes que le lecteur pourra tester sur sa propre machine.

Debug : le constructeur.

Cet outil, dont on ne vantera jamais assez les mérites, est fourni par le *DOS*, afin de mettre au point des programmes écrits dans le langage machine symbolique du processeur. Il s'agit d'un *constructeur* au sens plein. *DEBUG* permet de connaître le contenu des registres du processeur à un moment donné, d'afficher une zone définie de la mémoire vive, de lire des fichiers, ou même directement des secteurs d'une mémoire de masse (disquette ou disque dur), d'assembler et de désassembler du code, d'effectuer des opérations d'écriture ou de comparaison en mémoire vive, d'écrire sur des fichiers ou, directement, sur des secteurs d'une mémoire de masse, de communiquer avec tous les périphériques et, enfin, d'exécuter des séquences d'instructions en tout ou partie ou encore, en pas à pas.

> *Stricto sensu*, *DEBUG* n'affiche pas le contenu des registres du processeur, mais ne fait que *simuler* un tel affichage. Très exactement, lorsqu'on lui soumet des instructions que l'on fait exécuter par son intermédiaire, *DEBUG* indique l'état des registres après l'exécution de ces instructions, ce qui ne correspond pas au contenu *actuel* de ces registres puisqu'après les instructions qu'on lui soumet, *DEBUG* en exécute d'autres : celles qui rendent possible l'affichage des registres. Concrètement, *DEBUG* réalise cette simulation à l'aide d'un jeu de registres *bis*, souvent appelés registres *virtuels*, qu'il initialise au début de sa session et qu'il entretient en fonction des actions de l'utilisateur, notamment

des instructions qui lui sont soumises. Il suffit à *DEBUG* de recopier sur les registres bis le contenu des registres du processeur juste après l'exécution d'une instruction.

Nous donnerons plus loin les commandes principales de *DEBUG* avec leur syntaxe[1]. Mais pour permettre au lecteur de s'initier vraiment à l'assembleur, et il faut pour cela s'exercer le plus tôt possible, nous commençons par le minimum indispensable.

On appelle *DEBUG* en entrant son nom au clavier après le prompt du *DOS*. *DEBUG* répond par son propre prompt - après lequel il attend une commande dont le nom se réduit à une lettre :

```
  C:\>debug
-
```

Pour sortir de *DEBUG*, on entre la commande *q* (quitter) :

```
-q
  C:\>
```

Appelons *DEBUG* et donnons lui la commande *r* (suivi du retour chariot). On obtient[2] :

```
  C:\>debug
-r
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1358 ES=1358 SS=1358 CS=1358 IP=0100 NV UP EI PL NZ NA PO NC
1358:0100 06 PUSH ES
-
```

Par cette commande, *DEBUG* indique le contenu des treize ***registres du processeur*** désignés par un nom, soit, comme ci-dessus et dans cet ordre : *AX*, *BX*, *CX*, *DX*, *SP*, *BP*, *SI*, *DI*, *DS*, *ES*, *SS*, *CS* et *IP* puis, de façon conventionnelle, le contenu du ***registre des indicateurs*** par mots de deux lettres, *NV*, *UP*, *EI*, *PL*, *NZ*, *NA*, *PO* et *NC* dans l'exemple ci-dessus. A la troisième ligne, *DEBUG* indique l'instruction suivante, celle qui serait exécutée dans le code correspondant au contenu de la mémoire explorée par *DEBUG* si on exécutait ce code. L'instruction est entièrement caractérisée : *DEBUG* donne son adresse, ici **1358:0100** sous une forme expliquée plus loin, son code hexadécimal, ici **06**, et la traduction symbolique de ce code, ici **push es**.

On peut modifier, sous *DEBUG*, le contenu d'un registre. Il suffit de faire suivre le nom de commande *r* du nom du registre à modifier. *DEBUG* affiche le contenu du registre puis, à la ligne suivante, affiche : et attend la réponse de l'utilisateur. Si celui-ci ne veut pas modifier le registre, il répond au prompt : par un retour chariot :

```
-r
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1358 ES=1358 SS=1358 CS=1358 IP=0100 NV UP EI PL NZ NA PO NC
1358:0100 06 PUSH ES
-r cx
CX 0000
:36a7
-r cx
CX 36A7
:
-
```

---

[1] Il y a des langages partout! Celui de *DEBUG*, cependant, est très rudimentaire...

[2] Pour des raisons de place, nous avons choisi de reproduire ce que *DEBUG* affiche à l'écran en petits caractères, afin que le contenu d'une ligne d'écran n'occupe pas plus d'une ligne de texte. Pour la même raison, nous modifions la présentation de la commande d'affichage de *DEBUG* (*cf.* paragraphe 2.5.).

Pour obtenir le contenu du registre des indicateurs, on utilise la commande *rf* :

```
-rf
NV UP EI PL NZ NA PO NC -
```

et, comme précédemment, on répond par un retour chariot si on ne veut pas modifier ce registre.

Nous voyons à présent tous ces points en détail.

3.1. Registres et mémoire vive.

3.1.1. Registres

L'unité centrale du processeur dispose de registres qui lui servent à organiser et à faciliter l'exécution des instructions. Dans les processeurs de la famille 8086, ces registres sont au nombres de quatorze, désignés par les noms indiqués précédemment et habituellement classés selon leur emploi par les instructions du langage :

les registres généraux *X* : *AX*, *BX*, *CX*, *DX*;
les registres *P*ointeurs : *BP*, *SP*;
les registres *I*ndex : *DI*, *SI*;
les registres de *S*egments : *CS*, *DS*, *ES*, *SS*;
le registre d'*I*nstruction : *IP*;
le registre des indicateurs : (*F*).

Pour ce dernier, la parenthèse autour de *F* signifie que cette lettre ne peut pas être utilisée comme identificateur du registre auquel on accède à l'aide d'instructions particulières.

Les registres sont utilisés par le processeur lors de son fonctionnement. Leur rôle est essentiel : ils caractérisent complètement l'état de l'unité centrale après chaque exécution d'une instruction, ce qui permet d'interrompre un programme afin d'en exécuter un autre puis, une fois ce second programme terminé, reprendre le premier, là où on en était resté au moment de l'interruption. Ceci vaut non seulement pour deux programmes, mais, par récurrence, pour une suite finie quelconque de programmes, quelle que soit la façon dont ils sont appelés, mutuellement ou successivement. C'est ce qui rend possible le rôle moteur des interruptions dans le fonctionnement de la machine de von Neumann.

Les quatorze registres du 8086 comportent chacun seize bits : on dit que leur taille est celle d'un **mot**, c'est-à-dire deux **octets** puisque un octet représente huit bits. Afin de faciliter les manipulations sur certaines opérations et les accès à une mémoire organisée sur la base de l'octet, chacun des registres généraux et eux seuls, peut être décomposé en deux demi-registres d'un seul octet : le registre *RX* est partagé en *RH*, partie *haute* et *RL*, partie *basse*, nous verrons la raison de ces appellations au paragraphe 3.1.2. (notation hexadécimale). D'où les registres 'supplémentaires' *AH*, *AL*, *BH*, *BL*, *CH*, *CL*, *DH*, *DL*.

Exemple, sous *DEBUG* :

```
-a
1388:0100 mov ah,B4
1388:0102 mov al,7E
1388:0104
```

```
-t

AX=B400 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1388 ES=1388 SS=1388 CS=1388 IP=0102 NV UP EI PL NZ NA PO NC
1388:0102 B07E MOV AL,7E
-t

AX=B47E BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1388 ES=1388 SS=1388 CS=1388 IP=0104 NV UP EI PL NZ NA PO NC
1388:0104 008B0E80 ADD [BP+DI+800E],CL SS:800E=00
-
```

Comme on le constate sur cet exemple, le contenu des registres nous est pré par *DEBUG* sous la forme d'un nombre de quatre chiffres en base seize, c'est-à seize chiffres en base deux. Toutes les informations qui vont au processeur viennent passent par ce format que nous étudions au prochain paragraphe.

Cependant, comme nous l'avons vu au chapitre précédent, on ne ferait pas chose si on ne disposait que de quatorze registres de la taille d'un mot. En partic du point de vue de l'efficacité, l'apport d'une mémoire est essentiel. Cette mémoi décompose en unités repérées au moyen d'un système d'adressage que nous étu plus loin et qui utilise des nombres. De ce fait, les registres de la machine contienn plus souvent des données et des ***adresses***. A cet effet, les registres du proce reçoivent une spécialisation qui se traduit, notamment au niveau des instructions des restrictions dans l'utilisation de certains registres : nous le verrons en détail lo l'étude de l'adressage et des instructions. Nous résumerons ensuite l'utilisatio registres à la lumière de ces explications, sans jamais oublier qu'il s'agit de conven non obligatoires dans la limite des restrictions qu'impose le processeur.

La notation hexadécimale.

Nous avons indiqué, au chapitre 1 le codage en base seize des instruc machines, codage constituant un intermédiaire 'transparent' du codage bi fonctionnant réellement.

Convention :

Afin de pouvoir distinguer l'écriture hexadécimale d'un nombre de son éc décimale, nous écrirons l'écriture hexadécimale en caractères gras et toujours pré d'un zéro si la représentation ne contient pas de chiffre littéral (**A** à **F**). Nous ferons pas dans les textes de programmes écrits sous *DEBUG*, pour la simple raiso DEBUG ne connaît pas autre chose que le système hexadécimal.

Les huit bits d'un octet sont numérotés de 0 à 7 par ordre décroissant quan lit de gauche à droite. Ainsi l'octet 11001101 se détaille-t-il :

7	6	5	4	3	2	1	0
1	1	0	0	1	1	0	1

On utilise couramment cette numérotation des bits pour des groupements grands que l'octet. Pour le mot, constitué de deux octets, par définition, on parle de de 0 à 15 (nous écrirons toujours en base *décimale* les numéros de bit).

Un octet permet donc de représenter tous les entiers de 0 à $255 = 2^8-1$, en c dérant que la suite (de gauche à droite) des bits qui le constitue est l'écriture en deux d'un tel nombre. Comme $256 = 16^2$, le nombre représenté par l'octet s'éc base seize à l'aide de deux chiffres seulement. Le passage de la base deux à la seize et réciproquement est très facile. En effet, si $a = a_0 + ... + a_k2$

$j = 0,...h$, $b_j = a_{4j} + a_{4j+1}2 + a_{4j+2}2^2 + a_{4j+3}2^3$, avec, si nécessaire, à partir de $i = k$, les a_i complétés 'à gauche' par des 0 de façon à obtenir $k = 4h$. On en déduit l'algorithme suivant :

2 -> 16 : on regroupe les bits par quatre ($16 = 2^4$) et chaque groupe est remplacé par le chiffre hexadécimal qu'il représente ; ainsi : 11001101 s'écrit **CD** car 1100 représente 12 soit **C** en hexadécimal et 1101 représente 13 soit **D**.

16 -> 2 : on développe chaque chiffre hexadécimal sur quatre bits représentant l'écriture en base 2 de ce chiffre :
7C6 s'écrit en binaire 011111000110 puisque **7** s'écrit 0111, **C** s'écrit 1100 et **6** s'écrit 0110.

Les entiers en machine.

La notation hexadécimale constitue la base d'écriture du codage symbolique machine. Toutefois, la base numérique de ce codage est bel et bien l'octet, c'est-à-dire 16^2 soit 256, l'hexadécimal n'étant qu'un intermédiaire commode de représentation[1]. En outre, dans la famille des processeurs 80*x*86, les nombres sont écrits *à l'envers* : les décimales correspondent aux puissances croissantes de 256 sont écrites à des emplacements consécutifs de la mémoire d'adresses croissantes, ce qui est le sens contraire à la représentation usuelle des nombres quand on les lit de gauche à droite. De ce fait, les registres, qui ont une taille de 16 bits, présentent une inversion qu'il ne faut jamais oublier : si, par exemple, *AX* contient **1234**, la représentation machine en est **34 12**, car l'octet **34**, correspondant au coefficient de 256^0 vient avant l'octet **12** correspondant à 256^1. La même règle vaut, naturellement, pour les adresses absolues représentées par quatre mots : l'adresse **1234:5678** sera codée **78 56 34 12**, en vertu du principe que nous avons indiqué. La partie droite est écrite en premier, car elle correspond aux puissances *faibles* de 256. La partie gauche est écrite ensuite, puisqu'elle correspond aux exposants *forts* de la base dans le nombre considéré.

Dans un mot qui est, par définition, une unité de mémoire de 16 bits, on définit ainsi l'***octet de poids faible*** et l'***octet de poids fort***, respectivement **34** et **12** dans le mot **1234**. Nous avons là l'explication des noms donnés aux demi-registres associés aux registres généraux : *RH* pour la partie *haute* associée à l'octet *fort*, *RL* pour la partie *basse* associée à l'octet *faible*.

Nous avons à présent tous les éléments nécessaires pour décrire le système d'adressage du 8086. Cependant, ce processeur effectue des calculs **arithmétiques** sur des nombres toujours représentés de cette façon. Il convient de donner, à ce sujet, quelques explications.

Entiers relatifs.

En particulier, le processeur 'connaît' les entiers relatifs aussi bien que les entiers naturels que nous avons vus jusqu'ici. Comme indiqué au paragraphe 1.3, les machines de von Neumann se caractérisent par une borne imposée à la taille des entiers utilisés par le processeur. Dans le cas du 8086, cette borne est constituée par le mot. Les entiers naturels de la machine sont donc les nombres qui s'écrivent de **0000** à **0FFFF** (de 0 à 65535), c'est-à-dire de 0 à 2^{16}-1. Pour les entiers relatifs, la convention d'utilisation est la suivante : les calculs de la machine peuvent être interprétés comme des calculs modulo 2^{16}. Modulo 2^{16}, l'opposé de 1 est 2^{16}-1, c'est-à-dire **0FFFF**

---

[1] Pour nous, pauvres humains, 256 est une base beaucoup trop grande et 2, beaucoup trop petite. Si la base 8 est plus proche de notre notation décimale, elle allonge sensiblement l'écriture. D'où l'attrait de la base 16, pas trop éloignée : changer pour changer, on gagne au moins quelque chose, ne serait-ce que la condensation de l'écriture

(65535). D'une façon générale, l'opposé de a est 2^{16}-a, de sorte que, par convention, les entiers positifs ou nuls sont représentés par les nombres s'écrivant de **0000** à **7FFF** et les entiers strictement négatifs sont représentés par les nombres qui s'écrivent de **8000** à **0FFFF** (de -32768 à -1). On obtient ainsi les nombres relatifs de -32768 à 32767, c'est-à-dire depuis -2^{15} jusqu'à $2^{15}-1$ inclusivement.

Comme les registres généraux sont utilisés pour les opérations arithmétiques et qu'ils se subdivisent aussi en registres d'un octet, le processeur dispose d'instructions de calcul au format de l'octet. La même convention préside à la représentation des entiers : les entiers naturels iront donc de **00** à **0FF** (de 0 à 255), soit de 0 à 2^8-1, et les entiers relatifs de -128 à 127 soit **00** à **7F** (de 0 à 127 = 2^8-1) pour les nombres positifs ou nuls et **80** à **0FF** (de -128 = -2^7 à -1) pour les nombres négatifs.

Il importe de voir qu'il n'y a donc pas de différence, au niveau de la représentation dans le langage machine, entre les entiers naturels et les entiers relatifs. La différence est affaire d'*interprétation* : elle n'intervient qu'au moment de l'*action* et se fait donc *par les instructions*. De ce fait, il y a des instructions pour un type d'entiers et des instructions pour l'autre type. Dans la littérature spécialisée, les entiers relatifs représentés par la machine sont appelés entiers signés. C'est pourquoi l'on y parle d'opérations sur entiers signés ou sur entiers, sans précision, entendant par là les entiers naturels représentés en machine. D'où les quatre types d'expression qu'on rencontre dans la littérature spécialisée et qui se résument ainsi : ***opération*** (***octet***|***mot***) (***signée***). Cependant, la différence entre entiers naturels et relatifs étant affaire d'interprétation, le mot type sera réservé, dans ce qui suit, aux références : ce sera toujours l'indication mot ou octet.

3.1.2. Mémoire vive : l'adressage du 8086.

Différentes notions d'adresse.

L'unité de base de la mémoire du 80*x*86 est l'octet. La mémoire est divisée en octets repérés par un *numéro* qu'on appelle ***adresse*** de l'octet. L'ensemble de la mémoire que l'on peut numéroter est appelé ***espace adressable***. Les registres du processeur étant des mots et non des octets, l'espace adressable à partir d'un registre est donc de 16^4 octets soit 65536, les 64 Ko qui furent, un certain temps, une limite aux possibilités du *DOS*, sur laquelle nous reviendrons plus loin. L'amélioration des performances des matériels ont rendu possible la définition d'un espace adressable plus grand : 16^5 octets, soit 1048576 octets ou encore 1024 Ko. Le format de 20 bits étant peu commode et surtout, imposant une refonte importante du langage, les impératifs de compatibilité ont conduit à la représentation de l'adresse de 20 bits d'un octet de la mémoire, que nous appellerons ***adresse physique***, à partir de *deux mots* de la façon suivante.

On appelle ***adresse absolue*** (on dit encore ***adresse logique***), un couple M_1:M_2 de deux mots, M_1 étant appelé ***adresse de segment*** (en abrégé ***segment***) et M_2, ***adresse relative*** ou encore ***de déplacement***[1] (en abrégé ***déplacement***). Cette adresse est, par définition, celle de l'octet d'adresse physique $16*M_1 + M_2$. Ainsi, l'adresse absolue **1234:5678** définit-elle l'adresse physique **179B8** puisque, en hexadécimal, **10*1234 + 5678 = 179B8**. Symboliquement, une adresse absolue s'écrit donc :

segment : déplacement

[1] Depuis l'origine (du segment) que constitue l'adresse relative 0000.

Ne pas oublier, par conséquent, qu'on ne peut jamais écrire l'adresse physique en code machine.

Remarquons immédiatement que les adresses absolues définissent une surjection sur l'espace adressable qui est loin d'être injective : par exemple, **1244:5578**, **1254:5478**,... définissent la même adresse physique que **1234:5678**, à savoir **179B8**.

Zones, paragraphes, segments.

On appelle ***zone*** de la mémoire, toute partie de l'espace adressable dont les adresses physiques sont consécutives. On appelle ***segment***, une zone de la mémoire dont tous les octets peuvent être définis avec la même adresse de segment. Un segment comprend donc au plus 65536 octets. Enfin, s'agissant des adresses, on dit encore ***adresse effective*** pour l'adresse relative.

Notation : Nous conviendrons de désigner une zone par l'adresse de son premier octet suivi de l'adresse du premier octet qui suit la zone. S'il faut indiquer l'adresse de segment, elle le sera seulement pour le premier octet. La longueur (ou taille) de la zone est obtenue en retranchant la seconde adresse de la première.

Exemple : la zone de **0104** à **0340** comporte **023C** octets contigus depuis l'octet **0104** ; si l'adresse de segment est **13CE**, nous parlerons de la zone **13CE:0104** à **0340**.

Les seize bits de poids fort d'une adresse physique constituent un mot que l'on peut interpréter comme une adresse de segment. Il existe une unique zone mémoire telle que pour chaque adresse physique lui appartenant, le mot défini par les seize bits de poids fort soit toujours le même. Cette zone unique, composée de seize octets est appelée ***paragraphe***, et les seize bits de poids fort communs constituent, par définition, un ***numéro de paragraphe***. L'adresse absolue du début d'un paragraphe est donc de la forme **YYYY:XXX0**, où **Y** et **X** sont des chiffres hexadécimaux quelconques éventuellement différents d'une occurrence à l'autre. L'espace adressable comprend donc 65536 paragraphes. Notons enfin que s'il n'y a pas unicité de l'adresse absolue associée à une adresse physique, l'adresse physique définit par contre un numéro de paragraphe unique : ce numéro peut être regardé comme une *projection* de l'adresse physique.

Exemple : la zone de **13CE:0104** à **0340** n'est pas un paragraphe, ni même un nombre entier de paragraphe ; le début de cette zone se trouve dans le paragraphe **13CE:0100** à **0110** désigné simplement par **13CE:0100**.

EXERCICE :

Combien y a-t-il d'adresses absolues distinctes pour une même adresse physique?

On peut considérer cet adressage d'une façon plus géométrique. En effet, un déplacement peut être interprété comme une ***abscisse locale*** à l'intérieur d'un segment. Le numéro de paragraphe déterminant le segment peut être considéré comme une 'origine' des abscisses locales, ou mieux une abscisse, à une échelle différente, d'une telle origine. Notons, cependant, que par rapport aux abscisses habituelles, celles que nous venons de définir ont une propriété particulière : elles ne sont jamais négatives.

Quand l'adresse donnée est une adresse relative, il faut que, par ailleurs, l'adresse de segment correspondante soit connue. Elle est souvent implicite, l'instruction où est utilisée cette adresse étant, en général, associée à un des quatres registres de segments : *CS*, *DS*, *ES* ou *SS*. Nous verrons au chapitre 7 la raison fondamentale de ces associations implicites. Retenons qu'elles ont été définies de la façon suivante, permettant de retenir le nom de ces registres. Le registre *CS* est utilisé pour tout ce qui concerne le code du programme lui-même. Le registre *DS* est utilisé pour la manipulation des données. Le registre *SS* intervient dans les transferts vers et depuis la

pile[1], que nous verrons plus loin. Le registre *ES* est un registre pour données supplémentaires, intervenant dans certaines instructions de manipulation de caractères ou dans certaines interruptions.

Enfin, dernier point de terminologie : on appelle ***adresse mot***, respectivement, ***adresse paragraphe***, toute adresse multiple de deux, respectivement, de seize. Cela revient à considérer toute la mémoire, depuis 0000:0000, comme partagée en mots ou paragraphes et à donner l'adresse du premier octet de la nouvelle subdivision.

L'adressage indirect.

La représentation des adresses dans le code machine s'effectue selon *deux* modes qui donnent toute sa puissance à ce langage : l'***adressage direct*** et l'***adressage indirect***.

Le premier mode consiste à écrire l'adresse absolue, l'adresse relative ou encore un registre dont la valeur est l'adresse relative. L'adressage indirect est ainsi appelé car il ne fournit pas directement l'adresse d'un objet, mais l'adresse du premier octet d'une zone mémoire dans laquelle on trouvera l'adresse relative ou absolue, selon le cas, de l'objet visé. On dira qu'un adressage direct *donne* l'adresse de l'objet, tandis que l'adressage indirect *pointe* sur l'adresse de l'objet.

Cette conception fait de chaque octet de la mémoire une variable du langage puisque chaque octet a deux composantes : son adresse et sa valeur, cette dernière se trouvant donc associée à l'adresse. Un ***pointeur*** est un mot dont la valeur est considérée comme une adresse. Le code machine permet d'étendre le statut de pointeur à des expressions limitées constituées à partir de registres et de nombres. Ces expressions sont données par la formule :

[RB + RI + N]

où *RB* désigne l'un des deux registres de *base* : *BX* ou *BP*, *RI* l'un des deux registres *SI* ou *DI* et *N* un nombre représentable par un mot, un ou deux des trois termes de cette formule pouvant être absent. La raison des restrictions imposées à *RB* et *RI* apparaîtra au chapitre 7.

Dans tous ces cas, la valeur de l'expression fournit une adresse *relative*. L'adresse absolue nécessite un segment. Sauf mention expresse du contraire, le langage machine considère que l'adresse de segment est contenue dans *SS* lorsque *BP* est le pointeur de base et *DS* dans tous les autres cas. Nous verrons plus loin comment désigner à l'unité centrale un autre registre de segment que celui qu'elle utilise implicitement.

Aussi est-il temps d'aborder la description du langage machine symbolique utilisé par les processeurs 80*x*86. Ce que nous faisons dans l'ouvrage en deux temps : une première fois, de façon opératoire et syntaxique, à partir du paragraphe suivant, une deuxième fois, au chapitre 7, à nouveau sur un plan syntaxique, mais plus proche du code binaire lui-même. La description que nous allons donner ne sera pas exhaustive. Nous savons déjà qu'il suffit en fait de peu d'instructions pour exécuter tout ce qu'on veut, aussi nous contenterons-nous d'une partie du jeu des instructions. Toutefois, nous ne chercherons en aucune façon un ensemble minimal. Le lecteur choisira peut-être de lui-même un sous-ensemble de celui que nous lui proposons : la marque du style passe aussi par le vocabulaire.

---

[1] En anglais, *stack*, d'où le nom.

3.2. Les instructions du 80x86 : généralités.

L'alphabet du langage est très simple : il s'agit d'un sous-alphabet de l'alphabet latin, dans lequel on ne distingue pas les majuscules des minuscules, et auquel s'ajoutent les dix chiffres arabes usuels, la virgule, les deux points et le symbole blanc, ces trois derniers symboles faisant office de ***délimiteurs*** de mots, le principal étant le symbole blanc. La virgule est un délimiteur particulier : il s'agit d'un séparateur entre deux mots. Les mots admissibles du langage machine symbolique sont les ***mnémoniques*** des instructions, c'est-à-dire, une 'syllabe' évocatrice de leur nom ou de leur fonction, ainsi que les noms des registres et demi-registres. Il s'y ajoute les mots **byte**, **far**, **ptr** et **word**. Enfin, le langage d'assemblage admet un mot supplémentaire : **db**, qui ne fait pas partie du langage machine symbolique mais qui, néanmoins, peut figurer dans un code symbolique ordinaire soumis à *DEBUG* ; nous expliquerons ce statut singulier au moment opportun.

Nous commencerons par indiquer des propriétés générales de la syntaxe des instructions du langage. Puis nous exposerons les principes généraux sur lesquels repose le fonctionnement des instructions. Enfin nous présenterons celles que nous avons choisies en une sorte de *dictionnaire* qui les regroupe selon leurs fonctions.

La syntaxe des instructions.

Nous faisons un premier classement des instructions : syntaxique, c'est-à-dire selon la façon dont elles s'écrivent.

Les instructions s'écrivent toutes, chacune sur une même ligne, une ligne contenant au plus une instruction, selon l'une des trois formes suivantes :

```
MNEMO
MNEMO  opérande
MNEMO  but,source
```

représentant, respectivement, les instructions sans opérande, les instructions à une opérande et les instructions à deux opérandes, désignés symboliquement ci-dessus par les termes `but` et `source`. Le terme `MNEMO` désigne le nom symbolique de l'instruction et la virgule, rappelée en caractère gras, est le séparateur attendu entre le but et la source, étant entendu qu'on peut rajouter de part et d'autre de la virgule, autant de blancs que l'on veut dans la limite de la ligne. Enfin, les opérandes d'une instruction ont un ***type*** : octet ou mot, selon qu'ils représentent, respectivement, un octet ou un mot.

Les instructions à deux opérandes obéissent à trois principes généraux dont voici les deux premiers :

> *Le but désigne toujours une adresse, la source désigne toujours un contenu*, ce qu'on peut écrire, symboliquement :
>
> **MNEMO adresse,valeur.**
>
> *Le but et la source doivent être du même type.*

On ne doit pas oublier que le principe de non ambiguïté s'applique en permanence : il permet parfois de simplifier les expressions et presque toujours, il permet de retrouver rapidement la syntaxe appropriée au problème concret que l'on examine.

Retenons dès à présent la dissymétrie des deux opérandes. L'adresse sera toujours une adresse directe ou indirecte (calculée) et la valeur, soit le contenu d'une adresse (donnée directement ou indirectement), soit le contenu d'un registre, soit une donnée nécessairement numérique sur un ou deux octets (un *mot* dans ce dernier cas), que nous appellerons ***donnée immédiate*** ou ***entrée immédiate***.

Par ailleurs, l'adresse d'un octet peut aussi bien désigner l'adresse du premier octet d'un mot. De ce fait, si le type de l'opérande valeur de l'instruction n'est pas évident, ce qui est le cas d'une entrée immédiate n'occupant qu'un octet, il faut préciser le type du but. C'est la fonction des mots word, byte et ptr qui servent à *qualifier* l'adresse : **word ptr**, pour indiquer qu'il s'agit de l'adresse d'un *mot*, **byte ptr** pour l'adresse d'un octet. Ces expressions sont, de ce fait appelées ***qualifiants*** et elles peuvent être abrégés, respectivement, en **wo** et **by**, ce que nous ferons systématiquement sous *DEBUG*.

Troisième principe concernant les instructions comportant deux opérandes :

Si la valeur n'est pas une donnée immédiate, un des deux opérandes au moins doit être un registre (ou demi-registre).

Comme nous l'avons indiqué en donnant les formules ci-dessus, une instruction peut ne comporter qu'un seul opérande : nous en verrons des exemples ci-après, avec certaines des instructions de gestion de la pile. Nous en verrons également au chapitre suivant concernant certaines opérations. Dans le cas des instructions de cette sorte, l'opérande fonctionne soit comme adresse, soit comme valeur, selon la tâche effectuée par l'instruction. Nous le verrons sur l'exemple des instructions de gestion de la pile.

Une instruction peut également ne pas comporter d'opérande. Il en est ainsi de l'instruction nulle :

```
NOP
```

qui a pour fonction de combler des 'vides' dans le code ou bien d'introduire un 'micro'-délai (3 cycles de temps machine), nous en verrons plus loin des exemples.

Le fonctionnement d'une instruction.

Comme nous l'avons indiqué au début de ce chapitre, les processeurs 80x86, nous dirons, dans tout ce paragraphe *le processeur* pour alléger l'exposé, utilise deux registres, le registre *IP* et le registre *CS* pour exécuter le programme qu'on lui soumet. En effet, l'exécution d'un programme par le processeur s'effectue toujours selon le même scénario : on ***charge*** le programme en mémoire, ce qui veut dire qu'on le *copie* d'un support de masse (ou de la *ROM*) sur une zone de la mémoire vive, puis on donne à *CS* et *IP* des valeurs dites initiales, telles que *CS:IP* soit l'adresse absolue du premier octet du programme chargé. A partir de cet instant, le processeur exécutera toujours l'instruction qui se trouve en *CS:IP*.

L'exécution de l'instruction s'effectue alors de la façon suivante : le processeur charge l'instruction dans des registres internes qui constituent le ***pipeline*** auquel le programmeur n'a pas accès ; puis il la décode : il traduit les bits lus en mémoire en signaux qui se répartissent dans ses circuits, ce qui conduit, en un deuxième temps, à l'exécution de l'instruction. Sauf indication contraire par une des instructions prévues à cet effet dans le langage et qu'il aurait rencontrée, analogues de l'instruction ST de la machine à registre, le processeur réactualise *IP* de façon à ce que *CS:IP* pointe sur l'instruction qui suit, en mémoire, celle qu'il vient d'exécuter. Nous dirons que *CS:IP* pointe alors sur l'instruction suivante.

Ce cycle se répète indéfiniment jusqu'à la coupure de l'alimentation électrique de la machine : nous retrouvons la boucle infinie de la machine de von Neumann que nous avions signalée au chapitre précédent.

Nous examinons, à présent, les différents types d'instruction que le processeur peut rencontrer, illustrant chacun d'eux par une ou deux instructions fondamentales.

Pour en expliquer le fonctionnement, nous ouvrons, à chaque fois, une session de *DEBUG*. Dans la section 3.4. de ce chapitre, nous regrouperons les autres instructions sous forme d'un *dictionnaire*. Nous procéderons, autant que possible, en allant du plus simple au plus complexe.

3.2.1. L'affectation.

Dans une machine à registre.

Nous avons fréquemment rencontré une *séquence* d'instructions de la forme :

```
            NUL  RU
            NUL  RV
; 2 :
            DSZ  RA,5
            INC  RU
            INC  RV
            ST   2
; 5 :
```

Cette boucle reporte le contenu du registre *RA* sur les registres *RU* et *RV*, la copie sur un deuxième registre ayant pour fonction de sauvegarder la valeur initiale de *RA* afin de la restaurer éventuellement dans une autre partie du programme où elle est à nouveau utilisée. Il s'agit donc d'un *transfert* d'information entre deux registres.

Le processeur possède une famille d'instructions chargées d'effectuer de tels transferts. On les appelle instructions d'***affectation***.

Dans le transfert simple que nous venons de voir, la valeur du registre *RA*, la *source* du transfert, est ramenée à zéro à la fin du transfert. Dans le cas de l'affectation, cette valeur n'est pas altérée. L'affectation réalise une *copie* de l'information du registre source sur le registre but. Pour une machine de Minsky, on simulerait l'affectation du contenu du registre *RA* sur le registre *RU* en ajoutant aux instructions ci-dessus la boucle suivante :

```
;5 :
            DSZ  RV,7
            INC  RA
            ST   5
; 7 :
```

Dans le 8086.

Une seule instruction suffit pour copier le contenu d'un registre sur un autre : **mov cx,ax** copie le contenu du registre *AX* sur le registre *CX* ; **mov bl,dh** copie le contenu du demi-registre *DH* sur le demi-registre *BL*. Mais, ce qu'il fait pour un registre, le processeur peut le faire également pour une case mémoire : il suffit de lui donner son adresse. Ainsi, **mov [bx+2A0],ax** copie le contenu de *AX* sur le *mot* mémoire d'adresse *u* où *u* est la valeur obtenue en ajoutant **02A0** à la valeur de *BX* ; de même, **mov [di],al** copie le contenu du demi-registre *AL* sur l'*octet* mémoire dont l'adresse est fournie par la valeur du registre *DI*. On constate, sur ces deux exemples, l'application du troisième principe indiqué ci-dessus. De plus, en vertu du principe de non-ambiguïté, l'indication du registre fixe automatiquement le type du but : celui du registre.

Au niveau du *langage symbolique*, nous avons les variantes d'une même instruction en fonction de la nature et du type de ses opérandes : l'instruction **mov**. Les formes que peuvent prendre ces variantes sont régies par une syntaxe qui se résument, dans le cas de l'instruction **mov** par :

```
MOV  but,source
```

où `but` et `source` sont un nom de registre, une adresse indirecte ou une donnée immédiate, les trois principes que nous avons donnés devant être respectés.

Illustration sous *DEBUG*.

Ouvrant une session de notre *constructeur*, donnons lui la commande *a* afin d'*a*ssembler le code qui sera entré à l'écran. Fort civilement, *DEBUG* répond en offrant, à chaque ligne, l'adresse absolue du premier octet libre, l'adresse de segment étant la valeur de *CS*, le registre affecté, avec *IP*, au repérage du code à exécuter. Ainsi, peut-on obtenir, nous le reverrons plus loin, le nombre d'octets occupés par l'instruction. On sort de la commande *a* par un retour chariot et on fait exécuter pas à pas le code situé à partir du dernier appel de la commande *a* par la commande *t* (*t*race) qui, comme *r*, fournit l'état des registres après l'exécution de la dernière instruction examinée par *t* :

```
-a
1358:0100 mov ax,1234
1358:0103 mov [110],ax
1358:0106
-t

AX=1234 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1358 ES=1358 SS=1358 CS=1358 IP=0103 NV UP EI PL NZ NA PO NC
1358:0103 A31001 MOV [0110],AX DS:0110=027E
-t

AX=1234 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1358 ES=1358 SS=1358 CS=1358 IP=0106 NV UP EI PL NZ NA PO NC
1358:0106 0E PUSH CS

-d 110 l 10
1358:0110 34 12 89 0E 9A 03 BB 19 4.......
1358:0118 04 8A 2F C6 06 B3 03 00 ../.....
-
```

Notons que, dans cette session, le contenu affiché de *CS* est bien l'adresse de segment que *DEBUG* attribue aux instructions qui lui sont entrées par la commande *a*.

Lors de cette session, nous avons entré la commande *d* (anglais *d*ump) suivie des paramètres : **110** *L* **10**. Cette commande affiche une zone de la mémoire. Le premier paramètre (ici **110**) donne l'adresse (relative) du début de la zone, le segment étant implicitement contenu dans *DS*. Le second paramètre, ici *L* **10**, indique le nombre d'octets de la zone (toujours en hexadécimal pour *DEBUG*).

La reproduction in-extenso d'une session sous cette forme prendrait trop de place. Aussi la résumerons-nous sous la forme suivante :

```
0110 0200 0210 0219  AX   CX   IP
5678 0000 0000 0000 1234 0000 0106  MOV  CX,[0110]
5678 0000 0000 0000 1234 5678 010A  MOV  [0200],AX
5678 1234 0000 0000 1234 5678 010D  MOV  WO [0210],4321
5678 1234 4321 0000 1234 5648 0113  MOV  BY [0219],1C
5678 1234 4321 001C 1234 5648 0118  MOV  CH,[0219]
5678 1234 4321 001C 1234 1C48 011C
```

Les registres sont initialisés par la commande *r* de *DEBUG* et les octets ou mots de la mémoire par la commande *e* :

```
-e 110 78 56
-
```

a pour effet d'affecter la *valeur* **5678** depuis l'octet **0110**.

Notons que l'exécution de l'instruction **mov cx,[110]**, place dans *CX* le mot dont le premier octet a pour adresse **110**. Remarquons ce que nous avions dit au sujet de l'inversion de notation des nombres dans le code machine : le contenu de *CX* et celui des octets **110** et **111** en témoigne. De la même façon, que faire si nous n'avions voulu

placer dans *CX* que l'octet d'adresse **110**? C'est qu'il y a place, dans *CX* pour deux octets : lequel choisir? Comme nous l'avons déjà remarqué, il suffit de prendre pour but un demi-registre, par exemple *CH*, comme dans la session ci-dessus.

Données immédiates.

De cette session, nous tirons également le fonctionnement de l'instruction avec une donnée immédiate. Compte tenu de sa nature, une donnée immédiate ne peut-être utilisée qu'en opérande *source*, jamais comme but. Le type d'une donnée immédiate n'est pas toujours certain : **012** peut aussi bien être considéré comme l'octet **12** que comme le mot **0012**. Aussi, lorsque le but n'est pas un registre, faut-il obligatoirement donner le type de la donnée immédiate, même lorsque celui-ci ne peut être que le type mot. De même, si on tente d'affecter un demi-registre par un mot, *DEBUG* rejettera l'instruction de la façon suivante :

```
-a
1358:0100 mov al,1234
                  ^ Erreur
1358:0100
```

et il attendra une nouvelle instruction ou un retour chariot.

REMARQUES :

Les registres de segment sont de type mot, ils ne peuvent donc être affectés que par une source de même type. Cependant, les registres de segment ne peuvent pas être affectés ni par une donnée immédiate, ni par un registre de segment.

L'instruction **mov** peut figurer dans un programme avec le registre *CS* en opérande but. *DEBUG* l'acceptera aux risques et périls du programmeur. Le 8086 l'exéutera et le résultat sera ce qu'on attend, sauf erreur du programmeur. Les 80*x*86, $x \geq 2$, refuseront l'instruction ; conséquence : un plantage sans gravité.

Le mot *IP* n'appartient pas au langage *machine* symbolique et le registre des indicateurs n'a pas de nom dans ce langage.

Si l'assembleur du processeur ne permet pas d'accéder au registre *IP* ni à celui des indicateurs (du moins pour le 8086, pour ce dernier registre) *DEBUG* permet de le faire sous son égide par le biais de la commande *r* : *r ip* pour affecter le registre *IP*, *r f* (ou *rf*) pour affecter le registre des indicateurs.

Affectation particulière : la pile.

Le processeur connaît une pile, dont nous avons vu l'importance au chapitre précédent. Elle prend ici la forme d'une *zone* mémoire affectée au stockage temporaire d'informations du type mot au fur et à mesure des besoins : on y *empile* une information en la plaçant *en avant* (adresse mot immédiatement précédente) des informations déjà stockées ; l'adresse de la dernière information stockée constitue alors, par définition, le ***sommet de la pile***. Quand on *dépile* des informations, on prend toujours en premier la dernière information enregistrée et le sommet de la pile *remonte*. Pour le processeur, le sommet de la pile est pointé par *SS:SP*. Le registre *SP* est diminué de deux *avant* d'empiler une information et augmenté de deux *après* en avoir dépilé une, et ceci dans la limite du segment défini par le registre *SS*. La valeur initiale de *SS:SP* et la taille réelle de la pile sont laissés à l'initiative du programmeur avec toutefois la restriction suivante : *SP* ne doit jamais pointer sur **0FFFF** au moment de dépiler ni sur 1 au moment d'empiler. Il suffit, pour ce faire, que l'adresse initiale de la pile soit une adresse *mot*.

Instructions.

Dans le cas de l'empilement, comme l'adresse du but est connue *implicitement* du processeur, il suffit de lui indiquer la source. D'où une instruction particulière, **push** pour effectuer cette opération. Symétriquement, il lui correspond une instruction **pop**

effectuant le dépilement et pour laquelle il suffit d'indiquer le but, la source, cette fois, étant implicite. La syntaxe de ces instructions est :

```
PUSH  source
POP   but
```

Comme il peut être utile d'opérer sur la pile sans modifier la valeur de *SP*, le processeur associe au registre *BP* le registre de segment *SS*, de sorte que pour lui, *SS:BP* pointe aussi sur la pile. Si, au départ, *BP* = *SP*, les instructions **push ax** et **pop ax** équivalent, respectivement, aux instructions :

```
   push  ax                  pop  ax
DEC  BP                  MOV  AX,[BP]
DEC  BP                  INC  BP
MOV  [BP],AX             INC  BP
```

où, anticipant sur le dictionnaire, nous utilisons les instructions **dec** et **inc** : **inc**, analogue de l'instruction Minsky vue au chapitre précédent, ajoute 1 à la valeur de son opérande tandis que **dec** retranche 1 de la valeur de son opérande. Ceci montre l'avantage à utiliser **push** et **pop** pour gérer la pile : gain de place pour le code et gain de temps pour l'exécution, car l'exécution d'une instruction **push/pop** a la même durée que le **mov** correspondant dans le cas d'un registre. Si la source ou le but est adressé indirectement, le gain obtenu en utilisant **push/pop** est encore plus grand puisque une instruction ne peut contenir au plus qu'une adresse indirecte.

Précaution.

Très souvent, le code et les données, plus ou moins entre-mêlés sont à une extrémité du segment, tandis que la pile commence à l'extrémité opposée. En grandissant, la pile peut *descendre* jusqu'au code et même le recouvrir. Dans ce cas, comme *IP* renvoie à une instruction effacée par le contenu de la pile, c'est le contenu se trouvant à l'adresse donnée par *IP* qui est considéré comme un code d'instruction : ce morceau de la pile aura en général une signification tout à fait étrangère à l'instruction qu'il a effacée et il peut en résulter n'importe quoi. Le plus souvent, le plantage de la machine[1].

> Il importe de noter que ceci ne peut pas être prévenu par le processeur lui-même : rien ne permet de distinguer le 'bon' code du 'mauvais' puisque toute suite de quelques octets peut être considérée comme du code[2]. L'utilisateur peut cependant se prémunir contre un débordement de la pile : il lui suffit de simuler dans son programme ce que font les compilateurs intelligents. C'est-à-dire qu'il se fixe, par exemple dans les données, deux mots pour retenir l'adresse du point le plus extrême du code (dans la direction de la pile) et de vérifier, avant chaque modification de *SP* que la nouvelle valeur reste au-delà du code, bloquant l'exécution du programme avec un message adéquat en cas de débordement intempestif par la pile.

Signalons que la plupart des compilateurs de langage évolué (*PASCAL, FORTRAN, C, ADA*,...), utilisent systématiquement la pile pour le passage des arguments d'un sous-programme à un autre. Nous examinerons comment cela est fait au chapitre 5. On peut procéder autrement et, par exemple, se constituer soi-même une pile, en utilisant les séquences d'instructions ci-dessus employées avec le registre *BP* ou d'autres, selon les possiblités du programme, dans la mesure où ce sont des registres acceptés pour formuler une adresse indirecte.

---

1 Quoique très petite, la probabilité que ce morceau de pile soit un code qui provoque des destructions de données plus ou moins graves n'est pas nulle : donc, prudence!

2 C'est ainsi dans le 8086 et, dans une moindre mesure, dans le mode protégé du 80286 et de ses suivants.

Pile et registre des indicateurs.

La pile peut également être utilisée pour y copier ou y prendre la valeur du registre des indicateurs. Deux instructions sont prévues à cet effet et elles n'ont pas d'opérande puisque, dans ce cas, la source et le but sont également implicitement connus du processeur. Ces instructions sont :

```
PUSHF
POPF
```

Le lecteur aura deviné que **pushf** copie le contenu du registre des indicateurs sur la pile et que **popf** transfère la valeur du sommet de la pile dans le registre des indicateurs (il peut ne pas y avoir copie).

Voici quelques exemples :

```
0200 0202 FFE8 FFEA FFEC FFEE  AX   SI   SP   IP
5678 0000 0000 0000 0000 0000 0000 0000 FFEE 0100  MOV  AX,1234
5678 0000 0103 1388 0FD6 0000 1234 0000 FFEE 0103  PUSH AX
5678 0000 1388 0FD6 1234 0000 1234 0000 FFEC 0104  PUSH [0200]
5678 0000 0FD6 5678 1234 0000 1234 0000 FFEA 0108  POP  SI
5678 0000 1388 0FD6 1234 0000 1234 5678 FFEC 0109  POP  [0202]
5678 1234 010D 1388 0FD6 0000 1234 5678 FFEE 010D
```

pushf/popf :

```
FFEA FFEC FFEE  AX  (F)   SP   IP
0000 0000 0000 0000 0202 FFEE 0100  PUSHF
0FD6 0202 0000 0000 0202 FFEC 0101  POP  AX
1388 0FD6 0000 0202 0202 FFEE 0102  ADD  AX,1C
1388 0FD6 0000 021E 0202 FFEE 0105  PUSH AX
0FD6 021E 0000 021E 0202 FFEC 0106  POPF
1388 0FD6 0000 021E 021E FFEE 0107
```

Affectation particulière : les adresses absolues.

Pour certaines opérations, il peut être utile de charger directement une adresse absolue sur un registre de segment et un autre registre pour le déplacement. Comme il est fortement déconseillé (voire interdit) de charger *CS* et préférable de le faire avec précautions pour *SS*, le processeur ne dispose que de deux instructions pour effectuer un tel transfert. Ce sont les instructions **lds** et **les** dont voici la syntaxe :

```
LDS  registre,source
LES  registre,source
```

`registre` étant un registre général, pointeur ou d'indice, `source` étant l'adresse d'une zone mémoire de 4 octets qui, de ce fait, ne peut être définie que par adressage indirect (si on désigne la source par la valeur d'un registre, on n'obtient qu'un mot alors qu'on en demande deux). Les deux premiers octets sont affectés à `registre` (ils constituent la partie *basse* du double mot), les deux suivants à *DS* (resp.*ES*), la règle d'inversion à l'intérieur d'un mot étant observée comme d'habitude. Voici une illustration sous *DEBUG* du maniement de ces instructions :

```
0200 0202  CS   ES   DS   AX   BP   SP   IP
1234 9ABC 1388 1388 1388 0000 0000 FFEE 0100  LDS  AX,[0200]
1234 9ABC 1388 1388 9ABC 1234 0000 FFEE 0104  PUSH CS
1234 9ABC 1388 1388 9ABC 1234 0000 FFEC 0105  POP  DS
1234 9ABC 1388 1388 1388 1234 0000 FFEE 0106  MOV  WO [0200],5678
5678 9ABC 1388 1388 1388 1234 0000 FFEE 010C  LES  BP,[0200]
5678 9ABC 1388 9ABC 1388 1234 5678 FFEE 0110
```

Remarquer les instructions **push cs** et **pop ds** utilisées pour redonner à *DS* sa valeur initiale (sinon les valeurs en *ES:BP* ne seraient pas celles qui figurent ci-dessus mais celles qui sont en **9ABC:0200** pour *BP* et **9ABC:0202** pour *ES*).

→ *Dictionnaire* : cf. **in, mov, movsb, movsw, lahf, lodsb, lodsw, out, pop, popf, push, pushf, sahf, stosb, stosw, xchg.**

3.2.2. Les branchements.

En expliquant le fonctionnement d'une instruction du processeur, nous avons fait référence aux instructions ST et DSZ d'une machine à registre parce qu'elles peuvent modifier l'ordre usuel de l'exécution des instructions dans un programme. Le processeur dispose d'instructions ayant la même fonction. Trois d'entre elles sont des analogues de l'instruction de saut inconditionnel : ce sont les instructions **jmp, call** et **int**. Nous les étudierons en premier lieu. D'autres instructions ne réalisent que la partie saut conditionnel de DSZ. Nous les étudierons sur le modèle de l'instruction **jcxz.** Par ailleurs, le processeur réalise l'implantation d'une boucle, dont nous avons vu l'importance au chapitre 2, par le biais de plusieurs instructions. Nous en examinerons deux principales : la boucle **loop** et l'instruction de répétition **rep.**

a. Le saut inconditionnel.

Cette fonction est assurée par l'instruction **jmp** dont la syntaxe est :

```
JMP  adresse
JMP  FAR adresse
```

où adresse revêt des formes différentes selon que l'instruction est ou non qualifiée par le mot **far.**

Dans le premier cas, adresse peut revêtir quatre formes et sera toujours interprété par le processeur comme l'indication de l'adresse relative où il faut se rendre, l'adresse de segment correspondante étant prise dans le registre *CS* :

celle d'une donnée immédiate : l'adresse elle-même ;
celle d'un registre (segments exclus) : l'adresse est la valeur du registre ;
celle d'une adresse indirecte : l'adresse est pointée par l'adresse indirecte.

Dans ce dernier cas, le processeur va lire la valeur de l'adresse dans le mot se trouvant à l'adresse (relative) indiquée, l'adresse de segment de ce mot étant prise dans le registre de segment attaché à l'expression utilisée.

Quatrième forme :
celle d'une adresse absolue.

Dans le cas de l'instruction qualifiée par **far**, l'adresse est *toujours* indirecte, car elle indique l'adresse (relative) d'une zone de deux mots dont les valeurs fournissent une adresse absolue : le mot faible donne le déplacement de l'adresse, le mot fort donne son numéro de paragraphe.

Exemple sous *DEBUG* :

```
0200  AX   CS   IP   code
010A 0000 1388 0100  FF260002    JMP  [0200]
010A 0000 1388 0104  EB4A        JMP  150
010A 0000 1388 0150  E9AD13      JMP  1500
010A 0000 1388 1500  B81001      MOV  AX,0110
010A 0110 1388 1503  FFE0        JMP  AX
010A 0110 1388 0110  EA04027813  JMP  1378:0204
010A 0110 1378 0204  EB4A        JMP  250
010A 0110 1378 0250  E9AD13      JMP  1600
010A 0110 1378 1600  B81001      MOV  AX,0110
010A 0110 1378 1603  FFE0        JMP  AX
010A 0110 1378 0110
```

Les instructions ont été écrites jusqu'à **1388:0110** inclus. Lorsque l'instruction **jmp 1378:020A** est exécutée, *IP* pointe donc sur l'octet **1378:020A** qui définit la même adresse physique que **1388:010A**. D'où l'exécution à nouveau de **jmp 150**, pourrait-on penser. Mais, comme le montre ce tracé, en **1378:020A**, *DEBUG* lit l'instruction **jmp 250**, puis, se rendant à l'octet d'adresse **1388:0150 = 1378:0250**, le *constructeur* lit **jmp 1600**. Or, les instructions n'ont pas été modifiées par le programme si on en juge par le code hexadécimal affiché. Que se passe-t-il réellement?

Le fonctionnement des instructions jmp et jmp far.

D'après ce que nous avons dit de la syntaxe des instructions, qui est absolument correct, on pourrait penser que l'instruction **jmp** *adresse* réalise l'action **mov ip,***adresse*. C'est effectivement ce qui se passe lorsque l'adresse indiquée dans l'instruction **jmp** n'est pas une donnée immédiate.

> A noter que l'effet de l'instruction **jmp** *adresse absolue* peut être décrit par l'action **lcs ip,***double mot* qui *n'est pas* une instruction.

Dans le cas d'une adresse donnée sous forme immédiate, *DEBUG* assemble l'instruction machine en fournissant non pas l'adresse relative, c'est-à-dire l'abscisse locale du point visé dans le segment défini par *CS*, mais la *distance algébrique* de ce point au premier octet qui suit l'instruction. Ce qui explique le comportement constaté dans la session précédente de *DEBUG* et, notamment, que le second **jmp ax** exécute amène *CS:IP* en **13780110** et non en **1388:0110 = 1378:0210**. On constate en outre une différence de longueur de code entre les deux instructions **jmp 150** et **jmp 1500** : elle provient de ce que dans le premier cas, la différence algébrique peut être codée sur un octet, et dans le second cas, elle ne peut l'être que sur un mot. En effet : **0150 – 0106 = 4A** et **1500 – 0153 = 13AD**. De même, l'instruction **jmp 82** placée en **0100** demanderait deux octets puisque **0082 – 0102 = FF80 = -80**, tandis que **jmp 81** placée à la même adresse **0100** en demanderait trois car **0081 – 0102 = 0FF7F** qui ne tient pas sur un octet. On effectue donc **0081 – 0103 = 0FF7E**. Lorsque cette différence algébrique peut s'exprimer sur un octet *signé*, on dira que l'adresse de l'instruction est ***proche***. L'absence de ce qualificatif sous-entend une différence nécessitant un mot.

L'instruction **jmp far** *adresse indirecte* fonctionne comme le ferait la pseudo-instruction **lcs ip,***adresse indirecte* sur le modèle des instructions **lds/les**.

Illustration sous *DEBUG* :

```
0200 0202  CS   IP
0200 0000 1388 0100  MOV  [0202],CS
0200 1388 1388 0104  SUB  WO [0202],0010
0200 1378 1388 0109  JMP  FAR [0200]
0200 1378 1378 0200  MOV  [0202],CS
0200 1378 1378 0204  SUB  WO [0202],0010
0200 1368 1378 0209  JMP  FAR [0200]
0200 1368 1368 0200
```

L'appel : notion de sous-programme.

Si nous revenons à la simulation du schéma de récursion par une machine de Minsky, nous avons vu des instructions de la forme suivante :

```
; N :
          ST  P              ;
; N+1 :
          ...
; P :
          instructions
          ST  N+1            ;
```

Partant de l'instruction *N*, l'exécution est envoyée en *P* à une suite d'instructions. Les ayant effectuées, elle *revient* en *N*+1, c'est-à-dire à l'instruction qui suit *N*. Les instructions placées entre l'étiquette *P* et le *retour* **st N+1** constituent, par définition un ***sous-programme***. Nous dirons souvent, de façon synonyme, un ***programme***.

Le processeur implante un couple d'instruction qui permet de simuler ces actions : les instructions **call** et **ret**.

L'instruction **call** a une syntaxe analogue à celle de **jmp** (*cf.* dictionnaire) à cette exception près que l'instruction **call** ne dispose pas d'adresse proche : si l'adresse est une donnée immédiate, la distance algébrique de l'appel au point visé est toujours exprimée sur un mot. L'instruction **call** *adresse* exécute les deux actions suivantes :

```
PUSH IP
MOV  CS,adresse
```

où la valeur de IP placée sur la pile est l'adresse du premier octet qui suit l'instruction **call** et qu'on appelle ***adresse retour***.

Comme sa mnémonique le suggère, l'instruction **ret** permet d'effectuer le *retour* à cette adresse, donc à l'instruction qui suit le **call** correspondant. Comment cette instruction, sans autre indication, permet-elle de retrouver l'appel auquel elle doit renvoyer? Tout simplement parce qu'elle est exactement synonyme de l'action **pop ip** que le langage symbolique ne permet pas d'écrire ainsi.

La similitude de structure avec l'instruction **jmp** laisse penser qu'il existe une instruction **call far**. Tel est le cas. L'adresse indiquée dans cette instruction l'est aussi sous la forme d'un pointeur. Comme dans le cas de **jmp far**, le mot faible pointé contient le déplacement du point visé et le mot fort son numéro de paragraphe. Cette fois, l'action exécutée par l'instruction est un peu plus complexe que dans le cas du **call** puisqu'elle équivaut à l'exécution de trois actions élémentaires :

```
PUSH CS          ; sauvegarde sur la pile
PUSH IP          ; de l'adresse retour (absolue)
LCS  IP,adresse
```

où *adresse* est nécessairement indirecte. Le retour correspondant est assuré par l'instruction **retf** dont l'effet est équivalent aux deux actions ***simultanées*** :

```
POP  IP
POP  CS
```

Bien que ces dépilements sur la pile soient considérés comme simultanés, l'ordre indiqué, comme dans le cas de l'empilement, correspond à l'écriture d'une adresse absolue en mémoire : le mot faible pour le déplacement, le mot fort pour le segment.

Voici une session sous DEBUG pour nous convaincre de ce fonctionnement :

```
0200 0202 FFEA FFEC  SP   AX   BX   CS   IP
0250 0000 0000 0000 FFEE 0000 0000 1388 0100  MOV  [0202],CS
0250 1388 1388 0FD6 FFEE 0000 0000 1388 0104  SUB  WO [0202],0010
0250 1378 1388 0FD6 FFEE 0000 1378 1388 0109  CALL FAR [0200]
0250 1378 010D 1388 FFEA 0000 0000 1378 0250  MOV  AX,1234
0250 1378 010D 1388 FFEA 1234 0000 1378 0253  RETF
0250 1378 1388 0FD6 FFEE 1234 0000 1388 010D  MOV  BX,AX
0250 1378 1388 0FD6 FFEE 1234 1234 1388 010F
```

Toutes les instructions ont été écrites avant l'exécution. Les instructions **1378:0250** à **0254** l'ont été depuis **1388:0150** = **1378:0250**. L'instruction **1388:010D** suit l'instruction **call far [200]** qui occupe 4 octets.

Noter, aux instructions **0104**,**0109** puis **010D**,**010F**, que *DEBUG* utilise lui aussi la pile. La valeur **1388** est, visiblement, la sauvegarde du *CS* simulé.

Noter la nécessité de l'instruction **retf** : si l'on avait **ret**, *CS:IP* prendrait la valeur **1378:010D** et *SP* la valeur **FFEC**. Le lecteur est invité à tester l'exécution des autres formes possibles de cette instruction en prenant modèle sur ces instructions.

Compte tenu de ce que nous avons dit, **ret** et **retf** peuvent être utilisées pour simuler, respectivement, un **jmp** et un **jmp far.** En effet, **jmp ax** et **jmp far [bx]** sont respectivement équivalents aux instructions suivantes, ce que le lecteur peut vérifier sous *DEBUG* :

```
jmp  ax :        jmp  far [bx] :
   PUSH  AX         PUSH  [BX+02]
   RET              PUSH  [BX]
                    RETF
```

A noter qu'à partir du 80286 (pas avant), on peut simuler **jmp** *mot*, où *mot* est une donnée immédiate par **push** *mot* suivi de **ret** : cette forme de **push** est désormais admise.

Ce mode d'écriture d'un saut conditionnel est cependant peu recommandé, car il va à l'encontre d'une programmation claire.

b. Le branchement conditionnel.

Saut conditionné.

Nous avons vu, au chapitre 2 l'instruction **dsz R,*n*** d'une machine à registre. Rappelons qu'elle consiste d'abord à tester si le contenu de **R** est zéro. Si oui, on saute à l'instruction numérotée ***n***, sinon, on décrémente le registre **R**. Dans le processeur que nous étudions, on procède différemment. On dispose d'une famille d'instructions obéissant à la syntaxe suivante :

```
Scond  adresse
```

où `adresse` est une adresse proche. Si la réponse au test est oui, on exécute un saut à l'adresse indiquée, sinon, on passe à l'instruction suivante. Il y a autant de mnémoniques que de tests possibles.

Exemple : l'instruction **jcxz**. Dans ce cas, le test est : la valeur de *CX* est-elle nulle? Ceci permet de simuler l'instruction **dsz R,n** d'une machine de Minsky. Si le registre est représenté par un mot mémoire situé en `source` et si **n** a pour valeur une adresse proche `adresse`, il suffit d'écrire :

```
MOV  CX,source
JCXZ adresse
DEC  CX
MOV  source,CX
```

Illustration sous *DEBUG* :

```
 CX   IP
0000 0100  JCXZ 0150
0000 0150
0001 0100  JCXZ 0150
0001 0102
```

→ ***Dictionnaire*** : *cf.* instructions **j***cond*, en particulier, **jc**, **je**, **ja** et leurs négations.

Boucle.

L'instruction **jcxz** nous permet d'effectuer l'addition de deux registres, par exemple *AX* et *BX* avec le résultat dans *CX*, par un programme simulant celui de

l'exemple I du chapitre précédent :

```
; 100 :
          PUSH AX      ; on sauvegarde AX
          MOV  CX,BX   ; on copie BX dans CX
; 103 :                ; haut de la boucle
          JCXZ 0109    ; si CX = 0, fin de l'addition
          DEC  CX      ; on diminue CX de 1
          INC  AX      ; on augmente AX de 1
          JMP  0103    ; sinon, retour à 0103
; 109 :                ; l'addition est terminée
          MOV  CX,AX   ; résultat dans CX
          POP  AX      ; la valeur initiale de AX est restaurée
```

Si on dissocie la première exécution de **jcxz** qui teste en fait la nullité de *BX* des suivantes, on a l'association de trois instructions : **dec**, **jcxz** et **jmp**. Elles peuvent être remplacées par une seule : l'instruction **loop** conduisant au programme suivant :

```
; 100 :
          PUSH AX      ; on sauvegarde AX
          MOV  CX,BX   ; on copie BX dans CX
          JCXZ 108     ; si BX = 0, inutile d'additionner
                       ; sinon :
; 105 :                ; haut de la boucle
          INC  AX      ; on augmente AX de 1
          LOOP 105     ; retour au haut de la boucle
; 108 :                ; l'addition est terminée
          MOV  CX,AX   ; résultat dans CX
          POP  AX      ; la valeur initiale de AX est restaurée
```

Pour tester ces programmes sous *DEBUG* : les entrer depuis l'adresse **0100** par la commande *a* ; initialiser *AX* et *BX* par les commandes *r AX* et *r BX* ; initialiser *IP* à **100** par *r IP* ; afficher les registres par *r* puis exécuter pas à pas avec la commande *p* jusqu'à la fin. Choisir des valeurs petites pour *AX* et *BX* et tester le cas *BX* = 0.

L'instruction **loop** obéit à la même syntaxe que **jcxz**, le test étant également la nullité de *CX*. Elle ressemble, à première vue, à l'instruction **dsz** des machines à registres, mais ATTENTION : dans la machine à registres, on teste *avant* de décrémenter, tandis que l'instruction **loop** teste *après*. C'est pourquoi, dans l'exemple ci-dessus, nous avons coiffé la boucle par un test **jcxz** qui interdit l'entrée dans le corps de boucle si *CX* = 0 au départ. Sinon, que se passerait-il? L'instruction **loop** commencerait par décrémenter *CX* qui passerait de la valeur **0000** à **0FFFF**. De ce fait, la boucle serait parcourue 65536 fois avant de se terminer!

→ ***Dictionnaire*** : *cf.* instructions **j***cond*, **loop***cond*.

Répétition.

Un certain nombre d'instructions admettent la possibilité d'être répétées. On le signale en faisant précéder l'instruction visée par l'instruction **rep** (instruction sans opérande). On dit quelquefois que l'on préfixe l'instruction par **rep** encore appelée de ce fait ***instruction de préfixage***. Le processeur répète alors l'instruction ainsi préfixée le nombre de fois indiqué par le registre *CX*. ATTENTION : **rep** fonctionne comme **loop** en décrémentant ***CX avant*** de tester si ce registre est nul.

Exemple d'instructions utilisée avec **rep** : l'instruction **movs**. Cette instruction, sans opérande, consiste à copier un octet ou un mot d'une source sur un but, ces deux opérandes étant *implicites*. La source est *DS*:[*SI*] et le but, *ES*:[*DI*]. Mais ceci est

incomplet : il faut préciser le type des opérandes. C'est pourquoi l'instruction admet deux formes : **movsb** si l'on veut déplacer un octet, **movsw** si l'on veut déplacer un mot. Donc **movsb** copie by *DS*:[*SI*] sur by *ES*:[*DI*], tandis que **movsw** copie **wo** *DS*:[*SI*] sur **wo** *ES*:[*DI*]. Enfin, on doit indiquer le *sens* de progression de *SI* et *DI*. En l'absence d'indication, le processeur se règle sur l'un des indicateurs. En général, à l'initialisation, le sens est celui des adresses croissantes (sens dit *vers le haut*). Le programmeur a intérêt à fixer le sens qui lui convient. Il dispose pour cela de deux instructions : **cld** pour que *SI* et *DI* croissent, **std** pour que *SI* et *DI* décroissent. On a donc les équivalences indiquées par les instructions ci-après.

A remarquer que, dans ces instructions, on aurait **dec** au lieu de **inc** si l'on remplaçait **cld** par **std**. Noter l'instruction de préfixage **es:** pour indiquer que l'adresse de segment du pointeur est la valeur du registre *ES*.

```
0100 CLD      0100 PUSH AX              0100 CLD     0100 PUSH AX
0101 REP      0101 MOV  AL,[SI]         0101 REP     0101 MOV  AX,[SI]
0102 MOVSB    0103 ES:                  0102 MOVSW   0103 ES:
              0104 MOV  [DI],AL                      0104 MOV  [DI],AX
              0106 INC  DI                           0106 INC  DI
              0107 INC  SI                           0107 INC  DI
              0108 LOOP 0101                         0108 INC  SI
              010A POP  AX                           0109 INC  SI
                                                     010A LOOP 0101
                                                     010C POP  AX
```

Exemple sous *DEBUG* :

initialisation :

```
-d 150 1 10
1388:0150 00 01 02 03 04 05 06 07   ........
1388:0158 08 09 0A 0B 0C 0D 0E 0F   ........
-d 200 1 10
1388:0200 00 00 00 00 00 00 00 00   ........
1388:0208 00 00 00 00 00 00 00 00   ........
```

exécution :

```
 SI   DI   CX   IP
0150 0200 0010 0100  CLD
0150 0200 0010 0101  REP
0150 0200 0010 0102  MOVSB
0160 0210 0000 0103
```

résultat :

```
-d 150 1 10
1388:0150 00 01 02 03 04 05 06 07
1388:0158 08 09 0A 0B 0C 0D 0E 0F
-d 200 1 10
1388:0200 00 01 02 03 04 05 06 07   ........
1388:0208 08 09 0A 0B 0C 0D 0E 0F   ........
```

→ ***Dictionnaire*** : *cf.* instructions **lods, movs, stos, rep***cond*, **scas.**

3.2.3. Les opérations.

D'après ce que nous avons vu au chapitre 2, nous sommes théoriquement en mesure de tout programmer avec les instructions dont nous disposons.

Pour améliorer très sensiblement le confort du programmeur et l'efficacité à l'utilisation, le processeur fournit un certain nombre d'opérations sur les registres, interprétés comme nombres ou comme mots binaires, selon les besoins du programmeur. Ces dernières constituent en fait de véritables programmes inscrits dans certains des circuits électroniques du processeur. Le lecteur souhaitant savoir comment sont implantés de tels programmes se référera avec profit à [4].

Une instruction qui modélise une opération sur deux nombres aura deux opérandes ou bien un seul. Dans les deux cas, le lieu d'affectation du résultat ou de l'un des opérandes est implicite. On convient, de ce fait que le résultat sera toujours placé dans ce que nous avons appelé le but. Quant aux arguments, deux cas se présentent : si l'instruction a deux opérandes (voir **add** ci-après), ils occupent le but et la source et quand l'instruction aura été exécutée, la valeur initiale du but sera remplacée par celle du résultat ; si l'instruction n'a qu'un opérande (voir **mul** ci-après), l'opérande est considéré comme source et le but, implicite par conséquent, est le registre *AX*. Dans ce cas, *AX* reçoit un des arguments avant l'opération puis le résultat, une fois celle-ci effectuée.

a. Le registre des indicateurs.

Lorsqu'une opération se termine, le processeur fournit, dans le registre des indicateurs, un certain nombre d'informations complémentaires sur le résultat et le déroulement de l'opération.

Ce registre est un mot dont, en principe, chaque bit se voit attribuer une signification. En fait, tous les bits ne sont pas utilisés : c'est le cas, dans le 8086, des quatre bits de poids fort ainsi que les bits 1, 3 et 5.

Le tableau suivant donne la mnémonique des bits utilisés par le 8086 ainsi que celle des valeurs qu'ils peuvent prendre :

N° du bit	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
mnémonique	**	**	**	**	OF	DF	IF	TF	SF	ZF	**	AF	**	PF	**	CF
mnémonique pour 0					NV	UP	DI		PL	NZ		NA		PO		NC
mnémonique pour 1					OV	DN	EI		NG	ZR		AC		PE		CY

Ci-après, la fonction attachée à chacun de ces indicateurs.

OF	dépassement de capacité
DF	direction de l'incrémentation automatique (*cf.* **movs**)
IF	masquage des interruptions (cf. paragraphe 3.3.2.)
TF	traçage des instructions
SF	signe du résultat
ZF	résultat nul ou non
AF	retenue éventuelle du demi-résultat
PF	parité du nombre de bits à un dans le résultat
CF	retenue éventuelle du résultat

Pour agir sur le registre des indicateurs, le processeur dispose d'un certain nombre d'instructions agissant chacune sur l'un d'entre eux. Il fournit alors une instruction ***clf*** pour mettre le bit correspondant à zéro, d'une instruction ***stf*** pour le mettre à un. Ainsi, **clc** confère au bit *CF* la valeur *NC*, et **stc** lui confère la valeur *CY*. Peuvent être ainsi directement fixés par le programmeur les bits *CF*, *IF* et *DF*. Le bit *CF* possède le privilège d'avoir une instruction de bascule, **cmc**, fonctionnant de la façon suivante :

valeur de *CF* avant :	*NC*	*CY*
valeur de *CF* après :	*CY*	*NC*

Si l'on y tient, on peut 'bricoler' soi-même le registre des indicateurs grâce aux instructions **pushf** et **popf**, en utilisant les séquences suivantes :

```
PUSHF                    PUSH AX
POP  AX                  POPF
```

La séquence de gauche permet de recueillir dans *AX* la valeur du registre des indicateurs. Avec un **mov**, on donne à *AX* la valeur voulue pour le registre des

indicateurs. Avec la séquence de droite, on confère la valeur désirée à ce registre. Les opérations que nous verrons plus loin permettent d'insérer ce genre de manipulation dans un programme.

→ ***Dictionnaire*** : *cf.* instructions **lahf**, **sahf**.

Le registre des indicateurs est particulièrement utilisé par les instructions **j*cond***. Exemple sur l'instruction **jc** : **jc** *adresse*, où *adresse* est proche, renvoie à l'instruction suivante si *CF* = *NC* et à l'instruction commençant à *adresse* si *CF* = *CY*.

Dans ce qui suit, nous dirons 'si *NC*' ou 'si *CY*' pour, respectivement, 'si *CF* = *NC*', 'si *CF* = *CY*', et nous ferons de même pour les autres indicateurs.

b. Arithmétique

Le processeur offre de nombreuses instructions dans le but d'effectuer des opérations numériques. Nous ne les indiquerons pas toutes, pas même dans le dictionnaire. D'une part, nous nous bornons, dans cet ouvrage, à considérer les nombres naturels et d'autre part, nous prenons le parti de faire tous les calculs en hexadécimal pour n'exprimer les résultats en décimal que lorsque cela est nécessaire, utilisant dans ce but, des procédures que nous aurons construites. Les programmes n'en sont pas plus complexes et nous pensons qu'on y gagne à la fois en encombrement mémoire et en temps de calcul.

Addition et soustraction.

Nous retrouvons les cas particuliers que constituent les instructions **inc** et **dec** déjà rencontrées. Elles répondent aux syntaxes respectives :

```
INC  but
DEC  but
```

où la valeur de *but* est augmentée de 1 par **inc** et diminuée de 1 par **dec**. Dans les deux cas, l'opération est effectuée modulo $16^4 = 65536$ si *but* est de type mot, modulo $16^2 = 256$ s'il est de type octet. En outre, ces deux instructions agissent sur le registre des indicateurs, principalement sur *ZF*. Le dictionnaire donne la liste complète des indicateurs modifiés par l'instruction.

Nous ne donnons pas d'exemple sous *DEBUG* dans ce cas : le lecteur sait à présent ce qu'il faut faire pour effectuer tous les tests qu'il jugera utiles. Il pourra ainsi vérifier que si *AX* contient **0FFFF** avant l'exécution de **inc ax**, il contient **0000** après, que *ZF* prend alors la valeur *ZR*, et que *CF* gardera la valeur qu'il avait avant l'exécution de l'instruction.

Pour le cas général de l'addition et de la soustraction, nous considérerons, dans chaque cas, deux instructions à deux opérandes. Respectivement : **add/adc** et **sub/sbb**.

La syntaxe de ces instructions est exactement la même que celle de **mov**. Le fonctionnement est le suivant :

pour **add**, *but* := *but* + *source*
pour **sub**, *but* := *but* - *source*

où l'on prend, à droite du symbole := les valeurs avant exécution, et à gauche, la valeur après. En outre, le résultat a le même type que les deux opérandes. Il est effectué modulo $16^4 = 65536$ dans le cas d'une opération mot, modulo $16^2 = 256$ dans le cas d'une opération octet. De ce fait, que l'on considère les deux arguments comme entiers naturels ou tous deux comme entiers relatifs, le résultat est correct.

L'addition de deux nombres peut provoquer une retenue, puisque, en base b, si $0 \le a,c < b$, $a+c \le 2b-2 = b + b-2$. L'opération étant effectuée modulo une puissance de deux, cette retenue, si elle existe, n'apparaîtra pas dans le registre ou la place

mémoire contenant le résultat. Elle apparaîtra, par contre, dans le registre des indicateurs, sur le bit prévu à cet effet : *CF*.

C'est ici que prend place la différence entre **add** et **adc**. L'instruction **add** a pour arguments les opérandes seuls et fournit le résultat avec la retenue indiquée dans *CF* s'il y a lieu. L'instruction **adc** a pour arguments les opérandes ***et*** l'indicateur *CF*. Si *NC*, **adc** se comporte comme **add**. Si *CY*, **adc** ajoute 1 au résultat de **add** et la retenue indiquée par *CF* est celle de la somme globale puisque :

si $0 \le a,c < b$, $a+b+1 \le 2b-1 = b + (b-1)$.

Les instructions **sbb** et **sub** entretiennent une relation analogue. La soustraction effectuée par **sbb** retranche encore 1 si *CY*, la retenue engendrée dans *CF* correspondant à ce dernier résultat.

Multiplication et division.

Pour effectuer ces opérations sur les entiers, le processeur dispose de quatre instructions : **mul** et **imul** pour la multiplication, **div** et **idiv** pour la division.

Instructions *mul* et *imul* :

Elles sont à un opérande, source, considéré comme le multiplicateur. Le multiplicande, qui doit être du même type que la source, est implicitement attendu dans le demi-registre *AL* pour le type octet, dans le registre *AX* pour le type mot. Comme $(b-1)^2 = b^2-2b+1 = (b-2)b+1$, le produit est de type mot pour l'opération octet et de la taille d'un double mot pour l'opération mot. Implicitement, le résultat se trouve, respectivement, dans *AX* et *DX*:*AX*. Pour un résultat double mot, la partie haute est dans *DX* et la partie basse dans *AX*. Cependant, le produit, qui peut être d'une taille double des opérandes peut aussi être de la même taille qu'eux. Le processeur signale ce cas par *NV* et *NC* et le cas contraire, celui de la taille double, par *OV* et *CY*. Les autres indicateurs ont une valeur non déterminée et donc sans signification.

La différence entre **mul** et **imul** réside dans l'interprétation des entiers : **mul** effectue la multiplication entre entiers naturels, **imul**, la multiplication entre entiers relatifs. L'action sur les indicateurs porte sur *OF* et *CF* simultanément, comme pour **mul**, mais la signification est différente : *NV* et *NC* si et seulement si la partie haute du résultat est une extension du signe de la partie basse.

Illustration sur un exemple :

```
 AX   CX  CF OF                    AX   CX  CF OF
00FD 0002        MUL  CL          00FD 0002        IMUL CL
01FA 0002 CY OV                   FFFA 0002 NC NV
```

En notation décimale, on a effectué à gauche 2*253 et à droite 2*(-3). Observer que la partie basse du résultat est la même dans les deux cas puisque $-a = 2^u-a$, où $u = 8$ dans l'opération octet, $u = 16$ dans l'opération mot.

Instructions *div* et *idiv*.

Ces instructions fonctionnent sur le modèle précédent. Elles ont un opérande unique considéré comme source dont le type détermine celui de l'opération et dont la valeur ***qui doit être non nulle*** est censée être le diviseur. Pour une division de type octet, le dividende est attendu dans *AX* et le résultat, contenu dans *AX* se décompose ainsi : *AH* contient le reste et *AL*, le quotient entier. Pour une division de type mot, le dividende est attendu dans *DX*:*AX*, partie haute dans *DX*. Le reste est alors dans *DX* et le quotient entier dans *AX*.

L'instruction **div** effectue la division euclidienne de deux entiers naturels, tandis que **idiv** réalise la même opération sur des entiers relatifs. Cependant, le reste de **idiv** a

le même signe que le dividende, de sorte que l'équation de la division euclidienne est satisfaite avec le module des arguments et des résultats.

ATTENTION : si ces deux instructions n'acceptent pas la division par zéro, elles n'acceptent pas non plus que le quotient soit d'un type plus grand que le type de l'opération. Ainsi, si l'on place **8000** dans *AX* et **2** dans *CX*, le processeur déclenchera l'interruption **00** pendant l'exécution de **div cl** ce qui peut conduire à un plantage de la machine. La même restriction est valable pour **idiv**.

Les instructions n'ont pas, par contre, d'action significative sur les indicateurs.

Illustration sur un exemple avec **idiv** :

```
 AX   CX                          AX   CX
FFED 00FD   IDIV CL             FFED 0003   IDIV CL
FF06 00FD                       FFFA 0003
```

La division de gauche est : $-19 = (-3)*6 - 1$, et celle de droite : $-19 = 3*(-6) - 1$, ce qui correspond à $19 = 3*6 + 1$. Comme le dividende est négatif, le reste est -1 dans les deux cas, seul change le quotient avec le signe du diviseur.

→ *Dictionnaire* : *cf.* instructions **adc, add, div, idiv, mul, imul, neg, sbb, sub**.

c. Logique.

Le processeur fournit des instructions d'opérations logiques sur les octets et les mots considérés comme mots binaires, c'est-à-dire comme juxtaposition de zéros et de uns. Il fournit également des instructions de test.

Et, ou et xou.

Trois instructions à deux opérandes de syntaxe identique à celle de **mov** implantent les connecteurs logiques **et**, **ou** et **xou**, le ou exclusif (soit, soit). Ce sont les instructions **and**, **or** et **xor**. Elles placent dans le but le résultat de l'opération appliquée à chaque couple de bits source/but de même rang selon les relations :

```
0 et 0 = 0      0 ou 0 = 0      0 xou 0 = 0
0 et 1 = 0      0 ou 1 = 1      0 xou 1 = 1
1 et 0 = 0      1 ou 0 = 1      1 xou 0 = 1
1 et 1 = 1      1 ou 1 = 1      1 xou 1 = 0
```

ce qui peut encore s'exprimer par les formules :

a **et** b = min(a,b), a **ou** b = max(a,b) et a **xou** b = (a+b) mod 2

où a,b ∈ {0,1}.

Illustration :

```
 AX                    AX                    AX
F3C8  AND  AL,AH      F3C8  OR   AL,AH      F3C8  XOR  AL,AH
F3C0                  F3FB                  F33B
```

Par ailleurs, ces trois instructions agissent sur les mêmes indicateurs : *OF* et *CF* prennent toujours la valeur zéro, celle de *AF* n'est pas significative et les indicateurs significatifs sont ici *SF*, *ZF* et *PF*.

→ *Dictionnaire* : *cf.* instructions **and, not, or, xor.**

Tests.

A côté des tests définis par la valeur de un ou plusieurs indicateurs, le processeur fournit deux instructions de tests fondés sur la comparaison des opérandes. Ces deux instructions, **cmp** et **test**, ont la même syntaxe que **mov**, et ont un fonctionnement parallèle. L'instruction **cmp** *but,source* ne change pas les valeurs de *but* et *source*, mais agit sur les indicateurs exactement comme si l'on avait effectué **sub** *but,source*. L'instruction **test** *but,source* s'exécute de même, mais en se réglant sur l'instruction **and** *but,source*.

En association avec une instruction **j***cond*, l'instruction **cmp** permet de définir une grande variété de branchements conditionnels. Illustrons-le sur deux exemples :

```
 AX  CF  IP                          AX  OF SF  IP
8706     0100  CMP  AL,AH           8706       0100  CMP  AL,AH
8706 CY 0102  JB   0150             8706 NV PL 0102  JL   0150
8706 CY 0150  CMP  AH,AL            8706 NV PL 0104  CMP  AH,AL
8706 NC 0152  JB   01A0             8706 NV NG 0106  JL   01A0
8706 NC 0154                        8706 NV NG 01A0
```

L'explication est la suivante : **jb** *adresse* renvoie l'exécution à *adresse* si et seulement si *CY*, donc dans notre cas, si *AL* < *AH*, les valeurs étant considérées comme entiers naturels. Par contre, **jl** *adresse* renvoie l'exécution à *adresse* si et seulement si *SF* ≠ *OF*. Dans une soustraction, *OF* reste à la valeur *NV*, mais *SF* indique le signe du résultat considéré comme nombre relatif et en fonction du type de l'instruction. L'examen des quatre cas possibles montre que *SF* = 1 si et seulement si *AL* < *AH*, les valeurs étant interprétées comme nombres relatifs.

Parmi les instructions du groupe **j***cond*, citons **jbe** et **jle** pour la relation ≤ (respectivement entre nombres naturels, relatifs), **ja** et **jg** pour >, **jae** et **jge** pour ≥. On remarque que **ja** n'est pas autre chose que la négation de **jbe**, c'est pourquoi le langage accepte la mnémonique **jna** comme synonyme de **jbe**, c'est-à-dire que le code de l'instruction est le même dans les deux cas. Observer que **jb** est synonyme de **jc**.

→ ***Dictionnaire*** : *cf.* instructions **cmps**, **scas**.

d. Opérations sur des zones mémoire.

Certaines opérations concernent tout d'abord des zones octet ou mot, et donc des registres de même taille. Les autres concernent des zones de taille quelconque dans la limite d'un segment (64 Ko).

Octet ou mot.

Tout d'abord, l'échange entre registres ou de registres avec des zones de un ou deux octets : elle est réalisée par l'instruction **xchg** qui a deux opérandes. Illustration :

```
 AX   BX
1234 5678  XCHG AL,BH
1256 3478  XCHG AX,BX
3478 1256
```

D'autres opérations concernent la translation des bits d'un octet ou d'un mot. Le prototype des instructions correspondantes est l'instruction **shr** qui, possédant deux opérandes, obéit à une syntaxe différente de celle de **mov**, exception à la règle d'identité du type des opérandes :

```
SHR  but,amplitude
```

où `amplitude` indique le nombre de bits de la translation, ce nombre étant représenté de deux façons seulement. On indique 1 ou bien le demi-registre *CL*, dans lequel le

processeur lira le nombre de bits. Le sens de la translation est indiqué par la lettre **r** de la mnémonique : à droite. Cela revient donc à une division par 2^a, où a est la valeur de *amplitude*. Illustration :

```
0150 CF                          0150  CX  CF
2414    SHR  WO [0150],1         2414 0003    SHR  WO [0150],CL
120A NC                          0482 0003 CY
```

En outre, chaque bit sortant du but est placé sur *CF* qui retiendra donc le dernier.

Il existe une instruction **shl** effectuant la même opération vers la gauche. D'autres instructions effectuent une permutation circulaire sur les bits d'un octet ou d'un mot, l'amplitude de la rotation étant indiquée de la même manière : **ror** et **rol**, respectivement à droite et à gauche. Les instructions **rcr** et **rcl** effectuent une permutation circulaire dans laquelle entre l'indicateur *CF*.

Zones de un octet à un segment plein.

Il s'agit essentiellement d'instructions de comparaison susceptibles d'être préfixées par une instruction de répétition **rep** ou **rep***cond* : les instructions **scas** et **cmps**. Elles fonctionnent avec des opérandes implicites à la façon de **lods** ou de **movs**.

L'instruction **scas** effectue ce que fait **cmp** avec pour source *AL* (resp. *AX*) et pour but *ES*:[*DI*], modifiant *DI* selon l'indication de *DF* et le type de l'instruction : type octet désigné par la forme **scasb**, type mot désigné par la forme **scasw**. Ainsi a-t-on l'équivalence suivante où intervient **es:**, instruction de préfixage :

```
STD              ES:
SCASB            CMP  AL, [DI]
                 DEC  DI
```

L'instruction **scas** est très utilisée pour la recherche d'un octet ou d'un mot dans une zone. Cela peut s'appliquer aussi bien au traitement de chaînes de caractères que la recherche d'une entrée dans un tableau. Nous en verrons de multiples exemples dans la seconde partie de l'ouvrage.

L'instructions **cmps** a une syntaxe et un fonctionnement calqués sur celui de **movs**, l'opération effectuée étant **cmp** entre le but implicite et la source implicite, dans cet ordre. Cette instruction est également fort utile, notamment pour rechercher une chaîne de caractères dans un texte. Nous en verrons des applications au chapitre 5.

3.3. Le système d'exploitation.

La donnée du processeur et de ses instructions suffit, en théorie, à mettre en oeuvre cet outil. En fait, tout ce que nous avons vu jusqu'ici se déroule entre la *RAM* et le processeur ou à l'intérieur même du processeur. Ainsi, dans ces conditions, si un programme était lancé, il se passerait bien des événements dont nous ne saurions rien : la *communication* avec l'extérieur ne figure pas parmi les instructions que nous avons étudiées. Fort heureusement, cette partie essentielle du dialogue homme-machine a été prévue. C'est ce que nous examinons à présent.

3.3.1. Les entrées/sorties.

Le processeur ne dialogue pas directement avec l'extérieur. Il le fait par l'intermédiaire des périphériques. Deux instructions expriment le mécanisme de ces communications. L'instruction **in** permet au processeur de prendre connaissance des

informations recueillies par les périphériques. L'instruction **out** lui permet de leur délivrer des informations. Dans les deux cas, il y a un source et un but. Comme l'information transmise peut être un octet ou un mot, le registre choisi pour être tantôt le but, tantôt la source dans ces échanges est *AX* si l'information est un mot, le demi-registre *AL* si c'est un octet. Le périphérique, quant à lui, n'est pas désigné directement, mais par un des points des circuits de la machine connectés à la fois au processeur et à un périphérique. Un tel point est appelé ***port***. Du point de vue du processeur, les ports sont numérotés et ce numéro, de la taille d'un mot au plus, est appelé ***adresse*** du port.

Ceci nous permet de donner la syntaxe de ces deux instructions :

```
IN   registre,port
OUT  port,registre
```

où `registre` est *AX* ou *AL* selon la nature du port et où `port` est une donnée immédiate d'un octet ou le registre *DX*. L'adresse du port est égale à la valeur de cette donnée immédiate ou à celle de *DX*, selon ce qu'indique l'instruction.

On observe que ces deux instructions obéissent au schéma général :

```
MNEMO  but,source
```

mais que le principe d'identité des types des opérandes ne peut être respecté compte tenu de la nature des actions effectuées et de leur représentation.

Ainsi, pour transmettre un octet au clavier par le port **060**, on écrira :

```
OUT  60,AL
```

Lire un octet dans ce même port s'écrira :

```
IN   AL,60
```

Par contre, la communication avec le port **03F5** (lecteurs de disquettes), par exemple, nécessite les instructions :

```
MOV  DX,03F5          MOV  DX,03F5
IN   AL,DX            OUT  DX,AL
```

Les communications du processeur avec un périphérique ne sont pas directes. Elles passent par l'intermédiaire d'un *processeur* spécialisé généralement appelé ***contrôleur***. Les ports désignent en fait les points de communication entre ces organes et le processeur. Ils appartiennent aux contrôleurs. Un contrôleur dispose lui aussi de registres et peut accomplir un nombre limité d'actions. Il lui correspond de ce fait un *langage*, extrêmement rudimentaire dans lequel toutes ces actions s'expriment. Par exemple, pour formater une piste sur une disquette, il faut s'adresser au bon lecteur, indiquer le numéro de la piste, puis préciser les paramètres du formatage dont nous parlerons plus loin. En un mot, il faut programmer l'action du contrôleur, et un tel programme est communiqué au contrôleur par le processeur à travers les ports.

C'est une des raisons pour lesquelles il y a plusieurs ports pour un même contrôleur. Un des ports sert à sélectionner une des fonctions que le contrôleur sait effectuer. On passe alors les ordres destinés à cette fonction par un autre port puis on passe les données par ce dernier port ou un troisième. Le contrôleur renvoie souvent un accusé de réception quand tout s'est bien passé et un message adéquat en cas d'incident. Si le périphérique renvoie des informations, le contrôleur le signale et, au message 'prêt' rendu par le processeur, transmet les octets correspondants un à un par l'un des ports.

A l'aide des instructions **in** et **out**, on programme le processeur à programmer les contrôleurs. De tels programmes sont établis une fois pour toutes et servent toujours dans les mêmes conditions. On les appelle des ***utilitaires***. Généralement, ils servent de fonctions à un programme plus vaste chargé de gérer le périphérique et appelé le ***gestionnaire*** du périphérique.

Comme on ne demande pas à l'utilisateur de programmer lui-même les gestionnaires des périphériques, les fabricants d'ordinateur les fournissent. Ils sont généralement situés en *ROM* pour les périphériques prévus d'origine avec la machine. Pour des périphériques supplémentaires ou remplaçant des périphériques d'origine, ces programmes sont fournis sous forme de logiciels qu'il convient d'intégrer ou de signaler de façon adéquate au système d'exploitation. Pour utiliser ces utilitaires dans un programme que le processeur charge directement ou indirectement au démarrage de la machine, un mécanisme d'appel différent du **call** a été créé. Un mécanisme identique, quelle que soit la raison qui l'a déclenché. C'est lui que nous étudions à présent.

3.3.2. Les Interruptions.

a. L'instruction int.

Ce mécanisme est commandé par l'instruction **int** dont la syntaxe est la suivante :

```
INT  n
```

où n est un entier naturel de la taille d'un octet. L'effet de **int** correspond à la réalisation simultanée des trois actions suivantes :

```
PUSHF
PUSH CS
PUSH IP
```

l'ordre correspondant, comme dans le cas d'un **call far**, à ce qui se trouve sur la pile, en outre l'indicateur *IF* est mis à *0*, interdisant, en principe, l'action des interruptions que le processeur a le droit de bloquer . Vérification sous *DEBUG* :

```
FFE8 FFEA FFEC IF  SP   CS   IP
0000 0000 0000 EI FFEE 1388 0100 INT 00
0102 1388 0202 DI FFE8 0270 56FF
```

Il est particulièrement recommandé de n'utiliser que la commande *t* pour exécuter la seule instruction **int** et non le programme qui lui correspond.

Sur cet exemple, on constate que l'instruction **int** s'est comportée comme un **call far**, à ceci près que l'adresse du point visé n'a pas été *directement* indiquée. Elle a cependant été indiquée, mais sous une forme *indirecte.* L'instruction **int**, d'après ce qu'on a vu, appelle un programme. L'adresse absolue du début de ce programme est situé dans une table qui se trouve dans la zone **0000:0000-0400** : la ***table des interruptions***. Cette table comporte donc 256 entrées correspondant aux 256 numéros possibles. Chaque entrée, de quatre octets, définit un double mot qui constitue l'adresse d'appel d'un programme d'interruption. L'instruction **int** indique au processeur le numéro d'entrée de la table des interruptions.

Comme l'interruption est un appel, le programme correspondant doit se terminer par un retour à l'instruction qui suit l'appel au programme d'interruption. Comme un **retf** ne dépile que deux mots alors que trois ont été empilés par l'instruction **int**, il faut une instruction particulière pour signaler ce retour. C'est ce que fait l'instruction **iret**, sans opérande. Elle correspond à l'exécution simultanée des trois actions :

```
POP  IP          ; sauvegarde sur la pile
POP  CS          ; de l'adresse absolue de retour
POPF
```

ce qui peut être constaté sous *DEBUG.*

b. Mécanisme des interruptions.

Rappelons la boucle infinie sur laquelle repose la machine de von Neumann :

```
                Démarrage
; 1 :
                AttenteCommande
                EnregistrementCommande
                ExécutionCommande
                st 1
```

Elle se déroule de la façon suivante : l'attente d'une commande est signalée par un message à l'écran. L'utilisateur entre alors sa commande au clavier et la confirme par un retour chariot. Le processeur prend connaissance de la commande, l'analyse puis, la trouvant correcte, il l'enregistre – l'écran n'a pas changé – puis procède à son exécution. Avant d'aller plus loin, revenons à l'écran. La brillance d'une portion d'écran sur les moniteurs de micro-ordinateurs à tube cathodique est réalisée par l'excitation de particules de phosphore, ce qui produit un éclat fugitif. L'apparence d'éclairement continu d'une partie de l'écran est obtenue par la répétition, 50 à 70 fois par seconde, de l'excitation des *points* de l'écran concernés. Cette opération est appelée ***rafraîchissement*** de l'écran. Donc, pendant qu'il exécute, par exemple, la copie d'un fichier sur disque, ce qui n'est pas nécessairement instantané, le processeur rafraîchit l'écran tous les 1/50e de seconde.

Ainsi, le processeur fait plusieurs choses simultanément alors qu'à chaque instant, d'après ce que nous avons vu au second chapitre, il ne peut exécuter qu'une seule instruction. La raison en est, naturellement, que son temps n'est pas le nôtre. Les actions qui nous paraissent simultanées se succèdent, en réalité, à l'échelle du temps d'exécution des instructions. C'est le principe du ***temps partagé***.

Mais ceci ne résout pas entièrement le problème : lorsque le processeur est occupé à exécuter un programme de calculs mathématiques qui peut être volumineux, il ne peut pas exécuter une instruction puis lire l'heure, ses performances en seraient grandement dégradées. En fait, c'est l'heure qui doit elle-même se rappeler au processeur. A cet effet, la machine possède une horloge couplée au processeur, mais indépendante de ce dernier. A intervalles de temps réguliers, l'horloge envoie des signaux que les circuits du processeur considèrent comme prioritaires. A leur réception, il interrompt la tâche *T* à laquelle il est occupé, il appelle le gestionnaire d'écran pour lui demander de rafraîchir l'image, exécute encore quelques autres tâches puis, si aucun incident ne s'est produit, reprend l'exécution de la tâche *T* là, où elle avait été interrompue. Le processeur peut également être interrompu à tout moment par le clavier : chaque fois qu'une touche a été enfoncée ... ou relâchée. Enfin, l'horloge envoie également environ 18,2 fois par secondes un signal qui déclenche l'interruption **08** et permet donc à l'utilisateur de programmer lui-même des opérations liées au temps. Les logiciels qui affichent l'heure en permanence à l'écran ne font pas autre chose que gérer cette interruption.

Le même mécanisme est utilisé dans d'autres circonstances.

Ainsi, lorsque le processeur découvre une division par zéro ou qu'un opérande mot est adressé en **0FFFF**, ce qu'on appelle ***exception*** pour distinguer cette situation d'une interruption qui intervient *indépendamment* de la tâche en cours. Les organes de contrôle de la machine peuvent interrompre le cours normal des choses si un incident matériel se produit. Enfin, l'utilisateur lui-même peut souhaiter se constituer des interruptions pour ses propres besoins. Nous verrons comment faire au chapitre 7.

De cet examen, il convient de distinguer deux types d'interruption : celles qui sont déclenchées par un signal de celles qui sont programmées ou programmables; et il

convient de retenir que toutes les interruptions consistent en fait à transférer l'exécution à un programme.

c. Fonctions d'une interruption.

Lorsqu'une interruption est associée à un gestionnaire de périphérique quelque peu complexe, l'appel du gestionnaire ne suffit pas : il faut lui préciser le type d'action attendu. Pour cela, il est convenu que l'interruption dispose de ***fonctions*** numérotées sur un octet, à laquelle le programme appelant fournit des informations par le biais des registres du processeur. Quand l'interruption a terminé sa tâche, elle répond par le même canal. En général, la fonction demandée est définie par le demi-registre *AH* et l'indicateur *CF* contient, au retour de l'interruption, la valeur *NC* si l'utilitaire appelé a fonctionné correctement, et il contient la valeur *CY* dans le cas contraire. Si *CY*, l'interruption place dans *AX* un nombre indiquant la nature de l'erreur et appelé ***code d'erreur.***

Exemple :
La fonction **05** de l'interruption **013** permet de formater une piste sur une disquette. Les arguments à fournir sont les suivants :

AH = 05
AL = nombre de secteurs à formater
DL = numéro du lecteur de disquette
DH = numéro de face de la disquette (0 ou 1, respectivement dessus, dessous)
CH = numéro de la piste à formater
ES:BX = adresse d'une table de formatage :
cette table comporte autant d'entrées de 4 octets que de secteurs à formater ; à chaque entrée on indique, dans l'ordre suivant : le numéro de la piste, le numéro de la face, le numéro attribué au secteur (qui peut être différent du numéro d'entrée de la table), un numéro de taille du secteur (0 pour 128 octets, 1 pour 256, 3 pour 512, 4 pour 1024); les numéros attribués aux secteurs doivent être distincts deux à deux, commencer à 1 et se terminer par le nombre de secteurs.

En sortie, la fonction indique en *CF* si tout s'est bien passé. C'est le cas si *NC* et alors *AH* = 0. Si *CY*, ce n'est pas le cas et *AH* fournit un code d'erreur. Parmi ceux-ci :

04 : secteur demandé non trouvé ; **03** : disquette protégée contre l'écriture ; **40** : piste non trouvée.

Cet exemple met en évidence la frontière entre ce qui appartient au processeur et ce qui relève ensuite de ce que nous allons aborder : le système d'exploitation.

d. La boucle de von Neumann.

Le mécanisme de l'interruption fait partie du processeur. Lui appartiennent de ce fait, le concept de table d'interruption, l'adresse absolue de départ de cette table et sa taille. Par contre, les adresses indiquées aux entrées de la table et, *a fortiori*, les programmes installés à ces adresses ne dépendent plus de lui. Ils relèvent d'un ensemble de programmes dont nous allons parler maintenant, celui qui met en oeuvre la boucle infinie de la machine de von Neumann, et qu'on appelle le système d'exploitation.

Comme nous l'avons vu, cette boucle comporte deux phases : une phase de démarrage et une phase d'entretien du cycle.

Le démarrage.

Il n'est pas question ici, de décrire en détail cette opération, encore appelée ***initialisation*** du système. Nous n'en donnerons qu'une description succincte.

Le démarrage comporte habituellement trois parties. La première est micro-cablée dans le processeur. Elle consiste en une initialisation de ses registres et à l'exécution d'une première instruction : **jmp FFFF:0000**. Une telle adresse n'est pas située dans la *RAM* qui, au départ, s'étend sur 640 Ko, donc de **0000:0000** à **A000:0000**, mais en *ROM*. Après ce saut commence la deuxième phase : l'exécution d'un premier programme d'initialisation conçu par le constructeur de l'ordinateur. Ce programme met en action la *RAM*, installe une première table des interruptions et des programmes référencés dans la table, procède à diverses vérifications du matériel, fait l'inventaire des périphériques et les initialise, puis cherche le programme de lancement du système d'exploitation. Il le cherche tout d'abord sur le premier lecteur de disquette. En cas d'échec, il recommence la recherche sur le lecteur suivant, et ainsi de suite en terminant par les lecteurs de disque dur. En cas d'échec, il demande à l'utilisateur d'insérer dans le premier lecteur une disquette disposant d'un tel système et, sous l'effet d'un retour chariot, recommence toute l'opération de recherche. Si un tel programme est trouvé, le programme de la *ROM* charge ce programme à l'adresse **0000:7C00** (en *RAM*) et lui passe entièrement la main par l'instruction **jmp 0000:7C00**.

> La recherche consiste à vérifier que le premier secteur de la première piste du support considéré possède une structure déterminée par les trente deux premiers et les deux derniers des 512 octets du secteur. Les deux premiers octets définissent un **jmp** à une adresse de ce même secteur.

La troisième phase du démarrage commence : le système d'exploitation initialise la boucle que va entretenir un programme particulier, le ***maître d'oeuvre***. Pour cela, il commence par redéfinir la table des interruptions en lui conférant les adresses de ***ses*** gestionnaires et utilitaires qu'il charge en *RAM*, depuis le support de masse où il a été trouvé. Il initialise également des 'zones de travail' pour le maître d'oeuvre, le charge en *RAM* et lui passe alors la main.

Le maître d'oeuvre.

C'est ce programme qui demande à l'utilisateur d'entrer une commande, qui l'analyse, la rejette éventuellement en en redemandant une autre ou l'accepte et se met en devoir de l'exécuter, redemandant une commande lorsque la précédente est achevée.

Comme il y a plusieurs conceptions possibles de la mise en oeuvre des moyens et ressources de la machine, un système d'exploitation comporte toujours une certaine organisation des ressources. Il fixe également un langage dans lequel s'expriment les commandes[1], et un cadre à l'exécution des programmes que peut concevoir l'utilisateur.

Dans ce qui suit, le système d'exploitation que nous considérerons est le DOS, et son maître d'oeuvre est le programme *COMMAND.COM*.

Les commandes de *DOS* sont suffisamment connues pour que nous ne les rappelions pas en détail. Elles ont la forme d'un nom suivi éventuellement de paramètres qui sont, le plus souvent, la désignation d'un *fichier*. Ces commandes, une fois traduites par *COMMAND.COM* sont tout simplement associées à une fonction adéquate d'une interruption particulière, l'interruption **021** dite, pour cette raison, interruption de *DOS*.

Parmi les commandes qu'exécute le maître d'oeuvre figurent celles qui demandent l'exécution d'un programme écrit par l'utilisateur. Cette demande est extrêmement simple : on entre au clavier le nom (sans extension) du fichier contenant le programme. *COMMAND.COM* se contente d'appeler une fonction de l'interruption **021**

---

[1] Dans la littérature spécialisée, on appelle souvent langage de commande le programme que nous appelons ici maître d'oeuvre.

qui se charge du travail : la fonction ***EXEC*** dont nous simulerons le fonctionnement au chapitre 7 et que nous expliquerons au chapitre 4.

Aussi nous intéresserons-nous, à présent, à cette interruption.

3.3.3. L'Interruption 021.

Nous donnons la description des principales interruptions et fonctions d'interruption utilisées dans les programmes de l'ouvrage. L'interruption **021** comprend un peu plus de cent fonctions que nous appellerons ***fonctions DOS*** suivies du numéro d'appel dans l'interruption. Pour comprendre leur fonctionnement, il convient d'indiquer comment *DOS* conçoit les supports de masse et les périphériques. Nous verrons le cas de la mémoire vive au chapitre 4.

Disques et disquettes.

Une disquette comporte deux ***faces*** : 0 (dessus) et 1 (dessous). Chaque face comporte un certain nombre de ***pistes*** divisées en ***secteurs*** comportant un certain nombre d'octets. Nous avons vu précédemment que le contrôleur du lecteur de disquette avait une relative largeur d'esprit quant à l'organisation des secteurs sur une piste. DOS simplifie très impérativement : même nombre de secteurs par piste, même nombre d'octets (512) pour tous les secteurs. Le nombre de pistes et de secteurs par piste va dépendre du ***format*** de la disquette. Les principaux formats *DOS* sont, pour des disquettes double face :

caractéristiques	pistes par face	secteurs par piste	capacité totale
5"1/4, DD	40	9	360 Ko
5"1/4, HD	80	9	720 Ko
5"1/4, HD	80	17	1,2 Mo
3"1/2, DD	80	9	720 Ko
3"1/2, HD	80	17	1,2 Mo

Les pistes sont numérotées de **00** à **027** ou de **00** à **05F**, les secteurs d'une même piste de **01** à **09** ou **01** à **011**.

Le disque dur connaît un échelon supplémentaire : le ***cylindre*** qui contient plusieurs faces. Les secteurs d'un disque dur ont également une taille de 512 octets.

Cette structure est la *structure physique* du support, telle qu'elle est définie par le formatage. *DOS* lui superpose une *structure logique* quelque peu différente.

Fichiers et répertoires.

Structure.

Tout d'abord, *DOS* redéfinit la notion de secteur : un secteur *DOS* de disquette vaut deux secteurs physiques de 512 octets, un secteur *DOS* de disque dur vaut quatre, parfois huit, secteurs physiques. Ces secteurs *DOS* sont numérotés depuis **00**. Puis *DOS* introduit la notion de **fichier** et de **répertoire**. Un fichier est un ensemble de secteurs *DOS* situés sur un même support de masse, on dit encore : sur un même ***volume***, auquel l'utilisateur a donné un *nom*. Pour traiter les fichiers, *DOS* dispose de deux outils : le répertoire et une table dite ***table d'allocation des fichiers***, en abrégé ***TAF***. Le répertoire contient la liste des noms de fichiers avec, pour chacun d'eux des informations. Nous en retiendrons deux : le nombre d'octets total occupé par le fichier et le numéro de son premier secteur *DOS*. La table d'allocation est une sorte de photographie du volume. Chaque entrée correspond à un secteur *DOS* et contient **0FFF** si ce secteur est le dernier

du fichier, et si ce n'est pas le cas, elle contient le numéro d'entrée correspondant au secteur *DOS* suivant du fichier.

Afin de permettre à l'utilisateur de mieux gérer ses fichiers, *DOS* a introduit une division des répertoires en ***sous-répertoires***. A partir d'un répertoire appelé ***répertoire racine***, chaque répertoire peut contenir des noms de fichiers ***et*** des noms de répertoires. Ces répertoires, des fichiers à statut de répertoire, contiennent à leur tour des noms de fichiers et de répertoires. Il en résulte un arbre des ***répertoires*** que les utilisateurs de *DOS* connaissent bien. La liste des répertoires allant de la racine à celui qui contient un fichier donné s'appelle ***chemin*** conduisant au fichier.

La gestion des fichiers.

On traite un fichier de la façon suivante : si le fichier existe, on l'ouvre. Puis on effectue les opérations que l'on souhaite : lecture et/ou écriture, puis on le ferme, ce qui signifie que *DOS* remet à jour le répertoire et la *TAF*. Si le fichier n'existe pas, on le crée. Les fonctions *DOS* que nous utilisons dans ce but sont les fonctions **03C** (création), **03D** (ouverture), **03E** (fermeture), **03F** (lecture) et **040** (écriture) et elles achèvent leur tâche en rendant compte de son déroulement dans *CF*.

On appelle les fonctions **03C** ou **03D** en indiquant dans *DS:DX* une zone d'octets donnant le nom du fichier, avec ou sans l'indication du chemin complet. Si une information manque (lecteur ou répertoire), la fonction prend la valeur en cours correspondante. On délimite la fin du nom par un octet **00**. On indique également l'***attribut*** du fichier qui définit ses conditions d'accès : lecture/écriture (**00**), lecture seule (**01**), dissimulation à la commande *dir* (**02**). Pour la fonction **03D**, on indique en outre, via *AL*, le mode d'ouverture : lecture seule (**00**), écriture seule (**01**), lecture/écriture (**02**). Si *NC*, la fonction renvoie dans *AX* un numéro, dit ***sésame du fichier*** qui sera la seule référence au fichier pour les fonctions **03F**, **040** et **03E** et qui sera placé dans le registre *BX*.

Pour la fonction de fermeture, seule la donnée du sésame dans *BX* est nécessaire. Cette fonction est importante. C'est elle qui fixe les modifications apportées au fichier en réactualisant le répertoire : la taille du fichier peut avoir changé, et la *TAF* : des secteurs auront pu être libérés ou, au contraire, il aura fallu en allouer de nouveaux.

Lecture et écriture.

Les fonctions **03F** et **040** opèrent à partir d'un pointeur en mémoire vive fourni, à l'appel de la fonction, par *DS:DX* et un pointeur, dans le fichier, qui reste interne à *DOS*. Le fichier est identifié par son sésame, fourni par *BX*. La zone désignée par *DS:DX* est appelée **tampon** et sa taille est censée être indiquée par *CX*.

La fonction **03F** recopie sur le tampon *CX* octets du fichier depuis la position courante du pointeur et indique dans *AX*, le nombre d'octets recopiés si aucun incident ne s'est produit. De la sorte, si la fin du fichier est rencontrée avant la copie de *CX* octets, on le constate par *NC* (copie effectuée) et *AX* < *CX*.

La fonction **040** recopie les *CX* octets du tampon sur le fichier, depuis la position courante du pointeur du fichier. Comme précédemment, si *NC*, *AX* indique le nombre d'octets recopiés, ici, dans le fichier. La fonction se charge, en particulier, des allocations qui pourraient résulter de cette copie.

Pour ces deux fonctions, le tampon a donc au plus 64 Ko. Si le fichier est plus grand, il faut simplement répéter l'appel à la fonction. Lorsqu'on lit un fichier, pour savoir quand on atteint sa fin, on utilise la fonction **042** qui agit sur le pointeur du fichier de la façon suivante : on indique dans *AL* l'origine du déplacement à effectuer (**00** : début du fichier, **01** : position courante, **02** : fin du fichier) et on indique dans *CX:DX* la valeur, en octets, de ce déplacement. La fonction répond en plaçant le pointeur du fichier au nouvel emplacement dont il donne la distance *au début du fichier* dans *DX:AX*. Pour mesurer la taille du fichier à lire dans *DX:AX*, on place donc le

pointeur du fichier en fin de fichier en faisant *AL* = **02** et *CX:DX* = **00**. Ne pas oublier, ensuite, de le replacer au début si on veut lire quelque chose...

Ecran.

La gestion de l'écran occupe des ouvrages fort volumineux. Nous n'indiquerons que quelques fonctions très standard de *DOS* pour le mode texte. Ainsi, pour afficher un caractère à l'écran, on utilise la fonction **02**, plaçant dans *DL* le numéro *ASCII* du caractère à afficher. Pour afficher une chaîne de caractères, on utilise la fonction **09**, plaçant dans *DS:DX* l'adresse du premier octet à afficher. La fin du texte doit être indiquer par le caractère **$** (*ASCII* **024**) qui, naturellement, ne doit pas figurer dans la chaîne à afficher.

> Si on veut afficher **$** dans une chaîne, l'interrompre sur ce caractère, afficher le symbole *ASCII* **024** par la fonction **02** et reprendre l'affichage de la chaîne à partir du caractère suivant.

La lecture d'une chaîne de caractères entrés à l'écran (par le biais du clavier) est le fait de la fonction **0A**. Elle recopie ce qui a été entré sur une zone définie par *DS:DX*. Mais *DOS* n'écrit pas sur le premier octet de cette zone. En **by** *DS:DX*, le programmeur indique combien de caractères la fonction peut lire *au maximum*. L'utilisateur ne devra pas en entrer plus : s'il tente de le faire, *DOS* le lui signalera. En **by** *DS:DX*+1, *DOS* indique le nombre de caractères lus *avant* le retour chariot. La chaîne entrée à l'écran commence à être copiée sur la zone prévue à partir de l'octet *DS:DX*+2. Le caractère **0D** qui la termine est également recopié.

Nous utiliserons conjointement une autre méthode, plus rapide, que nous exposerons au chapitre suivant et qui consiste à écrire directement sur la *RAM* vidéo, c'est-à-dire, sur la *RAM* de la carte graphique qui est considérée, par le processeur, comme un espace adressable situé au-delà de l'adresse de segment **09FFF**. Le point de départ de la *RAM* vidéo dépend de la carte et du mode graphique utilisé. Les modes textes utilisent l'adresse de segment **0B000** ou **0B800** et les modes graphiques utilisent généralement l'adresse **0A000**.

Clavier.

Bien que *DOS* fournisse des fonctions de gestion du clavier, nous n'utilisons, dans cet ouvrage, que les fonctions **00** et **01** de l'interruption **016**.

La fonction **01** permet de tester si un caractère vient d'être entré au clavier, ce qu'elle constate en regardant l'état d'une zone de la *RAM* de seize octets appelée le tampon du clavier. Elle répond par *ZF* et *AX*. Si *ZR*, le tampon est vide : aucun caractère ne vient d'être entré au clavier. Si *NZ*, le tampon n'est pas vide et *AX* donne la description du dernier caractère entré au clavier. Si $AL \neq 0$, *AL* donne le code *ASCII* du caractère (ordinaire) et *AH* le code clavier de la touche actionnée. Si $AL = 0$, *AH* permet d'identifier la touche et donc, par convention, les touches spéciales : fonctions, curseurs, etc...

La fonction **00** lit un caractère en retournant, en *AX*, la description du caractère comme indiqué ci-dessus. Si le tampon n'est pas vide, elle obtient le dernier caractère entré au clavier. Si le tampon est vide, elle attend qu'un caractère soit entré au clavier et elle le fournit, lorsque cela a été fait.

Ces deux fonctions permettent donc d'organiser un dialogue par le clavier. Nous le verrons au chapitre 4. A noter que l'interruption **016** ne s'occupant que du clavier, il n'y a pas d'écho écran lorsque des caractères sont entrés au clavier de cette façon. Si le programmeur souhaite cet écho écran, il doit le programmer. Les fonctions d'affichage de *DOS* ou l'écriture directe en *RAM* permettent de le faire aisément.

3.4. Dictionnaire des instructions.

Le dictionnaire ci-contre est divisé en trois parties : affectation, branchement, opérations. Chaque section est divisée en articles consacrés à une instruction. Le dictionnaire ne contient que les instructions utilisées dans les programmes de l'ouvrage.

Les articles se présentent selon les deux modèles suivants :.

ADD but,source — *syntaxe de l'instruction*

```
add rm,rm   add ro,ro
add rm,ai   add ro,ai
add rm,dm   add ro,do
add ai,rm   add ai,ro
add wo ai,dm
add by ai,do
```

développement de la syntaxe pour toutes les formes possibles des opérandes

```
but := but+source
  CY := retenue
I :  SF,ZR,AC,PF,CF
```

fonctionnement de l'instruction
indicateurs modifiés

SUB

```
sy. :  ADD
but := but - source
  CY := retenue
I :  SF,ZF,AF,PF,CF
```

pour la syntaxe de cette instruction, se rendre à la référence indiquée

Le développement de la syntaxe utilise des abréviations indiquées ci-après.

Le fonctionnement, annoncé par l'utilisation de caractères en italique, fait appel au symbole := pour désigner l'affectation, c'est-à-dire la copie, dans le terme de gauche, de la valeur du terme de droite ; aux crochets droits [] pour désigner un octet ou un mot pointé par l'expression contenue dans les crochets. Celle-ci peut ne pas être un mot admissible en assembleur : exemple [SP] pour décrire le fonctionnement de **push** et **pop**. Les abréviations **wo** et **by** sont quelquefois concentrées en **w** et **b**.

Les exceptions propres à l'instruction sont annoncées par l'italique gras.

Exception valable pour presque toutes les instructions : le cas d'un opérande mot situé à une adresse relative **0FFFF** (à cheval sur deux segments).

Abréviations utilisées dans le dictionnaire.

rm : registre mot parmi *AX,DX,CX,BX,BP,SI,DI,SP*
ro : demi-registre (octet) parmi *AL,DL,CL,BL,AH,DH,CH,BH*
rs : registre de segment parmi *CS,DS,SS,ES*
ai : adresse indirecte, soit une des expressions ci-dessous :

[rb]	[rb+do]	[rb+dm]
[ri]	[ri+do]	[ri+dm]
[rb+ri]	[rb+ri+do]	[rb+ri+dm]

où **rb** désigne *BX,BP* et **ri** désigne *SI,DI*
dm : donnée immédiate mot
do : donnée immédiate octet
ar : *AX* ou *AL*
I : registre des indicateurs
IL : demi-registre faible de *I* : *SF ZF ** AF ** PF ** CF*

Une condition de la forme *SF* = 1 sera indiquée par *NG* seulement ; cette notation sera appliquée à tous les indicateurs.

IP pointe le début de l'instruction, *IP1*, le début de la suivante

AFFECTATION

CLC
```
NC ; I : CF
```

CLD
```
UP ; I : DF
```

CLI
```
DI ; I : IF
```

CMC
```
bascule :
  NC si CY, CY si CN
I : CF
```

IN ar,port
```
in AX,do  in AL,do
in AX,DX  in AL,DX
  AL := [port];
  AX := [port]
I : aucun
```

LAHF
```
  AH := IL
I :  aucun
```

LDS reg,adr
```
lds  rm,ai
  DS := [adr+02] ;
  reg := w [adr]
I : aucun
  exception si
    ai = 0FFFD
```

LES reg,adr
```
sy. :  LDS
  ES := [adr+02] ;
  reg := w [adr]
I : aucun
  exception : cf. LDS
```

LODSB
```
  AL := b DS:[SI] ;
  si UP, SI := SI+1 ;
  si DN, SI := SI-1
I :  aucun ; cf. REP
```

LODSW
```
  AX := w DS:[SI]
  si UP, SI := SI+2 ;
  si DN, SI := SI-2
I :  aucun ; + REP
  exception si
    SI = 0FFFF
```

MOV but,source
```
mov rm,rm  mov ro,ro
mov rm,ai  mov ro,ai
mov rm,dm  mov ro,do
mov ai,rm  mov ai,ro
mov rm,rs  mov ai,rs
mov wo ai,dm
mov by ai,do
mov rs,rm  mov rs,ai
     ^CS interdit^
  but := source
I :  aucun
```

MOVSB
```
b ES:[DI] := b DS:[SI]
UP => SI,DI := SI,DI+1
DN => SI,DI := SI,DI-1
I : aucun ; cf. REP
```

MOVSW
```
w ES:[DI] := w DS:[SI]
UP => SI,DI := SI,DI+2
DN => SI,DI := SI,DI-2
I :  aucun ; cf. REP
exception : SI = 0FFFF
```

OUT port,ar
```
out do,AX  out do,AL
out DX,AX  out DX,AL
  [port] := AL;
  [port] := [AX]
I :  aucun
```

POP but
```
pop rm  pop rs  pop ai
  but := [SP] ;
  SP := SP+2
I :  aucun
pop CS interdit,
pop SS : attention
```

POPF
```
I := [SP];  SP := SP+2
```

PUSH source
```
push rm  push rs
push ai
  SP := SP-2
  [SP] := source ;
I :  aucun
blocage si SP = 1
```

PUSHF
```
I :  aucun
SP := SP-2;  [SP] := I
blocage si SP = 1
```

SAHF
```
  IL := AH
I :  SF,ZF,AF,PF,CF
```

STC
```
CY ; I : CF
```

STD
```
DN ; I : DF
```

STI
```
EI ; I : IF
```

STOSB
```
  b ES:[DI] := AL ;
  si UP, DI := DI+1,
  si DN, DI := DI-1
I :  aucun ; cf. REP
```

STOSW
```
  w ES:[DI] := AX ;
  si UP, DI := DI+2,
  si DN, DI := DI-2
I :  aucun ; + REP
exception : SI = 0FFFF
```

XCHG but,source
```
xchg rm,rm  xchg ro,ro
xchg rm,ai  xchg ro,ai
xchg ai,rm  xchg ai,ro
  w := but;
  but := source;
  source := w
I :  aucun
```

Préfixage :

CS: DS: SS: ES:
indique le registre de segment à utiliser avec l'adresse indirecte de l'instruction suivante

BRANCHEMENT

CALL adr
```
call dm  call rm
  SP :=SP-2 ;
  [SP] := IP1 ;
  IP := adr
call ai
  IP := [ai]
I : aucun
```

CALL adr1:adr2
```
call dm:dm
  SP := SP-4 ;
  [SP] := IP1 ;
  [SP+2] := CS ;
  CS := adr1 ;
  IP := adr2
I : aucun
```

CALL FAR adr
```
call far ai
  SP := SP-4 ;
  [SP] := IP1 ;
  [SP+2] := CS ;
  CS := [adr+02] ;
  IP := [adr]
I : aucun
```

INT num
```
int  do
  SP := SP-6 ;
  [SP+4] := I ;
  [SP+2] := CS ;
  [SP] := IP1 ;
  CS :=  [0000:D+2];
  IP := [0000:D]
où D = 4*num
I : EI
  blocage si
    SP = 1, 3 ou 5
```

IRET

IP := [SP] ;
CS := [SP+2] ;
I := [SP+4] ;
SP := SP+6 ;

I : tous modifiés
blocage si SP = 0FFFF en cours d'exécution

Jcond adr

xxx dm
où xxx : ja jae jb jbe jc jcxz je jg jge jl jle jna jnb jnc jne jno jnp jns jnz jo jp js jz

IP1-80 ≤ dm ≤ IP1+7F
cond => IP := adr,
n cond => IP := IP+2

Conditions associées :

a	: > (N)	g	: > (Z)
ae	: ≥ (N)	ge	: ≥ (Z)
b	: < (N)	l	: < (Z)
be	: ≤ (N)	le	: ≤ (Z)
e,z	: ZR	ne,nz	: NZ
cxz	: CX = 0		
b,c	: CY	nb,nc	: NC
o	: OV	no	: NV
p	: PE	np	: PO
s	: NG	ns	: PL

I : aucun

JMP adr

jmp dm
IP1-80 ≤ dm ≤ IP1+7F
IP := adr
I : aucun

JMP adr

jmp dm jmp rm
IP := dm, rm
jmp ai
IP := [ai]
I : aucun

JMP adr1:adr2

jmp dm:dm
CS := adr1 ;
IP := adr2
I : aucun

JMP FAR adr

jmp far ai
CS := [adr+02] ;
IP := [adr]
I : aucun

LOOP adr

loop dm
IP1-80 ≤ dm ≤ IP1+7F
CX := CX - 1 ;
CX ≠ 0 => IP := adr
CX = 0 => IP := IP+2
I : aucun

LOOPE adr

LOOPNE adr

loop dm loopne dm
IP1-80 ≤ dm ≤ IP1+7F
f CX := CX - 1 ;
***e** :*
ZR et CX ≠ 0 =>
IP := adr,
NZ ou CX = 0 =>
IP := IP+2,
***ne** :*
NZ et CX ≠ 0,
IP := adr,
ZR ou CX = 0,
IP := IP+2
I : aucun

REP

répète l'instruction suivante tant que CX „ 0 pour les seules instructions MOVS, LODS, STOS

REPE

REPNE

***e** (répeter tant que égal) :*
répète l'instruction suivante tant que CX ≠ 0 et NZ
***ne** (répéter tant que inégalité) :*
répète l'instruction suivante tant que CX ≠ 0 et ZR
pour les seules instructions CMPS, SCAS

RET

IP := [SP] ;
SP := SP+2
I : aucun
blocage si SP = 0FFFF pendant l'exécution

RET quanta

ret dm
IP := [SP] ;
SP := SP+dm+2 ;
I : aucun
arrêt : cf. RET

RETF

IP := [SP] ;
CS := [SP+2] ;
SP := SP + 4
I : aucun
arrêt : cf. RET

RETF quanta

retf dm
IP := [SP] ;
CS := [SP+2] ;
SP := SP+dm+4;
I : aucun
arrêt : cf. RET

OPERATIONS

ADD, ADC, DEC, INC, SUB et SBB : calculs modulo 2^u où u = 16 si type mot et u = 8 si type octet.

ADC but,source

***sy.** : ADD*
but := but+source+CF
CY si et seulement si retenue
I : SF,ZR,AC,PF,CF

ADD but,source

add rm,rm add ro,ro
add rm,ai add ro,ai
add rm,dm add ro,do
add ai,rm add ai,ro
add wo ai,dm
add by ai,do
but := but+source
CY := retenue
I : SF,ZR,AC,PF,CF

AND but,source

***sy.** : ADD*
*but := but **et** source, bit à bit*
I : NV,SF,ZF,PF,NC

CMP but,source

***sy.** : ADD*
but et source inchangés,
I : cf. SUB,
SF,ZF,AC,PF,CF

CMPSB

comparaison :
b es:[di] et b ds:[si]
UP => DI,SI := DI,SI+1
DN => DI,SI := DI,SI-1
I : cf. CMP
cf. REPE,REPNE

CMPSW

comparaison :
w es:[di] et w ds:[si]
UP => DI,SI := DI,SI+2
DN => DI,SI := DI,SI-2
I : cf. CMP
cf. REPE,REPNE

DEC but

dec rm dec ro
dec wo ai,dm
dec by ai,do
but := but - 1
I : SF,ZF,AF,PF

DIV source

div rm div ro
div wo ai div by ai
*D := d*q + r où :*
type mot :
D := DX:AX ;
d := source ;
DX :- r ; AX := q
type octet :
D := AX ;
d := source ;
AH := r ; AL := q
I non significatif
exception : source = 0 ou q excède le type de l'instruction

IDIV source

sy. : DIV
même action que DIV sur |D| := |d||q|+|r| sauf pour le signe : signe r = signe D quel que soit le type*
I non significatif
exception : cf. DIV

IMUL source

sy. : DIV
type mot :
*DX:AX := AX*source ;*
NV,NC si DX étend le signe de AX,
OV,CY sinon
type octet :
*AX := AL*source ;*
NV,NC si AH étend le signe de AL,
OV,CY sinon
I : OF,CF

INC but

sy. : DEC
but := but+1
I : SF,ZF,AF,PF

MUL source

sy. : DIV
type mot :
*DX:AX := AX*source ;*
NV,NC si DX = 0 ;
OV,CY sinon
type octet :
*AX := AL*source ;*
NV,NC si AH = 0 ;
OV,CY sinon
I : OF,CF

NEG but

sy. : DEC
but := - but
I : SF,ZF,AF,PF,CF

NOP

AX := AX
I : aucun

NOT but

sy. : DEC
but := non but,
bit à bit
I : aucun

OR but,source

sy. : ADD
but := but ou source,
bit à bit
I : NV,SF,ZF,PF,NC

RCL but,quanta

rcl rm,1 rcl ro,1
rcl rm,CL rcl ro,CL
rcl wo ai,1
rcl by ai,1
rcl wo ai,CL
rcl by ai,CL
quanta = 1 : décalage des bits de but d'un cran vers la gauche ; b0 := CF ; CF := bm ;
quanta = CL : opération répétée CL fois
I : OF,CF

RCR but,quanta

sy. : RCL
comme pour RCL, mais décalage à droite

ROL but,quanta

sy. : RCL
comme pour RCL, mais CF n'entre pas dans le décalage ; CF reçoit une copie du bit qui a changé d'extrémité
I : OF,CF

ROR but,quanta

sy. : ROL,
cf. ROL, mais décalage à droite

SBB

sy. : ADD
but := but-source-CF
CF := retenue
I : SF,ZF,AF,PF,CF

SCASB

comparaison :
AL avec b es:[di] ;
UP => DI := DI+1,
DN => DI := DI-1
I : cf. CMP
cf. REPE,REPNE

SCASW

comparaison :
AX avec w es:[di] ;
UP => DI := DI+2,
DN => DI := DI-2
I : cf. CMP
cf. REPE,REPNE

SHL but,quanta

sy. : RCL
quanta = 1 :
but :- but DIV 2,
CF := but MOD 2
quanta = CL : opération répétée CL fois
I : OF,ZF,SF,PF,CF

SHR but,quanta

sy. : RCL
quanta = 1 :
*but := but*2,*
CF := retenue
quanta = CL : opération répétée CL fois
I : OF,ZF,SF,PF,CF

SUB

sy. : ADD
but := but - source
CY := retenue
I : SF,ZF,AF,PF,CF

TEST

sy. : ADD
but et source inchangés
I : cf. AND
NV,SF,ZF,PF,NC

XOR

sy. : ADD
but := but ***xou*** *source,*
bit à bit
I : NV,SF,ZF,PF,NC

CHAPITRE 4

STRUCTURATION.

Nous abordons à présent la partie 'savoir faire' la plus 'mouvante' des moyens et ressources de l'ordinateur : celle qui n'est pas enchâssée dans un utilitaire livré avec la machine et qui constitue d'une certaine manière le 'soft' du 'soft'. Sans être une conséquence obligée des théories exposées au chapitre 2, ces 'savoir faire' s'organisent, sous cet éclairage, en un instrument extrêmement précieux pour le programmeur.

4.1 Le cadre et l'outil.

Le chapitre 3 a mis à notre disposition un nombre suffisant d'instructions et d'utilitaires pour écrire des programmes. Pour les mettre en oeuvre, il nous faut éclaircir complètement deux points : l'exécution des programmes sous *DOS* et le maniement de *DEBUG* qui nous servira à *construire* nos programmes. Nous commençons par l'illustration de ce second point sur un programme donné au chapitre 1.

Un exemple.

Rappelons le texte du premier programme, en indiquant à présent, en commentaires, l'action des instructions :

```
0100 B91400          MOV  CX,0014        ; CX := 014
0103 BE0001          MOV  SI,0100        ; SI := 0100, donc début du code
0106:BOUCLE                              ; répétition tant que CX > 0
0106 8A14            MOV  DL,[SI]        ; DL := by DS:[SI]
0108 B402            MOV  AH,02          ; fonction 02 de DOS :
010A CD21            INT  21             ; afficher DL à l'écran DL
010C 46              INC  SI             ; on passe à l'octet suivant
010D E2F7            LOOP 0106           ; revenir à 0106 tant que CX > 0
010F B8004C          MOV  AX,4C00        ; boucle terminée : la fonction 4C
0112 CD21            INT  21             ; de DOS rend le contrôle de
0114                                     ; l'exécution à COMMAND.COM
```

Ce programme affiche donc une suite de **014** octets commençant à l'adresse relative **0100** et dont l'adresse de segment est dans *DS* (associé à *SI*). Or ici, *DS* n'est ni initialisé, ni modifié dans ce programme. Quelle est donc cette adresse de segment initiale? Qui l'a initialisée et selon quels principes? C'est évidemment le maître d'oeuvre de *DOS*, *COMMAND.COM*. Nous le verrons plus en détail ci-après, au début de l'exécution du programme, *DS* reçoit la même valeur que *CS*. Cela signifie que, dans notre exemple, *DS:SI* pointe sur le début du code du programme. Or, la colonne de gauche du listage ci-dessus donne l'adresse du premier octet de chaque instruction. On en déduit que la longueur du programme est de **014** octets : notre programme affiche donc à l'écran son propre code soit un mot de 20 lettres pris dans le jeu des caractères

ASCII. Ce n'est pas très parlant, mais n'en constitue pas moins une application immédiate du théorème du point fixe...

Pour faire exécuter ce programme par *COMMAND.COM*, il faut le transformer en un fichier *.COM*. Indiquons comment faire sous *DEBUG* :

Tout d'abord, on recopie les instructions ci-dessus dans un fichier sous la forme suivante :

```
a
;
;  programme qui redonne son code machine
;
          mov       cx,0014     ;
          mov       si,0100     ;
;  106:                         ;
          mov       dl,[si]     ;
          mov       ah,02       ;
          int       21          ;
          inc       si          ;
          loop      106         ;
;                                ;  DOS:
          mov       ax,4C00     ;
          int       21          ;

r cx
14
r bx
0
nptfixe.com
w
q
```

La commande *a* permettra de faire assembler les instructions par *DEBUG*. La ligne vide (retour chariot) permet de sortir de cette commande. Par *r*, on place dans *BX:CX* la taille du fichier à créer et par *n*, on donne son nom. La commande *w* crée le fichier sur le disque en lui donnant comme contenu les **014** octets depuis *CS*:**0100**. Il est important de donner un nom d'extension *.COM*.

Mais puisque *DEBUG* fonctionne en mode conversationnel avec l'utilisateur, comment lui faire assembler ce qui est dans un fichier? Tout simplement, grâce à la ***redirection*** des commandes du *DOS*. Si on appelle, par exemple, **ptfixe.csm** le programme ci-dessus, on entrera à l'écran :

```
C:\>debug < ptfixe.csm > ptfixe.bsm
```

Le symbole < indique à *DEBUG* que les ordres qu'il attend doivent être lus dans le *fichier source* **ptfixe.csm** et le symbole > indique que tout ce qui ordinairement sort à l'écran doit être écrit dans le *fichier but* **ptfixe.bsm**. Un développement de ce programme sur le thème du point fixe est proposé page ci-contre.

IMPORTANT : ne pas oublier, dans le fichier source, de terminer la session de *DEBUG* par la commande *q suivie du retour chariot*.

Dans la suite de l'ouvrage, pour disposer d'une information plus dense, les programmes seront présentés en la forme donnée à la page 63.

Le lecteur peut vérifier la méthode sur un programme 'amélioré' redonnant son propre code sous forme hexadécimale, en appliquant la redirection au fichier **.csm** ci-après :

```
a
;
;  programme qui redonne son code machine écrit en hexadécimal
;
          db        B9 3C 00 BE 00 01 BB 10 00 51 30 E4 8A 04
          db        F6 F3 88 E6 88 C2 E8 15 00 86 D6 E8 10 00
          db        B2 20 B4 02 CD 21 46 59 E2 E3 B8 00 4C CD
          db        21 90 80 C2 30 80 FA 3A 72 03 80 C2 07 B4
          db        02 CD 21 C3

u 100 L 3C
g = 100 13C
q
```

Le mot **db** annonce que ce qui suit sur la ligne est une suite d'octets à lire tels quels, on dit encore, des ***données***. La commande ***g* = 100 13C** permet de faire débuter l'exécution à partir du premier octet des données ci-dessus (**0100** comme on le verra plus loin) et d'aller jusqu'à l'instruction d'adresse **013C**. La commande ***u* 100 *L* 3C** permet de ***désassembler*** la zone **0100-013C** interprétée par *DEBUG* comme code hexadécimal. Ceci permet d'obtenir le texte de ce programme. Vérifier qu'il comprend deux appels à un sous-programme **12C** affichant un chiffre hexadécimal à l'écran :

```
012C 80C230        ADD  DL,30    ; conversion en caractère
012F 80FA3A        CMP  DL,3A    ; DL < 3A? (chiffre décimal?)
0132 7203          JB   0137     ; oui : -> chiffre obtenu
0134 80C207        ADD  DL,07    ; non : additif hexadécimal
0137 B402          MOV  AH,02    ; fonction 02 de DOS :
0139 CD21          INT  21       ; affichage du caractère
013B C3            RET           ; retour à l'appelant
```

4.1.1. L'exécution des programmes sous DOS.

Le maître d'oeuvre de *DOS* définit trois formes de programmes exécutables. La première est celle des fichiers *.COM*, la seconde, celle des fichiers *.EXE* et la troisième, celle des fichiers *.BAT*. Cette dernière forme définit un fichier contenant des commandes à *COMMAND.COM* telles que l'utilisateur les entre lui-même à l'écran, reliées par un langage très rudimentaire. Nous ne considérerons ici que les deux formes définissant des fichiers contenant du code binaire : les fichiers *.COM* et *.EXE*.

La grande majorité des programmes de cet ouvrage sont des fichiers *.COM*. Comme il n'est pas raisonnable d'écrire sous *DEBUG* des fichiers *.EXE*, nous étudierons leur fonctionnement au chapitre 8, où nous étudierons un macro-assembleur.

Le mécanisme de l'exécution.

Un fichier *.COM* est réputé contenir un code binaire que nous représentons par sa forme hexadécimale, et il se caractérise par une taille ne dépassant pas 65278 octets. Lorsque l'utilisateur demande son exécution en entrant, dans la ligne de commande, le nom du fichier avec ou sans l'extension *.COM*, le maître d'oeuvre appelle une fonction de *DOS* chargée de ce travail : la fonction *EXEC* encore appelée ***chargeur*** de *DOS*. Ce nom provient du mode de fonctionnement de la fonction. En effet, le chargeur commence par faire une *copie* du fichier en mémoire vive avant de transférer le contrôle de l'exécution au code ainsi installé.

Revenons sur les différentes étapes de ce mécànisme.

La copie du fichier en *RAM* suppose une organisation de la *RAM*. Après l'installation initiale, une des tâches du maître d'oeuvre consiste, précisément, à gérer cette structure en fonction des commandes qui lui sont présentées : pour charger un fichier *.COM*, encore faut-il qu'il y ait de la place. Pour ce faire, *DOS* conçoit l'organisation de la *RAM*, espace plus restreint qu'un support de masse, sur un modèle plus simple que celui d'un répertoire associé à une table d'allocation. *DOS* adopte, dans ce cas, un système de pile : chaque fois que la réservation d'une zone est demandée, *DOS* fait précéder cette zone d'une ***marque*** qui consiste en un segment de seize octets dans lequel figure la taille, en paragraphes, de la zone. Cette marque indique en outre, par quel programme la zone est réservée et si la zone est suivie ou non par une autre zone également réservée. Il suffit donc de connaître la première marque et de remonter la chaîne des marques de réservations jusqu'à la dernière pour connaître la place libre restant en mémoire.

Quand il est appelé, le chargeur de *DOS* réserve toute la mémoire disponible pour le programme *.COM* qui va être exécuté. Sur le premier segment de 64 Ko de la place réservée, il définit une zone de 256 octets appelée le ***préfixe*** ou encore PSP (*Program Segment Prefix*) et le fichier *.COM* est recopié à partir du premier octet qui suit le préfixe : le programme est installé. Puis il initialise les registres du processeur. Si *seg* est le numéro de paragraphe du préfixe du programme, les quatre registres de segment prennent la valeur *seg*. Les autres registres n'ont pas de valeur significative sauf SP. En effet, le chargeur de DOS initialise la pile connue du processeur par *SS*:*SP* en *seg*:**0FFFE** ou bien, si la place réservée au chargement est égale, en octets, à *taille* avec taille < 64 Ko, en *seg*:*taille*-2. Au sommet de la pile, le chargeur place le mot 00. Lorsque la pile est installée, le chargeur transfère l'exécution au code installé par un **jmp far** à l'adresse *seg*:**0100**, adresse du premier octet du code chargé en *RAM*.

Pour que l'exécution revienne au maître d'oeuvre lorsque le programme est terminée, ce dernier doit se conclure (du point de vue du fonctionnement) principalement de deux façons : soit par un appel à la fonction **04C** de l'interruption **021**, soit par un appel à l'interruption **020**.

Une troisième méthode existe, préconisée par certains et décriée par beaucoup, remontant à la première version du *DOS* : l'exécution d'un **ret**. Comme un **ret** équivaut à l'action **jmp [sp]** et que [*SP*] = 00 si les **push** et **pop** du programme s'équilibrent, ceci équivaut à **jmp 0000**. Or, cette adresse renvoie au deux premiers octets du préfixe du programme dont le contenu est : **CD 20**, c'est-à-dire, **Int 20**.

On peut également se demander pourquoi l'exécution n'est pas transférée au programme depuis le chargeur de *DOS* par un **call far**. Admettons que le chargeur procède de la sorte. Le saut au programme est la dernière instruction du chargeur avant passage au code installé. La pile est donc en place et les adresses de retour seront donc sur cette pile quand le programme appelé se terminera. Pour que le retour se produise, il faudrait terminer le programme par **retf**. Ce qui suppose deux conditions : que cette instruction renvoie à la même pile et que cette dernière soit restée équilibrée. En appelant un utilitaire extérieur qui sait, depuis l'initialisation sous *DOS*, où retourner chaque fois qu'un programme se termine, on peut affranchir le programmeur de ces deux contraintes.

La gestion de la RAM.

La gestion de la *RAM* par *DOS* repose sur un certain nombre de fonctions de l'interruption **021** dont plusieurs ne sont pas documentées. Pour les programmes que nous écrirons, nous nous contenterons de quatre opérations : mesurer la place disponible, réduire la place allouée au programme *.COM*, réserver une zone mémoire, la rendre au maître d'oeuvre.

La première et la troisième opération sont réalisées par la fonction **048** de l'interruption **021**, que nous illustrons par les deux exécutions suivantes :

```
 AX   BX   CF   IP                        AX   BX   CF    IP
0000 0000       0100  MOV AH,48          0000 0000       0100  MOV AH,48
4800 0000       0102  MOV BX,1000        4800 0000       0102  MOV BX,FFFF
4800 1000       0105  INT 21             4800 FFFF       0105  INT 21
36A5 1000  NC   0107                     0008 879A  CY   0107
```

A gauche, nous le savons déjà puisque *NC*, la réservation, on dit aussi ***allocation***, a été faite : par *BX*, on a réservé **01000** paragraphes, c'est-à-dire 64 Ko, et la fonction répond en plaçant en *AX* le numéro de paragraphe du début de la zone allouée. A droite, on a demandé **FFFF** paragraphes, soit 1024 Ko. Ceci sort des normes classiques : d'où le rejet de la demande signifié par *CY*. Cependant, le maître d'oeuvre s'explique : le code d'erreur **08** dit que la mémoire disponible est insuffisante. La preuve, précise-t-il : le registre *BX* fournit le nombre de paragraphes disponibles. Les instructions de droite indiquent en fait ce qu'il faut faire pour obtenir la mesure de la place disponible : en demander trop pour que le système fournisse, avec ses 'excuses' le renseignement dont on a besoin.

La seconde fonction est celle par laquelle on doit commencer. En effet, lors du chargement du programme *.COM*, le maître d'oeuvre lui a alloué toute la mémoire disponible. Si le programme veut réserver une place, il doit commencer par rendre de la place au système en modifiant la place qui lui a été allouée. C'est la tâche de la fonction **4A**. Ses arguments sont *AH* = **4A**, *BX*, la nouvelle taille demandée en paragraphes, *ES* l'adresse de segment du début de la zone allouée, et pour la nouvelle taille, on peut demander moins, mais on peut aussi demander plus. Au retour, la valeur de *CF* indique l'issue de la tâche effectuée. Si *NC*, la taille de la zone du programme a été réajustée. Le programmeur devra en tenir compte pour les instructions suivantes. Si *CY*, c'est, en général, qu'on a voulu augmenter la taille de la zone et que cela est impossible. *BX* indique alors le nombre de paragraphes disponibles.

La quatrième fonction est ce que l'on doit faire, en principe, quand on s'en va... On rend la place dont on n'a plus besoin au système. C'est ce qu'exécute la fonction **49** à qui l'on fournit *AH* = **49** et dans *ES*, l'adresse de segment du début de la zone que l'on restitue. La fonction rend compte par *CF*. Il y aura erreur, par exemple, si l'adresse donnée n'est pas celle d'un début de zone réservée : on l'aura, dans ce cas, oubliée.

A noter que pour ces quatres fonctions, les registres *CX*, *DX*, *SI*, *DI* et *BP* restent inchangés. Le registre *AX* peut l'être par les fonctions **4A** et **49** et l'est toujours par la fonction **48**. Le registre *BX* n'est pas modifié par la fonction **49**.

Structure d'un programme .COM.

Les explications précédentes donnent la raison de la limite de 65278 octets à la taille du fichier contenant le code du programme : 65278 = 65536 - 256 - 2. Comme le fichier, tel qu'il est, sera chargé à une adresse **0100**, il faut l'assembler sous *DEBUG* à partir de cette adresse, ce que nous voyons plus loin. Il faut aussi que le programme commence par une instruction puisque l'exécution, sous son égide, débutera à partir de là. Ensuite, il pourra s'assurer que les quatres registres de segments ont la valeur requise en alignant la valeur des trois autres sur celle de *CS* par des **mov** judicieux. Il peut éventuellement restreindre sa taille : ne pas oublier de modifier *SP*, si besoin, avant tout emploi de la pile. Il peut aussi procéder à de premières allocations de mémoire. Ensuite, ce qui sera écrit dépendra de l'objet du programme.

Mais le programme chargé en *RAM* est plus que la copie du fichier *.COM* car il dispose d'un *préfixe*. Cette zone contient un certain nombre d'informations inscrites par le chargeur de *DOS*. Nous ne détaillerons pas tout mais retiendrons surtout que les deux

premiers octets sont occupés par le code d'une instruction : CD 20, soit **int 20**[1], et qu'à partir de l'octet **080** de cette zone, se trouve la ***ligne de commande***. Sur cette ligne, le chargeur inscrit depuis **081** ce que l'utilisateur a entré à l'écran, entre le nom du fichier *.COM* et le retour chariot, y compris le retour chariot (**0D**), et inscrit en **080** la longueur de ce qu'il a copié, retour chariot exclu.

Exemple :

On entre à l'écran la commande

```
C:\>ptfixe deux paramètres
```

On obtient alors, dans le préfixe de ptfixe, depuis l'octet **080** :

```
10 20 64 65 75 78 20 70 61 72 61 6D 8A 74 72 65 73 0D
      d  e  u  x     p  a  r  a  m  è  t  r  e  s
```

Les logiciels qu'on peut appeler en entrant à l'écran une liste d'options exploitent tout simplement cette possibilité.

Le préfixe contient d'autres zones affectées à la gestion des fichiers selon les normes des premières versions de *DOS*. Si on utilise le système des sésames, ces zones ne seront jamais utilisées par le programme.

La fonction EXEC.

La fonction *EXEC* de *DOS* est la fonction **4B** de l'interruption **021**. Cette fonction constitue le chargeur du maître d'oeuvre. Elle demande, en entrées : *AH* = **4B** ; *AL* = **00** ou **03**, selon qu'il s'agit de charger et exécuter (**00**) ou de charger seulement. *DS*:*DX* pointe une zone mémoire contenant le nom du *fichier* à charger. La fin du nom doit être délimitée par **00** et le nom doit être un nom valide de fichier *DOS* pouvant comporter un chemin (cf. chapitre 3) ; *ES*:*BX* pointe sur une zone mémoire contenant des pointeurs sur les paramètres que peut utiliser le programme chargé. Les paramètres seront essentiellement passés sous la forme d'une *ligne de commande* fonctionnant sur le modèle de ce que nous avons vu dans le préfixe d'un programme *.COM*. Lorsque la fonction *EXEC* chargera le programme désigné par *DS*:*DX*, elle recopiera cette ligne de commande sur celle du préfixe qu'elle aura installé pour le programme à charger. Il ne faut donc pas oublier de mettre dans le premier octet de la ligne fournie par le programme appelant, le nombre d'octets transmis, qui ne devra pas comptabiliser le retour chariot final (recopié lui aussi). La zone pointée par ES:BX comportera 0E octets structurés de la façon suivante (cf. convention d'adressages de la page 30) :

octets 00-02	adresse de segment des paramètres d'environnement
octets 02-06	adresse absolue de la ligne de commande
octets 06-0A	adresse absolue du premier bloc de paramètres fichiers
octets 0A-0E	adresse absolue du second bloc de paramètres fichiers.

Dans le système de gestion des fichiers par sésame, ces deux blocs de paramètres sont inutiles. On peut donc donner aux octets concernés les valeurs que l'on veut. Pour passer au programme appelé les mêmes paramètres d'environnement que le programme appelant, il suffit de donner la valeur 0000 à l'adresse de segment.

RAPPEL : un environnement est un ensemble de paramètres confiés à *DOS*, chacun étant de la forme **nom=chaine**. Dans le bloc des paramètres d'environnement, chaque groupe de cette forme doit être ponctuée par **00** et le dernier groupe doit être suivi par deux **00**. Un environnement vide se signale par trois **00** consécutifs.

---

[1] Voir page précédente, en 'seconde lecture'.

Comme la plupart des fonctions *DOS*, la fonction *EXEC* rend compte du déroulement de sa tâche par *CF*. Si *CN*, elle a exécuté ce qu'il faut, si *CY*, elle signale par *AX* ce qui l'a empêché de charger le programme.

A noter qu'elle ne peut pas *détecter* un mauvais fonctionnement du programme chargé. Mais si c'est le cas, l'utilisateur s'en apercevra de lui-même.

Comme la fonction *EXEC* modifie les registres du processeur pour passer l'exécution au programme qu'elle charge, il faut éventuellement les sauvegarder afin de les restaurer au retour, en particulier, *SS:SP*. Comme au retour, le seul registre dont on soit sûr est *CS* (et pour cause!), la sauvegarde de *SS:SP* ne se fera pas sur la pile, mais dans une zone mémoire repérée par rapport à *CS*.

4.1.2. Le maniement de DEBUG.

La vérification et la mise au point sont un moment crucial de la programmation. Dans ce but, le mode d'exécution des programmes en pas à pas de *DEBUG* est extrêmement précieux. Mais ce constructeur offre bien d'autres possibilités.

Retour au problème du chronomètre.

Nous avons donné, au chapitre 1, les instructions d'une fonction de soustraction de deux *heures* afin d'en déduire une *durée*. Le programme comprend en fait deux parties comme l'indique le listage que voici :

```
0100 BF6001          MOV  DI,0160        ; DI pointe sur l'heure d'arrivée
0103 BE5001          MOV  SI,0150        ; SI pointe sur l'heure de départ
0106 BB8001          MOV  BX,0180        ; BX : tableau des correctifs
0109 B90400          MOV  CX,0004        ; la durée comprend 4 nombres
010C FC              CLD                 ; (remplissage pour adresse paire)
010D F8              CLC                 ; CF neutralisé
010E:BOUCLE                              ;
010E 51              PUSH CX             ; compteur de boucle sauvé
010F 8A05            MOV  AL,[DI]        ; composante heure d'arrivée
0111 8A24            MOV  AH,[SI]        ; composante heure de départ
0113 8A0F            MOV  CL,[BX]        ; correctif pour ces composantes
0115 E8E800          CALL 0200           ; soustraction 'complexe'
0118 884510          MOV  [DI+10],AL     ; inscription du résultat
011B 47              INC  DI             ; mise à jour des pointeurs
011C 46              INC  SI             ;
011D 43              INC  BX             ;
011E 59              POP  CX             ; compteur de boucle retrouvé
011F E2ED            LOOP 010E           ; retour à BOUCLE
```

Le programme 0200 est celui que nous avons vu au chapitre 1 :

```
0200 80D400          ADC  AH,00          ; report de la retenue précédente
0203 38E0            CMP  AL,AH          ; AL >= AH?
0205 7307            JNB  020E           ; oui : soustraction normale
0207 00C8            ADD  AL,CL          ; non : ajouter le correctif
0209 28E0            SUB  AL,AH          ; soustraction
020B F9              STC                 ; retenue
020C C3              RET                 ; retour à l'appelant
020D 90              NOP                 ;
020E 28E0            SUB  AL,AH          ; soustraction
0210 F8              CLC                 ; pas de retenue
0211 C3              RET                 ; retour à l'appelant
```

Nous insérons ces deux programmes dans un fichier **duree.csm**, et nous installons le *contexte* de la vérification comme indiqué sur le listage du haut de la page suivante, dans lequel ne figurent que les commandes de *DEBUG*.

Après les instructions ci-contre reproduites sous une commande a d'assemblage, on insère les données : l'heure de départ et l'heure d'arrivée. Les commentaires annoncés par % ont été ajoutés au listage :

```
f 150 L 80 00                 % remplissage par 00 pour lire le
                              % résultat plus commodément
e 150 1F 0C 05 17             % heure de départ : 23h 05mn 12s 31
e 160 1F 09 05 00             % heure d'arrivée : 00h 05mn 09s 31
e 180 64 3C 3C 18             % les correctifs : 24h 60mn 60s 100

g = 100 121                   % exécution depuis 0100 jusqu'à 0121
d 150 L 40                    % afficher la zone 0150-0190
q                             % et ne pas oublier le retour chariot!
```

Après l'exécution, par *g* des instructions **100 à 121**, on obtient le résultat suivant, affiché par la commande *d* :

```
-d 150 L 80
13CE:0150  1F 0C 05 17 00 00 00 00-00 00 00 00 00 00 00 00
13CE:0160  1F 09 05 00 00 00 00 00-00 00 00 00 00 00 00 00
13CE:0170  00 39 3B 00 00 00 00 00-00 00 00 00 00 00 00 00
13CE:0180  64 3C 3C 18 00 00 00 00-00 00 00 00 00 00 00 00
```

Le résultat est donc : 0h 59mn 57s 00

Pour des raisons typographiques, nous n'avons pas reproduit le 'décodage' par *DEBUG* sous forme 'lisible' des seize octets de chaque ligne.

Dans ce programme de test, nous avons utilisé les commandes suivantes de *DEBUG* :

La commande *f* remplit une zone dont on précise le point de départ et la taille en octets en recopiant dans la liste d'octets suivant l'indication de la taille, la liste étant recyclée autant que de besoin.

La commande *e* permet d'inscrire des valeurs à partir d'une adresse donnée : on entre d'abord l'adresse du premier octet à écrire, puis les valeurs à inscrire en mémoire à partir de cette adresse, contigûment et par adresses croissantes.

Syntaxe des commandes de DEBUG.

Rappelons que toute donnée sous *DEBUG* s'écrit en hexadécimal. Une commande est constituée par *une lettre* suivie éventuellement de *paramètres*. Ceux-ci peuvent prendre les formes suivantes : *adresse*, *plage*, *valeur* ou *liste*.

Une valeur est un *mot* hexadécimal, une liste, une suite d'*octets* ou bien une chaîne de caractères comprise entre deux apostrophes. Une adresse a la forme *valeur* ou bien *valeur1*:*valeur2* ou bien *RS*:*valeur*, où *RS* est un des quatre noms des registres de segment. Une *plage* est de la forme *adresse* **L** *valeur* ou bien *adresse*,*valeur*, où *valeur* est l'adresse relative d'un octet venant après celui d'*adresse* dans le même segment. Dans le premier cas, *DEBUG* comprend la zone *adresse*,*adresse+L* et dans le second cas la zone *adresse*,*adresse+T* où, si *adresse* = *seg*:*place*, *T* = *valeur-place*+1.

Une commande de la forme *z adresse* signifie que *z* agit à partir de l'adresse indiquée. Une commande de la forme *z plage* signifie que *z* agit dans les limites de la zone délimitée par *plage*.

Les commandes principales sont données dans le premier tableau de la page ci-contre, regroupées selon leur syntaxe. On constate que, dans de nombreux cas, le paramètre peut être omis. Lorsqu'il en est ainsi, *DEBUG* lui donne en fait une valeur dite ***par défaut***. Lorsqu'une plage peut être omise, elle peut être donnée de façon incomplète en n'indiquant que l'adresse de départ. La taille de la plage est alors fixée

```
a,d,u [plage]                              % assemble, affiche, désassemble
c,m adresse plage                          % compare, recopie
e [adresse] liste                          % écrit en RAM
f,s plage liste                            % remplit, cherche
g [=adresse] [adresse [adresse] ... ]      % exécute
h valeur1 valeur2                          % calcule
l,w [adresse]                              % charge en RAM, écrit sur disque
n nom[complet]                             % nomme
p,t [=adresse] [valeur]                    % pas à pas, trace
q                                          % quitte
r                                          % affiche registres
```

Tableau des commandes de *DEBUG*. Les crochets signifient que le paramètre qu'ils entourent est facultatif.

`d`	*DS:D1*, *L* = **080** (*D1* = **0100** au début de la session)
`a`	*CS:IP* (*IP* = **0100** au début de la session)
`u`	*CS:U1*, *L* = **020** (*U1* = **0100** au début de la session)
`g`	de *CS:IP* à la fin
`l,w`	depuis l'adresse **0100**
`p,t`	*CS:IP* (*IP* = **0100** au début de la session)
`r`	tous les registres ; après l'action de *r*, *U1* = *IP*

Tableau des valeurs par défaut. *D1* pointe sur l'octet qui suit le dernier octet affiché par la commande *d*. Même relation entre *U1* et *u*.

également par défaut. Le second tableau ci-contre donne les valeurs par défaut des paramètres pour les commandes concernées.

Les commandes de l'exécution contrôlée.

Traces et pas à pas.

Les commandes de pas à pas sont les commandes *p* et *t* que nous avons déjà utilisées. Appelées sans paramètre, toutes deux exécutent l'instruction pointée par *CS:IP*. Le premier paramètre indique l'adresse *A* de l'instruction à exécuter. Ainsi *p* = **0112** applique *p* à l'instruction débutant à l'octet **0112**. Le second paramètre indique le nombre d'instructions auxquelles on applique successivement la commande depuis *A*. Par défaut, ce nombre est 1. Si on indique un nombre mais pas l'adresse *A*, celle-ci est prise par défaut selon le tableau ci-dessus.

La différence entre *p* et *t* concerne les instructions 'complexes' : **loop**, **call**, **int** et les instructions préfixées par **rep**. Pour les autres instructions, *p* et *t* agissent de la même façon que nous connaissons déjà.

Pour les instructions indiquées, *p* exécute l'ensemble de l'action impliquée par l'instruction jusqu'à rencontrer l'instruction suivante, tandis que *t* n'exécute que l'action stricte définie par l'instruction.

Prenons l'exemple de la boucle **010E** du programme de chronométrage et soit *IP* = **011F**, de sorte que *CS:IP* pointe sur **loop 010E**. La commande *p* = **011F** aura pour effet d'aller à l'instruction **0121** en exécutant tout ce que le processeur fait entre le moment où *IP* pointe sur **011F** et où *IP* pointe sur **0121**, à savoir l'exécution de la boucle, autant de fois que l'exige la valeur de *CX* au moment où l'on a déclenché *p*[1]. Si par contre on exécute *t* = **011F**, on se retrouve en **0121** si *CX* = **0001** à l'application de *t* et en **010E** sinon.

La commande *t* permet donc de vérifier le fonctionnement d'une boucle en la retraversant autant de fois qu'on le jugera nécessaire. Une fois la boucle vérifiée, la

---

[1] Donc, veiller à ce qu'alors *CX* ≠ 00.

poursuite de la mise au point se fera en utilisant la commande *p* lorsqu'on rencontrera l'instruction **loop** correspondante.

Il en est de même pour **call** et **int**. Signalons qu'en dehors des interruptions que l'utilisateur créera lui-même, la commande *t* n'est pas très utile pour les interruptions fournies par le *DOS* ou le fabricant de la machine[1].

Points d'arrêts.

Lorsque l'on est plus avancé dans la mise au point, le pas à pas devient vite fastidieux. On souhaite alors 'passer' sur les plages vérifiées pour concentrer son attention sur les points délicats. La commande *g* est là pour cela. Dans sa forme complète, on indique l'adresse de l'instruction de départ *I*, introduite par le signe =, puis une liste de ***points d'arrêts***, *A*1,...,*Ak*. L'exécution part de *I* et se poursuit jusqu'à ce que l'une des instructions d'adresses *A*1,...,*Ak* soit rencontrée.

Il faut être ***très prudent*** avec cette commande, car elle transfère le contrôle du processeur à la suite d'instructions commençant avec l'instruction *I*, ce qui est le meilleur moyen de tester 'en vrai grandeur' l'action du programme. Comment alors arrêter le processeur lorsque *IP* prendra une des valeurs désignées par *A*1,...,*Ak*? Tout simplement en remplaçant chaque octet défini par cette suite d'adresse par l'interruption **03** qui se code sur un octet. Lorsque le processeur rencontre cet octet, il exécute donc l'interruption **03** que *DEBUG* a reconstruite dans ce but au début de la session, et qui consiste à renvoyer en un point précis du programme de *DEBUG*. Naturellement, *DEBUG* conserve la copie de ces octets et des points d'arrêts pour restaurer le code initialement soumis à la commande *g* lorsque le contrôle du processeur lui sera revenu.

D'où vient le danger? C'est que si l'on se trompe dans le placement des points d'arrêt, tout peut se produire. Illustrons le sur l'exemple du programme **0200** suivant :

```
0200 80D400 ADC AH,00      0200 80D400 ADC AH,00      0200 80D400 ADC AH,0
0203 38E0   CMP AL,AH      0203 38E0   CMP AL,AH      0203 38E0   CMP AL,AH
0205 7307   JNB 020E       0205 7307   JNB 020E       0205 7307   JNB 020E
0207 00C8   ADD AL,CL      0207 CC     INT 3          0207 00CC   ADD AH,CL
0209 28E0   SUB AL,AH      0208 C8     DB  C8         0209 28E0   SUB AL,AH
020B F9     STC            0209 28E0   SUB AL,AH      020B F9     STC
020C C3     RET            020B F9     STC            020C C3     RET
020D 90     NOP            020C C3     RET            020D 90     NOP
020E 28E0   SUB AL,AH      020D 90     NOP            020E CC     INT 3
0210 F8     CLC            020E CC     INT 3          020F E0F8   LOOPNZ 0209
0211 C3     RET            020F E0F8   LOOPZ  0209    0211 C3     RET
                           0211 C3     RET
```

A gauche, le code initial. Au centre, le même après insertion de points d'arrêts en **0207** et **020E** pour tester l'exécution de **jnb 020E** depuis **0200**. A droite, une erreur a été introduite : le premier point d'arrêt est placé en **0208**.

Analyse de l'erreur : le point d'arrêt n'existe plus mais l'instruction visée a été modifiée. Si *AL* < *AH*, on passe par **0207** qui introduit un ***calcul faux***. Puis on continue jusqu'au **ret**. Si [*SP*] = 0000, ce qui se produit si on est en début de session, et qu'on n'a pas encore agi sur la pile, un **jmp 0000** sera exécuté induisant le message : **Fin normale du programme**, ce qui est évidemment une anomalie puisqu'on aurait du se retrouver à l'instruction **0208**. Sinon, on se trouve comme si la commande *g* était exécutée sans paramètres de sorte que si la mise au point n'est pas terminée, le risque d'un plantage est très élevé. Faire donc très attention quand on modifie des adresses.

Lorsque le programme est mis au point, on peut encore effectuer un test global par la commande *g* sans paramètres qui fera exécuter le programme depuis l'instruction pointée par *CS:IP* et sans introduire de point d'arrêt. Attention donc à cette commande

---

[1] Evidemment, cette commande permet d'aller voir comment sont faites les dites interruptions. Ce qui prendra beaucoup, beaucoup de temps à tout ceux qui le feront.

qui supporte mal les fautes de frappe. Il est en outre préférable de la tester en mode conversationnel avant de l'inclure dans un fichier destiné à *DEBUG*.

Manipulation d'un fichier .COM sous DEBUG.

Les principales commandes utilisées sont *n*, *l*, *w* et *e*. La commande *n* désigne le fichier sur lequel on travaille. Par défaut, c'est le nom du fichier entré dans la ligne de commande à l'appel de *DEBUG* et sinon, il n'y a pas de tel fichier. La commande *n* permet donc d'initialiser le nom du fichier de travail ou d'en changer si on veut passer à autre chose. Mais *nommer* un fichier n'est pas le *charger* en RAM. Ceci est la tâche de la commande *l*. Elle prend donc le fichier de nom connu, affiche **fichier introuvable** si aucun nom n'a été donné, et le charge depuis l'adresse indiquée. Par défaut, cette adresse est *CS*:**0100**. Pour un fichier *.COM*, l'adresse *CS*:**0100** est ***obligatoire***. Après le chargement, *DEBUG* affiche dans *BX*:*CX* la taille du fichier en octets.

Pour écrire dans le fichier, y faire des corrections par exemple, on utilisera principalement *e* qui fonctionne aussi en mode interactif. Dans ce mode, *e* répond en affichant la valeur actuelle de l'octet indiqué suivie d'un point après lequel le curseur clignote, attendant la réponse. On peut alors entrer une valeur octet. On revient à *DEBUG* par le retour chariot. Si l'on veut entrer des valeurs pour les octets voisins, la barre d'espacement fait passer à l'octet d'adresse +1, la touche - à l'octet d'adresse –1. Pour écrire des textes importants, la méthode la plus simple est de passer par la redirection comme nous l'avons indiqué précédemment.

La commande *w* est l'inverse de la commande *l*. Connaissant un nom, elle l'écrit sur disque en prenant pour source la zone mémoire débutant à l'adresse indiquée, *CS*:**0100** par défaut, et s'étendant sur *BX*:*CX* octets. Attention donc à la valeur fournie par *BX*:*CX*. Pour cette raison, si le fichier *.CSM* créé sur les modèles précédents contient des ordres d'écriture et des ordres d'exécution à point d'arrêt, il vaut mieux placer les ordres d'écriture avant.

L'adresse **0100** n'est pas obligatoire dans ce cas pour un fichier *.COM*. Nous utiliserons largement cette possibilité pour constituer de grands programmes *morceaux par morceaux*. Nous le verrons à partir du chapitre suivant.

4.2 La structuration d'un programme.

A présent que nous disposons d'outils importants pour tester les programmes que nous écrirons, concentrons-nous sur l'aspect *logique* de la programmation. Illustrons la démarche sur un exemple que nous revisiterons plus loin : la très célèbre suite des nombres de Fibonacci dont nous nous proposons de calculer le $n^{ème}$ terme.

4.2.1. Exemple : les nombres de Fibonacci.

Implantation de l'algorithme.

La suite de Fibonacci est définie par les équations suivantes :

$f_0 = 0, f_1 = 1,$
$f_{n+1} = f_n + f_{n-1}$ pour $n \geq 1$.

Pour programmer cette suite, il nous suffit de réserver une place pour f_n et une autre pour f_{n-1} : nous dirons qu'au rang n, f_n est le premier élément et f_{n-1} le second. Quand on passe du rang n au rang $n+1$, le premier élément du rang n devient le second du rang $n+1$, et le premier élément du rang $n+1$ est la somme des deux éléments du

rang n, ce que l'on est tenté de transcrire en assembleur de la façon suivante, en attribuant le registre *AX* à f_n et le registre *BX* à f_{n-1} :

```
0100      ADD  AX,BX
0102      MOV  BX,AX
```

Cependant, ce schéma est incorrect : quand l'instruction **add** a été exécutée, *AX* n'a pas la valeur f_n, mais la valeur $f_n + f_{n-1}$. Donc *BX* recevra f_{n+1} et non f_n. Il suffit donc de sauver *AX* sur la pile et de récupérer la valeur dans *BX* comme ceci :

```
0100      PUSH AX
0101      ADD  AX,BX
0103      POP  BX
```

pour obtenir une traduction correcte de la transformation. On l'insère ensuite dans une boucle permettant d'obtenir, quand on en sort, la valeur de f_n. Il suffit, pour obtenir la valeur correcte du $n^{ème}$ terme, d'initialiser les variables judicieusement.

Comment initialiser? Tout d'abord, on éliminera les deux cas particulier $n = 0$ et $n = 1$ qui ne se calculent pas par la boucle. L'utilisation de la boucle commence pour $n = 2$ et on passe alors *une fois* dans la boucle. Plus généralement, pour $n \geq 2$, on passe $n-1$ fois dans la boucle. D'où le programme :

On suppose n placé dans *CX* et le résultat dans *AX*.

```
0100 31C0          XOR  AX,AX    ; AX := f0
0102 E30F          JCXZ 0113     ; si CX = 0, c'est fini
0104 89C3          MOV  BX,AX    ; BX := f0
0106 40            INC  AX       ; AX := f1
0107 83F901        CMP  CX,+01   ; CX <= 1?
010A 7607          JBE  0113     ; oui : c'est fini
010C 49            DEC  CX       ; non : CX-1 boucles
010D:BOUCLE                      ;
010D 50            PUSH AX       ; conserver fn
010E 01D8          ADD  AX,BX    ; AX := fn + fn-1
0110 5B            POP  BX       ; BX := fn
0111 E2FA          LOOP 010D     ; retour à BOUCLE
0113 C3            RET
```

Nous avons là le 'coeur' d'un programme de calcul des nombres de Fibonacci. Si on l'insère dans un fichier analogue à **ptfixe.csm** comportant une initialisation du registre *CX* et une commande *g*=**100 113**, on obtiendra, dans le fichier *.BSM* produit par *DEBUG* redirigé le listage ci-dessus et le résultat du calcul dans le registre *AX*.

Les entrées/sorties : conversions.

Cette solution n'est pas satisfaisante : on peut légitimement souhaiter entrer le nombre n à l'écran et lire le résultat de la même manière. Il convient donc de modifier le programme pour y insérer une partie *entrée des données* avant le calcul du $n^{ème}$ nombre de Fibonacci et une partie *résultat* après ce calcul.

Or, dans la partie *calcul* du programme, les nombres sont exprimés en base seize, tandis qu'à l'écran, nous sommes habitués à les entrer ou les lire en base dix. Nous devons donc résoudre deux problèmes : la conversion d'un entier de la base seize à la base dix et l'opération inverse, le passage de la représentation décimale à la représentation hexadécimale.

Conversion de décimal en hexadécimal.

Commençons par la lecture du nombre entré au clavier. Le programme figure page ci-contre. On affiche tout d'abord un message demandant à l'utilisateur d'entrer le nombre, ce qui passe par l'appel à la fonction *DOS* **09**. L'argument de *DX* est l'adresse du début du message. Le nombre entré à l'écran est copié sur le tampon **0100** prévu à

```
0100 06 00 00 00 00 00 00 00          % TAMPON
0108 BA0802          MOV  DX,0208     ; message d'annonce
010B B409            MOV  AH,09       ; fonction affichage de DOS
010D CD21            INT  21          ;
010F BA0001          MOV  DX,0100     ; lecture sur le tampon
0112 B40A            MOV  AH,0A       ; par la fonction DOS ad hoc
0114 CD21            INT  21          ;
0116 31C9            XOR  CX,CX       ; CX := 0
0118 8A0E0101        MOV  CL,[0101]   ; CL := nombre de chiffres
011C B80000          MOV  AX,0000     ;
011F B30A            MOV  BL,0A       ; BL := 0A (dix)
0121 BD0201          MOV  BP,0102     ;
0124 31D2            XOR  DX,DX       ;
0126:BOUCLE                           ; sur le nombre des chiffres lus
0126 F7E3            MUL  BX          ;
0128 8A5600          MOV  DL,[BP+00]  ; lecture du chiffre courant
012B 80EA30          SUB  DL,30       ; convertî en décimale
012E 01D0            ADD  AX,DX       ; ajouté à la somme actuelle
0130 45              INC  BP          ; prochain symbole
0131 E2F3            LOOP 0126        ; retour à BOUCLE
0133 89C1            MOV  CX,AX       ; résultat dans CX
0135 C3              RET              ;
```

```
1. S := 0 ; (dans le programme, S :- AX) ;
2. ajouter à S la décimale courante (DL) ;
3. multiplier par 0A ;
4. incrémenter le pointeur de lecture ;
5. si non terminé, revenir au point 2.
```

cet effet par la fonction *DOS* **0A**. Commence ensuite le programme de conversion ci-dessus qui consiste à appliquer l'algorithme de Hörner.

Conversion d'hexadécimal en décimal.

Le programme est un simple intermédiaire : il convertit un mot (au plus, un double mot) placé dans un tampon en une chaîne de caractères prête à l'affichage. L'algorithme consiste en une division successive des quotients obtenus par dix. La terminaison est obtenue lorsque le quotient est zéro. Les décimales sont fournies une à une par les restes successifs, le premier reste correspondant au chiffre des unités.

Avant cette boucle, *DX* est initialisé à **00** et *BX* à **0A** (dix) ; *BP* pointe sur le chiffre hexadécimal de tête du nombre à convertir; le nombre divisé est sur un double mot, puisqu'on utilise l'instruction **div** de type mot :

```
0110:BOUCLE                           ; tant que le quotient est non nul
0110 A10A01          MOV  AX,[010A]   ; [010A] sert de lieu de stockage
0113 F7F3            DIV  BX          ; division par dix
0115 A30A01          MOV  [010A],AX   ; on sauve le quotient
0118 8812            MOV  [BP+SI],DL  ; on sauve la décimale courante
011A 31D2            XOR  DX,DX       ; on annule DX
011C 3D0000          CMP  AX,0000     ; quotient nul?
011F 7403            JZ   0124        ; oui : terminé
0121 46              INC  SI          ; non : actualiser le pointeur
0122 EBEC            JMP  0110        ; retour à BOUCLE
0124 C3              RET              ;
```

Le résultat obtenu doit être ensuite affiché. L'inclusion du résultat dans un texte ou, plus généralement, des éléments de présentation constitue un problème qui se résout simplement : la fonction **09** de *DOS* permet d'afficher une très grande variété de chaînes de caractères. Le résultat peut être affiché globalement par la même fonction, à condition d'avoir été transcrit en caractères, au fur et à mesure du calcul et d'avoir été constitué selon des adresses décroissantes, puisque le chiffre des unités est écrit le dernier. Ceci ne correspond pas à l'algorithme ci-dessus où l'on ne fait que convertir.

L'affichage de la conversion est l'objet d'un autre algorithme qui affiche, caractère par caractère, les décimales trouvées lues dans le bon ordre et converties en caractères, au fur et à mesure qu'on les lit. Ce que fait la boucle suivante :

La conversion est faite par lecture dans un tableau : si le chiffre lu est *x*, on lit le caractère correspondant à l'entrée *x* du tableau.

```
0100 DB             '0123456789'        ; tableau de conversion
0128 890E0A01       MOV  [010A],CX      ; initalisation de 0110
012C 31F6           XOR  SI,SI          ; initialisation des pointeurs
012E BD5201         MOV  BP,0152        ; SI, BP
0131 BB0A00         MOV  BX,000A        ; pour divisions par dix
0134 31D2           XOR  DX,DX          ; initialisation 1ère division
0136 E8D7FF         CALL 0110           ; CONVERSION
0139 BB0001         MOV  BX,0100        ; pointeur du tableau
013C 89F1           MOV  CX,SI          ;
013E 41             INC  CX             ; nombre de chiffres en décimal
013F:BOUCLE                             ; sur le nombre des chiffres
013F 31C0           XOR  AX,AX          ; AL := 0
0141 31D2           XOR  DX,DX          ; DL := 0
0143 8A02           MOV  AL,[BP+SI]     ; on lit la décimale courante
0145 89C7           MOV  DI,AX          ; et, par le tableau,
0147 8A11           MOV  DL,[BX+DI]     ; conversion en caractère
0149 B402           MOV  AH,02          ; qu'on affiche
014B CD21           INT  21             ;
014D 4E             DEC  SI             ; décimale suivante
014E E2EF           LOOP 013F           ; retour à BOUCLE
0150 C3             RET                 ;
```

Le programme final de calcul aura l'aspect suivant :

```
0100 E81D00         CALL 0120           ; lecture de N mis dans CX
0103 E84A00         CALL 0150           ; AX := Fib(CX)
0106 E8E700         CALL 01D8           ; affichage de AX, le résultat
0109 B8004C         MOV  AX,4C00        ; retour au DOS
010C CD21           INT  21             ;
```

Il suffit de recopier les sous-programmes précédents aux adresses indiquées, (le programme de conversion en **1B0** puisque son entrée, **0128**, correspond à **01D8**) en ajoutant quelques messages afin d'annoncer le résultat, pour 'assembler' définitivement ce programme. Nous verrons plus loin comment faire la copie de programmes 'utilitaires' mis au point une fois pour toutes.

D'ores et déjà, nous voyons que, d'une façon très générale, la structure d'un programme répond aux trois **call** indiqués ci-dessus, définissant trois étapes fondamentales :

Saisie des données
Exécution de l'algorithme.
Sortie des résultats

Nous allons nous intéresser, à présent, au second de ces points.

4.2.2. La traduction de l'algorithme.

A partir de cet exemple, nous pouvons aborder les problèmes soulevés par l'implantation d'un algorithme dans un programme.

a. L'articulation d'un programme.

Nous avons vu, au chapitre précédent le mécanisme offert par le processeur pour la réalisation de sous-programmes. Cette facilité est extrêmement importante car elle

permet d'articuler un programme selon la logique de l'algorithme mis en oeuvre. De ce point de vue, il nous paraît très important de chercher à concilier trois exigences qui se contrarient mutuellement : la transparence de fonctionnement, la lisibilité du programme et l'efficacité.

Revenons à notre exemple. Il peut paraître étrange de réduire ce programme de calcul en trois **call**. Pourtant, même dans ce cas très élémentaire, un tel découpage a son utilité : si on écrivait les programmes à l'endroit de chaque appel, on perdrait vite de vue cette articulation. En effet, le programme le plus court est, en l'occurrence, le programme mathématique. Le plus long, est l'affichage du résultat qui prend à lui seul plus de place que les deux autres programmes réunis.

> En effet, cet affichage se décompose en deux opérations : conversion en base dix puis affichage d'un nombre, la conversion dans le sens hexadécimal-décimal étant plus compliquée à programmer que la conversion inverse car elle est basée sur des divisions, opérations plus complexes que les multiplications de l'algorithme d'Hörner.

Le programme mathématique se trouverait donc noyé dans un problème d'entrée/sortie, qui, pour important qu'il soit, n'est pas l'essentiel dans ce programme. Du point de vue de l'efficacité, le découpage ajoute au temps d'exécution la durée de trois instructions **call** et de trois instructions **ret** : négligeable devant le temps que l'utilisateur met pour entrer ses données, une 'éternité' aux yeux du processeur...

Toujours du point de vue de l'efficacité, l'algorithme d'affichage n'est pas optimal : il décompose en deux temps séparés deux actions que l'on peut imbriquer, ce que nous ferons dans un programme du chapitre 6. Par contre, il est plus clair et partant, plus facile à vérifier.

Dans cette articulation, nous appellerons ***programme principal*** le programme qui contient la première instruction exécutée par le processeur après le chargement du programme. Il a la même structure que tout autre programme : des ***points d'entrée*** constitués par les adresses de la première instruction exécutée (pas nécessairement la même à chaque appel), un ***corps*** d'instruction et des ***points de sortie*** qui sont les adresses des instructions ***ret*** qu'il contient. Comme un programme constitue une unité du point de vue du contrôle de l'exécution, il nous paraît utile de faire apparaître cet aspect dans l'écriture du programme : un sous-programme ne doit pas être écrit au milieu des instructions du programme qui l'appelle, mais en dehors de ces instructions. Tant du point de vue de l'exécution que de celui du code, un programme doit être conçu comme un ***bloc*** possédant la propriété fondamentale suivante dite d'***absorption*** : si un bloc contient une instruction appartenant à un autre bloc, il contient aussi tout ce bloc. La structure des appels induit une ***hiérarchie*** des programmes à l'intérieur d'un même code respectant la propriété d'absorption : tout en haut, le programme principal et, tout en bas, les programmes qui ne contiennent pas d'autre bloc qu'eux-mêmes. Dans le programme correspondant à l'implantation de l'algorithme, cette hiérarchie doit refléter la structure de l'algorithme. Nous le verrons dans les exemples de structure plus complexe de la seconde partie de l'ouvrage.

b. Des ressources d'un programme.

De son point d'entrée jusqu'à ce qu'un **ret** soit rencontré, un programme contrôle entièrement le processeur, quel que soit sa place dans la hiérarchie du code soumis à la machine. Il dispose donc, en théorie de toutes les ressources de l'ordinateur. En réalité, il n'en va pas ainsi : d'une part, il n'a pas, le plus souvent, besoin de toutes ces ressources ; d'autre part, les autres programmes ont également besoin de ressources et souvent, un programme ne doit pas interférer avec les ressources d'un autre. Nous partirons du principe qu'un programme constitue une entité comprenant à la fois une partie du code bien délimitée, une partie également délimitée des ressources de

l'ordinateur que l'on appelle ***environnement du programme***, un moment précis dans l'exécution globale du code entier.

Prenons le programme de conversion d'hexadécimal en décimal. L'environnement de ce programme est constitué par :

les registres *AX,BX,CX,DX,SI,DI,BP*,
la ***variable*** de mots **[01BA]**
les ***données*** du tableau de conversion **[01B0]-[01BA]**,
la zone d'écriture des chiffres du résultat en base **010** : 4 octets depuis l'adresse **0212**.

Sauf mention expresse du contraire, les ressources d'un programme lui sont propres. Elles sont dites ***locales*** à ce programme. De plus, le statut des ressources locales peut être de deux sortes : ***temporaire*** si elles sont créées par le sous-programme au moment de son appel et détruites par lui lorsque le contrôle de l'exécution revient au programme appelant ; ***permanent*** si elles sont créées une fois pour toutes et retrouvées à chaque passage par le sous-programme.

Parmi les ressources, les ***variables*** jouent un rôle fondamental. Une variable est un octet ou un mot mémoire réservé par un programme pour des transferts d'information. La valeur d'une variable est a priori modifiable ce qui la différencie d'une ***constante*** dont la valeur est fixé une fois pour toute. Remarquons alors que réserver un emplacement ne lui confère aucune valeur particulière. Or, il y a toujours quelque chose à un endroit donné de la *RAM*. Comme cette valeur *brute* n'est pas nécessairement celle qu'attend le programme lorsqu'il en prend connaissance pour la première fois, il faut donc le faire au préalable : il faut ***initialiser*** la variable. Ce point est capital et tout oubli est sanctionné par un comportement inattendu du programme : différent, en général, pour des exécutions différentes avec les mêmes données. Il est quelquefois difficile de retrouver la variable fautive. C'est pourquoi le programmeur doit toujours faire l'inventaire exhaustif des ressources de chacun de ses programmes et vérifier, à cette occasion, l'initialisation des variables.

> Remarquons que la distinction introduite entre variables temporaires et permanentes peut conférer un statut particulier à certaines variables locales permanentes : elles peuvent jouer le rôle de constantes pendant l'exécution du programme *P* dont elles dépendent, tandis qu'elles seront réinitialisées par un autre programme *Q* dans l'intervalle qui sépare deux appels consécutifs de *P*.

Les ressources qui appartiennent en commun à plusieurs programmes sont dites ***partagées***. Celles qui sont partagées par le programme principal et des sous-programmes de celui-ci sont dites ***globales***. Cela dit, il ne faut pas confondre des ressources partagées avec des arguments de passage d'un programme à un autre. Ainsi, le programme d'affichage du résultat se décompose en fait en deux programmes : un programme de conversion et un programme d'affichage d'un nombre. Ces deux programmes partagent la zone **0212** sur laquelle le programme de conversion *écrit* et que le programme d'affichage *lit*. Quand le programme de conversion a fini d'écrire, il *passe* l'adresse du dernier octet écrit au programme d'affichage par le biais du registre *SI*, lui permettant ainsi de trouver le nombre d'octets à afficher.

c. Du passage des arguments

La décomposition en sous-programmes induit un problème complexe : le transfert d'information entre tous ces programmes. Examinons les différentes façons de le résoudre en assembleur.

Les programmes que nous avons vus jusqu'ici utilisent essentiellement un mode : le recours aux registres du processeur, ce qui est le mode de passage des

arguments pour ces programmes particuliers que constituent les *interruptions*. Deux autres modes sont possibles : l'utilisation de variables et l'emploi de la pile.

Par des variables.

L'utilisation des variables généralise le passage d'arguments par les registres. Il repose sur le même principe : placer par le programme appelant sur la ressource concernée du programme appelé la valeur que cette ressource doit prendre. Cette écriture sur l'appelé par l'appelant se produit, évidemment, *avant* l'appel. On peut modifier cette façon de procéder en plaçant la variable en 'terrain neutre' : si elle se trouve placée en un point du programme global qui n'est inclus dans aucun des blocs de code constitués par ces programmes particuliers, ceci revient à ériger la variable en ressource partagée n'appartenant ni à l'un ni à l'autre programme. Il peut alors convenir de pointer une partie de la zone commune. Cela peut se faire par un registre ou, à défaut, par une autre variable partagée d'adresse fixe.

Un lieu 'naturellement' partagé par beaucoup de sous-programmes d'un programme est évidemment la pile. D'où le rôle particulier qu'elle joue dans la programmation en assembleur. C'est le mode presque exclusivement utilisé par les compilateurs de langage évolué.

Nous décrivons à présent ce mode qui n'est pas nécessairement le mode le plus efficace, ni même le plus transparent...

Par la pile.

L'utilisation de la pile pour passer des arguments est facilitée par les instructions du processeur prévues à cet effet. Nous illustrerons abondamment ce mécanisme au chapitre 6. En voici le principe.

Si un programme *A* appelle un programme *B*, la première façon de procéder consiste à placer sur la pile, avant l'appel à *B*, les arguments que *A* doit lui passer. Comme après cet empilement, l'exécution de l'instruction **call** amène le processeur à placer sur la pile l'adresse de retour, le programme appelé ne peut pas directement obtenir les arguments voulus en dépilant. S'il veut dépiler, il doit placer d'abord l'adresse de retour sur une variable (ou, si c'est possible, un registre), puis dépiler les arguments avant de replacer l'adresse de retour sur la pile. C'est ce qui est indiqué par les instructions centrales de l'illustration ci-après.

Mais on peut procéder autrement : comme *BP* est associé au même registre de segment que *SP*, à savoir *SS*, on peut utiliser *BP*, initialisé par *SP*, pour aller 'chercher' les arguments de l'appelant sur la pile, et ceci, par les instructions de l'appelé. Cette méthode présente le gros avantage de ne pas modifier la pile, du moins, pas l'adresse de retour, de prendre les arguments au moment le plus opportun et dans l'ordre que l'on veut, mais présente l'inconvénient de prendre plus de temps du fait de la référence à une adresse indirecte. Cette méthode est illustrée par les instructions de droite ci-après :

A gauche, l'empilement des arguments avant l'appel au programme **0210**. Au centre, dans le programme **0210**, accès aux arguments par dépilement. A droite, une autre version du programme **0210** où l'accès aux arguments est effectué en utilisant le registre *BP*.

```
0100 PUSH AX            0210 POP  [0200]        0210 MOV  BP,SP
0101 PUSH SI            0214 POP  DI            0212 MOV  AX,[BP+06]
0102 PUSH DI            0215 POP  SI            0215 MOV  DI,[BP+02]
0103 CALL 0210          0216 POP  AX            0218 MOV  SI,[BP+04]
                        0217 PUSH [0200]
```

Cependant, le passage des arguments par la pile, s'il constitue une méthode simple et sûre, présente l'inconvénient de ne pas être transparent : dans la première méthode, on dépile à l'intérieur du programme appelé, de sorte que les **pop** qui équilibrent les **push** précédant l'appel n'apparaissent pas, sur le listage, après

l'instruction **call**. Cela peut se faire dans la seconde méthode, mais, en général, dans ce cas, on dépile 'd'un seul coup' en utilisant l'instruction **ret** avec un argument : dans l'exemple ci-dessus à droite, le programme **0210** se terminerait par l'instruction **ret 06**. Cette instruction équivaut à la séquence suivante :

```
0100 RET
0101 MOV  BP,SP
0103 ADD  BP,06
0104 MOV  SP,BP
```

Dans le cas où *n* arguments ont été empilés avant l'appel, on terminera par **ret** *m*, où $m = 2*n$. Ceci, présente le même inconvénient du point de vue de la transparence de fonctionnement. En revanche, la lisibilité des programmes peut s'en trouver augmentée.

d. L'articulation logique dans un bloc.

Les structures d'articulation logique d'un programme sont, classiquement, au nombre de trois : le branchement conditionné, la boucle et l'appel. Nous traiterons plus loin le cas des sous-programmes. Les deux premières structures se subdivisent en deux : le *IF* et le *CASE* pour le branchement, le *WHILE* et le *FOR* pour la boucle.

Rappelons que le *IF* définit une alternative et que le *CASE* généralise le *IF* à un nombre de choix fixe plus grand que deux. Le *WHILE* est la forme la plus générale de la boucle, le *FOR*, un cas particulier où le nombre de passages dans la boucle est fixé une fois pour toutes avant la première entrée dans la boucle.

IF et CASE.

Nous indiquons, ci-après, la traduction de cette notion en *PASCAL* et en assembleur :

```
IF Condition                    0100 CALL 0150
  THEN                          0103 JC   010A
     Instruction1               0105 CALL 0180
  ELSE                          0108 JMP  010D
     Instruction2               010A CALL 01B0
                                010D
```

On voit aussitôt que la traduction en assembleur suppose de nombreux sous-entendus. En particulier, que la condition est vérifiée par un sous-programme d'adresse **0150** (naturellement, il faudra vraisemblablement la redéfinir) dont le résultat : ***vrai*** ou ***faux***, est traduit par l'indicateur *CF*, la valeur *NC* étant associée à la réponse ***vrai***, la convention opposée pouvant fort bien être adoptée : seul le contexte peut en décider. Enfin, au cas où la condition et chaque branche se traduirait par un petit nombre d'instructions, on pourrait, si la clarté n'en était pas non plus compromise, remplacer les appels aux suites d'instructions concernées par ces séquences elles-mêmes.

Le *CASE*, quant à lui, proposant plusieurs choix, peut se traduire par une suite de *IF* emboîtés. Si *a*1,...,*ak* sont les valeurs d'une variable *c* définissant les différents choix possibles d'une variable, le *CASE* est équivalent à l'action suivante :

```
IF  a1
    THEN  I1
    ELSE  IF  a2
              THEN  I2
              ELSE  ...
                    IF  ak
                        THEN  Ik
                        ELSE J ;
```

où l'instruction *J* traite, par exemple, le cas où aucune des valeurs sélectionnées n'est atteinte. Lorsque le nombre des choix est petit, une telle suite d'emboîtements peut être

construite. Pour un plus grand nombre, le recours à un tableau et un **jmp** indirect est préférable. On fera en sorte que les différents choix soient associés à des valeurs croissantes et paires d'une certaine variable que le programme de sélection placera dans *SI*, les valeurs de *SI* allant de 00 à un certain maximum *M*. A partir d'une adresse *B*, on place les adresses d'appel de programmes traitant chacun des cas considérés, l'adresse placée en *B+SI* correspondant à la valeur *SI* sélectionnée. Ceci se réalise très simplement par les instructions ci-après :

```
0100 E84D00     CALL 0150       ; définition du choix, résultat dans SI
0103 83FE09     CMP  SI,+09     ; valeur admissible de SI?
0106 7218       JA   120        ; non: traitement de ce cas
0108 FFA41001   JMP  [0110+SI]  ; saut au programme de traitement ad hoc
010C

0110 DB         80 01 BA 01 E8 01 20 02 32 02 % adresses liées aux choix
011A
```

Nous verrons plusieurs exemples d'application de cette méthode dans les programmes que nous étudierons dans la seconde partie de l'ouvrage : en particulier, dans l'analyseur de syntaxe du programme décrit au chapitre 5.

Boucle WHILE, boucle FOR.

Les programmes donnés en exemple jusqu'ici nous ont offert plusieurs exemples de ces boucles. Pour la boucle *FOR*, nous avons tous les **loop** vus jusqu'ici. Pour la boucle *WHILE*, nous avons le programme de conversion de la base seize à la base dix. Dans ce dernier cas, la structure des instructions peut être résumée de la façon suivante :

```
0100 E84D00     CALL 0150       ; vérification de la condition
0103 7205       JC   010A       ; -> sortie de la boucle
0105 E87800     CALL 0180       ; exécution du corps de boucle
0108 EBF6       JMP  0100       ;
010A                            ; sortie de la boucle
```

Quant à la structure d'une boucle *FOR*, nous avons vu que c'est en fait celle d'un **loop**, à deux conditions expresses : le compteur de boucle est la variable *CX*, et la variable *CX* ne doit pas être modifiée par le corps de boucle. Enfin, signalons qu'un *FOR* du *PASCAL* n'est pas exécuté si le nombre de passages dans une boucle déterminé avant l'exécution de la première boucle est nul. De ce fait, en toute généralité, la traduction d'un *FOR* se présente comme suit :

```
0100 E84D00     CALL 0150       ; détermine CX et initialise
0103 E307       JCXZ 010C       ; sortie de boucle si CX = 0
0105:BOUCLE                     ;
0105 51         PUSH CX         ; sauvegarde du compteur
0106 E87700     CALL 0180       ; exécution du corps de boucle
0109 59         POP  CX         ; restauration du compteur
010A E2F4       LOOP 0105       ; retour à BOUCLE
010C                            ; sortie de la boucle
```

Naturellement, si l'on sait que *CX* n'est jamais nul, et seulement dans ce cas, l'instruction **jcxz** est inutile. De même, les **push** et **pop** sont inutiles si le corps de boucle ne modifie pas le registre *CX*.

On peut, évidemment, inclure le couple **push/pop** dans le programme **0180**. Pour la transparence du programme ***global***, le modèle indiqué ci-dessus nous paraît préférable. Par ailleurs, la présence d'un couple **push/pop** n'est pas une garantie absolue de préservation du registre *CX*. Le programme **0180** peut très bien faire des manipulations sur la pile à l'aide du registre *BP*... Au programmeur de prendre ses responsabilités.

REPEAT.

Si l'on sait que le corps de la boucle doit être exécuté au moins une fois, on peut placer le test de sortie après l'exécution du corps de boucle et, de ce fait, si la taille de ce corps dans le code le permet, on peut utiliser l'instruction **loop** pour contrôler la boucle. Cela revient à exprimer la condition de sortie sous la forme $CX = 0001$ puisque l'instruction **loop** fait passer à l'instruction qui la suit si et seulement si $CX = 0001$ au moment de l'exécuter. Dans cet emploi de **loop**, la valeur de *CX* est au contraire modifiée dans le corps de boucle puisque c'est elle qui conditionne la sortie, mais d'une façon qui n'est plus automatique. Cette façon de programmer une boucle peut présenter un intérêt. Bien que nous ne le recommandions pas, nous signalons cet emploi de **loop** pour souligner la grande flexibilité de l'assembleur.

4.2.3. Structuration des données et entrées/sorties.

La structuration des données est un des aspects de l'organisation des ressources d'un programme. D'une façon générale, il s'agit toujours de zones de mémoire affectées à un usage particulier. Une telle zone est repérée par l'adresse de son premier octet, sa taille, si elle est fixe, sa taille maximale et sa taille actuelle si elle est susceptible de varier, et un pointeur ou plusieurs pointeurs pour repérer des éléments de la zone. L'interprétation que peut recevoir cette forme de départ assez neutre conduit à un très grand nombre de structures. Les programmes que nous développerons dans la seconde partie de l'ouvrage mettent en oeuvre plusieurs types de structures que nous signalerons alors.

Dans les programmes donnés jusqu'ici, nous avons essentiellement rencontré le ***tableau*** et le ***tampon.***

Le tableau est une zone fixe ne contenant qu'un pointeur, appelé ***indice.*** Son exploitation se fait généralement dans une boucle **loop**. C'est ce que nous voyons dans le programme d'affichage d'un entier converti en base dix. Le tampon apparaît deux fois dans le programme des nombres de Fibonacci : un tampon de lecture traité par la fonction *DOS* **09** et un tampon d'écriture créé par le programme. C'est une sorte de modèle réduit du *fichier* que nous avons étudié au chapitre précédent.

Le modèle du fichier occupe une place très importante pour les entrées/sorties, car d'une certaine manière, toute sortie, même l'écriture à l'écran, peut être considérée comme une écriture dans un fichier. La caractéristique du fichier, que partage le tampon : voir le programme de Fibonacci, est son caractère *dynamique* et son traitement par *séquences*. Sa taille peut varier (à l'intérieur de limites fixes). Lorsqu'on agit dessus, on le fait, en général, pour une suite d'octets, la *séquence*, l'ensemble des opérations se partageant en deux catégories : lecture et écriture.

En lecture, le pointeur du premier octet à lire est toujours situé *avant* les données à traiter. En écriture, il est toujours situé après le dernier octet traité. En lecture, on se préoccupe, à chaque pas, de savoir si la fin du fichier est atteinte. En écriture, où l'on est, en général, à la fin du fichier, on se préoccupe de savoir si la *limite* à la taille maximale du fichier est atteinte. Comme la vérification, pour chaque octet traité, du non dépassement des limites ralentirait considérablement les opérations, on a intérêt, autant que faire se peut, à les insérer dans des boucles **loop** *sûres* : on sait que le nombre de passages dans la boucle ne fera pas sortir de ces limites.

Cette conception a de larges applications pratiques. *DOS* la met en oeuvre avec la notion de redirection que nous exploitons avec *DEBUG*. Cela s'applique en particulier aux fonctions **040** et **03F** de lecture et d'écriture sur un fichier. Pour afficher un message à l'écran, on peut tout aussi bien utiliser la fonction de *DOS* **040**. Il suffit d'indiquer le sésame associé à l'écran : **02**. Tous les périphériques de la machine sont

en fait associés à un sésame ***prédéfini***. Il convient de remarquer qu'il ne faut pas ouvrir un tel sésame, car la fonction d'ouverture affecte un sésame au fichier ouvert. On peut par contre le fermer, mais c'est à manipuler avec précaution : si d'aventure on ferme le clavier, on ne peut plus s'en servir tant qu'il reste fermé...

L'avantage de cette méthode, pour écrire à l'écran, est sa très grande portabilité. C'est elle, par exemple, que *DEBUG* utilise quand il dialogue avec nous. Elle présente deux inconvénients : pour des présentations plus élaborées, il faut déplacer le pointeur du fichier, ce qui suppose de faire la correspondance entre les coordonnées écran, qui sont à deux dimensions et la position dans le fichier qui est une abscisse : donc à une dimension. Ce n'est pas difficile, mais ce n'est pas très commode. L'autre inconvénient est la lenteur de cette méthode. Elle ne se remarque pas lorsqu'il s'agit de lire ce que *DEBUG* affiche, mais elle est très perceptible si l'on affiche de cette façon le contenu d'un fichier. C'est pourquoi on lui préfère souvent l'écriture directe sur la *RAM* vidéo par les instructions usuelles de l'assembleur, **movs** et autres. Si cette méthode pose le même problème de représentation linéaire d'un espace à deux dimensions, la programmation en est toutefois plus simple et la rapidité obtenue en vaut la peine. Quant à la portabilité, théoriquement moins grande, elle n'est pas si réduite : l'écriture directe en *RAM* vidéo est simple pour le mode texte de tous les écrans et ne pose de problèmes que pour des machines *anciennes*. En effet, si on peut écrire directement sur les cartes vidéo de ces machines, le résultat n'est pas très bon : il se produit un effet de neige qui rend la lecture moins confortable. Au paragraphe suivant, nous donnons un exemple qui fournit les principaux outils de cette méthode.

Par contre, nous recommandons vivement l'utilisation des sésames prédéfinis pour l'écriture sur l'imprimante. Il s'agit de **04** pour la sortie parallèle. C'est le moyen le plus sûr de communiquer avec ces machines fort capricieuses.

4.2.4. Assemblage de programmes sous DEBUG.

Il convient à présent d'indiquer comment procéder à la confection de programmes *.COM* éventuellement complexes sous *DEBUG*.

Nous avons vu comment utiliser la redirection pour obtenir l'assemblage d'un code. Rappelons qu'il suffit d'écrire les commandes de *DEBUG* dans un fichier *ASCII* ordinaire. Après une commande *a*, on insère les instructions en assembleur, en utilisant librement le symbole ; pour séparer les commentaires du code et mettre en place des éléments de présentation permettant d'améliorer la lisibilité. Dans le fichier but donné à *DEBUG* on obtient l'assemblage des instructions affiché de façon symbolique mais avec l'adressage exact des instructions. Voici en regard l'un de l'autre (sans commentaires) l'exemple du programme **ptfixe.csm** du début de ce chapitre :

```
a
;  programme ptfixe
;
     mov  cx,0014
     mov  si,0100
;  106 :
     mov       dl,[si]
     mov       ah,02
     int       21
     inc       si
     loop      106
;  DOS :
     mov       ax,4C00
     int       21
```

```
13CE:0100 ;  programme ptfixe
13CE:0100 ;
13CE:0100      mov  cx,0014
13CE:0103      mov  si,0100
13CE:0106 ;  106 :
13CE:0106      mov  dl,[si]
13CE:0108      mov  ah,02
13CE:010A      int  21
13CE:010C      inc  si
13CE:010D      loop 106
13CE:010F ;  DOS :
13CE:010F      mov  ax,4C00
13CE:0112      int  21
13CE:0114
```

La commande *u* permet d'obtenir, sans les commentaires, les listages sous la forme donnée dans cet ouvrage. La commande *n* permet de donner un nom au fichier contenant le code (pas nécessairement *.COM*), la commande *w*, après avoir initialisé *BX*:*CX* comme il convient, permet d'écrire le fichier sur le disque, dans le répertoire depuis lequel on a appelé *DEBUG*.

Si les trois programmes appelés aux adresses **120**, **150** et **1D8** ci-dessus sont assemblés sous forme binaire dans les fichiers **entree.spm**, **fibo.spm** et **sortie.spm**, on obtient le code binaire d'un programme de calcul des nombres de Fibonacci en les assemblant sous *DEBUG*, à l'aide des commandes *l* et *n*, par le fichier suivant :

```
a;
     call 120                corrections d'adresses pour 0120
     call 150                e 121 08 02
     call 1D8                e 128 18 01
;                            e 132 19 01
     mov  ax,4C00            e 13A 1A 01
     int  21
                             corrections d'adresses pour 01B0
nentree.spm                  e 181 7A 01
l 118                        e 186 7A 01
nfibo.spm                    e 189 7C 01
l 150                        e 18E 7C 01
nsortie.spm                  e 1A4 7A 01
l 1B0                        e 1A9 7C 01
r cx                         e 1AF E4 01
150                          e 1BA 70 01
r bx
0
nfibo.com
w
```

Dans le cas considéré, le programme **01D8** (le point d'entrée est en **01D8**, mais le programme commence en **01B0**) contient une variable repérée par rapport à l'adresse **0100** dans le fichier **sortie.spm**. Pour ne pas écrire sur une partie du code, on corrige ces adresses pour que les bonnes valeurs figurent dans le fichier **fibo.com**. On le fait par des ordres *e* adéquats, indiqués ci-dessus à droite du listage (dans le fichier soumis à *DEBUG* ces commandes s'intercalent entre *l* **1B0** et *r cx*). La détermination des adresses se décompose en deux phases : on repère l'adresse relative dans le fichier **.bsm** obtenu avec la confection du programme **.spm** et on ajoute l'adresse de départ indiquée dans le fichier ci-dessus. Il n'y a pas de correction à faire pour les instructions **jmp** ou *jcond* d'après ce que nous avons vu au chapitre 3. Par contre, il faut le faire pour les **call**, sauf si les programmes concernés ont été 'compilés' dans un même fichier. Nous approfondirons cette technique dans un cadre plus complexe aux chapitres 5 et 6.

4.3. La récursivité en assembleur.

Un programme est dit ***récursif*** lorsqu'il contient au moins un appel à lui-même. Cette appellation est étroitement liée aux fonctions récursives que nous avons étudiées au chapitre 2. Nous retiendrons, de ce mode d'écriture des programmes est qu'il revêt deux aspects très différents. Un aspect *existentiel* : il existe un objet satisfaisant à l'équation que constitue cette définition ; un aspect *effectif* : l'objet à définir est l'aboutissement d'un *processus* dont chaque étape s'exécute en fonction du résultat *à venir* des étapes précédentes. Nous commençons par deux exemples.

4.3.1. Exemples.

a. Factorielle.

Le calcul de *n*! est l'exemple ultra-classique de la récursivité. Rappelons la définition de cette fonction :

0! = 1,
(S*n*)! = S*n*.(*n*!), où S*n* est le suivant de *n*.

L'hypothèse de récurrence d'une solution connue à l'ordre *n*-1, mais non encore calculée, est extrêmement féconde. Elle fournit un outil de programmation très puissant dont l'intérêt premier réside dans le fait qu'en général, cette méthode contient la démonstration de sa correction. Elle constitue aussi un moyen de calcul : la seconde équation s'applique au 'cas général', la première étant réservée au cas particulier de la valeur 0. Le programme suivant permet de nous en convaincre :

```
0100 31C0          XOR  AX, AX    ; CX = n
0102 40            INC  AX        ; AX initialisé à 1
0103 E306          JCXZ 010B      ; si n = 0 : -> fin
0105 F7E1          MUL  CX        ; sinon : AX := CX*AX
0107 49            DEC  CX        ; n := n-1
0108 E8F8FF        CALL 0103      ; remultiplier
010B C3            RET            ;
0110 E8EDFF        CALL 0100      ; appel du programme récursif
```

Si on exécute ce programme, on observera, une fois n! trouvé une suite de dépilements correspondant aux empilements de l'adresse de retour provoqués par les appels successifs du programme **0103**.

Ces appels sont parfaitement inutiles : l'instruction **jmp 103** à l'octet **0108** conduirait au même résultat (corriger **jcxz 10B** en **jcxz 10A**) et plus rapidement. Nous verrons que cette observation sur la programmation récursive n'est pas propre à n! mais à une grande partie, sinon toute, du domaine récursif, ce que nous verrons au second paragraphe.

Il existe cependant de nombreux problèmes où la récursivité fournit le moyen le plus efficace d'en programmer la solution. Le célèbre problème des tours de Hanoï en est une illustration.

b. Les tours de Hanoï.

Suite au célèbre canular du mathématicien Lucas, de nombreux 'récits de voyageurs' retour d'Orient, rapportaient, à la fin du XIX*ème* siècle, une curieuse pratique des prêtres du temple de Brâhma : soixante quatre disques d'or percés en leur milieu sont enfilés sur des aiguilles serties de pierres précieuses ; les disques sont tous de tailles différentes et chaque jour, les prêtres déplacent un disque d'une aiguille sur une autre ne contenant que des disques *plus grands* ; commencé depuis des temps immémoriaux, répétait-on après Lucas, le jeu annoncera la fin du monde par son achèvement.

Reformulons la règle du jeu :
On dispose de n disques, percés en leur milieu, de diamètres distincts, et de trois plots numérotés 1, 2 et 3. Au départ, tous les disques sont rangés sur le plot 1, par ordre de tailles décroissantes, le plus grand disque en bas et le plus petit en haut. On ne peut déplacer qu'un disque à la fois. Quand on effectue cette action, on ne peut poser le disque que sur un plot sans disque ou sur un plot dont le disque supérieur a un diamètre supérieur au sien. Le but du jeu est d'amener tous les disques sur le plot 3 dans le même ordre qu'au départ, sur le plot 1.

Désignons par $a \rightarrow b$ l'action consistant à prendre le disque supérieur du plot a pour le poser sur le disque supérieur du plot b.

Supposons, par récurrence sur n que nous sachions résoudre le problème pour n disques. Si on a n+1 disques, on se 'ramène' au cas n de la façon suivante : soit $N \cup i$ les n+1 disques en 1, i étant le plus grand ; par hypothèse de récurrence, on sait amener N en 2, 3 étant le plot intermédiaire sans déplacer i puisque i est le disque le plus grand ; on peut alors placer i en 3, puis, pour la même raison, et toujours par hypothèse de récurrence, on peut placer les N disques de 2 sur 3, le plot intermédiaire étant 1.

On observe que ce raisonnement contient le cas $n = 2$ dont la solution peut s'écrire $1 \rightarrow 2$, $1 \rightarrow 3$, $2 \rightarrow 3$.

Si d_n est le nombre des déplacements de disques effectués selon cet algorithme pour n disques, on démontre aisément que $d_n = 2^n - 1$.

Programmons cet algorithme. Si **0100** désigne l'adresse d'un programme résolvant le problème et **0110** l'adresse d'un programme écrivant l'action $a \rightarrow b$, on peut écrire :

```
0100 E8FDFF   CALL 0100          ; déplacer N de 1 sur 2
0103 E80A00   CALL 0110          ; effectuer 1 -> 3
0106 E8F7FF   CALL 0100          ; déplacer N de 2 sur 3
0109 C3       RET                ; retour
```

Ce programme ne fait évidemment pas ce qu'on en attend : la différence entre l'appel de l'instruction `0100` et celui de l'instruction `0106` n'apparaît que dans les commentaires. Mais ce n'est pas tout. Exécutons ce 'programme'. On obtient :

```
 SP   IP
FFFE 0100  CALL 0100
FFFC 0100  CALL 0100
FFFA 0100  CALL 0100
```

et ainsi de suite jusqu'à ce que la pile descende à l'adresse **0102**, écrasant le code et provoquant vraisemblablement l'arrêt du système.

Pour remédier à ce défaut, il suffit de se rappeler ce que nous avons dit sur l'environnement des programmes. Le programme **0100** demande des informations complémentaires : N, le nombre de disques, a, le plot de départ, b, le plot intermédiaire et c, le plot d'arrivée, ce que nous placerons, respectivement, dans *AL*, *AH*, *CL*, *CH*. Ces registres définiront donc le ***contexte*** dans lequel travaille le programme **0100**. Lorsqu'on appelle le programme **0100**, on lui passe donc un certain contexte, par exemple, par la pile. Il appelle à nouveau le programme **0100**, mais dans un autre contexte. Quand ce second appel aura été exécuté, on se retrouve dans le même programme mais dans le contexte du premier appel et, comme l'adresse de retour a été sauvegardée sur la pile, à l'adresse de l'instruction qui suit l'appel qui vient de s'achever. Il faut donc, à ce moment-là retrouver le contexte du premier appel. Nous voyons donc, que la pile doit stocker non seulement l'adresse de retour à l'instruction suivante, mais aussi le contexte à retrouver, lors de ce retour. Nous en profitons pour insérer dans notre programme une instruction de contrôle du débordement de la pile. D'où le programme ci-contre que nous appellerons *programme Hanoï* dans ce qui suit.

c. Digression graphique : programme de Hanoï animé.

Nous proposons à présent au lecteur une illustration des techniques indiquées au paragraphe précédent, notamment l'organisation des ressources d'un programme : il s'agit d'écrire un programme réalisant une animation graphique de ce jeu. Ce programme aura par ailleurs un aspect conversationnel car il permettra à l'utilisateur, en

Rappel, en entrée : $AL = N$, $AH = a$, $CL = b$, $CH = c$

```
0102 3C00          CMP  AL,00          ; N = 0?
0104 7702          JA   0108           ; non : suite
0106 C3            RET                 ; si : retour

0108 89E3          MOV  BX,SP          ;
010A 3B1E0001      CMP  BX,[0100]      ;
010E 761A          JBE  012A           ;
0110 50            PUSH AX             ; sauver le contexte de la
0109 51            PUSH CX             ; présente exécution
0112 86E9          XCHG CH,CL          ; initialiser le contexte du
0114 FEC8          DEC  AL             ; 1er appel
0116 E8E9FF        CALL 0102           ; APPEL pour permutation b,c
0119 59            POP  CX             ; restauration du contexte
011A 58            POP  AX             ; de la présente exécution
011B FFD6          CALL SI             ; DEPLACEMENT a -> b
011D 50            PUSH AX             ; sauver le contexte actuel
011E 51            PUSH CX             ;
011F 86E1          XCHG AH,CL          ; initialisation du contexte
0121 FEC8          DEC  AL             ; du second appel
0123 E8DCFF        CALL 0102           ; APPEL pour permutation a,b
0126 59            POP  CX             ; restauration du contexte de
0127 58            POP  AX             ; la présente exécution
0128 C3            RET                 ; fin
0129 90            NOP                 ;
```

Compte tenu des notations, le contexte passé au premier appel est N,a,c,b, ce qui correspond au premier déplacement selon notre algorithme ; le contexte passé au second appel correspond à N,b,a,c puisque le contexte initial, a,b,c a été rétabli depuis la pile. Noter l'appel par un registre du programme $a \rightarrow b$.

Le saut depuis **010E** renvoie à l'affichage d'un message 'débordement de la pile'.

L'exécution de ce programme pour $N = 3$ donne le résultat ci-dessous, après insertion dans le programme du premier appel à **0102** d'instructions adéquates pour l'affichage d'un message.

```
Entrer le nombre de disques
3
1 -> 3
1 -> 2
3 -> 2
1 -> 3
2 -> 1
2 -> 3
1 -> 3
```

actionnant les curseurs de modifier la vitesse d'exécution du programme ou de faire un 'arrêt image' s'il le désire.

Articulation du programme d'animation.

Faute de place, nous ne donnons, en annexe 8, que des extraits du texte complet du programme. Nous ne pouvons présenter ici que les séquences d'instruction les plus marquantes. Et tout d'abord, le plan du programme :

```
CALL  0128          ; lecture des données
CALL  01E0          ; initialisation graphique
CALL  0160          ; programme Hanoï proprement dit
```

Le programme **0128** est le programme de lecture du nombre N que nous avons vu au paragraphe précédent. Pour des raisons évidentes de durée et de représentation, la valeur de N est limitée à 10. Le programme **0160** comporte un appel récursif au programme Hanoï. Ce qui relie ce programme au 'graphisme' mis en place par le

programme **01E0** est le programme **call si** du programme Hanoï soit ici, le programme **0580**. Ce dernier, en effet, consiste à effacer le disque supérieur du plot de départ pour le replacer sur le plot d'arrivée.

Le programme **01E0** affiche la première image, l'état initial du jeu de Hanoï en même temps qu'il met en place la structure du contrôle ultérieur de cette image qui consiste en une table reproduisant, d'une certaine façon, l'image de l'écran. On peut la représenter de la façon suivante :

```
          BH   I  d0  d1  d2  d3  d4  d5  d6  d7  d8  d9
DB        0B  02  04  03  02  00  00  00  00  00  00  00
DB        0B  02  07  06  01  00  00  00  00  00  00  00
DB        0C  01  08  05  00  00  00  00  00  00  00  00
```

Chaque ligne de ce tableau représente l'état d'un plot : de haut en bas, les plots 1, 2, 3. Le premier octet, marqué *BH*, car il donne le contenu de ce registre lorsque (*BL*,*BH*) représente les coordonnées d'un point de l'écran sous la forme (colonne,ligne), sachant qu'un écran standard comporte 80 colonnes et 25 lignes et que (0,0) désigne le coin supérieur gauche. Le second octet donne l'adresse depuis d_0 (=00) du disque supérieur sur ce plot. A partir de d_0, l'identification de chaque disque du plot par un numéro proportionnel à son diamètre, d_0 contenant le numéro du disque inférieur, d_1 celle du disque placé immédiatement au-dessus, et ainsi de suite. Sur la figure ci-dessus, la table indique huit disques se répartissant ainsi :

de bas en haut, les disques 4, 3 et 2 sur le plot 1,
de bas en haut, les disques 7, 6 et 1 sur le plot 2,
de bas en haut, les disques 8 et 5 sur le plot 3.

Le programme **01E0** effectue les actions suivantes :

choix du mode écran (**0200**), effectué entre le mode **07** et le mode **02** selon que l'octet **0040:0063** contient **B4** (carte monochrome en service) ou **D4** (carte couleur) ; dans le premier cas, la *RAM* vidéo se situe à l'adresse de paragraphe **0B000**, dans le second, à **0B800** ;
initialisation des paramètres graphiques (**0270**), dont ceux de la table décrite ci-dessus qui occupe la zone **0240-026E** ;
tracé de la première image : le trait horizontal, support commun aux trois plots (**02B8**) ; les disques du plot n°1 (**0300**) ; les plots n°2 et n°3 ;
point d'arrêt pour définir via le clavier (**0468**) la suite du programme.

Le programme **0580** effectue le déplacement d'un disque de la façon suivante :

il commence par sauver les registres *AX*, *CX* (qui contiennent le contexte d'appel au programme récursif) et les registres *DS* et *SI* ;
puis, il initialise ses variables (zone **06A0-06AE**) ;
il prépare ensuite les variables du programme d'effacement du disque supérieur (**0394**) ;
après l'effacement du disque, il corrige le dessin à l'intersection du plot avec le nouveau disque supérieur (ou le plateau) et réactualise la table **0240** ;
ensuite, il dessine un disque 'égal' au disque déplacé sur le disque supérieur (ou le plateau) du plot de destination (**0358**) après avoir préparé les arguments à passer à ce programme ;
il corrige le nouveau dessin à l'intersection du nouveau disque avec le plot et, la tâche étant terminée, interroge le clavier (**06B0**).

Le passage des arguments entre programmes intérieurs au bloc **0580** et ce programme se font par les registres, le registre *ES* étant réservé pour donner l'adresse de segment de la *RAM* vidéo. Ainsi, pour dessiner ou effacer un disque, **0580** passe au

programme **0358** ou **0394** les registres *BX*, *DX* et *AH* contenant, respectivement, les coordonnées (*BL*,*BH*) du coin supérieur gauche du disque représenté par un rectangle de largeur deux lignes, la largeur du disque (en nombre de colonnes), l'attribut vidéo du trait à dessiner, ce que nous expliquons à présent.

Les programmes reçoivent aussi dans *SI* l'adresse d'appel d'un programme de recalcul des adresses : les sous-programmes constituant le programme d'animation ont été compilés sous *DEBUG* suivant la méthode que nous avons donnée et il commencent, habituellement, à l'adresse **0100**. Pour éviter de recalculer les adresses de variables, une solution consiste à corriger le registre *DS* de la translation correspondant au point d'entrée réel du sous-programme dans le fichier définitif. Comme on se situe dans un *.COM*, il n'intervient que des déplacements par rapport au numéro de paragraphe du préfixe et la correction est faite par rapport à cette adresse de segment. La seule condition est que l'adresse du point d'entrée (ou plutôt du premier octet du bloc) soit une adresse de paragraphe, c'est-à-dire se terminant, en hexadécimal, par 0. Dans cette optique, on résout le problème des **call** en les écrivant sous la forme **call RZ** où *RZ* est un registre dont la valeur est fournie au moment de l'appel du bloc.

Cette méthode reste d'un maniement simple pour des programmes dont la taille ne dépasse pas 2 Ko. Au-delà, elle mobilise trop de registres, utilisés à d'autres fins.

Dessiner sur la RAM vidéo.

Lorsqu'on écrit sur la RAM vidéo, on modifie aussitôt l'aspect de l'écran. En effet, en mode texte, chaque ***mot*** de la RAM vidéo est mis en correspondance avec un point de l'écran défini par ses coordonnées (colonne,ligne), les numéros de colonne variant de **000** à **04F** et les numéros de lignes, de **000** à **018** ; le point de coordonnées (0,0) étant le coin supérieur gauche de l'écran.

Un point est le lieu d'affichage d'un caractère figurant parmi les 256 du jeu *ASCII* habituel. Chaque caractère est un *dessin* défini par les utilitaires du constructeur de la machine, le dessin étant décrit par une matrice 8*8 qui se trouve à l'entrée d'une table installée en *ROM*.

Cette correspondance s'effectue de la façon suivante : le mot d'adresse **0000** en *RAM* vidéo définit le point (0,0) ; le mot suivant définit le point contigu à droite sur la même ligne et ainsi de suite sur la ligne ; le mot associé au dernier point à droite d'une ligne a pour suivant le mot associé au premier point à gauche de la ligne suivante.

Le mot associé au point définit non seulement le caractère écrit, puisqu'un octet suffit pour cela, mais aussi un certain nombres de paramètres vidéo qui constituent, par définition, l'***attribut vidéo*** du point. L'attribut est l'octet fort du mot associé au point et l'octet faible contient le code *ASCII* du caractère à afficher. Dans l'octet de l'attribut, le quartet faible définit la couleur du caractère et le quartet fort, la couleur du fond et la permanence.

En noir et blanc, les couleurs sont 0 pour le noir et 7 pour le blanc. Ainsi, le mode habituel blanc sur noir a pour attribut **07** et le mode inverse **70**. Le bit 3 définit la luminosité : 0 pour l'intensité normale, 1 pour la surbrillance. Le bit 7 indique le mode d'affichage : 0 pour permanent, 1 pour clignotant.

Les écrans couleurs peuvent définir, de cette façon, seize couleurs (seize nuances de gris pour les écrans monochromes *EGA* ou *VGA*), seules les huit premières étant disponibles pour le fond puisque le bit 7 est réservé pour le clignotement.

Le lecteur peut vérifier tout cela sous *DEBUG* avec lequel on peut lire la *RAM* vidéo et y écrire pour vérifier les points indiqués ci-dessus.

Les disques étant des rectangles empilés les uns sur les autres par diamètres décroissants en allant de bas en haut, il suffit en fait de dessiner trois des côtés du rectangle : les deux côtés latéraux et le côté supérieur. Ce dernier utilise le caractère **0C4** et les deux autres côtés, le caractère **0B3**. Le coin supérieur gauche est tracé par le

caractère **0DA**, le droit par **0BF**. Le programme de tracé et le programme d'effacement sont en fait presque identiques : pour effacer, on redessine avec le symbole blanc.

> On pourrait également repasser sur le dessin avec l'attribut **00**, c'est-à-dire convertir ce qui est écrit blanc sur noir en l'écriture du même symbole noir sur noir. La solution choisie dans le programme fournit une image qui reste correcte si on se restreint aux seuls octets faibles.

Le tracé d'une ligne horizontale s'effectue par un **stosw** avec *AX* = **07C4**, une fois déterminé son point de départ. Il faut donc convertir les coordonnées écran (*BL*,*BH*) en l'adresse *RAM* vidéo du mot correspondant. C'est ce que fait le programme **0440** :

```
0440 50          PUSH AX          ; on sauve les variables
0441 51          PUSH CX          ; contenant le contexte
0442 891E1F01    MOV  [011F],BX   ; [011F] := BL, [0120] := BH
0446 31C0        XOR  AX,AX       ;
0448 B0A0        MOV  AL,A0       ; AX := 0A0 (= 80 mots)
044A 8A0E2001    MOV  CL,[0120]   ; CL := nombre de lignes
044E F6E1        MUL  CL          ; AX := dépl. début ligne ad hoc
0450 8A0E1F01    MOV  CL,[011F]   ;
0454 30ED        XOR  CH,CH       ;
0456 00C9        ADD  CL,CL       ; CX := 2*nbre de colonnes
0458 01C8        ADD  AX,CX       ; AX := adresse
045A 89C3        MOV  BX,AX       ; placée dans BX
045C 59          POP  CX          ; restauration
045D 58          POP  AX          ; du contexte
045E C3          RET              ;
```

Dialogue par le clavier.

Ceci repose sur l'appel aux fonctions **00** et **01** de l'interruption **016**. Les instructions **06BE-06CB** du programme **06B0** montrent comment les employer :

```
06BE B401        MOV  AH,01       ; un caractère a-t-il été lu?
06C0 CD16        INT  16          ;
06C2 7501        JNZ  06C5        ; oui : -> le lire
06C4 C3          RET              ; non : retour, donc suite
06C5 B400        MOV  AH,00       ; lire le caractère lu
06C7 CD16        INT  16          ;
06C9 3C00        CMP  AL,00       ; début d'analyse du caractère :
06CB                              ; touche spéciale?
```

Cependant, comme l'exécution par le processeur du programme de tracé d'un disque est très rapide, pour que l'animation soit *visible*, il faut introduire un délai d'attente. C'est ce que le programme **06B0** commence par faire :

```
06B0 B98000      MOV  CX,0080     ; cadence initiale
06B3:BOUCLE1                      ; modulable
06B3 51          PUSH CX          ; compteur sauvé
06B4 B90010      MOV  CX,1000     ;
06B7:BOUCLE2                      ; fixe : 12288 cycles
06B7 90          NOP              ;
06B8 E2FD        LOOP 06B7        ; retour à BOUCLE2
06BA 59          POP  CX          ; compteur retrouvé
06BB 90          NOP              ;
06BC E2F5        LOOP 06B3        ; retour à BOUCLE1
```

La boucle intérieure, fixée à 12288 cycles, est invariable. On peut agir sur le délai d'attente en modifiant la donnée immédiate placée dans *CX* par l'instruction **06B0**. Chaque action sur la touche → accélère le délai d'un facteur 2, chaque action sur la touche ← le ralentit par addition de **040**. On contrôle l'accélération afin de ne jamais mettre **0000** dans *CX*. Le choix est fixé par des tests successifs sur la valeur de *AH* lue par la fonction **00** de l'interruption **016**. Comme les valeurs possibles sont 'dispersées',

ces tests ne sont pas effectués sur le modèle *CASE* du paragraphe 4.2.3., mais sur celui de *IF* emboîtés. Le lecteur peut aisément reconstruire lui-même ces instructions.

c. La fonction d'Ackermann.

Cette fonction apparaît dans la théorie des fonctions récursives comme l'exemple désormais classique[1] d'une fonction récursive totale qui n'est pas primitive récursive. De ce fait, cette fonction présente un *intérêt pratique* peu exploité, semble-t-il. En effet, le calcul de cette fonction étant *intrinsèquement* complexe, son temps de calcul est suffisamment long pour qu'on puisse en mesurer, de façon significative, les temps d'exécution de différentes implantations.

Cette fonction est définie par les équations suivantes :

$$\psi(0,y) = Sy,$$
$$\psi(Sx,0) = \psi(x,1),$$
$$\psi(Sx,Sy) = \psi(x,\psi(Sx,y)).$$

Un calcul simple, par récurrence sur n nous donne les formules suivantes, utiles pour la vérification du programme :

$$\psi(1,n) = n + 2,$$
$$\psi(2,n) = 2n + 3,$$
$$\psi(3,n) = 2^{n+3} - 3.$$

Comme nous le verrons, il est exclu de vérifier $\psi(4,1) = 2^{16} - 3$ avec les programmes que nous ferons, et cela l'est davantage pour $\psi(4,2) = \psi(3,\psi(4,1))$.

En appliquant la méthode utilisée pour programmer en assembleur le jeu des tours de Hanoï, on obtient aisément le programme donné à la page 93 (avec une numérotation des instructions ramenée depuis **0100**) et que nous désignerons dans ce qui suit par ***programme Ackermann***. C'est un programme analogue que fournissent la plupart des compilateurs de langage évolué.

On peut le tester directement sous *DEBUG* en utilisant la commande *r* pour initialiser les registres *CX* et *DX* et la commande *g* pour l'exécution. On peut également l'insérer dans un programme comportant une lecture de *X* et *Y* à l'écran et d'affichage du résultat. Le programme donné en annexe 3 fonctionne avec une implantation différente de cette même fonction, mais comportant la mesure de son temps de calcul. En remplaçant, dans l'annexe 3 le programme **0320** par le programme Ackermann, on obtiendra, en plus la durée d'exécution de ce dernier.

La mesure du temps d'exécution est obtenue par le programme que nous avons vu au début de ce chapitre. Nous renvoyons les précisions sur l'obtention de l'heure au paragraphe suivant où les mesures que nous obtiendrons nous permettront de comparer entre elles diverses méthodes de programmation.

4.3.2. La dérécursification.

Retour à l'exécution des programmes : blocs et récursivité.

Nous avons vu, au paragraphe 4.2.2. l'articulation des programmes en blocs unifiant la syntaxe du code lui-même et le déroulement de l'exécution. Ce schéma ne

---

[1] La fonction construite par Ackermann est plus complexe. Cette version en est une simplification apportée par Rózsa Péter.

s'applique pas, au premier regard, dans le cas de programmes récursifs. Cependant, si on ajoute à la notion de programme et de bloc l'environnement dans lequel il fonctionne, un même bloc *syntaxique* se décompose en plusieurs blocs *exécutants* distincts car d'environnement différent.

On peut interpréter ces blocs exécutants comme des ***copies*** fictives du même bloc syntaxique : en effet, lorsque, par exemple, le programme Hanoï s'appelle lui-même, tout se passe comme si ce programme appelait une copie de lui-même se trouvant à un autre endroit du code. On peut, d'ailleurs envisager le processus suivant : lorsque le programme *A* s'appelle lui-même, l'exécutif commencerait par installer une copie de *A* dans une zone libre de la mémoire puis transférerait le contrôle de l'exécution à cette copie après lui avoir passé les arguments redéfinissant le contexte.

Nous l'avons vu dans le cas du programme Hanoï. Il se produit ici la même chose, qui explique les sauvegardes de contexte sur pile du programme Ackermann.

Récursivité croisée.

Un contexte plus général de récursivité, que nous n'aborderons pas, faute de place, dans cet ouvrage, est celui de la récursivité croisée. C'est celle que l'on obtient lorsque, par exemple deux programmes *A* et *B* s'appellent mutuellement. La distinction d'environnement (dans laquelle on peut inclure, abstraitement, l'heure de l'appel) permet de replacer ce cas dans la problématique des blocs unités syntactiques et d'exécution. Le cas de deux programmes se généralise à un nombre quelconque s'appelant de façon cyclique, par exemple. Les conclusions que nous tirerons, au sujet de la dérécursification, s'appliquent aussi au cas de la récursivité croisée.

Récursivité terminale.

Le cas de n!

Si on se reporte au programme de n!, on observera que l'appel récursif du programme de la page 85 n'est pas 'encadré' par les sauvegardes et restitutions de contexte portés dans le cas des programmes de Hanoï et d'Ackermann. Dans le cas de n!, nous avions en fait observé que l'appel récursif lui même était inutile. Si nous revenons sur cette analyse, nous constaterons que les appels récursif n'avaient servi à rien car, après l'instruction d'appel, on revenait aux appels précédents. D'une façon général, dans un programme récursif, un appel est dit ***terminal*** si l'action qui suit immédiatement cet appel est un retour.

Le programme Ackermann.

Reprenons cette réflexion dans le cas du programme Ackermann.

Ce programme comporte deux appels récursifs terminaux : l'appel de l'instruction **0121** et l'appel de l'instruction **0143**. En effet, les deux **pop** qui suivent chacune de ces instructions n'ont pas d'autre fonction que de restaurer le contexte et elles sont suivies immédiatement d'une instruction **ret**. De ce fait, les instructions de dépilement sont inutiles : après ce **ret**, on se retrouvera dans le contexte d'un autre bloc exécutant du programme Ackermann. Comme il n'était pas utile de dépiler ce contexte, il n'était pas utile non plus de l'empiler, et, comme dans le cas de n!, il n'était pas nécessaire, en fait de sauver l'adresse de retour du programme lui-même, car, à cet endroit de l'exécution, la rencontre du **ret** conduira au dépilement de tous les appels terminaux sans autre action. Les bons compilateurs de langage évolué savent effectuer cette simplification.

Ce qu'ils ne peuvent faire, par contre, est l'observation suivante : la valeur de *DX* (c'est-à-dire *Y*) n'a pas besoin d'être sauvée pour le seul appel récursif encore utile dans le programme Ackermann, à savoir celui de l'instruction **0133** : en effet, après le **pop dx** de **0137**, le **push dx** suivant se rapporte à un appel récursif terminal et est donc inutile ; l'instruction qui suit, et qui précède le second appel récursif, est **mov dx,ax**.

Programme Ackermann :

Les arguments de la fonction sont *CX* et *DX*, respectivement, *X* et *Y*, résultat : *AX*.

```
0102 83F900        CMP  CX,+00        ; X = 0?
0105 7709          JA   0110          ; non : -> suite
0107 42            INC  DX            ; oui :
0108 89D0          MOV  AX,DX         ; psi := Y+1
010A C3            RET                ; retour

0110 83FA00        CMP  DX,+00        ; Y = 0?
0113 7715          JA   012A          ; non : -> suite
0115 3B260001      CMP  SP,[0100]     ; oui : peut-on empiler?
0119 7635          JBE  0150          ; non : -> arrêt et message
011B 52            PUSH DX            ; oui : on empile le contexte
011C 51            PUSH CX            ;   de l'appel récursif
011D BA0100        MOV  DX,0001       ;
0120 49            DEC  CX            ;
0121 E8DEFF        CALL 0102          ; psi := psi (X-1,1)
0124 59            POP  CX            ; restauration du contexte
0125 5A            POP  DX            ; pour la présente exécution
0126 C3            RET                ;

012A 3B260001      CMP  SP,[0100]     ; X,Y <> 0 ; peut-on empiler?
012E 7620          JBE  0150          ; non : -> arrêt et message
0130 52            PUSH DX            ; oui : on empile le contexte
0131 51            PUSH CX            ;   du premier appel
0132 4A            DEC  DX            ;
0133 E8CCFF        CALL 0102          ; psi := psi (X,Y-1)
0136 59            POP  CX            ; restauration du contexte pour
0137 5A            POP  DX            ; la présente exécution
0138 3B260001      CMP  SP,[0100]     ; peut-on empiler?
013C 7612          JBE  0150          ; non : -> arrêt et message
013E 52            PUSH DX            ; oui : on empile le contexte
013F 51            PUSH CX            ;   du second appel
0140 89C2          MOV  DX,AX         ; W := psi (X,Y-1)
0142 49            DEC  CX            ;
0143 E8BCFF        CALL 0102          ; psi := psi (X-1,W)
0146 59            POP  CX            ; restauration du contexte pour
0147 5A            POP  DX            ; la présente exécution
0148 C3            RET                ; retour
```

Programme Ackrec1 sans récursivité terminale :

```
0102 83F900        CMP  CX,+00        ; CX = 0?
0105 7709          JA   0110          ; non : -> suite
0107 42            INC  DX            ; oui :
0108 89D0          MOV  AX,DX         ; psi := Y+1
010A C3            RET                ; retour

0110 83FA00        CMP  DX,+00        ; Y = 0?
0113 770B          JA   0120          ; non : -> suite
0115 BA0100        MOV  DX,0001       ;
0118 49            DEC  CX            ;
0119 EBE7          JMP  0102          ; psi := psi (X-1,1)

0120 3B260001      CMP  SP,[0100]     ; peut-on empiler?
0124 761A          JBE  0140          ; non : arrêt et message
0126 51            PUSH CX            ; sauvegarde du contexte
0127 4A            DEC  DX            ;
0128 E8D7FF        CALL 0102          ; psi := psi (X,Y-1)
012B 59            POP  CX            ; restauration du contexte
012C 89C2          MOV  DX,AX         ; W := psi (X,Y-1)
012E 49            DEC  CX            ;
012F EBD1          JMP  0102          ; psi := psi (X-1,W)
```

REMARQUE : Dans tous ces programmes, la pile est contrôlée par un 'butoir' fixé à l'adresse **CS:0600**, la valeur **0600** étant conservée dans le mot d'adresse **0100**.

Comme il est donc inutile de sauver *DX*, on obtient, en fonction de toutes ces remarques, le programme simplifié figurant au bas de la page 93 et que nous appellerons ***programme Ackrec1***.

Le cas général.

On peut le représenter par les programmes suivants donnés côte à côte, où un seul appel récursif, terminal, est indiqué, le programme récursif à gauche sur ce listage, et le programme non récursif, à droite. Dans ces deux programmes, le programme **0300** représente la transformation effectuée par le programme **0100**, le programme **0200** représente la préparation des arguments pour l'appel récursif suivant.

```
0100 E30B    JCXZ 010D     si X = 0, retour
0102 E8FB01  CALL 0300     sinon, calcul
0105 51      PUSH CX
0106 E8F700  CALL 0200       appel récursif ->
0109 E8F4FF  CALL 0100   <- appel récursif
010C 59      POP  CX
010D C3      RET
```

```
0100 E308    JCXZ 010A
0102 E8FB02  CALL 0300
0105 E8F800  CALL 0200
0108 EBF6    JMP  0100
010A C3      RET
```

Le raisonnement justifiant le passage au programme de droite est le même que précédemment : inutilité de la sauvegarde du contexte car non utilisé après son dépilement, et, pour la même raison, inutilité de la sauvegarde de l'adresse de retour puisqu'alors on ne fait pas autre chose que revenir aux programmes précédents, ... récursivement!

Principes de dérécursification.

Le principe théorique.

Nous avons vu un cas important où il y a intérêt à transformer un programme récursif en programme non récursif, tant du point de vue de l'étendue du code que, surtout, du temps d'exécution et de l'économie des ressources. On peut se demander si, d'une part, la propriété est générale et, d'autre part, si une telle transformation est toujours possible.

Nous allons d'abord répondre à cette deuxième question.

La théorie des fonctions récursives permet de démontrer le *principe général de dérécursification* : tout programme récursif admet un programme équivalent non récursif. D'une certaine manière, l'équivalence entre les fonctions récursives et les machines de Minsky fournit une démonstration de cette propriété, dans la mesure où ces machines, au sens où nous les avons définies, ne sont évidemment pas récursives (les machines de Minsky ne connaissent même pas le **call**). Cependant, si une telle démonstration, répondant à la *question de principe*, fonde la possibilité de ***dérécursifier*** tout programme récursif, elle ne fournit pas de moyen pratique pour le faire.

Y a-t-il intérêt à faire une telle transformation?

Le plus souvent, oui. En effet, on a fréquemment intérêt à rechercher une solution récursive au problème, dans tout les cas où il se prête à cette approche par sa nature. Cette approche présente deux qualités importantes : d'une part, une facilité de conception et d'écriture des programmes, d'autre part, la démonstration de la correction du programme.

> Précisons ce point. On procède le plus souvent par récurrence sur un paramètre entier naturel n. On démontre tout d'abord que pour $n = 0$, le programme fait ce qu'on attend de lui. Puis, supposant, par hypothèse de récurrence, qu'il le fait pour tous les objets associés à la valeur n du paramètre, on démontre alors qu'il le fait nécessairement pour tout objet associé à la valeur $n+1$ du paramètre. Le jeu de Hanoï en est une illustration frappante.

Dans un second temps, on a intérêt à dérécursifier, notamment pour des programmes répétés fréquemment. Si on ne peut supprimer complètement le recours à la pile que, pratiquement, dans le cas d'une récursivité terminale, dans les autres cas *concrets*, on peut souvent *réduire* les sauvegardes du contexte de façon à réduire le recours à la pile au minimum. Nous donnons, pour ce faire, une méthode générale.

La méthode du parcours : application au programme Ackermann.

Replaçons-nous dans le cadre d'un programme récursif. Nous avons interprété les appels du programme à lui-même comme des appels à des blocs exécutants qui se différencient par leur environnement. Au cours de l'exécution, les valeurs arguments constituant le contexte du programme récursif définissent un certain ensemble et, à l'intérieur de cet ensemble, elles décrivent un certain ***parcours***. L'idée de la méthode est de reproduire ce parcours d'une façon non récursive et, si possible, plus économique.

Nous illustrons comment procéder sur l'exemple du programme Ackermann.

Nous avons déjà beaucoup gagné en éliminant la récursion terminale et en remarquant qu'il suffit de sauver la valeur de X seulement, nous le verrons plus loin. Il nous reste donc à éliminer l'unique appel récursif du programme Ackrec1, ce que nous faisons en cherchant le parcours de X pendant l'exécution du programme. Nous le traduisons sur le listage ci-dessous où les indentations permettent de suivre les étapes du raisonnement, l'adresse **0100** ayant été donnée au début de la boucle principale et non au début du programme.

En *temps normal*, on franchit les deux tests $X = 0$ et $Y = 0$ pour passer à l'instruction **0120** du programme Ackrec1. On empile CX, on décrémente et on revient au début : instructions **010D-0112** du listage ci-après. Il y a là une boucle que nous faisons débuter en **0100**, et dont on ne sort que si $X = 0$. Dans cette boucle, il peut se produire que $Y = 0$ (test de **0102**). Dans ce cas, **0105 à 010D**, on effectue $X := X\text{-}1$ et $Y := 1$.

```
00F6 89E6     MOV  SI,SP·           ; SI : niveau zéro de la pile
00F8:BOUCLE1                        ; tant que X > 0 et pile > 0
00F8 39F6     CMP  SP,SI            ; pile = 0?
00FA 7204     JB   0100             ; non : -> suite
00FC E31C     JCXZ 011A             ; X = 0 et pile = 0 : -> fin
00FE EB02     JMP  0102             ; X > 0 : -> test sur Y
0100:BOUCLE2                        ; tant que X > 0
0100 E310     JCXZ 0112             ; si X = 0 : sortie de boucle
0102 83FA00   CMP  DX,00            ;
0105 7706     JA   010D             ;
0107 49       DEC  CX               ;                 % comme ce programme ne
0108 BA0100   MOV  DX,0001          ;                 % comporte ni call ni
010B EBF3     JMP  0100             ; -> BOUCLE2      % variable, on peut le
010D 51       PUSH CX               ;                 % translater par la commande
010E 4A       DEC  DX               ;                 % m de DEBUG afin de
010F EBEF     JMP  0100             ; -> BOUCLE2      % commencer à l'adresse 0100
0112 42       INC  DX               ;                 % ou toute autre adresse.
0113 89D0     MOV  AX,DX            ;
0115 59       POP  CX               ;
0116 49       DEC  CX               ;
0117 EBDF     JMP  00F8             ; -> BOUCLE1
011A 42       INC  DX               ;
011B 89D0     MOV  AX,DX            ;
011D C3       RET                   ;
```

Que faire si $X = 0$? Ceci marque la fin d'un appel récursif : il faut dépiler le contexte, soit CX. Mais il faut, auparavant incrémenter Y, l'affecter à ψ et décrémenter X, d'où les instructions **0112-0119**. Mais ce retour concerne-t-il un appel récursif ou le premier appel du programme? Pour le savoir, il faut regarder où en est la

pile : il s'agira de l'appel initial quand *SP* retrouvera la valeur que ce registre avait au départ. D'où la nécessité de stocker cette valeur, ce qu'on fait, ici, dans *SI*. La condition de terminaison du programme est donc *CX* = 0 et *SP* = *SI*, c'est-à-dire, la pile revenue 'à zéro'. Mais on doit à nouveau incrémenter *Y* et porter cette valeur dans *AX*, obtenant ainsi le résultat cherché.

On trouvera à l'annexe 3 le programme Ackermann dérécursifié, avec quelques instructions supplémentaires de contrôle du débordement de la pile.

Mesures.

Le programme de chronométrage que nous avons construit au début de ce chapitre va nous permettre de comparer les performances, du point de vue du temps d'exécution des trois programmes dont nous disposons pour calculer la fonction d'Ackermann : le programme Ackermann, que nous désignerons dans le tableau ci-après par Ackrec0, le programme Ackrec1 et le programme dérécursifié de l'annexe 3 que nous appellerons Ackdr.

Le déclenchement et l'arrêt du chronomètre s'obtiennent à chaque fois en faisant appel à le même fonction *DOS* : la fonction **02C** de l'interruption **021** qui fournit l'heure au format (*h,m,s,c*), c'est-à-dire, heure, minutes, secondes, centièmes. Son fonctionnement est le suivant :

```
 AX   CX   DX   IP
0000 0000 0000 0100  MOV  AH,2C
2C00 0000 0000 0102  INT  21
2C00 090D 3151 0104
      h m  s c                    % heure indiquée : 9h 13mn 49s 81c
```

On place les deux instants ainsi définis aux adresses **0290** et **0294**. Le programme de chronométrage place la durée dans *CX:AX* et le programme d'affichage convertit les octets trouvés en nombres décimaux, ce que nous savons faire et ce que le programme de l'annexe 3 donne intégralement.

On obtient alors les résultats suivants :

	ψ(3,7)	ψ(3,8)	ψ(3,9)	ψ(3,10)	ψ(3,11)
valeur	1021	2045	4093	8189	16381
Ackrec0	6s38	25s54	1m42s	6m49s	8m42s*
Ackrec1	4s50	18s07	1m12s	4m50s	18m01*
Ackdr	4s45	17s79	1m11s	4m53s	19m02s

L'astérisque signifie que le calcul a été interrompu au bout du temps indiqué par débordement de la pile.

On constate ainsi que, du point de vue du temps d'exécution, la dérécursification porte ses fruits. Le plus gros du travail a été fait par l'élimination de la récursion terminale et par la simplification apportée à la sauvegarde. La dérécursification complète a permis, dans ce cas, de porter le calcul plus loin, car le débordement de la pile se manifeste pour ψ(3,11) en ce qui concerne Ackrec1. On peut vérifier les performances de mobilisation des ressources de ces programmes en modifiant l'adresse de butoir de la pile fixée à **0600** dans les programmes considérés (voir l'annexe 3). Si l'on fixe ce butoir à l'adresse *SS*:**0F800**, Ackrec1 calcule ψ(3,6) mais pas ψ(3,7) tandis que Ackdr calcule ψ(3,7) mais pas ψ(3,8).

L'explication est la suivante : la plus grande séquence d'empilements successifs sans dépilement comporte un nombre d'empilements pratiquement égal à la valeur que la fonction doit calculer. Pour calculer ψ(3,7) = 1021, il faut environ 1024 mots soit 2048 octets, soit, en hexadécimal, **0800**. La place laissée par la borne **0F800** est donc

suffisante pour le programme Ackdr. Mais, pour le même argument, Ackrec1 demande une place double puisqu'il empile un contexte **et** une adresse retour. Comme on a, approximativement $\psi(3,n+1) = 2*\psi(3,n)$, cette place est suffisante pour que Ackrec1 puisse calculer $\psi(3,6)$.

REMARQUE : L'exécution du programme est sensible aux instructions utilisées. Ainsi, le programme Ackdr de l'annexe 3 comporte, pour le contrôle du débordement de la pile, une instruction **cmp si,[31E]**. Si l'on remplace cette instruction par **cmp si,bx** après avoir initialisé *BX* une fois pour toute hors boucle, on obtient une amélioration du temps d'exécution comme l'indique le tableau ci-après où l'on compare les performances du programme Ackdr à un programme Ackdr1 fonctionnant avec le registre *BX* :

	$\psi(3,7)$	$\psi(3,8)$	$\psi(3,9)$	$\psi(3,10)$	$\psi(3,11)$
valeur	1021	2045	4093	8189	16381
Ackdr	4s45	17s79	1m11s	4m53s	19m02s
Ackdr1	4s01	16s04	1m04s	4m17s	17m10s

La raison en est que l'exécution de l'instruction **cmp si,bx** reste 'interne' au processeur tandis que l'exécution de **cmp si,[31E]** oblige à en 'sortir' pour consulter la *RAM* qui se trouve 'loin' de lui, ce qui prend plus de temps.

Autres applications.

La méthode du parcours que nous avons appliquée au programme Ackermann s'applique *mutatis mutandis* à tout programme comportant deux appels récursifs dont l'un soit terminal.

C'est le cas, par exemple, du programme Hanoï. On pourrait en plus, dans le cas de ce programme, compte tenu de la structure du contexte, ne pas empiler *a*,*b* et *c*, mais effectuer leurs 'restaurations' par calcul puisqu'il s'agit de simples permutations. On gagnerait peu, sur le temps d'exécution, mais on gagnerait trois fois plus de place pour la pile. Malgré cela, l'effort de programmation correspondant serait assez mal récompensé, surtout dans le cas d'une application graphique : le temps d'exécution dépend en fait des entrées/sorties qui prennent plus de temps que le calcul nécessité par la récursion. Qui plus est, si on implante un dialogue avec l'utilisateur, les délais introduits pour ce dernier sont infiniment plus importants et rendent la dérécursification inutile. Enfin, dans le programme Hanoï, l'utilisation de la pile est négligeable : la plus grande séquence d'empilements successifs sans dépilement est *n*, le nombre de disques. Or, le temps d'exécution, avec ou sans récursivité, est incomparablement plus grand : de l'ordre de 2^n. Pour une application pratique, les valeurs de *n* que l'on choisira feront que la pile nécessitera moins de place que la structure de contrôle de l'image graphique...

Par contre, la méthode est utile dans un autre cas, que nous n'abordons pas dans cet ouvrage : celui de l'utilisation d'arbres binaires, par exemple, dans des problèmes de tri. La démarche récursive permet d'obtenir très facilement des programmes que l'on peut ensuite dérécursifier si le programme de tri est lui même fréquemment appelé par un autre programme ou si sa durée est suffisamment longue.

DEUXIEME PARTIE : MISE EN OEUVRE.

CHAPITRE 5

MODELISATION.

Comme indiqué au chapitre 2, nous abordons ici la simulation, en assembleur, d'une machine de Minsky.

Pourquoi cet exemple? Tout simplement parce que dans ce problème d'apparence *purement théorique*, on est immédiatement confronté à un problème *pratique* fondamental : celui de la gestion des ressources de l'ordinateur. La raison en est celle que nous avons déjà signalée au chapitre 2 et que nous rappelons. Les registres d'une machine de Minsky sont censés contenir des nombres naturels *quelconques*, y compris des nombres arbitrairement grands. Dans la *représentation* de tels nombres en machine, nécessairement fondée sur une *base fixe* donnée, apparaît alors la notion de *longueur* ou *taille* de la représentation : dans le cas des ordinateurs actuels, le nombre d'*octets* (ou de *mots*) nécessaires pour écrire le nombre considéré. Comme, naturellement, les ressources de l'ordinateur sont *finies*[1], on doit prévoir une limite à la représentation. Nous examinons à présent les *modalités concrètes* d'une solution à ce problème.

5.1 Le cahier des charges.

Notre programme, que nous appellerons ***simulateur***, devra reproduire le fonctionnement de toute machine de Minsky écrite dans le langage défini au chapitre 2 qu'on lui soumettra. Il commencera par analyser la correction syntaxique des instructions soumises. Si l'analyse se conclut positivement, il demandera à l'utilisateur d'entrer à l'écran les valeurs initiales des registres. Si l'entrée est faite correctement, le simulateur exécutera une à une les instructions de la machine de Minsky. Si l'exécution se termine normalement, le simulateur affichera le contenu des registres désignés par l'utilisateur comme devant contenir le résultat.

Ces indications contiennent beaucoup de *si* et des sous-entendus qui vont se révéler à l'occasion de la réalisation de ce programme. Nous sommes conduits, de ce fait, à préciser les *spécifications* du cahier des charges.

L'analyse syntaxique suppose une définition très précise de la syntaxe des programmes à simuler que nous appellerons, dans ce qui suit, ***programmes Minsky***. Nous commençons par rappeler ce point et par le compléter.

Syntaxe des programmes Minsky.

Les instructions sont les quatre instructions **nul**, **inc**, **st** et **dsz** que nous avons vues au chapitre 2. Les adresses seront écrites en *nombres décimaux*. Pour les raisons de

---

[1] Ultra-finies, lit-on dans certains articles théoriques où l'on distingue entre le fini *accessible* et celui qui ne l'est guère comme, par exemple, le nombre $10^{10000!}$.

limitation des machines *réelles* que nous connaissons, on restreindra les adresses à des nombres s'écrivant avec au plus quatre chiffres : on est sûr qu'ils se convertissent en base seize en des nombres dont chacun n'occupera pas plus d'un *mot* mémoire. De la même manière, on précise l'écriture du *nom* des registres. Nous avions indiqué, au chapitre 2, l'écriture *RX*, où *X* est une des 26 lettres de l'alphabet, et l'écriture *R***k**, où **k** est un entier naturel. Ce nombre **k** sera soumis au mêmes limites que les nombres figurant dans les adresses des instructions Minsky. Enfin, nous conviendrons de ne pas distinguer la majuscule de la minuscule d'une même lettre de l'alphabet.

Nous avons vu l'importance des *commentaires* dans un programme. Les programmes *Minsky* doivent bénéficier, à part entière, de cette possibilité. Nous fixerons la même règle qu'en assembleur : nous conviendrons de n'écrire au plus qu'une instruction sur une même ligne et d'introduire les commentaires par le symbole ;. Ce qui est inclus entre ce symbole et la fin de la ligne n'est pas traité par notre simulateur.

Enfin, pour simplifier grandement l'écriture du programme et pour ne pas masquer l'essentiel par les problèmes[1] d'un éditeur de texte avec vérificateur de syntaxe intégré (ce que fait *DEBUG* sous une forme certes rudimentaire, mais néanmoins précieuse), nous supposerons que le programme Minsky à simuler se trouve *dans un fichier*. Nous imposerons également une limite à la *taille* de ces fichiers : pour des raisons de programmation évidentes, notre simulateur refusera d'analyser un fichier de plus de 64 Ko[2].

Arguments initiaux et résultats.

L'entrée des valeurs initiales des registres se fera à l'écran. Chaque nombre entré, réputé *nombre décimal* aura au plus 64 chiffres. Le nombre des registres est limité à 1000, ce qui reste confortable.

Les résultats seront affichés à l'écran.

La redirection des entrées/sorties sous *DOS* permet d'utiliser des fichiers tant pour l'initialisation des paramètres que pour les résultats.

Traitement des erreurs.

Comme nous l'avons vu au chapitre 2, un programme ne peut pas contrôler complètement l'exécution d'une machine de Minsky, puisqu'on ne dispose d'aucun moyen *général* pour tester si l'exécution d'une machine fixée sur des données fixées s'arrête ou se poursuit indéfiniment. C'est pourquoi, nous ne chercherons pas à détecter des erreurs de logique. Par exemple, le simulateur ne cherchera pas à déceler si on divise par zéro ou non dans une division. Si on divise par zéro, il exécutera la division, tombant ainsi dans une boucle infinie.

> A vrai dire, le simulateur ne peut pas *interpréter*, au sens mathématique, les instructions qu'il exécute. Il n'a donc aucun moyen de détecter *à coup sûr* qu'une suite d'instructions données programme effectivement une division : il y a une infinité de façons de le faire. Il y a par contre une solution pour le programmeur qui est censé savoir ce qu'il fait : celui-ci peut incorporer au programme Minsky un test de division par zéro et fixer un registre pour recevoir le résultat du test.

C'est une situation générale : la prévention des erreurs de logique est du ressort du programmeur. C'est pourquoi notre simulateur s'efforce de détecter ce qui est dans ses possibilités : les erreurs de syntaxe. Ainsi, si le programme Minsky soumis

---

1 Au demeurant non triviaux!

2 Nous pensons que peu de programmeurs voudraient goûter au plaisir d'écrire d'authentiques programmes Minsky de plusieurs centaines de Ko...

comporte des instructions mal écrites pour une raison ou une autre, le simulateur le reconnaîtra et affichera un message adéquat, en principe, pour toutes les erreurs du texte. Il surveille également l'entrée des données à l'écran. Enfin, pendant l'exécution, il veille en permanence à ce que les limites d'espace mémoire fixées pour l'exécution soient respectées.

5.2. Conception du simulateur.

5.2.1. Transcodage des programmes Minsky.

Afin de faciliter l'exécution du programme Minsky qu'on lui soumet, notre simulateur procède à un *codage* des instructions Minsky. Chacune d'elles est codée sur plusieurs octets. Le premier indique le type de l'instruction selon le tableau suivant :

00 pour **nul**,
01 pour **inc**,
02 pour **st**,
03 pour **dsz**

Dans le cas des instructions **nul** et **inc**, ce premier octet est suivi par un *mot* qui contient le numéro d'ordre du registre. Quand l'analyseur rencontre le nom d'un registre, il regarde s'il figure déjà dans la liste, depuis l'adresse **02000**, de ceux qu'il a relevés. Si oui, il connaît le numéro d'ordre du registre, sinon, il lui en attribue un et ajoute son nom à la liste. Au départ, la liste est vide. Dans tous les cas, l'analyseur inscrit ce numéro d'ordre depuis le second octet du code de l'instruction.

Dans le cas de l'instruction **st**, le premier octet est suivi d'un mot : l'adresse de l'instruction, dans le code du programme à exécuter que construit le simulateur. Dans le cas de l'instruction **dsz**, le premier octet doit être suivi de deux mots : dans le mot de poids faible on placera le numéro d'ordre du registre, et dans le mot de poids fort, on placera l'adresse de l'instruction référencée par **dsz**.

La définition de l'adresse des instructions référencées dans **st** ou **dsz** est réalisée par une analyse en deux temps.

Dans un premier temps, l'analyseur peut coder le type des instructions et le nom des registres (par les numéros d'ordre). S'il ne connaît pas les adresses des instructions référencées qu'il n'a pas encore examinées, il peut déterminer pour chaque instruction analysée, au moment où il la rencontre, le nombre d'octets que prend son code : trois octets pour **nul**, **inc** et **st**, cinq octets pour **dsz**. Lors de ce premier passage, il recopie simplement sur le code le numéro (qu'il convertit en hexadécimal) de la référence et inscrit, dans une table, l'adresse, depuis **08000**, de l'instruction qu'il vient de coder.

Dans une seconde étape, l'analyseur relit le code qu'il vient de construire et remplace les numéros d'instruction Minsky par l'adresse correspondante dans le code.

Ci-contre, nous avons,sur un même listage, le programme de la multiplication donné au chapitre 2 (page 9), et le code correspondant construit par le simulateur.

Au cours de la première lecture, les trois premières instructions du programme auront été codées (adresse du premier octet du code devant chaque transcodage d'une instruction) :

```
00  00 00 00
03  03 01 00 0E 00
08  03 02 00 0E 00
```

Sur chaque ligne, on a, de gauche à droite : l'adresse du premier octet du code, le code de l'instruction, le numéro (décimal) de l'instruction Minsky correspondante et cette instruction elle-même :

```
00  00 00 00                    0  NUL  RR
03  03 01 00 34 00              1  DSZ  RA,14
08  03 02 00 34 00              2  DSZ  RB,14
0D  01 01 00                    3  INC  RA
10  01 02 00                    4  INC  RB
13  00 03 00                    5  NUL  RU
16  03 01 00 34 00              6  DSZ  RA,14
1B  03 02 00 29 00              7  DSZ  RB,11
20  01 00 00                    8  INC  RR
23  01 03 00                    9  INC  RU
26  02 1B 00                   10  ST   7
29  03 03 00 16 00             11  DSZ  RU,6
2E  01 02 00                   12  INC  RB
31  02 29 00                   13  ST   11
34                             14
```

et la table de correspondance entre les instructions sera :

adresses d'entrée :	**00**	**02**	**04**
entrées :	**00**	**03**	**08**

l'adresse d'une entrée de la table étant le double du numéro de l'instruction qui lui correspond.

La vérification de la correction syntaxique s'effectue au cours du codage, pendant la première étape de l'analyse. Elle porte sur l'orthographe du nom des instructions et sur le respect des règles de formation rappelées ci-dessus.

5.2.2. La représentation des registres.

C'est le problème clé de notre simulation. De multiples solutions peuvent être envisagées pour résoudre ce problème. Nous avons choisi le parti de calquer l'organisation des objets qui joueront le rôle des registres de la machine de Minsky dans notre modélisation, sur la structure des fichiers de *DOS* et leur organisation par le système d'exploitation. Voyons, à présent, comment cela fonctionne concrètement.

Espace des registres.

Sans restreindre la généralité du modèle proposé ci-après, nous nous bornerons à gérer la partie de la mémoire vive que le programme définira au début de son exécution. Pour fixer les idées, notre programme se restreindra à 64 Ko pour son code et diverses tables qu'il construira à partir du programme Minsky qu'on lui soumet et il s'allouera une zone de 64 Ko pour le traitement et pour l'exécution de ce programme.

De façon précise, cette zone, pointée par *ES* une fois qu'elle aura été allouée, sera utilisée une première fois pour recevoir le texte du programme Minsky. C'est sur cette copie du texte que travaillera l'analyseur de syntaxe du simulateur. Le résultat de l'analyse, lorsqu'il n'y a pas d'erreur est placé dans le premier espace de 64 Ko, à partir de l'adresse **08000**. La seconde zone sera affectée aux registres et nous l'appellerons, de ce fait, l'***espace des registres***.

Table d'allocation.

Le principe de cette affectation est le suivant : par analogie avec la structure logique des supports de masse, on découpe l'espace des registres en ***secteurs*** de seize octets chacun. Chaque secteur est représenté par un *mot* dans une zone située au début

de l'espace des registres, la ***table d'allocation des secteurs***, en abrégé : ***TAS***. Les 64 Ko ne peuvent à la fois contenir 4096 secteurs de seize octets et les 8192 octets de la *TAS* correspondante. Un calcul simple nous montre que 64 Ko permettent de contenir et représenter 3640 secteurs avec un reste de seize octets.

Chaque entrée de la *TAS* d'adresse relative 2**S* représente le secteur commençant à l'adresse relative hexadécimale (**01C8**+*S*)***010**. Afin de simplifier les calculs, on numérote les secteurs de **01C8** à **0FFF** (de 456 à 4095) inclusivement. Si *S* est le ***numéro*** d'un secteur, celui-ci commence donc à l'adresse **010****S* et le mot qui le représente en *TAS* a pour adresse 2*(*S*-**01C8**). Un ***registre*** est, par définition, une suite ordonnée de secteurs distincts rattachés à un ***nom***. Soit $S_0,...S_k$ la suite, dans cet ordre, des numéros des secteurs qui composent le registre *R* représentant le nombre *N*. Les seize octets de S_0, dans l'ordre de leurs adresses croissantes, sont les décimales associées aux puissances 0 à 15 de 256 dans la décomposition de *N* en base 256, les seize octets de S_1 sont les décimales suivantes (puissances 16 à 31), et ainsi de suite. De la sorte un registre possède un ***premier secteur*** et, de même, un ***dernier secteur***[1]. On ne suppose pas que la suite des numéros des secteurs du registre soit une suite croissante. Le nombre *N* ainsi représenté par le registre sera appelé la ***valeur*** du registre.

Répertoire.

Pour que le registre soit entièrement déterminé par la connaissance du numéro de son premier secteur, il suffit de placer, dans le mot qui représente le secteur *de* numéro S_i, le nombre S_{i+1} pour i = 0,...,*k-1* et une marque de dernier secteur dans le mot qui représente le secteur de numéro S_k, dernier secteur du registre. Par ailleurs, on définit la ***longueur*** de la représentation de *N* dans le registre *R* comme le nombre de décimales dans la représentation de *N* en base 256[2] (=65536). Pour des raisons de commodité des calculs et de vérification, on conservera cette information dans une table que nous appellerons le ***répertoire des registres***.

Une entrée de la *TAS* aura pour valeur :

0000	si le secteur désigné n'est occupé par aucun registre,
0100	si le secteur désigné est le dernier secteur du registre,
S'	où *S'* est le numéro du secteur suivant occupé par le registre.

Quant au répertoire des registres, qui prendra place dans la première zone de 64 Ko, depuis l'adresse **04000**, chacune de ses entrées occupe deux mots :

le numéro d'ordre du registre, dans le mot de poids faible,
la taille en mots de la valeur du registre dans le mot de poids fort.

Ce numéro d'ordre sera défini dans l'étape d'analyse syntaxique du programme Minsky : l'ordre des registres est, par convention, leur ordre d'apparition dans le texte des instructions.

Exemple :

Le registre est constitué par les secteurs, dans cet ordre, **01CD**, **01D0**, **01C9**. S'il représente le nombre suivant, écrit en hexadécimal :

```
1224 56A3 BC1E D932 1654 89D1 23FF A564 3123 13D1 23C5 7BB4
5AD2 3AB4 5645 F1CD 312A 564A
```

On lira en *TAS* :

```
adresses d'entrées :
   00    02    04    06    08    0A    0C    0E    10
entrées :
 ????  0100  ????  ????  ????  01D0  ????  ????  01C9
```

[1] Qui coïncident si le registre ne comporte qu'un seul secteur.

Et on lira, dans l'espace des registres, à partir de l'adresse **1C90** :

```
1C90 : A3 56 24 12 00 00 00 00 00 00 00 00 00 00 00 00
                        ...
1CD0 : 4A 56 2A 31 CD F1 45 56 B4 3A D2 5A B4 7B C5 23
                        ...
1D00 : D1 13 23 31 64 A5 FF 23 D1 89 54 16 32 D9 1E BC
```

Nous aurons à définir quelques opérations sur les registres. D'une part, ce que l'on peut appeler les ***manipulations*** : la création et l'initialisation de registres qui sont effectuées lors de la lecture des arguments. D'autre part, les opérations de type arithmétique, liées à l'exécution du programme Minsky : la remise à zéro d'un registre, son incrémentation, la détermination de la nullité de sa valeur, sa décrémentation lorsque cette valeur n'est pas nulle.

5.3. Programmation du simulateur

Nous donnons en annexe 1 le texte intégral de ce programme, de sorte que le lecteur désireux de consulter la source pour vérifier certains détails pourra le faire sans difficulté.

Nous donnons simplement quelques extraits permettant d'illustrer la mise en oeuvre des principes définis au paragraphe précédent.

Suivant le plan que nous avons suivi, nous commençons par la partie analyse de la syntaxe et par le transcodage.

Le programme principal.

Il consiste essentiellement en une suite d'appels à des sous-programmes correspondant à chacune des étapes du travail du simulateur. Chaque appel est suivi du test de l'indicateur *CF*. En effet, chaque procédure est censée renvoyer *NC* si sa tâche s'est effectuée normalement, et *CY* au cas où une erreur aurait été détectée. On a donc une suite d'instructions sur le modèle :

```
CALL 02E0     ; transcodage du fichier source
JC   013F     ; traitement des erreurs
```

Après l'exécution du programme de traitement des erreurs, on est renvoyé à la terminaison du programme par un **jmp 118**.

Nous nous contenterons donc, ici, de donner la liste des tâches effectuées, dans l'ordre d'appel des sous-programmes, que nous appellerons encore ***utilitaires*** ou ***procédures*** :

restriction à 64 Ko de la zone prise par le simulateur[1];
allocation de 64 Ko supplémentaires;
chargement du fichier source;
transcodage du fichier source;
lecture des arguments;
exécution;
traitement des erreurs;
affichage des registres.

Naturellement, s'il y a des erreurs, il n'y a pas exécution du programme et, *a fortiori*, pas d'affichage des registres. La terminaison du programme est placée à l'adresse **0118** sous la forme de la très classique **mov ax,4C00** suivie de **int 21**.

---

[1] On rappelle qu'un programme *.COM* se voit attribuer initialement par le chargeur de *COMMAND.COM* la totalité de la place libre en mémoire vive.

5.3.1. Le transcodage du fichier source.

Première passe.

Comme indiqué précédemment, cette opération s'effectue en deux temps. Nous reproduisons ci-après les instructions correspondant à la première 'passe' de l'analyseur :

```
02D8 FF366601      PUSH [0166]          ; 1ère PASSE
02DC 07            POP  ES              ; ES : sur l'espace des registres
02DD 31FF          XOR  DI,DI           ; ES:DI : pointeur de lecture
02DF 31F6          XOR  SI,SI           ; DS:08000+SI : pointe l'écriture
02E1 31D2          XOR  DX,DX           ; DX : n° de l'instruction lue
02E3:BOUCLE1                            ; tant qu'il y a des instructions
02E3 E88A00        CALL 0370            ; placer ES:DI sur la suivante
02E6 E87703        CALL 0660            ; calculer SI et enregistrer
02E9 7222          JB   030D            ; si incident -> saut à la ligne
02EB 3B3E3202      CMP  DI,[0232]       ; fin du fichier?
02EF 7310          JNB  0301            ; oui : 2ème passe
02F1 31DB          XOR  BX,BX           ; non : AL contient le type
02F3 88C3          MOV  BL,AL           ;   de l'instruction
02F5 00DB          ADD  BL,BL           ;   transformé en adresse
02F7 FF976403      CALL [BX+0364]       ; traitement ad hoc (CASE)
02FB E89202        CALL 0590            ; traiter la fin de ligne
02FE 42            INC  DX              ; n° d'instruction suivante
02FF EBE2          JMP  02E3            ; retour à BOUCLE1
0301 803E701000    CMP  BY [1070],00    ; y a-t-il des erreurs?
0306 740B          JZ   0313            ; non : suite
0308 F9            STC                  ; oui : signal CY
0309 C3            RET                  ;   retour

030D E88601        CALL 0496            ; saut à la ligne
0310 42            INC  DX              ; n° d'instruction suivante
0311 EBD0          JMP  02E3            ; retour à BOUCLE1
0313 E84A03        CALL 0660            ; adresse de HLT
0316 8936050B      MOV  [0B05],SI       ; la noter
031A 89166C03      MOV  [036C],DX       ; le numéro aussi
031E 87DB          XCHG BX,BX           ; remplissage (pour parité)
```

Dans ce sous-programme, nous retiendrons l'utilisation de tous les registres utilisables dans les adresses indirectes comme pointeurs des différentes zones associées au travail de transcodage. Nous remarquons qu'une toute première analyse est faite par la recherche de la première lettre du nom de l'instruction (sous-programme **0370**). Comme l'instruction a été repérée, on en profite pour écrire, dans la table adéquate (depuis **6000**) son adresse dans le codage et, provisoirement, on écrit le numéro de cette instruction dans le codage. Puis, convertissant le type de l'instruction renvoyé par **0370** en adresse dans *BX* sous forme d'un nombre, on utilise cette information pour appeler un utilitaire de transcodage adéquat par un **call** indirect calculé, à l'image de ce que nous avions vu pour implanter le *CASE* : **call [0364+bx]**, les vraies adresses des programmes concernés étant stockées depuis l'adresse **0364**. L'utilitaire **0590** passe à la ligne suivante et on incrémente *DX* puisque le **jmp 2EB** ramène au programme **0370** de recherche du début de l'instruction suivante. On termine la première passe en inscrivant, en **0B05** l'adresse de l'instruction Minsky **hlt** implicite.

L'analyse de la syntaxe.

Elle est faite au fur et à mesure de la *lecture* du fichier. Les programmes **0370** et les programmes appelés par **call [0364+bx]**, reposent sur la même boucle fondamentale que résume la séquence du haut de la page ci-contre :

```
0100 BOUCLE                          ; tant que l'on peut lire
0100 3B3E3202       CMP  DI,[0232]   ; fin du fichier atteinte?
0104 7202           JB   0108        ; non : suite
0106 F8             CLC              ;
0107 C3             RET              ;
0108 26             ES:              ; (ES : sur la copie du fichier)
0109 8A05           MOV  AL,[DI]     ; lire l'octet courant
010B E8F200         CALL 0200        ; traitement
010E 47             INC  DI          ; octet suivant
010F EBEF           JMP  0100        ; retour à BOUCLE
```

Dans le programme **0370**, la boucle consiste à passer sur les blancs (*ASCII* **020**) jusqu'à lire un caractère. Si c'est un point-virgule, on passe à la ligne suivante, ce qui veut dire lire jusqu'à ce qu'on rencontre soit la fin du fichier, soit la séquence **0D0A** du retour chariot et, dans ce cas, on positionne *DI* sur l'octet qui suit immédiatement cette séquence. Si c'est un caractère, on regarde si c'est le premier caractère du nom d'une instruction.

Ce test est réalisé par **rep scasb** convenablement initialisé, le contenu de *AL*, l'octet lu, est comparé aux données du tableau : NnIiSsDd débutant en **0358**.

Chacun des programmes [**0364**+*BX*] est fondé sur la même boucle. Comme on connaît la première lettre du nom de l'instruction, on sait combien de lettres on doit lire et lesquelles. On lit ces lettres et, par **rep cmps**, on vérifie l'orthographe du nom de l'instruction en comparant ce qui a été lu à 'l'étalon' figurant dans le tableau **0486**.

Dans le cas des instructions comportant un nom de registre, on renvoie au programme **04B8** pour vérifier la syntaxe du nom qui s'écrit *Rx* ou *Rdddd* (*d* chiffre décimal, *x* lettre). Dans la table **02000**, on écrit la valeur en hexadécimal de *dddd* dans le cas de *Rdddd* et le code *ASCII* de *x* (octet fort) suivi de **00** dans le cas de *Rx*. Comme *dddd* < **03000** et comme le code *ASCII* de *x* est supérieur à **040**, les deux cas se distinguent aisément. Voir le détail des instructions en annexe 1.

Seconde passe.

Il s'agit encore d'une boucle de lecture, instruction par instruction, *SI* pointant sur le code écrit, depuis *CS*:**08000** lors de la première passe. Comme précédemment, à chaque lecture, on regarde si la fin du code est atteinte. Sinon, on analyse l'octet lu, premier du code correspondant à l'instruction courante. Cet octet détermine si l'instruction contient ou non une adresse. Si c'est le cas, on a écrit, lors de la première passe, le numéro de l'instruction. La table **06000** fait la correspondance entre le numéro d'une instruction et son adresse *dans le code*. Ce numéro, multiplié par 2, donne l'adresse de l'entrée de la table qui lui correspond. Ce que détaille la séquence ci-après :

```
032E 3C02           CMP  AL,02          ; instruction NUL ou INC?
0330 721C           JB   034E           ; oui : -> on passe
0332 3C02           CMP  AL,02          ; instruction ST?
0334 7503           JNZ  0339           ; non : DSZ
0336 46             INC  SI             ; oui : ST, un octet à passer
0337 EB03           JMP  033C           ;
0339 83C603         ADD  SI,+03         ; référence après le registre
033C 8B9C0080       MOV  BX,[SI+8000]   ; BX := numéro de la référence
0340 01DB           ADD  BX,BX          ; converti en adresse de tableau
0342 8B870060       MOV  AX,[BX+6000]   ; AX := adresse de la référence
0346 89840080       MOV  [SI+8000],AX   ; portée sur le code
034A 46             INC  SI             ; passer
034B 46             INC  SI             ; deux octets
034C EBD6           JMP  0324           ; retour à la lecture
034E 83C603         ADD  SI,+03         ; SI := SI+taille de l'instruction
0351 EBD1           JMP  0324           ; retour à la lecture
```

5.3.2. La lecture des arguments.

Cette étape de la simulation pose, *a priori*, deux problèmes : la réservation de place pour les registres et la conversion des données entrées à l'écran.

Le premier problème se résout très simplement : puisque l'espace des registres est vierge au début de cette étape, le plus simple consiste à saisir les registres un par un, dans l'ordre qui leur a été attribué à l'étape précédente. A chaque fois, on affecte au registre traité une *zone* de l'espace des registres. En effet, lorsqu'on initialise un registre, si l'espace affecté aux registres déjà initialisés constitue une zone commençant à l'adresse **0000**, ce qui reste constitue également une zone à partir d'une certaine adresse. Si on en retranche une zone contiguë aux précédentes, l'espace attribué constituera de nouveau une zone ainsi que ce qui restera. Comme convenu, la zone affectée à un registre est composée d'un nombre entier de *secteurs*. L'attribution concrète des secteurs nous conduit au second problème.

Puisqu'on entre les nombres à l'écran, on a fixé une limite au nombre de leur chiffres : 64 avons nous dit dans le cahier des charges. Ces chiffres constituent les décimales en base dix qu'il faut donc, au fur et à mesure qu'on les lit, convertir en un nombre hexadécimal.

Multiplication de plusieurs mots par un seul.

Comme nous le savons, cette conversion repose sur des multiplications successives. Nous utiliserons à nouveau l'algorithme de Hörner, mais dans une implantation différente puisqu'il s'agit d'entiers 'très longs'. C'est-ce qu'effectue le programme **0820**. Les instructions *clés* sont les suivantes (ramenées à l'adresse **0100**) :

```
0133      CALL 0898          ; initialiser le registre R
0136:BOUCLE                  ; CX prêt
0136      CALL 0908          ; multiplier la valeur de R par cent (064)
0139      JC   0148          ; erreur : -> sortie
013B      CALL 0970          ; ajouter décimale 'suivante'
013E      JC   0148          ;
0140      LOOP 0136          ; retour à BOUCLE
0142      CALL 09F0          ; ré-initialisation des paramètres
0145      JC   0148          ;
```

Le nombre de chiffres du nombre entré constitue sa longueur en *octets*. Si ce nombre est impair, rajouter un **0** à gauche ne change rien à la valeur du nombre mais permet de considérer sa longueur *en mots*. Chaque groupe de deux chiffres est ainsi interprété comme une décimale en base cent. C'est la raison de la multiplication par cent effectuée par le programme **0908**. On utilise la même instruction **mul** que si l'on multipliait par dix, mais on va deux fois plus vite puisque le groupement des décimales divise leur nombre par deux.

Les instructions précédant cet extrait consistent précisément à se ramener à une longueur du nombre entré en mots en rajoutant un **0** à gauche si nécessaire. Cette longueur est placée dans *CX*, le compteur de la boucle, juste avant l'initalisation du registre à zéro sur l'unique secteur qui lui a été alloué.

Examinons de plus près le programme **0908** qui multiplie par cent la valeur courante du registre. L'algorithme suivi reprend la technique enseignée à l'école élémentaire. Le nombre à multiplier est écrit, en base **10000** :

$$a_k \quad \ldots \quad a_0,$$

et on le multiplie par un nombre $a < 10000$ (ici, $a = 064$). On effectue la multiplication 'décimale' par 'décimale' en partant de a_0. Soit a_i la décimale à examiner. En ajoutant 'mentalement' un zéro à gauche au besoin, on peut supposer que le nombre obtenu

jusqu'ici s'écrit avec i+1 décimales. Les i premières décimales sont celles du résultat final de la multiplication. Soit r la retenue obtenue à cette étape. Comme le produit a_i*a s'écrit c***010000** + a'_i et que la retenue s'ajoute à ce produit, la i+1ème décimale du résultat est donc le reste modulo **010000** de c***010000** + a'_i + r, c'est-à-dire le reste de a'_i+r = e***010000** + u_i où e = 0 ou 1. La nouvelle retenue est donc $c + e$, et comme a_i*$a \leq$ (**0FFFF**)^2 = **0FFFE*****010000** + 1 d'où $c <$ **0FFFF**, on a donc c+$e <$ **010000**. Quand nous en arrivons, à présent, à a_{i+1}, nous nous trouvons dans les mêmes hypothèses que précédemment. Or cette hypothèse est vérifiée pour a_0 si l'on convient de considérer que la retenue r vaut 0 lorsque i = 0.

La boucle principale du sous-programme **0908** qui multiplie la valeur courante du registre par cent est la suivante :

```
091B 31D2          XOR  DX,DX          ; retenue nulle au départ
091D:BOUCLE                           ;
091D 51            PUSH CX            ; sauver le compteur de boucle
091E 52            PUSH DX            ; sauver la retenue MUL précédente
091F 26            ES:                ;
0920 8B05          MOV  AX,[DI]       ; lire ai, la décimale courante
0922 F7E3          MUL  BX            ; DX:AX := AX*064 (BX = 064)
0924 59            POP  CX            ; rappel retenue MUL précédente
0925 01C8          ADD  AX,CX         ; addition
0927 7301          JNB  092A          ; retenue causée par l'addition?
0929 42            INC  DX            ; oui : l'ajouter à la retenue MUL
092A 26            ES:                ;
092B 8905          MOV  [DI],AX       ; inscrire le résultat partiel
092D 47            INC  DI            ; passer
092E 47            INC  DI            ; au mot suivant
092F 59            POP  CX            ; rappel du compteur
0930 E2EB          LOOP 091D          ; retour à BOUCLE
```

Allocation de secteurs.

Mais que se passe-t-il lorsqu'on sort de cette boucle? Il peut y avoir une retenue, auquel cas $DX > 0$. S'il en est ainsi, il se peut que le *mot* à écrire ne trouve plus place dans les secteurs alloués au registre. Si ce n'est pas le cas, on écrit donc le mot dans le secteur considéré et on augmente de 1 la longueur en mots de la valeur du registre. Mais si tel est le cas, il faut 'agrandir' le registre en lui allouant un nouveau secteur.

Dans ce cas, un nouveau test est nécessaire : reste-t-il encore des secteurs libres dans l'espace des registres? Si oui, on alloue le premier secteur disponible, sinon, le programme doit s'arrêter : on produit alors une 'erreur' sous la forme du message 'plus de place disponible'. C'est ce que fait la suite du programme **0908** auquel nous renvoyons à l'annexe 1. En particulier, ce programme fait appel à un utilitaire, le programme **09CC** que nous reproduisons ci-après, effectuant en *TAS* les marquages associés à l'allocation en cours de réalisation :

```
09CC 8B12240A  MOV  BX,[0A24]    ;
09D0 26        ES:               ; ES:BX sur TAS
09D1 8937      MOV  [BX],SI      ; n° inscrit
09D3 26        ES:               ; puis
09D4 FF07      INC  WO [BX]      ; mis à jour
09D6 43        INC  BX           ; ES:BX sur l'entrée
09D7 43        INC  BX           ; suivante en TAS
09D8 26        ES:               ; marquée alors comme
09D9 C7070001  MOV WO [BX],0100  ; dernier secteur
09DD 46        INC  SI           ; nouveau numéro
09DE 8936220A  MOV  [0A22],SI    ; enregistré
09E2 891E240A  MOV  [0A24],BX    ; adresse notée
09E6 C3        RET               ; retour
```

On note, dans ce dernier programme, l'utilisation de variables *partagées*. Ainsi **0A22** note le numéro du dernier secteur en cours du registre, **0A24** l'adresse de l'entrée de la *TAS* qui lui correspond, et **0A20**, l'adresse du premier octet de ce secteur dans l'espace des registres. Le contenu de ces variables est utilisé par d'autres procédures, ce que nous avons indiqué dans l'annexe 1.

Le programme **0970** qui additionne la décimale courante 'lue à l'écran'[1] présente quelques particularités que nous signalons :

Tout d'abord, cette décimale courante est constituée par les deux caractères se trouvant à la position courante de lecture :

```
0970 B30A     MOV  BL,0A          ; multiplicateur dix
0972 8A4600   MOV  AL,[BP+00]     ;
0975 2C30     SUB  AL,30          ; multiplicande prêt
0977 F6E3     MUL  BL             ; multiplication
0979 31DB     XOR  BX,BX          ;
097B 45       INC  BP             ; octet suivant
097C 8A5E00   MOV  BL,[BP+00]     ;
097F 80EB30   SUB  BL,30          ; idem (caractère -> nombre)
0982 01D8     ADD  AX,BX          ;
0984 45       INC  BP             ; octet suivant
```

Puis, elle est additionnée à la décimale en cours de traitement de la valeur courante. Ce qui peut poser un problème : en effet, cette addition peut produire une retenue de 1 qui, ajoutée à la décimale suivante peut en produire une autre et ainsi de suite, jusqu'à la décimale de tête de la valeur du registre comprise. On peut alors se trouver, comme précédemment, dans la nécessité d'allouer un nouveau secteur au registre ce qui nécessite le test que nous avons déjà signalé.

5.3.3. L'exécution du programme.

Par elle-même, la programmation de la boucle principale de l'exécution ne présente pas de difficulté. En voici, page ci-contre, une variante différente de celle de l'annexe 1.

On lit le transcodage du fichier source en regardant si l'instruction **hlt** implicite est atteinte. Le registre *SI* joue le rôle de pointeur d'instruction.

> L'adresse de l'instruction **hlt** implicite n'est connu qu'au moment du transcodage. Le transcodeur réécrit donc l'instruction **cmp sl,8000** ainsi écrite afin qu'elle occupe quatre octets car, comme nous le verrons au chapitre 7, cette instruction peut prendre trois ou quatre octets, selon la valeur de la donnée immédiate.

Le premier octet rencontré, **00**, **01**, **02** ou **03** nous donne le type de l'instruction codée et le nombre d'octets qu'elle occupe. Nous saurons donc où lire le premier octet de l'instruction suivante s'il n'y a pas de saut. S'il doit y en avoir un, la référence établie lors du transcodage est l'adresse (depuis **08000**) de l'instruction : on saura donc, dans tous les cas, où lire l'instruction suivante.

Le transfert de l'exécution est ensuite transféré par un **jmp** calculé selon la valeur lue en *AL*, sur le modèle du *CASE*. Dans les cas de **nul**, **inc** et **dsz**, on transfère l'exécution de l'instruction courante à un programme. L'exécution de l'instruction **st** ne nécessite pas d'appel à un programme. Elle se résume à l'exécution de l'instruction :

```
MOV  SI,[8001+SI].
```

---

[1] En fait, le nombre entré à l'écran a été recopié sur un tampon.

```
0B00 31F6          XOR  SI,SI          ; SI pointe dès le premier octet
0B02:BOUCLE                            ; tant que HLT n'est pas atteinte
0B02 31DB          XOR  BX,BX          ; BX := 0
0B04 81FE0080      CMP  SI,8000        ; donnée immédiate réécrite (HLT)
0B08 7341          JNB  0B4B           ; fin de l'exécution
0B0A 8A840080      MOV  AL,[SI+8000]   ; lire le 1er octet du code
0B0E 3C04          CMP  AL,04          ; correct?
0B10 7206          JB   0B18           ; oui : -> suite
0B12 E8BB04        CALL 0FD0           ; erreur du simulateur
0B15 F9            STC                 ;
0B16 C3            RET                 ;

0B18 88C3          MOV  BL,AL          ; n° de type d'instruction
0B1A 00C3          ADD  BL,AL          ; converti en adresse
0B1C FFA74C0B      JMP  [BX+0B4C]      ; saut calculé type CASE

0B20 8B840180      MOV  AX,[SI+8001]   ; instructions INC et NUL
0B24 FF97540B      CALL [BX+0B54]      ;
0B28 83C603        ADD  SI,+03         ;
0B2B EBD5          JMP  0B02           ;

0B2E 8BB40180      MOV  SI,[SI+8001]   ; instruction ST
0B32 EBCE          JMP  0B02           ;

0B34 8B840180      MOV  AX,[SI+8001]   ; instruction DSZ
0B38 E87502        CALL 0DB0           ; registre nul?
0B3B 720E          JB   0B4B           ; incident : sortie en erreur
0B3D 7506          JNZ  0B45           ; non nul (donc décrémenté)
0B3F 8BB40380      MOV  SI,[SI+8003]   ; nul : saut
0B43 EBBD          JMP  0B02           ;
0B45 83C605        ADD  SI,+05         ; (décrémentation faite)
0B48 EBB8          JMP  0B02           ;
0B4A F8            CLC                 ;
0B4B C3            RET                 ;

0B4C 20 0B 20 0B 2E 0B 34 0B           % adresses des JMP
0B54 90 0C C8 0B                       % adresses des CALL
```

En effet, avant son exécution, *SI* contient l'adresse du premier octet de l'instruction de saut. **08001**+*SI* pointe donc sur la référence qui est l'adresse depuis **08000** de la prochaine instruction à exécuter. D'où la nouvelle valeur de *SI* après exécution de l'instruction **mov** ci-dessus.

L'exécution des trois autres instructions, **nul**, **inc** et **dsz** présente un point commun : il faut d'abord retrouver le registre référencé dans l'instruction. Le programme **0B70** se charge de cette opération. Il dispose, en entrée de la référence du registre qui a été lue dans *AX* (le numéro d'ordre du registre). On va alors lire dans la table depuis **04000** à l'adresse relative égale à **4***AX* le numéro du premier secteur du registre puis la longueur en mots de sa valeur. A partir de là, on explore la *TAS* depuis l'image, dans cette table, du premier secteur du registre jusqu'au dernier secteur qui lui est alloué. Cette procédure fournit en sortie :

DI :	*ES:DI* est l'adresse du premier secteur du registre,
SI :	numéro du secteur correspondant,
BP :	adresse de ce secteur dans la *TAS*,
BX :	adresse (depuis **0000**) du registre dans le répertoire,
CX :	nombre des secteurs alloués au registre,
0A20 :	adresse du premier secteur du registre (témoin),
0A26 :	longueur en mots de la valeur du registre.

On rappelle que la longueur en mots affectée à un registre nul est 1.
On remarque les variables partagées **0A20** et **0A26**

Instruction nul.

L'exécution de **nul** consiste à remplir de 00 les seize octets du premier secteur du registre et à désallouer tous les autres. Ceci se fait par une boucle s'appuyant sur l'exploration de la *TAS*. Cette boucle a une structure particulière afin d'éviter le test encombrant du cas où un seul secteur est alloué au registre. En voici le schéma :

```
0CB2 E202          LOOP 0CB6          ; si CX = 1, c'est terminé
0CB4 EB2D          JMP  0CE3          ; et donc : -> suite
0CB6 3D0001        CMP  AX,0100       ; sinon : dernier secteur?
0CB9 7507          JNZ  0CC2          ; non : suite
0CBB E81203        CALL 0FD0          ; erreur du logiciel
0CBE F9            STC                ; (le test de contrôle de l'ex
0CBF 5E            POP  SI            ; dernier secteur est effectué
0CC0 C3            RET                ; hors boucle)

0CC2 51            PUSH CX            ; compteur sauvé
0CC3 50            PUSH AX            ; n° du secteur sauvé
0CC4 B90400        MOV  CX,0004       ; puis transformé en adresse
0CC7 D3E0          SHL  AX,CL         ; du début du secteur
0CC9 89C7          MOV  DI,AX         ;
0CCB E82A00        CALL 0CF8          ; mise à zéro du secteur
0CCE 58            POP  AX            ; AX := n° du secteur
0CCF 2DC801        SUB  AX,01C8       ; transformé
0CD2 D1E0          SHL  AX,1          ; à présent
0CD4 89C5          MOV  BP,AX         ; en adresse en TAS
0CD6 26            ES:                ;
0CD7 8B4600        MOV  AX,[BP+00]    ; lire le n° du secteur suivant
0CDA 26            ES:                ; puis le remplacer par la
0CDB C746000000    MOV  WO [BP+00],0000; marque : secteur libre
0CE0 59            POP  CX            ; rappel du compteur
0CE1 EBCF          JMP  0CB2          ; retour au test LOOP
```

Instruction inc.

L'exécution de l'instruction **inc** pose le même problème, mais en plus simple, que celui de l'addition de la décimale courante à la valeur actuelle du registre, ce que nous avons vu en étudiant le programme **0970**. En effet, au lieu d'ajouter une décimale au registre, on n'ajoute que le nombre 1, ce qui conduit à une simplification de la procédure. Cependant, ici, on opère l'incrémentation secteur par secteur, ce qui conduit à définir un sous-programme d'incrémentation d'un secteur que nous donnons ci-après. En effet, comme nous l'avons vu, l'instruction **inc** n'agit pas sur l'indicateur *CF*. Afin d'utiliser cet indicateur de maniement commode, puisqu'on peut le modifier à volonté, on propose les instructions suivantes, pour le corps de boucle :

```
0C15 26       ES:                ;
0C16 8B05     MOV  AX,[DI]       ; lecture du mot courant
0C18 050100   ADD  AX,0001       ; addition de 1
0C1B 26       ES:                ;
0C1C 8905     MOV  [DI],AX       ; écriture du résultat
0C1E 7304     JNB  0C24          ; si NC, terminé
0C20 47       INC  DI            ; si CY, passer au mot suivant
0C21 47       INC  DI            ;
```

Naturellement, l'addition de 1 peut conduire à sortir du dernier secteur alloué au registre afin d'écrire le 1 qui constitue la nouvelle décimale de tête du nombre représenté. La procédure est alors celle que nous avons vu dans le programme **0820**.

Instruction dsz.

Cette instruction est exécutée selon le même principe que l'instruction **inc**. On définit donc la décrémentation d'un secteur qui ressemble, en définitive à l'incrémentation : on remplace **add** par **sub**. De plus, comme on diminue la valeur du

registre, on ne sort pas du dernier secteur de celui-ci. Par ailleurs, comme on a testé au préalable si la valeur du registre est nulle ou non (rappelons qu'elle est nulle si et seulement si la longueur est 1 et le premier mot du premier secteur est **0000**), on sait, que l'opération de décrémentation finira par s'achever avec une valeur de *CF* égale à *NC*. Il n'y aura donc plus de retenue à ce moment là. Cependant, comme on diminue le nombre, il se peut que le dernier secteur se trouve réduit à zéro après l'opération[1]. Il faut alors le désallouer et marquer en *TAS* le secteur précédent comme dernier secteur du registre. Ceci nécessite de reprendre l'exploration de la *TAS* en ayant soin de noter le secteur qui précède celui que l'on considère. C'est ce que fait le programme **0D50** auquel nous renvoyons en annexe 1 et dont la structure s'apparente à celle du programme **0C90**, à ceci près qu'on ne libère que le dernier secteur.

Si le registre est nul, le programme **0DB0** le signale par *ZR* et l'exécution du saut à l'adresse référencée est faite dans le programme **0B00**, instruction **0B3D**, sur le modèle de ce qui est fait pour l'instruction Minsky **st**.

REMARQUE : De nombreux programmes du simulateur possèdent deux sorties : une sortie en erreur, signalée au retour par *CY*, et une sortie sans erreur signalée par *NC*. Souvent, l'indicateur prend la bonne valeur à la suite d'un test caractéristique dont le résultat est *CY* ou *NC*. Cependant, si entre ce test et la sortie sans erreur, on effectue des opérations secondaires qui ne sont pas source d'erreur dans la logique de notre programme, le résultat de ces opérations peut cependant modifier l'indicateur *CF*. Il convient, dans ce cas, de positionner *CF* sur la valeur correcte juste avant l'instruction **ret**.

5.3.4. L'affichage des résultats.

Lorsque l'exécution du programme se termine sans erreur (les seules erreurs détectées par le simulateur sont les dépassements de capacité), on affiche le contenu de chacun des registres de la machine de Minsky. Ceci pose le problème de conversion inverse de celui que nous avons traité pour l'initialisation des registres de la machine.

La valeur qui se trouve dans un registre peut occuper plusieurs secteurs. Aussi donnerons-nous également une limite à l'affichage des résultats. Nous déclarerons *trop grande* une valeur dont la représentation décimale nécessite plus de 128 chiffres. Pour faciliter les opérations qui ne concernent, pour chaque registre, que quatre secteurs au plus, on commence par recopier la valeur du registre sur une zone de quatre secteurs contigus. Puis on effectue une boucle de divisions par cent (on obtient ainsi les chiffres décimaux par groupes de deux) : chaque division fournit deux chiffres de l'écriture décimale en commençant par le chiffre des unités. On stockera ces chiffres dans une autre zone, les convertissant au passage en caractères, et le calcul s'arrêtera, soit parce que la conversion sera terminée, soit parce qu'ayant déjà utilisé 128 chiffres, elle n'est pas encore terminée.

Nous ne donnons pas, ci-après la boucle des divisions mais simplement l'exécution d'une seule d'entre elles qui représente la division d'un 'grand nombre' écrit en base *b* par un nombre *d* avec $d < b$. Afin de conduire le plus aisément cette division, on considèrera la valeur du registre en base 256 et non 256[2], de sorte que les décimales sont les *octets* composant le registre et non les mots. Pour cette division, on transformera la longueur en mots en une longueur en octets, ce qui nécessite d'examiner le mot de tête : si *lo* est la longueur en octets et *lm* la longueur en mots, $lo = 2*lm$ si le nombre représenté par le mot de tête ne tient pas sur un octet, et $lo = 2*lm - 1$ dans le cas contraire.

---

[1] Penser par exemple au passage de **01000** à **0FFF**.

Voici la partie concernée du programme :

```
0F96 57       PUSH DI              ; sauver valeur initiale de DI
0F97 30E4     XOR  AH,AH           ; au départ, pas de retenue
0F99:BOUCLE                        ; sur le nombre de mots
0F99 4F       DEC  DI              ; position sur l'octet courant
0F9A 8A05     MOV  AL,[DI]         ; lu dans AL
0F9C F6F3     DIV  BL              ; division par cent = 064
0F9E 8805     MOV  [DI],AL         ; écriture (pour la prochaine)
0FA0 E2F7     LOOP 0F99            ; retour à BOUCLE
0FA2 30C0     XOR  AL,AL           ; conversion du reste en .
0FA4 86E0     XCHG AH,AL           ; chiffres décimaux
0FA6 B30A     MOV  BL,0A           ;
0FA8 F6F3     DIV  BL              ;
0FAA 86E0     XCHG AH,AL           ; permutation pour affichage
0FAC 053030   ADD  AX,3030         ; AX devient deux caractères
0FAF 5F       POP  DI              ; valeur initiale retrouvée
```

Le corps de la boucle **0F99** se fonde sur la remarque suivante : si on divise $u.b + v$ par a où $a < b$ (ici $b = 256$ et $a = 100$) et $u < a$, le quotient q vérifie $q < b$ et, de même, le reste r vérifie $r < b$ (puisque $r < a$ et $a < b$).

5.3.5. Le diagnostic des erreurs.

Le lecteur aura déjà remarqué, dans les extraits de programme donnés ci-dessus, les appels au sous-programme **0FD0** assortis du commentaire : appel du gestionnaire des erreurs. Il aura peut-être également remarqué que cet appel n'est précédé d'aucune initialisation particulière de registre ou de variable qui pourrait indiquer à l'utilitaire de quelle erreur il s'agit. C'est qu'il n'est pas besoin de le faire.

Principe.

En effet, dans la perspective d'un diagnostic de *plusieurs* erreurs du programme source, s'il en contient, il est naturel d'affecter à chaque erreur un numéro. Lorsque le gestionnaire d'erreur est appelé, il stocke le numéro dans une file d'attente afin de dresser la liste des erreurs lorsque l'analyse sera terminée. Il affichera alors, pour chaque erreur, la nature probable de celle-ci. Rappelons-nous que l'appel à l'utilitaire a pour effet de placer sur la pile l'adresse de retour de l'exécution une fois que ce programme aura été exécuté. Cette adresse est caractéristique de l'endroit du programme où le gestionnaire d'erreur a été appelé, ce qui caractérise donc l'erreur. Il suffit de prendre cette adresse comme numéro (ce qui ne coûte qu'un mot).

Pour accéder à cette adresse de retour quand on se trouve dans le gestionnaire des erreurs, il suffit d'aller la chercher sur la pile : à l'entrée dans ce programme, cette adresse est le premier mot qui se trouve sur la pile. L'instruction

```
0100      MOV  BP,SP
0102      ADD  BP,02
```

permet d'aller le lire sans altérer la pile puisque le registre *BP* est associé, pour les adresses absolues, au registre de segment *SS*, tout comme le pointeur de la pile, *SP*. En fonction de la valeur lue, le gestionnaire peut éventuellement décider de stocker aussi le numéro de ligne (dans le texte) correspondant à l'erreur. Dans ce but, le programme de saut à la ligne met à jour un compteur de lignes. Il suffit au gestionnaire des erreurs de le consulter à ce moment en cas de besoin.

Dans le programme **0FD0**, l'instruction correspondant à l'instruction **0102** ci-dessus est l'instruction **0FD8** : **add bp,0C**. En effet, ce programme commence par empiler six registres au cas où il devrait rendre la main au programme appelant. L'adresse de retour se trouve donc après ces six mots.

Donnez-moi deux tableaux...

La mise en oeuvre de ce principe utilise deux tableaux *A* et *M*. Les entrées du tableau *A* sont les adresses de retour de tous les appels au gestionnaire des erreurs dans le programme. Les entrées du tableau *M* sont les adresses des messages d'erreur à afficher. La correspondance entre les deux tableaux est faite par l'adresse de l'entrée : si l'adresse d'une erreur *a* est à l'entrée *i* du tableau *A*, l'adresse du message à afficher dans ce cas est à l'entrée *i* du tableau *M*.

En outre, la nécessité d'inscrire le numéro de ligne ou non est donnée par l'adresse elle-même : ceci ne se produit que pour des erreurs décelées au cours de l'analyse syntaxique ; or, la partie correspondante du programme est comprise dans une zone qui ne chevauche aucune autre partie du code. De la même façon, les erreurs qui se produisent avant ou après interdisent l'exécution du programme Minsky.

La recherche de l'adresse dans le tableau *A* utilise les instructions **repne scasw**. L'indicateur *ZF* indique si l'entrée a été trouvée. Ceci est à utiliser pendant la mise au point afin de parer à tout oubli de la part du programmeur. Mais ce point réglé une fois pour toute, le registre important est *DI* car la valeur *DI*-2 fournit l'adresse de l'entrée du tableau et donc, l'adresse de l'entrée du tableau des messages. Pour faciliter les opérations, il y a intérêt à prendre les adresses des entrées depuis le début de chacun des tableaux. La valeur à utiliser de *DI* est alors la même dans les deux cas.

Fin du programme d'erreur.

En cas de retour à l'appelant, on dépile au préalable ce qu'on avait empilé à l'entrée (attention à l'ordre!). S'il y a trop d'erreurs ou si leur gravité interdit l'exécution ultérieure du programme Minsky, on affiche toutes les erreurs trouvées : on efface l'écran, on positionne le curseur (rappel : interruption **010**), on annonce le relevé des erreurs qu'on affiche dans une boucle. Voir en annexe 1 le détail des instructions.

5.4. Assemblage et mise au point.

Le nombre d'instructions de ce programme, environ un millier, rend impossible sa confection, 'd'un seul tenant', sous DEBUG, ou même sous un macro-assembleur. Le 'montage' du fichier *.COM* est réalisé de façon *fractionnée*.

Après la définition du programme principal, qui peut en fait être remanié après l'écriture détaillée des procédures appelées, on procédera par *blocs*, chacun d'eux représentant une étape de l'algorithme de simulation.

Chaque étape est à son tour décomposée en blocs plus restreints du point de vue de l'action effectuée. La taille d'un bloc doit être limitée, à notre sens, à la page de listage de façon à l'avoir globalement sous le regard[1].

A noter que dans ce programme il n'y a pas de procédure récursive.

Assemblage.

Chaque bloc est écrit sous *DEBUG* à l'adresse **0100**, par exemple et ne devrait pas occuper, sauf exception, plus de cent à cent cinquante octets. Sous cette forme, le premier problème est le 'réglage' des adresses. Un peu d'habitude et la connaissance du chapitre 7 permettront de trouver à quelques unités près les bonnes valeurs et, de ce fait,

---

[1] Il ne s'agit pas d'une coquetterie, mais d'un point important : les programmes sont également faits pour les programmeurs.

à limiter le nombre de compilations sous *DEBUG*. Ce nombre, au demeurant, est nécessairement grand. Seule la réflexion méthodique peut l'empêcher de devenir astronomique.

Le second point consiste à faire une première mise au point : vérifier que les j*cond* fonctionnent selon la logique du programme et que, par conséquent, on n'a pas mis **ja** quand il fallait **jbe**[1]. Vérifier également l'équilibre de la pile et que les **loop** ne commencent pas avec un compteur de boucle égal à **0000**.

Une fois cette première façon achevée, il s'agit de placer le bloc dans une unité plus grande, voir à sa place définitive. Le plus simple sous *DEBUG*, est d'utiliser la commande *m*. Par les multiples compilations effectuées, on connaît la taille du bloc à déplacer et l'endroit précis du code où il doit s'insérer. Comme nous l'avons déjà remarqué, cette translation ne nécessite pas la modification des j*cond* ni des **loop** ou **loop***cond*, ni des **jmp** proches. Par contre, les autres **jmp** à adresse immédiate ainsi que la même catégorie de **call** doivent être modifiés. Doivent l'être aussi les références par l'adresse à des variables *locales*.

Ces modifications se feront par la commande *e*. Comme il y a de fortes chances que l'on doive retoucher plusieurs fois à ce qu'on a ainsi fixé, il est conseiller de noter les adresses à corriger dans une ***table de redressement*** que l'on inclura dans le fichier *.CSM*. En voici un exemple avec le programme **0B00** de la page 109.

Le programme source, compilé à partir d'une adresse **0100**, est suivi par une table de redressement et des instructions adéquates pour que *DEBUG* produise en fin de parcours le code demandé :

```
;  table de redressement :
;  en      D+1E       écrire     0B4C
;           +26                   0B54
;           +4D                   0B
;           +4F                   0B
;           +51                   0B
;           +53                   0B
;  calls :
;  en      D+13       écrire     04BB        (0FD0)
;          D+39                  0275        (0DB0)

m 100 L 58 B00
e B1E 4C 0B
e B26 54 0B
e B13 BB 04
e B39 75 02
e B4D 0B
e B4F 0B
e B51 0B
e B53 0B
```

Plusieurs de ces programmes, constituant des zones de code se succédant dans le texte du programme global, peuvent être rassemblés dans un même fichier *.CSM*. Si, par exemple, le premier programme du fichier commence à l'adresse **0B00** (après translation par la commande *m*) et le dernier se termine à l'adresse **0FC0** (adresse du premier octet qui suit le dernier du code), on écrira, pour la commande *w* :

```
r cx
4C0
r bx
0
nmodule3.com
w 0B00
```

[1] L'auteur de ces lignes reconnaît volontiers que cela lui est arrivé plusieurs fois.

Le fichier *.COM* ainsi obtenu commencera effectivement à l'adresse 0100, mais une fois translaté à l'adresse **0B00**, il fournira le code attendu avec les adresses correctes, y compris les adresses de variables, les **call** à adresse immédiate et les **jmp** à adresse immédiate non proche.

Lorsque tous les fichiers de la série 'module' définis à l'image de cet exemple sont constitués, on les regroupe en un fichier *.COM* que nous appellerons **minsky.com** en sollicitant à nouveau *DEBUG*, par fichier *.CSM*, avec les commandes suivantes :

```
nmodule1.com                % chargement du premier fichier
1                           % commande sans paramètre :
m 100 L 510 2100            %   un fichier .COM est toujours
nmodule2.com                %   chargé à l'adresse 0100
1                           %
m 100 L 4E0 2628            % commande m pour recopier le
nmodule3.com                %   morceau de code correspondant
1                           %   à sa place définitive dans
m 100 L 4C0 2B00            %   le programme .COM final
nmodule4.com                % le paramètre de L est toujours
1                           %   pris un peu 'large'
m 100 L 650 2FD0            %
r cx                        % CX est initialisé à la longueur
1510                        %   du programme .COM final
r bx                        % par précaution, BX initialisé
0                           %   à 0000
nminsky.com                 % désignation du fichier à créer
w 2100                      %   adresse du 1er octet source
```

Mise au point.

Avec sous les yeux le listage fourni par les fichiers *.BSM* construits par *DEBUG* et par redirection de la sortie écran, la mise au point est très simple. En combinant trace, pas à pas et exécution jusqu'à un point d'arrêt que l'on peut choisir avec la précision absolue, on dispose d'un outil très efficace pour suivre le déroulement de l'exécution d'un programme.

Nous conseillons la démarche suivante :

Tester les 'petits blocs' au cours de la phase d'élaboration, ainsi que nous l'avons indiqué. Dans cette phase, les fichiers n'étant pas très gros, il est aisé de le faire en conversationnel. Toutefois, si l'on doit effectuer de nombreux tests dans un contexte identique ou variant peu, il peut devenir fastidieux de devoir réentrer au clavier, à chaque nouvel appel de *DEBUG*, les données constituant l'environnement du programme testé. Aussi peut-il être utile de confectionner, dans ce cas, ce qu'il convient de faire à un stade ultérieur : la réalisation du test sous fichier, ce que nous appellerons un ***test différé***. Un exemple très élémentaire est donné au chapitre 4, page 65 : il concerne la sortie du résultat d'un autre programme de point fixe.

Lorsque les blocs ont été testés individuellement, tester le programme assemblé. Pour cela, prévoir plusieurs essais en fonction des étapes de l'algorithme. Dans le cas du simulateur que nous venons d'étudier, on a d'abord mis au point l'analyseur syntaxique. Pour cette étape, on a testé tout d'abord l'efficacité de la procédure **0370** de recherche du début d'une instruction. Puis les programmes de reconnaissance des noms d'instruction puis de la vérification de la correction syntaxique de chaque instruction. On a ensuite testé l'assemblage du programme Minsky en 'code machine'.

Cette étape réalisée, on a vérifié, en conversationnel cette fois, la saisie des arguments à l'écran puis, la correction des algorithmes de conversion. Les instructions INC, NUL et DSZ ont été testées avec des données que probablement aucune exécution réelle ne pourra fournir : certains tests ont porté sur des registres comptant plusieurs

secteurs. Ce n'est qu'après cette étape que les tests de l'exécution du programme Minsky par le simulateur ont été conduits.

Ces tests d'exécutions sont, au demeurant fort instructifs. Un test doit, en principe être effectué sur des données pour lesquelles on connaît le résultat que doit donner le programme. Ainsi, si l'on teste le simulateur, ce doit être sur un programme Minsky que l'on sait correct. Or, est-on toujours sûr de cette correction?

L'indécidabilité de l'arrêt.

Nous savons, d'après les théorèmes vus au chapitre 2, qu'il n'est pas possible de construire un programme qui permettrait de vérifier la correction des autres programmes. Cela ne veut pas pour autant dire qu'aucune vérification n'est possible. Cela signifie simplement que le problème est très complexe[1] et qu'il n'y a pas d'approche globale. Une méthode, au demeurant, est de partir d'éléments connus pour être certains et de progresser vers des éléments plus complexes en s'appuyant sur des démonstrations. Nous avons indiqué l'intérêt de la démarche récursive dans cette perspective.

Dans le problème qui nous concerne, le comportement du simulateur en cours de test peut fort bien être testé sur un programme Minsky supposé correct et qui, en réalité ne l'est pas. En effet, à l'aide de *DEBUG*, on peut obtenir, par test différé, l'exécution pas à pas du programme Minsky : un premier test différé permet d'obtenir le code construit par le compilateur, ce que montre le listage suivant :

commandes à *DEBUG*

```
nminsky.com                    % chargement du simulateur
l                              %   implicitement, à l'adresse 0100
e 260 'mult.msk' 00            % nom du fichier source Minsky
g 190                          %  exécution jusqu'au chargeur du simulateur
g = 1AB 130                    % on saute l'entrée du nom à l'écran
p                              % exécution du programme de transcodage
d 8000 L 100                   % affichage du transcodage
```

on lit, depuis *CS*:**08000** :

```
13E8:8000  00 00 00 03 01 00 34 00-03 02 00 34 00 01 01 00
13E8:8010  01 02 00 00 03 00 03 01-00 34 00 03 02 00 29 00
13E8:8020  01 00 00 01 03 00 02 1B-00 03 03 00 16 00 01 02
13E8:8030  00 02 29 00 01 01 01 01-01 01 01 01 01 01 01 01
```

En remplaçant alors la donnée immédiate dans l'instruction **0B04 cmp si,8000** par l'adresse depuis **08000** de l'instruction Minsky à laquelle on doit s'arrêter, on peut réaliser sur fichier une exécution pas à pas du programme Minsky. Il est alors possible de déterminer qui, du simulateur ou du programme Minsky, est la source de l'erreur et, si les deux sont erronés, de les corriger tous les deux.

Nous verrons au chapitre suivant des exemples de programmes pour lesquels les modalités de vérification seront plus aisées.

---

[1] C'est une complexité *incommensurable* par rapport aux problèmes *décidables*. Dans ce domaine de complexités fondamentalement *irréductibles*, la complexité du problème de la halte est la plus petite. On sait en concevoir d'incomparablement plus grandes...

CHAPITRE 6

APPLICATION A LA THEORIE DES NOMBRES.

La modélisation, par le simulateur construit au chapitre précédent, de la manipulation de nombres entiers arbitrairement grands donne la possibilité d'appliquer les mêmes idées à la confection d'outils destinés à effectuer les opérations de l'arithmétique élémentaire sur ces nombres. Nous obtiendrons ainsi ce qu'il est convenu d'appeler des programmes d'arithmétique multi-précision.

Le champ d'application de tels outils ne se restreint pas à la théorie des nombres. La cryptologie s'est emparée de concepts arithmétiques dont la mise en oeuvre à des fins *pratiques* repose sur l'existence de ces programmes.

Un exemple célèbre : on dispose de techniques assez efficaces pour savoir si un nombre entier donné, même de grande taille, est premier ou non ; par contre, si on donne un nombre assez grand qui s'avère ne pas être premier, le temps de calcul de ses facteurs premiers peut être si grand qu'il sera impossible, *pratiquement*, de les trouver. Il suffit de donner des nombres d'une centaine de chiffres en base dix pour qu'aucun des ordinateurs actuels ne puisse trouver leurs facteurs, même au bout de plusieurs jours de calculs... C'est sur la base de la complexité de ce problème de factorisation des nombres entiers que fonctionne le très célèbre système de codes d'accès à clés publiques utilisé par certains organismes économiques.

Registres et nombres.

Nous repartirons de la notion de registres attribués à la représentation d'un entier naturel en base 256. Mais cette fois, nous considérerons chaque secteur du registre comme une décimale de la représentation du même nombre en base 256^{16}. Fondamentalement, c'est la même opération que le passage de la base deux à la base seize, seule change la *dimension* de la transformation.

Naturellement, nous nous réserverons toujours la possibilité de considérer les registres comme une représentation de leur valeur en base 256. De ce fait, les algorithmes de calculs que nous allons indiquer reposent sur une conception 'à deux étages'. Supposant résolu le problème du calcul des opérations élémentaires sur des nombres à un ou deux chiffres de la base 256^{16}, nous appliquerons aux registres les algorithmes usuels de l'addition, de la multiplication et de la division que nous appellerons ***algorithmes scolaires*** puisque ce sont ceux que l'on enseigne à l'école primaire.

> Ces algorithmes, dont la valeur historique n'échappe à personne, n'en continuent pas moins d'être intéressants. Dans le mode de fonctionnement des machines actuelles, la multiplication scolaire, pour prendre cet exemple, n'est pas l'algorithme le plus performant. On peut faire beaucoup mieux au prix de méthodes sophistiquées de discrétisation faisant appel à la fameuse transformée de Fourier rapide. Cependant, on peut démontrer que sur une machine connexionniste, l'ordre de grandeur du temps de calcul de l'algorithme scolaire de la multiplication est optimal.

Naturellement, le processeur ne nous fournit aucun programme de calcul dans une base 256_{16}. Il nous faut donc, au préalable, développer des utilitaires jouant le rôle des instructions **add, mul, sub** et **div**.

Nous verrons qu'en fait, nous ne construirons que des analogues en base 256_{16} des instructions **adc** et **mul**. Nous ne traiterons pas la soustraction, trop semblable à l'addition[1], et nous appliquerons une méthode un peu différente pour la division. Cependant, l'application des algorithmes scolaires aux super-décimales que sont les secteurs, ou aux octets, dans le cadre des secteurs, cas de la division, est loin d'être aussi évidente qu'il pourrait le paraître à première vue. La difficulté n'est pas non plus insurmontable et le chemin conduisant au résultat mérite d'être emprunté.

Les programmes que nous construirons se trouvent intégralement dans l'annexe 2. Ils peuvent facilement être adaptés à une taille différente des secteurs, notamment à une taille plus grande. Les instructions à transformer sont indiquées.

Espace des registres et organisation des registres.

Au chapitre précédent, nous avons travaillé avec un espace des registres fixe largement suffisant pour les applications concrètes que nous pouvions faire de notre simulateur. Ici, l'espace des registres pourra évoluer de façon dynamique jusqu'à atteindre, s'il le faut, toute la mémoire disponible. Nous supposerons que le plus grand espace mémoire accessible à notre ordinateur est de 1024 Ko, soit seize segments de 64 Ko. De ce fait, avec des secteurs de seize octets, la taille d'un registre pourrait atteindre **0100*01000**, nombre qui s'écrit sur trois octets. Pour faciliter les opérations de manipulation par secteurs, nous mesurerons la taille d'un registre par deux nombres : le nombre des secteurs composant le registre et le nombre de mots dans le dernier secteurs du registre, c'est-à-dire, ce qui reste de la taille du registre en mots lorsque l'on déduit tous les secteurs qui précèdent le dernier. Ainsi, la taille du registre suivant, commençant en **1CD0** et finissant en **1C90** :

```
1C90 00 00 00 00 01 A2 01 00 00 00 00 00 00 00 00 00
1CD0 12 34 56 78 9A BC DE F0 FE DC BA 98 76 54 32 10
```

est définie par les deux nombres **02** et **04**.

Le nombre de segments d'un registre est au plus **010*01000**. Il est en fait nécessairement moins car, l'espace des registres contient la *TAS*. Celle-ci aura un caractère *local* : chaque zone de 64 Ko allouée pour augmenter la taille des registres sera organisée comme l'espace des registres du chapitre précédent. Nous avons alors vu que les numéros de secteur ne nécessitaient en fait que trois quartets. Comme nous autoriserons la possibilité, pour un registre, d'avoir des secteurs sur plusieurs zones, nous utiliserons le quatrième quartet pour numéroter la zone à laquelle le secteur appartient. Comme nous ne pouvons disposer que de seize zones au plus, numérotées de **00** à **0F**, ce quartet suffira. Ainsi **02A31** est le 2154ème secteur de la troisième zone de 64 Ko de l'espace des registres.

Le nombre de secteurs d'un registre ne demande donc pas plus d'un mot. Quant au nombre de mots du dernier secteur, nous lui affectons 'généreusement' un mot. Comme précédemment, le répertoire des registres débute à *CS*:**04000**. La table est complétée par une 'sous-table' débutant à *CS*:**04040** pour le nombre de mots du dernier secteur. Ceci ne laisse que seize registres, ce qui sera en fait suffisant dans les applications que nous verrons. Enfin, les zones elles-mêmes nécessitent deux tables de seize mots chacune, puisque l'espace des registres ne peut en contenir plus de seize. En *CS*:**3800** débute la ***table des adresses*** et en **03820**, la ***table des occupations***. A l'entrée

---

[1] Seule différence, sur le plan informatique : la nécessité de libérer, éventuellement, le dernier secteur alloué au registre. Nous avons traité ce point au chapitre précédent.

d'adresse 2**i* depuis **03820**, correspondant à la *i*+1ème zone (*i* de **00** à **0F**), se trouve le nombre de secteurs libres de cette zone. Ce nombre, que nous appellerons ***taux d'occupation***, est égal à **0E38** lorsque la zone correspondante vient d'être allouée. En 3840, on indique le nombre de zones allouées à l'espace des registres.

Enfin, question de vocabulaire : nous appellerons aussi le dernier secteur du registre ***secteur de tête*** par analogie avec l'expression ***décimale de tête*** qui correspond au coefficient de la plus grande puissance de la base que contient le nombre. Le secteur de tête est le représentant de la décimale de tête lorsqu'on écrit le nombre en base 256_{16}.

6.1. L'addition.

C'est, de loin, l'opération la plus simple. Le programme général repose sur l'addition de deux secteurs, aussi commencerons nous par ce point.

Cas de deux secteurs.

Comme indiqué plus haut, nous définissons l'analogue de l'instruction **adc**. Cette même instruction est utilisée dans une boucle ordinaire comme l'indique le programme 0200 ci-dessous :

Les adresses de début des secteurs sont dans *ES:DI* et *DS:SI* respectivement. Le résultat va dans le secteur pointé par *ES:DI* ; la retenue initiale est dans *CF*.

```
0200 B90800          MOV  CX,0008        ;
0203: BOUCLE
0203 8B04            MOV  AX,[SI]        ; (segment dans DS)
0205 26              ES:                 ;
0206 1305            ADC  AX,[DI]        ; (segment dans ES)
0208 26              ES:                 ;
0209 8905            MOV  [DI],AX        ; écriture
020B 47              INC  DI             ;
020C 47              INC  DI             ;
020D 46              INC  SI             ;
020E 46              INC  SI             ;
020F E2F2            LOOP 0203           ; retour à BOUCLE
0211 C3              RET                 ;
```

On observe que s'il y a une retenue, elle se trouve dans *CF* lorsque l'on sort de la boucle **0203**.

Sur deux registres : l'addition *stricto sensu*

Pour obtenir un programme additionnant deux registres, on pourrait penser qu'il suffit de reprendre le schéma du programme ci-dessus en remplaçant l'instruction **adc** par le programme **0200**, et les instructions d'incrémentation de *SI* et *DI* par des programmes de passage au secteur suivant, ce que nous avons vu au chapitre précédent dans le cadre de la simulation d'une machine de Minsky. Cependant, une difficulté apparaît, qui se retrouve dans l'algorithme mathématique que nous avons donné au chapitre 1. En effet, les nombres à additionner peuvent ne pas être de même longueur. Dans ce cas, on a effectivement l'algorithme ci-dessus tant que le plus petit des deux nombres n'a pas été parcouru. Mais ensuite, s'il y a propagation d'une retenue de 1 'jusqu'au bout', on peut devoir parcourir tout ce qui reste du plus grand nombre comme l'illustre l'exemple suivant en base dix : 197 + 999844 = 1000041. Du point de vue de la programmation, l'implantation de ce point se double d'une difficulté supplémentaire : les deux registres à additionner n'ont pas un rôle symétrique : l'un est uniquement source, l'autre est à la fois source et but, comme dans l'instruction **add**. Ceci signifie que si le but est, au départ, plus court que la source, il faut, à partir d'un certain moment, copier la source sur le but, ce qui, au passage, pose un problème d'allocation de secteurs.

On obtient alors le programme suivant, reproduction de l'utilitaire **04A0** de l'annexe 2 :

```
04A0 51          PUSH CX            ; sauver CX (peut être variable)
04A1 E89CFD      CALL 0240          ; initialisation secteurs courants
04A4 9C          PUSHF              ; sauver retenue (CF = NC)
04A5:BOUCLE1
04A5 9D          POPF               ;
04A6 42          INC  DX            ; mise à jour nombre de secteurs
04A7 E856FD      CALL 0200          ; ADC des secteurs courants
04AA 9C          PUSHF              ; sauver retenue
04AB E802FE      CALL 02B0          ; passer au secteur suivant
04AE 73F5        JNB  04A5          ; -> retour à BOUCLE1
04B0 3C00        CMP  AL,00         ; registre source épuisé?
04B2 752B        JNZ  04DF          ; non : -> cas AL = 01
04B4 9D          POPF               ;  retrouver la retenue
04B5:BOUCLE2
04B5 7311        JNB  04C8          ; pas de retenue : -> cf. longueur
04B7 42          INC  DX            ; mise à jour nombre de secteurs
04B8 E875FE      CALL 0330          ; secteur but suivant
04BB 7205        JB   04C2          ; pas trouvé : -> erreur
04BD E860FD      CALL 0220          ; retenue : +1 au résultat
04C0 EBF3        JMP  04B5          ; retour à BOUCLE2
04C2 59          POP  CX            ; erreur : rééquilibrer la pile
04C3 E80A0B      CALL 0FD0          ; gestionnaire d'erreurs
04C6 F9          STC                ; signal erreur
04C7 C3          RET                ;
04C8 E8A5FF      CALL 0470          ; nombre de mots dernier secteur
04CB 2E          CS:                ;
04CC 8B1E0A0A    MOV  BX,[0A0A]     ; adresse répertoire registre but
04D0 2E          CS:                ;
04D1 89970240    MOV  [BX+4002],DX  ; taille du but (en secteurs)
04D5 D1EB        SHR  BX,1          ; conversion de BX en adresse
04D7 2E          CS:                ; pour le tableau 04040
04D8 88874040    MOV  [BX+4040],AL  ; nbre de mots du dernier secteur
04DC 59          POP  CX            ; CX retrouvé
04DD F8          CLC                ; pas d'erreur
04DE C3          RET                ; retour
04DF: BOUCLE3                       ; source non épuisé mais but oui
04DF 42          INC  DX            ; mise à jour taille
04E0 E84DFE      CALL 0330          ; réserver nouveau secteur but
04E3 72E7        JB   04C2          ; non : -> capacité dépassée
04E5 E878FF      CALL 0460          ; copier secteur source sur but
04E8 9D          POPF               ; rappel retenue
04E9 7303        JNB  04EE          ; nulle : -> secteur suivant
04EB E832FD      CALL 0220          ; non nulle : +1 au résultat
04EE 9C          PUSHF              ; sauver retenue
04EF E82EFF      CALL 0420          ; secteur source suivant
04F2 7202        JB   04F6          ; plus : -> terminé?
04F4 EBE9        JMP  04DF          ; retour à BOUCLE3
04F6 9D          POPF               ; rappel retenue
04F7 73CF        JNB  04C8          ; nulle : -> terminé
04F9 42          INC  DX            ; mise à jour nbre de secteurs
04FA E833FE      CALL 0330          ; réserver dernier secteur but
04FD 72C3        JB   04C2          ; non : -> capacité dépassée
04FF E81EFD      CALL 0220          ; 1 final (+1 à 00)
0502 EBC4        JMP  04C8          ; -> réajustement taille
```

REMARQUE :

Lorsque le nombre de secteurs du registre source est plus grand que celui des secteurs du registre but, la copie que l'on doit faire des secteurs sources sur les secteurs buts, créés au fur et à mesure, revient à simuler, abstraction faite du problème de la retenue, l'algorithme de la commande **copy** mis en oeuvre par *COMMMAND.COM* quand on demande de concaténer un fichier à un second fichier, déjà existant.

La mise au point.

Pour un programme de cette taille et de cette importance, puisqu'il va servir de base au programme qui va suivre de la multiplication, il est impératif d'être certain de son fonctionnement correct. Habituellement, pour acquérir cette certitude, on effectue un certain nombre de tests à partir de données bien choisies du problème. Dans notre cas, la mise en place de données concluantes nécessiterait plusieurs secteurs et serait particulièrement fastidieuse. Un moyen de parvenir à ce but à bon compte consiste à reprendre l'idée d'un ***modèle réduit***. Or, nous avons un modèle réduit fort simple : celui de la base 256. En effet, dans notre interprétation, les secteurs ne sont pas autre chose que les décimales, dans une certaine base b, d'un entier. Nous avons choisi $b = 256^{16}$, mais l'algorithme, s'il est correct l'est pour toute autre base et réciproquement. Si l'on prend $b = 256$, la vérification devient facile. On vérifie le programme ci-dessus en prenant, dans les sous-programmes d'addition des décimales ou de passage à une décimale suivante, des programmes adaptés à la base 256, c'est-à-dire, à la mesure de l'octet.

Exemple pour le programme **0200** (addition de deux décimales) :

```
0200 8A04          MOV  AL,[SI]        ;
0202 1205          ADC  AL,[DI]        ;
0204 8805          MOV  [SI],AL        ;
0205 C3            RET                 ;
```

et pour le programme **0330** (passage à la décimale suivante) :

```
0330 89F0          MOV  AX,SI          ;
0332 3B060010      CMP  AX,[1000]      ; [1000] : taille registre source
0336 7310          JNB  0348           ;
0338 46            INC  SI             ;
0339 89F8          MOV  AX,DI          ;
033B 3B060210      CMP  AX,[1002]      ; [1002] : taille registre but
033F 7303          JNB  0344           ;
0341 47            INC  DI             ;
0342 F8            CLC                 ; sortie sans erreur
0343 C3            RET                 ;
0344 B001          MOV  AL,01          ; erreur n°1
0346 F9            STC                 ;
0347 C3            RET                 ;
0348 30C0          XOR  AL,AL          ; erreur n°0
034A F9            STC                 ;
034B C3            RET                 ;
```

La simplicité du modèle permet de réduire la recherche du secteur suivant ou de l'allocation d'un nouveau secteur à la seule incrémentation d'un registre d'adresse.

Le test peut se faire en mode interactif sous *DEBUG* ou par test différé. On entre les nombres à tester par la commande *e* et on initialise *SI*, *DI* et *IP* par la commande *r*. Les valeurs à placer dans **0212** et **0214** le sont par la commande *e*.

La gestion des secteurs.

Dans le programme 'grandeur nature', on reprend ces mêmes sous-programmes en remplaçant les instructions du modèle réduit par des suites d'instructions ou des programmes correspondant au modèle réel. Nous avons donné plus haut le programme d'addition de deux secteurs. Nous connaissons, d'après le programme de simulation d'une machine de Minsky, le principe de détermination du secteur suivant d'un registre et le mécanisme d'allocation s'il faut agrandir le registre. Toutefois ici, comme l'espace des registres peut être étendu en cours d'exécution du programme à plusieurs segments de 64 Ko, il faut ajouter le mécanisme d'allocation d'une nouvelle zone de 64 Ko si besoin est, assorti de la gestion d'arrêt du programme au cas où l'allocation ne pourrait plus se faire, faute de place en *RAM*. Il faut également prévoir que l'on traite *deux* registres qui peuvent se trouver dans des zones différentes. On prévoit, à cet effet de

mettre l'adresse absolue du secteur courant du registre source dans *DS:SI* et celle du secteur courant du registre but dans *ES:DI* (ce qui facilite, par exemple les opérations de copie). On aura assez souvent à convertir un numéro de secteur lu en *TAS* en adresse absolue ou en adresse en *TAS*, de l'entrée correspondant à ce secteur. Les instructions pour effectuer ces conversions sont données ci-après côte à côte :

Dans les deux cas, le numéro de secteur est dans *BX* et l'adresse absolue fournie dans le couple de registres *DS:SI* :

```
0100 PUSH BX          conversion en     0100 PUSH BX
0101 AND  BX,F000       adresse         0101 AND  BX,F000
0105 MOV  CL,0B         absolue         0105 MOV  CL,0B
0107 SHR  BX,CL         d'abord         0107 SHR  BX,CL
0109 CS:                le segment      0109 CS:
010A MOV  AX,[BX+3800]                  010A MOV  AX,[BX+3800]
010E MOV  DS,AX         DS prêt         010E MOV  DS,AX
0110 POP  SI            puis le         0110 POP  SI
0111 AND  SI,0FFF     déplacement       0111 AND  SI,0FFF
0115 MOV  CL,04                         0115 SUB  SI,01C8
0117 SHL  SI,CL         SI prêt         0119 SHL  SI,1
0119 RET                                011B RET
```

Un certain nombre de variables sont *partagées* par tous les sous-programmes du programme **04A0** d'addition. Ce sont les mots suivants :

```
0A00 :  longueur du registre source
0A02 :  longueur du registre but
0A04 :  n° du secteur initial source (quartet le plus fort à 00)
0A06 :  n° du secteur initial but (quartet le plus fort à 00)
0A08 :  adresse de la zone du registre but dans la table des adresses de zones
0A0A :  adresse en répertoire du registre but
```

Ces variables sont initialisées par le programme **0240**. Elles sont utilisées par les programmes **02B0**, **0330** et **0420** de passage aux secteurs suivants ainsi que le programme **0470** de calcul de la longueur du registre.

Allocation de zone et de secteur.

L'allocation d'une nouvelle zone de 64 Ko se fait lorsque voulant copier un secteur source sur un secteur but, le programme de recherche du secteur but suivant ne trouve plus de secteur à allouer. L'allocation fait appel à la fonction *DOS* **048**, le programme utilisant les utilitaires multiprécision ayant pris soin de réduire son propre espace de travail par la fonction *DOS* **04A**.

L'allocation d'un nouveau *segment* à un registre s'apparente à la recherche d'une place de stationnement dans un parking à plusieurs étages. On regarde tout d'abord s'il reste de la place au niveau où l'on est : c'est le but de la place mémoire **A00+08** où l'on garde l'adresse, dans la table des adresses, de la zone où l'on est. Le tableau des occupations dit immédiatement si cette zone est complète ou non. S'il reste des secteurs disponibles, on les cherche d'abord avec des adresses plus grandes que celle du secteur courant. En cas d'échec, on cherche dans les adresses plus petites et on n'oublie pas de mettre à jour l'entrée concernée de la table des occupations ainsi que la *TAS* de la zone. La recherche d'un secteur libre se fait à l'aide de l'instruction **scasw** préfixée par **repne**, avec *AX* = **0000** puisque telle est la marque d'un secteur libre.

Voir en particulier le programme **0330** pour la mise en oeuvre de ces principes.

Application : l'accès aux grands nombres de Fibonacci.

Ce programme d'addition nous permet d'obtenir des nombres de Fibonacci avec un grand nombre de chiffres.

Pour ce faire, il nous faut tout d'abord adapter le programme de calcul de la fonction de Fibonacci que nous avons vu au chapitre 4. Nous obtenons le programme suivant qui diffère, par une instruction, du programme **0150** de l'annexe 4 :

```
0150 E8BD03        CALL 0510          ; réserver deux registres à 00
0153 31FF          XOR  DI,DI         ; DI : adresse (rép.) 1er registre
0155 BE0400        MOV  SI,0004       ; SI : adresse (rép.) 2me registre
0158 E316          JCXZ 0170          ; si CX = 0, -> fin (Fib(0) = 0)
015A E84300        CALL 01A0          ; +1 à (DI)
015D 83F901        CMP  CX,+01        ; argument égal à 1?
0160 760E          JBE  0170          ; oui : -> fin (Fib(1) = 1)
0162 49            DEC  CX            ; X := X-1
0163 57            PUSH DI            ; permutation entre
0164 56            PUSH SI            ; DI et SI
0165: BOUCLE                          ;
0165 5F            POP  DI            ; (DI) = F(n+1), (SI) = Fn
0166 5E            POP  SI            ;
0167 57            PUSH DI            ; (DI) := Fn
0168 56            PUSH SI            ; (SI) := Fn+1
0169 E83403        CALL 04A0          ; (DI) := Fn+2
016C E2F7          LOOP 0165          ; retour à BOUCLE
016E 5E            POP  SI            ; restitution de SI et DI
016F 5F            POP  DI            ;
0170 C3            RET                ;
```

Comme ce programme n'utilise que deux registres, *DI* et *SI* prennent alternativement les valeurs **0000** et **0004**, de sorte que le résultat est pointé par *DI* = **0000** si *N*, l'argument de la fonction, est impair et par *DI* = **0004** si *N* est pair.

On constate ainsi que, pour l'essentiel, ce programme remplace l'instruction **add** du programme donné au chapitre 4 par un appel au sous-programme **04A0**. Notons qu'ici aussi, la pile est utilisée pour effectuer le 'glissement' de $n, n+1$ sur $n+1, n+2$ en permutant le rôle de *SI* et *DI* qui désignent les deux registres utilisés par ce programme.

L'utilisation de deux registres suppose qu'au départ ceux-ci soient ***initialisés***. C'est le rôle du programme **0510** qui réalise l'***interface*** entre le programme mathématique **0150** et l'organisation des ressources nécessitées par le programme d'addition multiprécision **04A0**. En particulier, la première incrémentation du registre (*DI*), programme **01A0**, est organisée comme suit :

```
01A0 51            PUSH CX            ; CX compteur de boucle
01A1 57            PUSH DI            ; DI : n° du registre
01A2 BF801C        MOV  DI,1C80       ; adresse 1er registre
01A5 E87800        CALL 0220          ; +1 au secteur courant
01A8 5F            POP  DI            ;
01A9 59            POP  CX            ;
01AA C3            RET                ;
```

On note la sauvegarde de *DI* et *CX* qui sont modifiés par le programme **0220**.

L'affichage des résultats.

Dans le programme de simulation, nous avions convenu de n'afficher à l'écran que des résultats dont la longueur ne dépasse pas quatre secteurs. Comme ce programme de simulation a essentiellement un rôle de mise au point, une telle possibilité dépasse largement les besoins réels de l'affichage dans le cadre d'une utilisation concrète de ce programme.

Il n'en est pas de même pour notre programme de Fibonacci et, d'une façon plus générale, pour un programme utilisant la multi-précision. Un tel programme ne sert à rien si on ne peut pas afficher les résultats.

Nous conviendrons toutefois, de n'afficher à l'écran que des résultats n'occupant pas plus d'une quinzaine de secteurs : si le nombre de chiffres affichés dépasse la page

d'écran, ce qu'on obtient est illisible. On conviendra donc que les résultats occupant plus de place seront fournis à l'utilisateur dans *deux* fichiers : un fichier contenant le résultat sous forme hexadécimale brute et un fichier contenant la représentation décimale. Le fichier 'hexadécimal' pouvant ainsi servir à un autre programme de calcul en multiprécision et le fichier 'décimal' pourra être utilisé pour l'*édition* du résultat sous une forme appropriée[1].

Toutefois, dans une première étape de mise au point, on peut se contenter d'un affichage 'sommaire' qui, a priori, ne sera pas limité, sachant que l'on ne demandera pas de nombre de Fibonacci pour un indice supérieur à 65536.

C'est ce que nous supposerons. On peut alors procéder de la façon suivante.

On définit une zone 'tampon' à partir de *CS*:**08000** sur laquelle on recopie le registre résultat, une fois le calcul terminé. Puis, nous inspirant du programme d'affichage du simulateur de machines de Minsky, on stocke, en partant de la fin du tampon et en progressant vers le bas, les décimales, traduites en caractères *ASCII*, au fur et à mesure des divisions par **064** (= cent). Nous avons déjà vu, au chapitre 5 le programme de conversion, car celui que nous utilisons ici est fondamentalement le même. Nous donnons, page ci-contre, le programme de copie sur la zone tampon :

Dans ce programme, où la copie est écrite depuis **08000**, *DI* contient au départ, l'adresse, dans le répertoire, de l'entrée associée au registre à copier. Puis *DS*:*DI* est le pointeur d'écriture avec, ici, *DS* = *CS* et *DI* initialisé à **08000**. On écrit dans la même zone le nombre en hexadécimal, depuis **08000**, adresses croissantes, et sa transcription décimale, depuis **0F7FA**, adresses décroissantes. On ne part pas depuis **0FFFE** afin de garder une place (confortable) à la pile puisque le programme comporte des instructions **push** et **pop**.

On obtient, par exemple, l'affichage suivant :

```
Entrer N
   > 2000
Fib (2000) =
422469633339230487870672560234148278257985284025068109801028013731430858437013070722412359963914151108844608753890960360764019471164359602927198331259873732625355580260699158591522949245390499872225679531698287448247299226390183371677806060701161549788671987985831146887087626459736908672288402365442229524334796448013951534956297208765265606952980649984197744872015561280266540455417171788193032402520431208251681712 5
```

6.2. La multiplication.

L'algorithme de la multiplication que nous considérerons ici reprend également l'algorithme scolaire du calcul de cette opération en base *b* quelconque et s'appuie essentiellement sur l'algorithme de l'addition.

Il nous faut, cependant introduire de nouvelles opérations :

la multiplication de deux secteurs,
la multiplication d'un registre par un secteur.

---

[1] Ce que ferait, par exemple un programme de calcul des *x* premières décimales du nombre π.

Programme de copie du résultat sur le tampon :

```
05A0 2E          CS:                ; DI : adresse (rép.) du registre
05A1 8B850040    MOV  AX,[DI+4000]  ; à copier, AX : n° 1er secteur
05A5 31ED        XOR  BP,BP         ; BP : compte les secteurs copiés
05A7 BF0080      MOV  DI,8000       ; on recopie depuis 8000
05AA 0E          PUSH CS            ;
05AB 07          POP  ES            ; ES pointe sur le code
05AC:BOUCLE                         ;
05AC 89C2        MOV  DX,AX         ; conversion du n° de secteur lu
05AE 81E200F0    AND  DX,F000       ; en adresse DS:SI du secteur
05B2 25FF0F      AND  AX,0FFF       ; source à copier
05B5 50          PUSH AX            ;
05B6 B10B        MOV  CL,0B         ;
05B8 D3EA        SHR  DX,CL         ;
05BA 89D3        MOV  BX,DX         ;
05BC 2E          CS:                ;
05BD 8B870038    MOV  AX,[BX+3800]  ;
05C1 8ED8        MOV  DS,AX         ;
05C3 58          POP  AX            ;
05C4 50          PUSH AX            ;
05C5 B104        MOV  CL,04         ;
05C7 D3E0        SHL  AX,CL         ;
05C9 89C6        MOV  SI,AX         ;
05CB B90800      MOV  CX,0008       ; mise en place MOVSW pour
05CE FC          CLD                ; huit mots
05CF F3          REPZ               ;
05D0 A5          MOVSW              ;
05D1 45          INC  BP            ;
05D2 58          POP  AX            ; rappel n° de secteur
05D3 2DC801      SUB  AX,01C8       ; conversion en
05D6 D1E0        SHL  AX,1          ; adresse TAS
05D8 89C6        MOV  SI,AX         ;
05DA 3E          DS:                ;
05DB 8B04        MOV  AX,[SI]       ; lecture entrée TAS
05DD 3D0001      CMP  AX,0100       ; dernier secteur?
05E0 7402        JZ   05E4          ; oui : copie terminée
05E2 EBC8        JMP  05AC          ; non : -> retour à BOUCLE
05E4 C3          RET                ; (AX n° du secteur à copier)
```

Ces deux opérations correspondent, pour la première, à l'instruction **mul** de type mot, et, pour la seconde, à une des multiplications du multiplicande par une décimale du multiplicateur dont la somme fournit le résultat de la multiplication.

Nous verrons tout d'abord le programme de la multiplication de deux secteurs, puis celui de la multiplication d'un registre par un secteur pour terminer par le produit de deux registres.

Multiplication de deux secteurs.

On a déjà pu remarquer que le résultat d'une instruction **mul** de type octet est un mot et que le résultat d'une instruction **mul** de type mot est représenté sur deux mots (*DX:AX* comme nous l'avons vu). Il s'agit d'un résultat général : $(b\text{-}1)^2 = b^2 - 2b + 1$ qui nécessite donc au plus deux chiffres en base *b*. Par conséquent, le produit de deux nombres représentés chacun par un secteur est un nombre dont la représentation demande au plus deux secteurs.

Regardons de plus près la relation : $(b\text{-}1)^2 = b^2 - 2b + 1$. Comme on en tire aussitôt $(b\text{-}1)^2 = (b\text{-}2).b + 1$, il en résulte que la seconde décimale du produit *a.c* avec $0 \leq a,c \leq b$, et que nous appellerons **retenue mul**, est au plus égale à $b\text{-}2$.

Rappelons brièvement l'algorithme du produit d'un nombre quelconque *M* par un nombre naturel *m* inférieur à la base *b* considérée.

Si $M = a_k b^k + ... + a_0$ avec $0 \leq a_i < b$ pour $i \doteq 0,...,k$ et si m vérifie $0 \leq m < b$, on obtient : $M.m = a_k.mb^k + ... + a_0.m = u_{k+1}b^{k+1} + ... + u_0$ avec $0 \leq u_i < b$ pour $i = 0,...,k+1$, les u_i étant obtenus de la façon suivante : si $a_i.m = r_i b + s_i$ (s_i, produit faible et r_i produit fort ou retenue mul), et si $s_i + r_{i-1} = e_i.b + v_i$, avec $e_i = 0$ ou 1, alors : $u_i = v_i + e_{i-1}$ pour $i = 0,...,k+1$ car $r_i \leq b-2$; on convient que les nombres r_{-1} et e_{-1} des relations précédentes sont nuls. Ceci peut se résumer par la figure suivante fondée sur la numération de position :

a_k	a_i	a_1	a_0	multiplicande	
			m	*	multiplicateur
s_k	s_i	s_1	s_0	produit faible	
r_k	r_{k-1}	r_{i-1}	r_0	r_{-1}	produit fort
e_k	e_{k-1}	e_{i-1}	e_1	e_{-1}	retenue 'ADC'
u_{k+1}	u_k	u_i	u_1	u_0	résultat

Le programme ci-après traduit, en langage machine l'algorithme consistant à effectuer le calcul des u_i en prenant la somme de s_i, r_{i-1} et e_{i-1}, engendrant la retenue e_i, s_i et r_i étant, quant à eux, définis par le produit $a_i.m$.

Le calcul se décompose comme précédemment : on multiplie le secteur multiplicande par une décimale du multiplicateur (en l'occurrence, un mot) et ce produit est additionné sur le résultat, mais avec un décalage correspondant à la position de la décimale dans le secteur multiplicateur. On pourrait obtenir une programmation simple de ces opérations en copiant multiplicateur et multiplicande sur des zones adéquates du programme et calculer le résultat sur une zone du programme occupant deux secteurs contigus. Il faudrait alors copier ce résultat sur les secteurs de destination. Il en résulte une perte de temps certaine. On peut l'éviter au prix d'une complication de la programmation afin de gérer l'écriture, sur les secteurs destinataires, des résultats partiels de multiplication par une décimale et d'addition correspondante avec décalage.

Pour effectuer ce calcul, le programme utilise une zone de la mémoire divisée en mots qui représentent autant de variables *partagées* avec les autres programmes concourant à la multiplication de deux registres :

`[A26]:[A24]` : secteur source multiplicande
`[A2A]:[A28]` : secteur source multiplicateur
`[A2E]:[A2C]` : secteur but faible
`[A32]:[A30]` : secteur but fort

Le programme initialise ainsi *ES:DI*, *DS:SI* et d'autres combinaisons de registres de segment et de registres généraux, pour calculer le produit du secteur multiplicande par la décimale courante puis pour ajouter ce résultat partiel, convenablement décalé, au couple de secteurs résultat.

Nous ne donnons que des extraits de ce programme qui, pour raison d'efficacité ne comporte pas d'appel à d'autres programmes et nécessite ainsi près de 0100 octets. En voici la structure et quelques extraits, le programme complet se trouvant à l'annexe 2 :

On initialise à zéro les secteurs but :

```
051D 2E            CS:                 ;
051E C43E2C0A      LES  DI,[0A2C]      ; ES:DI pointe sur but faible
0522 B90800        MOV  CX,0008        ; huit mots
0525 31C0          XOR  AX,AX          ;
0527 FC            CLD                 ;
0528 F3            REPZ                ; répétition inconditionnelle
0529 AB            STOSW               ; avec STOS
```

Remarquer l'utilisation de l'instruction **les** ainsi que l'instruction **stosw** associée au pointeur *ES:DI*.

Puis, avant la multiplication proprement dite, on teste si l'un des deux secteurs source est nul :

```
052A 2E           CS:                ;
052B C43E2C0A     LES  DI,[0A2C]     ; ES:DI : sur but faible
052F 2E           CS:                ;
0530 C536240A     LDS  SI,[0A24]     ; DS:SI : sur multiplicateur (ici)
0534 B90800       MOV  CX,0008       ; huit mots
0537 FC           CLD                ;
0538 F3           REPZ               ; répéter tant que
0539 A7           CMPSW              ; ES:WO [DI] = DS:WO [SI]
053A 7502         JNZ  053E          ; multiplicateur non nul
053C C3           RET                ; multiplicateur nul : retour
```

Ici on notera la double association *ES:DI* et *DS:SI* permettant d'utiliser l'instruction **cmpsw**, car *ES:DI* pointe sur un secteur but qui vient d'être initialisé par des zéros. Le secteur testé est donc nul si et seulement si le test de comparaison avec **jnz** est négatif auquel cas la multiplication des deux secteurs est terminée puisque le but se trouve à zéro comme il se doit.

Le produit du multiplicande par une décimale du multiplicateur est effectué dans une zone tampon contenue dans le programme à l'adresse *D*+**0E8** où *D* est l'adresse du premier octet du programme. Ceci est fait dans une boucle **loop** ordinaire sur huit mots. Attention à la retenue : il y a celle qui provient de la multiplication de type mot ($DX \neq 0$) et celle qui provient de l'addition de la retenue précédente à la décimale au pas précédent de la boucle. D'où l'utilisation de l'instruction **adc** :

On a initialisé *DS:SI* par **lds** depuis **0A24**, *ES* depuis **0A2A** et *DI* depuis **0A28**, puis *DX* à zéro, *BP* qui pointe sur la décimale multiplicatrice par *BX+DI*, *CX* par **08** (huit mots) et *CF* par *NC*, puisqu'au départ la retenue est nulle.

```
0571:BOUCLE
0571 8B04         MOV  AX,[SI]       ;
0573 89D3         MOV  BX,DX         ; BX devient retenue précédente
0575 9C           PUSHF              ; annuler effet de MUL sur (F)
0576 F7E5         MUL  BP            ; BP : décimale active
0578 9D           POPF               ; (DX : retenue créée par MUL)
0579 11D8         ADC  AX,BX         ; AX := AX + 'ex'DX
057B 2E           CS:                ;
057C 8905         MOV  [DI],AX       ; écriture du résultat sur tampon
057E 46           INC  SI            ;
057F 46           INC  SI            ;
0580 47           INC  DI            ;
0581 47           INC  DI            ;
0582 E2ED         LOOP 0571          ; retour à BOUCLE
0584              ADC  DX,+00        ; retenue dernière décimale
```

L'addition du contenu de la zone tampon au résultat se fait en deux temps. D'abord, ce qui s'additionne sur le secteur but faible, puis sur le secteur but fort. Comme on sait, la multiplication par la *ième* décimale correspond à la multiplication par cette décimale suivie d'une multiplication par b^{i-1}, où *b* est la base considérée. Dans notre situation, rappelons que la base est 256^2, soit 65536. Or, le décalage correspond exactement à la valeur de *BX* dans l'instruction **0567 mov bp,[bx+di]**, où *DI* a reçu l'adresse du premier octet du secteur multiplicateur. Il suffit de sauvegarder cette valeur jusqu'au moment où on l'utilisera pour l'addition. C'est pourquoi cette boucle est encadrée par une 'parenthèse' **push bx/pop bx**. Voir annexe 2, programme **0510**.

Lorsqu'on en est à l'addition de ce produit sur les secteurs buts, on procède d'abord sur le secteur but faible pour **010** - *BX* octets soit (**010**-*BX*)/2 mots, puisqu'on utilise les instructions **adc** de type mot et on ajoute la valeur de *BX* ainsi déterminée à *DI*, cependant que *CS:SI* pointe sur le tampon propre au programme, voir instructions **0587-059A** du programme **0510**.

Il convient de préciser comment on gère la dernière retenue créée par la dernière instruction **mul** de type mot. On conserve cette retenue dans *DX* jusqu'à la fin de l'addition du tampon sur les secteurs but. Mais comme au moment où on utilisera cette

retenue il faudra également tenir compte de la retenue engendrée par la dernière instruction **adc** utilisée, on incorpore cette dernière retenue à *DX* par l'instruction **adc dx,00** que l'on placera à l'issue de la boucle listée à la page 127 (instruction **0584**). En effet, comme nous l'avons noté, la retenue engendrée par l'instruction **mul** de type mot vaut au plus **0FFFE** et on peut donc lui ajouter 1 sans risque de débordement. La même propriété de $(b\text{-}1)^2$ nous assure que les seize mots des secteurs but seront toujours suffisants pour recueillir le résultat.

Multiplication d'un registre par un secteur.

Ce programme reprend presque mot pour mot, mais 'à grande échelle', la partie du programme précédent effectuant le produit d'un secteur multiplicande par une décimale.

Deux nouveautés apparaissent : le nombre des secteurs du registre n'est pas limité : on reprendra le sous-programme de recherche du secteur suivant de l'addition de deux registres. On doit copier le registre multiplicande sur un registre 'tampon'. Pour ce faire, l'appel à la 'grande multiplication' réservera un registre à cet usage. A chaque appel, de la multiplication registre*secteur, on allouera pour ce registre autant de secteurs qu'en comporte le registre multiplicande. Comme on aura réservé de la place à l'étape précédente, les mêmes places serviront tout le temps de la grande multiplication. Par ailleurs, le rôle de la retenue de **mul** sera joué par une zone tampon du programme registre*secteur.

Ce programme est le programme **0610** de l'annexe 2. La boucle correspondant à la boucle **0571** précédente n'est plus une boucle **loop**. Le retour est effectué par un **jmp** et le test de sortie est constitué par la lecture, en *TAS*, du numéro du secteur suivant :

```
0699 2E              CS:              ;
069A C536240A        LDS  SI,[0A24]   ; on cherche à présent
069E B104            MOV  CL,04       ; le prochain secteur de (SI)
06A0 D3EE            SHR  SI,CL       ; on convertit SI
06A2 81EEC801        SUB  SI,01C8     ; en adresse TAS
06A6 D1E6            SHL  SI,1        ;
06A8 8B04            MOV  AX,[SI]     ; on lit l'entrée TAS retrouvée
06AA 3D0001          CMP  AX,0100     ; dernier secteur?
06AD 7449            JZ   06F8        ; oui : -> sortie de boucle (fin)
```

Le programme **0610** utilise les mêmes variables partagées que le programme **0510** et initialise *ES:DI* et *DS:SI* depuis ces variables par des instructions **les** et **lds**. La boucle gère également plusieurs retenues : la retenue **mul** qui, cette fois, occupe un secteur et la retenue **adc** éventuellement provoquée par l'appel au programme **0200**. Au départ de chaque boucle, la retenue **mul**, partie haute du résultat déjà porté dans le registre, est copiée sur une partie du tampon prévue à cet effet afin de pouvoir être additionnée au produit calculé par le programme **0510**. Les instructions associées à cette copie sont reproduites sur le listage ci-contre. Voir l'annexe 2 pour la place de cet extrait, ainsi que du précédent, dans le programme **0610**.

Application : la factorielle d'un nombre (presque) sans restriction.

Le programme précédent nous permet de programmer le calcul exact de $n!$ pour des valeurs relativement grandes de n : on se contentera, en fait, de la limite $n < 65536$, qui sera suffisante pour cette démonstration.

Ce programme reprend, pour son cadre général, celui du programme de la suite de Fibonacci. Outre l'emploi de la multiplication d'un registre par un secteur, ce programme fait appel à une utilisation particulière de la pile, dont une partie va fonctionner comme ressource partagée par les procédures du programme principal. La zone correspondante sert au contrôle de la retenue **mul** produite éventuellement par le programme **0610** de multiplication d'un registre par un secteur et à la gestion des

```
0670 9C           PUSHF              ; retenue ADC sauvée
0671 2E           CS:                ;
0672 C536300A     LDS  SI,[0A30]     ; DS:SI : secteur fort
0676 2E           CS:                ; (retenue MUL)
0677 C43E300A     LES  DI,[0A30]     ; on recopie cette retenue sur le
067B 83C710       ADD  DI,+10        ; tampon prévu à cet effet
067E B90800       MOV  CX,0008       ; (instruction MOVSW préfixée
0681 FC           CLD                ; par REP)
0682 F3           REPZ               ;
0683 A5           MOVSW              ;
0684 E889FE       CALL 0510          ; multiplication de deux secteurs
0687 2E           CS:                ;
0688 C43E2C0A     LES  DI,[0A2C]     ; ES:DI pointe sur le secteur but
068C 2E           CS:                ; faible
068D C536300A     LDS  SI,[0A30]     ; DS:SI pointe sur l'ex retenue
0691 83C610       ADD  SI,+10        ; engendrée par MUL
0694 9D           POPF               ; rappel retenue ADC
0695 E868FB       CALL 0200          ; ADC deux secteurs
0698 9C           PUSHF              ; sauver retenue ADC
```

adresses des deux registres utilisés pour calculer n! désignés, dans ce qui suit par (*SI*) et (*DI*) en raison du registre utilisé pour conserver cette adresse.

On garde sur la pile le dernier secteur du registre source (*SI*) ainsi que le dernier secteur du registre (*DI*) sur lequel le programme **0610** écrit le résultat du produit. Au passage suivant de la boucle, les rôles de (*SI*) et (*DI*) sont échangés, puisque le programme **0610** effectue la multiplication *sur un autre registre*. Ceci a pour effet de simplifier le déroulement du programme **0610** : la réservation d'un nouveau secteur se produit au plus une fois, lors de la multiplication de la décimale de tête du multiplicande par le secteur multiplicateur.

Le calcul de n! choisi ici est, évidemment, itératif, faisant appel à une boucle **loop** chapeautée par une instruction **jcxz**.

Le programme de calcul de n! se trouve à l'annexe 5. Voici, ci-après la boucle principale du programme à comparer avec le programme du chapitre 4 :

```
015A E326         JCXZ 0182          ; si CX = 0, fini
015C 51           PUSH CX            ; sauver le compteur (N)
015D 56           PUSH SI            ; sauver l'adresse des registres
015E 57           PUSH DI            ;
015F 55           PUSH BP            ;
0160 E8AD04       CALL 0610          ; (DI) := (SI)*[0A2A]:[0A28]
0163 5D           POP  BP            ;
0164 E8E106       CALL 0848          ; dernière retenue MUL non nulle?
0167 740B         JZ   0174          ; nulle : -> suite
0169 E8F406       CALL 0860          ; non : ajouter retenue à (DI)
016C 7306         JNB  0174          ;       et étendre à (SI)
016E 59           POP  CX            ;
016F E85E0E       CALL 0FD0          ; non exécuté : erreur
0172 F9           STC                ;
0173 C3           RET                ;
0174 2E           CS:                ; exécuté :
0175 C43E280A     LES  DI,[0A28]     ; préparation 0220
0179 E8A400       CALL 0220          ; +1 au S. multiplicateur
017C 90           NOP                ;
017D 5E           POP  SI            ; (SI) <-> (DI)
017E 5F           POP  DI            ; (dernier résultat dans SI)
017F 59           POP  CX            ; compteur retrouvé
0180 E2D8         LOOP 015A          ; retour à 015A
```

Par contre, la permutation des registres *SI* et *DI* pose le problème de savoir à quel registre *réel* sont associées les informations pointées sur la pile par *BP*. Afin de gagner en temps d'exécution, on n'effectuera de permutation de ces informations (**0860**

consulte le début de la zone réservée censée informer sur (*DI*)) que si *DI* ne correspond pas au registre attendu dans wo [*BP*]. Un problème analogue se pose en fin de calcul pour savoir quel registre *réel* contient le résultat afin de compter sa taille en mots, ce qui est indispensable à la conversion précédant l'affichage.

Les instructions ci-après concernent la rotation des informations sur la pile. Elles sont extraites du programme **0860** (cf. annexe 5) :

```
0860 8B46FA          MOV  AX,[BP-06]      ; on lit DI sur la pile
0863 3D0800          CMP  AX,0008         ; comparé à la valeur actuelle.
0866 7428            JZ   0890            ; égalité : on ne fait rien
0868 2E              CS:                  ; différence :
0869 A06408          MOV  AL,[0864]       ; on réécrit l'instruction 0863
086C 2E              CS:                  ; en vu du prochain passage
086D 8A267E09        MOV  AH,[097E]       ; on charge la bonne valeur
0871 2E              CS:                  ; stockée en 097E
0872 88266408        MOV  [0864],AH       ; et l'échange avec la valeur
0876 2E              CS:                  ; actuelle en 0864
0877 A27E09          MOV  [097E],AL       ;
087A 55              PUSH BP              ; on sauve BP
087B B90600          MOV  CX,0006         ; pour six mots
087E:BOUCLE                               ;
087E 8B5E00          MOV  BX,[BP+00]      ; on échange
0881 8B460C          MOV  AX,[BP+0C]      ; [BP] et [BP+0C],
0884 894600          MOV  [BP+00],AX      ; BP allant de BP0 (adr. sauvée)
0887 895E0C          MOV  [BP+0C],BX      ; à BP0+0A inclusivement
088A 45              INC  BP              ;
088B 45              INC  BP              ;
088C E2F0            LOOP 087E            ; retour à BOUCLE
088E 5D              POP  BP              ; BP restitué
0890: adjonction d'un secteur à DI (cf. annexe 5)
```

La mise en place de la zone sur pile est effectuée dans le programme **0740** après l'allocation d'une première zone de 64 Ko pour l'espace des registres. Aux instructions concernées correspondent des instructions de 'désinstallation' dans le programme **0830**. Voici ces deux séquences d'instructions en regard l'une de l'autre :

```
075E 89E5      MOV  BP,SP          0834 4D        DEC  BP
0760 83ED18    SUB  BP,+18         0835 4D        DEC  BP
0763 FA        CLI                 0836 8B4600    MOV  AX,[BP+00]
0764 89EC      MOV  SP,BP          0839 894618    MOV  [BP+18],AX
0766 FB        STI                 083C 89E5      MOV  BP,SP
0767 8B4618    MOV  AX,[BP+18]     083E 83C518    ADD  BP,+18
076A 894600    MOV  [BP+00],AX     0841 FA        CLI
076D 45        INC  BP             0842 89EC      MOV  SI,BP
076E 45        INC  BP             0844 FB        STI
```

Le parallélisme de l'incrémentation finale de *BP* dans **0740** et de la décrémentation initiale de *BP* dans **0830** correspond au déplacement des adresses retour de **0740** dans le premier cas et **0830** dans le second : lorsque l'exécution rencontrera l'instruction **ret**, il faut que *SP* pointe effectivement sur l'adresse retour. Noter l'encadrement de **mov sp,bp** par le couple d'instructions **cli/sti** afin de ne pas perturber la valeur de *SP* par d'éventuelles interruptions.

Le programme **0860** d'extension du registre résultat (*DI*) par un secteur destiné à recueillir la retenue MUL non nulle reprend nombre d'instructions données dans le programme **0330** déjà vu dans l'algorithme de l'addition multi-précision. Voir l'annexe 5 pour le détail des modifications apportées pour l'adaptation à ce contexte. On notera, en particulier l'utilisation d'instructions similaires à l'exemple suivant :

```
08DA C57608          LDS  SI,[BP+08]
```

Ces instructions s'emploient sans préfixage puisque *BP* est associé, implicitement, au registre de segment *SS*.

Pour la mesure de la taille, voir le programme **0980** de l'annexe 5.

On obtient, avec le même programme d'affichage que pour les nombres de Fibonacci (avec une simple modification de présentation de la fonction que nous laissons à la sagacité du lecteur), l'exemple suivant :

```
Entrer N
   > 200
200! =
78865786736479050355236321393218506229513597768717326329474253
32443594499634033429203042840119846239041772121389196388302576
42790242637105061926624952829931113462857270763317237396988943
92244562145166424025403329186413122742829485327752424240757390
32403212574055795686602260319041703240623517008587961789222227
89623703897374720000000000000000000000000000000000000000000000
000
```

Multiplication de deux registres.

Pour la bonne règle, nous donnons, ci-après, quelques extraits du programme de multiplication d'un registre par un autre.

Ce programme reprend le principe de la gestion par la pile des registres et secteurs 'courants' sur lesquels s'appliquent les opérations, comme nous venons de le voir pour le programme du calcul de n!. Les variables partagées ainsi dégagées sont affectées de la façon suivante :

```
[BP+00] :                adresse répertoire du registre auxi iaire
[BP+04]:[BP+02] :        premier secteur du registre auxiliaire
[BP+08]:[BP+06] :        dernier secteur '610' du registre auxi iaire
[BP+0C]:[BP+0A] :        adresse TAS du même
[BP+10]:[BP+0E] :        secteur suppl. du registre auxi iaire
[BP+14]:[BP+12] :        adresse TAS d'icelui
[BP+16] :                son n° TAS complet
[BP+1A]:[BP+18] :        premier secteur en décalage
[BP+1E]:[BP+1C] :        adresse TAS du même
[BP+20] :                son n° TAS complet
[BP+24]:[BP+22] :        secteur courant du multiplicateur
[BP+28]:[BP+26] :        adresse TAS du même
[BP+2E] :                SI initial
[BP+2C] :                BX initial
[BP+2A] :                DI initial
```

On notera la place relativement importante prise par ces variables. Ceci s'explique par le fait que ce sont, le plus souvent des adresses absolues que l'on place sur pile afin de les charger sur les registres adéquats par des instructions **les** ou **lds**.

Dans le programme de contrôle de la multiplication à deux registres, le programme **07A0**, la pile est mise en place et démantelée par ce sous-programme lui-même. Cependant, l'initialisation de la pile est dévolue à un autre sous-programme, le programme **07F0** qui met en place et initialise le registre auxiliaire. Ce registre, noté AUX ci-après et d'adresse répertoire [*BP*+**00**] est destiné à recueillir le résultat du produit partiel effectué par le programme **0610**. Enfin, on a noté AUX_d le produit partiel 'décalé' pour l'addition de ce nombre au résultat. On trouvera, page 133, la boucle principale du programme **07A0** (texte complet à l'annexe 2).

Ce programme utilise le programme d'addition **04A0** sous une forme modifiée. L'entrée dans ce programme ne se fait pas par l'adresse usuelle, mais à une adresse plus lointaine, car l'initialisation est différente de celle qui est attendue de façon 'normale' par le programme **04A0**. Comme, en outre, une sauvegarde sur la pile a lieu dans **04A0** *avant* cette seconde entrée, on n'effectue pas l'appel par une instruction **call**, mais de la façon suivante. Après avoir initialisé les variables de l'addition (zone [**A00**]–[**A10**]), on

place sur la pile l'adresse de retour dans le programme **0950** (soit, ici **09C8**), et on met sur la pile *CX* comme cela est fait dans **04A0** avant le second point d'entrée. On initialise *DX* (pour tenir compte du décalage) et *CF* (retenue **ADC**) et on 'appelle' le programme d'addition au second point d'entrée par une instruction **jmp** :

```
09BA B8C809          MOV  AX,09C8          ;
09BD 50              PUSH AX               ;
09BE 51              PUSH CX               ;
09BF 2E              CS:                   ;
09C0 8B16F809        MOV  DX,[09F8]        ;
09C4 F8              CLC                   ;
09C5 E9DCFA          JMP  04A4             ;
09C8 7306            JNB  09D0             ;
```

Lorsqu'on arrivera à la fin du programme **04A0** (voir texte en annexe 2), on dépilera *CX* et le **ret** terminal fera appel à ce qu'il y aura alors sur la pile, à savoir l'adresse **09C8** soit, comme on le voit ci-dessus, la suite du programme **0950**.

Le test en **09C8** indique que l'addition s'est déroulée normalement, la seule erreur possible, dans ce cas, étant le dépassement de capacité.

Le texte du programme **07F0** contient deux instructions (**0947-0948**) dont la fonction est de ***réécrire*** l'instruction **0486** du programme **0470** de détermination de la longueur, en mots, du dernier secteur. Dans une addition normale, lorsqu'il y a plusieurs secteurs, le dernier, qui correspond à la décimale de tête, n'est, par construction, jamais nul. Le programme vérifie cette condition pour s'assurer que l'addition s'est bien déroulée. Dans notre cas, si le produit partiel est nul (cas d'un 'chiffre' nul dans le multiplicateur), il faut neutraliser ce test qui prévoit l'arrêt du programme en cas de réponse négative. C'est la raison d'être de deux ordres de réécriture : dans le programme **07F0**, pour neutraliser le test, et dans le programme **0A40**, après la multiplication, en quelque sorte, pour le rétablir.

Pour terminer sur ce point de la multiplication, un mot sur les méthodes de vérification. Comme indiqué au chapitre 4, il suffit de mettre en place l'environnement attendu du programme dans un fichier soumis à *DEBUG* par le moyen de la redirection. Voici des extraits du fichier résultat produit par *DEBUG* dans le cas suivant, où pour économiser l'espace, des commentaires *a posteriori* ont été introduits ; ils sont signalés par le symbole **%**.

Les nombres initialisés sont à la base 2^{128} ce que 999 et 99 sont à la base 10. Le résultat correspond, bien évidemment, à 98901 :

```
0100 ;  test de la multiplication de deux registres
0100 ;
-nmulreg.com
-l
-f 3000 L 8000 00              % mise à zéro d'un espace de travail
-f 3000:00 L 8000 00           % espace des registres (simulation
                               % d'une allocation
-e 3800 00 30                  % initialisation de la table 3800
-e 3820 38 0E
-e 3840 01                     % puis, ci-dessous, du répertoire
-e 4000 C8 01 03 00 CB 01 02 00 CD 01 01 00
-e 4040 08 00 08 00 01 00
                               % installation de la TAS
-e 3000:00 C9 01 CA 01 CB 01 00 01 CD 01 CE 01 00 01 00 01
-f 3000:1C80 L 50 FF           % initialisation des secteurs
```

On initialise ensuite les registres du processeur : *SI*, *DI* et *BX* par des commandes *r* afin d'initialiser correctement le contexte de lancement du programme **07A0**. On initialise ensuite *IP* à **07A0** et l'exécution est lancée par la commande *g* **7EA**, **07EA** étant l'adresse du **ret** terminal du programme.

On affiche ensuite l'état du répertoire, de la *TAS*, et le résultat, par l'affichage du début de l'espace des registres à partir de l'adresse **01C80** à l'aide de la commande *d*.

Le programme commence par sauver les adresses répertoire des registres utilisés et par installer la pile. En sortie de boucle, la pile est démantelée et les adresses répertoires dépilées sur les registres *DI*, *BX* et *SI*.

```
07AC E84100          CALL 07F0            ; initialisation des variables
07AF 8B5E2C          MOV  BX,[BP+2C]      ; BX := adr. répertoire de m
07B2 2E              CS:                  ;
07B3 8B8F0240        MOV  CX,[BX+4002]    ; CX := |m| (nombre de mult.)
07B7 8B762E          MOV  SI,[BP+2E]      ; SI := adr. répertoire de M
07BA 8B7E00          MOV  DI,[BP+00]      ; DI := adr. répertoire de AUX.
07BD:BOUCLE                               ; produits partiels
07BD 56              PUSH SI              ; sauver les paramètres utilisés
07BE 57              PUSH DI              ; dans la boucle, notamment son
07BF 51              PUSH CX              ; compteur : CX
07C0 8B4622          MOV  AX,[BP+22]      ; initialiser s, secteur courant
07C3 2E              CS:                  ; de m ('décimale' courante)
07C4 A3280A          MOV  [0A28],AX       ; pour le programme de produit
07C7 8B4624          MOV  AX,[BP+24]      ; partiel 0610
07CA 2E              CS:                  ;
07CB A32A0A          MOV  [0A2A],AX       ;
07CE 55              PUSH BP              ; sauver BP
07CF E83EFE          CALL 0610            ; AUX := M * s
07D2 5D              POP  BP              ; retrouver BP
07D3 E87A01          CALL 0950            ; P := P + AUX_d
07D6 E86702          CALL 0A40            ; réinitialisation des variables
07D9 59              POP  CX              ; sur la pile
07DA 5F              POP  DI              ; restitution des paramètres de
07DB 5E              POP  SI              ; la boucle
07DC E2DF            LOOP 07BD            ; -> BOUCLE
```

D'où la suite du listage précédent :

```
-d CS:4000 L 10                % état du répertoire
13E8:4000  C8 01 03 00 CB 01 02 00-CD 01 05 00 CE 01 01 00
-d CS:4040 L 10
13E8:4040  08 00 08 00 08 00 01 00-00 00 00 00 00 00 00 00
-d 3000:00 L 20                % état de la TAS
3000:0000  C9 01 CA 01 00 01 CC 01-00 01 D2 01 D0 01 00 01
3000:0010  D1 01 00 01 D3 01 D4 01-D5 01 00 01 00 00 00 00
-d 3000:1C70 L 150             % contenu des secteurs
3000:1C70  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
3000:1C80  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF
3000:1C90  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF
3000:1CA0  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF
3000:1CB0  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF
3000:1CC0  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF
3000:1CD0  01 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
3000:1CE0  01 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
3000:1CF0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
3000:1D00  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF
3000:1D10  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF
3000:1D20  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
3000:1D30  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF
3000:1D40  FE FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF
3000:1D50  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF
3000:1D60  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

% on relève, grâce au répertoire et à la TAS que le résultat
% commence au secteur 1CF0 et occupe les secteurs suivants, dans
% l'ordre des puissances croissantes de la base :

% 1CD0-1D20-1D30-1D40-1D50

6.3. La division.

C'est, de loin, l'opération la plus complexe. Nous avons déjà construit un programme de division sur des nombres de longueur arbitraire pour la conversion de la représentation hexadécimale d'un nombre en représentation décimale. Mais il s'agit de la division d'un tel nombre par un nombre à un seul chiffre.

La programmation de la division pose deux difficultés.

6.3.1. Le problème mathématique.

a. Division euclidienne.

Reprenons l'algorithme de la division. Supposons que nous sachions effectuer la division euclidienne des nombres au plus égaux à b, la base considérée.

Soit D et d deux nombres quelconques, $d \neq 0$ s'écrivant, respectivement : $D = a_k b^k + ... + a_0$ et $d = c_h b^h + ... + c_0$. Sans restreindre la généralité, on peut supposer que $c_h \neq 0$. Si $D < d$, le quotient euclidien de D par d est 0 et le reste est D. On peut donc supposer $D \geq d$ d'où $k \geq h$. Posons $m = k-h$. Il existe alors un entier naturel u_m unique tel que $u_m b^m d \leq D < (u_m+1)b^m d$ puisque d n'est pas nul. Comme, de plus, $bb^m d > D$, on en déduit que $u_m < b$. Par construction de u_m, le nombre D_1 défini par $D_1 = D - u_m b^m d$ vérifie $D_1 < b^m d$. Donc il existe de même un unique entier naturel u_{m-1} pour lequel on ait $u_{m-1} b^{m-1} d \leq D_1 < (u_{m-1}+1)b^{m-1} d$. De plus, on a $u_{m-1} < b$ puisque, par définition de u_m, $D_1 < bb^{m-1} d$.

Nous en déduisons un algorithme fournissant les entiers q et r de la division euclidienne puisque le dernier nombre D_i de la suite ainsi obtenue vérifie $D_i < d = b^0 d$. De plus, on obtient l'écriture de q sous la forme $u_m b^m + ... + u_0$, où $0 \leq u_m < b$.

> Notons, dès à présent que, dans ces hypothèses, il peut se faire que $u_m = 0$, où rappelons-le, $m = k-h$. Prendre, par exemple, $D = 21000$ et $d = 211$ (dans toute base $b \geq 3$). Notons également, que si $k > h$ et $a_k \neq 0$, si $u_m = 0$, nécessairement $u_{m-1} \neq 0$. Sinon, on aurait $D < b^{m-1} d = c_h b^{k-1} + ... c_0 b^{m-1} < D$. Si $u_m = 0$ et $k = h$, des inégalités ci-dessus, on déduit $q = 0$ et $r = D$.

La principale difficulté consiste à trouver les u_m. Même si l'on sait diviser les nombres à un chiffre en base b, ceci ne donne pas u_m immédiatement car on n'a pas nécessairement $u_m = [a_k/c_h]$. Peut-être n'en sommes nous pas loin?

Il faut regarder ceci de très près.

Ecrivons à présent $D = ab^s + a'$ et $d = cb^t + c'$, où b est la base et a, a', c et c' sont quelconques avec les seules restrictions : $0 \leq a' < b^s$, $0 \leq c' < b^t$, et posons à présent $m = s-t$. Des relations immédiates $ab^s \leq D < (a+1)b^s$ et $(c+1)b^t > d \geq cb^t$, on tire :

$$(a/(c+1))b^{s-t} < D/d < ((a+1)/c)b^{s-t}.$$

En écrivant $D = dq + r$ avec $0 \leq r < d$ et $q = ub^m + w$ avec $0 \leq w < b^m$, en divisant par b^m et en tenant compte des inégalités vérifiées par r et w (en particulier $w+r/d < w+1$), il vient, du fait que u est un entier, que l'on a l'encadrement suivant :

$$(1) \qquad [a/(c+1)] \leq u \leq [(a+1)/c].$$

En revenant à la définition de la partie entière, on en déduit que :

(2) $\Delta = [(a+1)/c] - [a/(c+1)] < 1 + 1/c + a/c(c+1)$.

Considérons, à présent, le cas où $s = k-1$ et $t = h-1$. On a alors $u = u_m$, de sorte que l'encadrement (1) s'applique à u_m pour $a = a_k b + a_{k-1}$ et $c = c_h b + c_{h-1}$. Comme $a \leq b^2-1$ et $c \geq b$, on en déduit $a/c(c+1) \leq (b^2-1)/b(b+1)$ et $1/c \leq 1/b$. Il vient donc :

$$[(a+1)/c] - [a/(c+1)] < 1 + 1/b + (b-1)/b = 2,$$

puisque on part d'une inégalité stricte dans (2). Comme les parties entières sont des entiers, l'inégalité stricte avec 2 donne une inégalité large avec 1, soit :

(3) $0 \leq \Delta = [(a+1)/c] - [a/(c+1)] \leq 1$

Si $u_m \neq 0$, on a la décimale de tête de q, et on applique à $D_1 = D - u_m b^m d$ les considérations précédentes avec, à présent $s = k-2$ et $t = h-1$. On observe qu'alors $s-t = m-1$, de sorte que nous trouverons u_{m-1}. Si $u_m = 0$, la formule définissant D_1, nous donne $D_1 = D$. Ce que nous avons dit précédemment s'applique de la même manière si nous faisons comme dans le cas $u_m \neq 0$. Nous en déduisons que la décimale de tête de q sera alors u_{m-1} si $a_k \neq 0$.

Si $a = a_{k-1}b + a_{k-2}$ (décimales de tête de D_1 sans hypothèse sur a_{k-1}), les majorations conduisant à (2) s'appliquent *mutatis mutandis* et nous obtenons donc (3). Comme le cas $u_m = 0$ le suggère, mais ce n'est pas la seule éventualité, il se peut que l'on ait $a = a_k b^2 + a_{k-1}b + a_{k-2}$ avec $a_k \neq 0$.

Plaçons-nous dans ce cas. Comme $s-t = m-1$, on obtient également les relations (1) et (2). Comme $D_1 < b^m d$, on a alors $a < (c+1)b$. Sinon, $a \geq (c+1)b$ d'où l'on tire $D_1 \geq c_h b^k + c_{h-1}b^{k-1} + b^{k-1} + a_{k-2}b^{k-2} + \ldots + a_0$. Or :

$$c_h b^k + c_{h-1}b^{k-1} + b^{k-1} = b^m(c_h b^k + c_{h-1}b^{k-1} + b^{k-1}) > b^m d.$$

Nous distinguerons, à présent deux sous-cas :

Tout d'abord, le cas $a \leq cb$. Si a vérifie cette inégalité, alors $a/c(c+1) \leq b/(c+1)$ d'où $\Delta < 1 + 1/c + a/c(c+1) \leq 1 + 1/c + b/(c+1)$. Si $c \geq b+1$, on en déduit $\Delta < 1 + 1/(b+1) + b/(b+2) = 2 + 1/(b+1) - 2/(b+2) = 2 - b/(b+1)(b+2)$, de sorte qu'on obtient à nouveau la relation (3). Si, par contre, $c < b+1$, nécessairement $c = b$ et $a = b^2$ puisque $a \geq b^2$ ($a_k \neq 0$). On a immédiatement $[a/(c+1)] = b-1$ et $[(a+1)/c] = b$, d'où (3) et $u_{m-1} = b-1$.

Si $a > cb$, on a donc $a \geq cb + 1$ d'où $a/(c+1) \geq ((c+1)b - (b-1))/(c+1) = b - (b-1)/(c+1)$.

Comme $b \geq c$, on a donc $a/(c+1) > b-1$ d'où $[a/(c+1)] = b-1$ et donc $u_{m-1} = b-1$ puisque, comme nous l'avons vu, on a toujours $0 \leq u_i < b$. Comme $a+1 \leq (c+1)b$, on a donc $(a+1)/c \leq (c+1)b/c = b + b/c$ et donc $[(a+1)/c] = b$, d'où de nouveau (3), lorsque $c \geq b+1$. Si $c = b$, le cas le plus extrême est $a = b^2 + b - 1$, pour lequel $\Delta = 2$, car on a $(a+1)/c = b+1$. Mais dans ce cas, $u_{m-1} = b-1$, car a provient d'un reste $D - u_m b^m d < d$.

Comme on peut réitérer ce raisonnement pour le nouveau reste D_2 obtenu et ainsi de suite, jusqu'au reste r de la division, on obtient l'algorithme suivant, dans lequel $|R|$ désigne le nombre de chiffres du nombre R représenté en base b :

1. Fixer D_t, D_p et D_u, positions respectives : de la décimale de tête du dividende D, de la décimale qui précède et de la décimale de rang m, où $m = k-h$ (notations ci-dessus) ou encore $|D|-|d|$; fixer d_t et d_u pour le diviseur d ; calculer c et $c+1$ et les noter ; poser $|q| = |D|-|d|+1$; fixer la première valeur de a ; fixer q_t et $q_c := q_t$ où q_c est le pointeur de la prochaine décimale de q, le quotient.

2. Calculer $u = [a/(c+1)]$ et $v = [(a+1)/c]$; poser $w := 0$ si $u = v$ et $w = 1$ sinon.

3. Calculer $D - dub^m$ et mettre le résultat dans D.

4. A-t-on $w = 0$?
Si oui, passer au point 5.
Sinon : vérifier que $D - u b^m d < b^m d$; si la relation est vérifiée, passer au point 5 ; sinon, soustraire $b^m d$ de la valeur actuelle de D et augmenter u de 1.
5. Décaler les pointeurs : $D_u := D_u - 1$, $D_p := D_p - 1$; si la décimale pointée par D_t est non nulle, laisser D_t inchangé ; si elle est nulle, faire $D_t := D_t - 1$.
6. S'il s'agit du premier calcul de u seulement : si $u = 0$, fixer q_t à $q_t - 1$ et poser $q_c := q_t$; retourner au point 2 si $q_t > 0$, au point 7 sinon.
Pour les calculs ultérieurs de u, enregistrer u et décaler q_c par $q_c := q_c - 1$; retourner au point 2 si $q_c > 0$, au point 7 sinon ;
7. Fin de l'algorithme.

Cet algorithme suppose qu'on soit en présence d'entiers D et d s'écrivant avec au moins deux chiffres en base b pour d et éventuellement trois pour D. Si on n'est pas dans ce cas, on se trouve toujours dans une situation où la division est censée résolue : soit le cas de la division d'un nombre d'au plus deux chiffres par un nombre de deux chiffres, soit le cas de la division d'un nombre quelconque de chiffres par un nombre à un chiffre. Nous connaissons déjà l'algorithme de la division dans ce dernier cas. Le premier est un cas particulier de la division d'un nombre à trois chiffres par un nombre à deux chiffres.

b. La division 'à virgule'.

Cette division constitue un algorithme classique permettant d'obtenir la représentation, en base b, d'un nombre fractionnaire, c'est-à-dire de la forme N/D, où N et D sont des entiers naturels, $D \neq 0$.

Comme enseigné à l'école primaire, cet algorithme commence comme celui de la division euclidienne. Puis, lorsque l'on a trouvé un reste inférieur au diviseur, on prolonge l'algorithme de la même façon en adjoignant, à chaque division partielle, un zéro à droite du reste établi lors de la division précédente.

Regardons de plus près cet algorithme :

Si r est le reste obtenu par la division euclidienne de D par d, le nombre r/d est plus petit que 1. On peut écrire, en base b :

$$r/d = a_1/b + a_2/b^2 + \ldots$$

avec $0 \leq a_i < b$ d'où, en multipliant les deux membres par le produit db :

$$br = a_1 d + a_2 d/b + \ldots$$

soit :

$$0 \leq br - a_1 d < d(b-1)(1/b)(1 + 1/b + 1/b^2 + \ldots)$$

L'inégalité stricte étant obtenue du fait que s'il y avait égalité à droite, r/d serait un nombre de la forme N/b^k et donc, on aurait eu l'égalité à gauche auparavant.

Comme la somme de la série est $1/(1-1/b) = b/(b-1)$, on obtient que :

$$br = a_1 d + r_1, \text{ avec } 0 \leq r_1 < d.$$

Le même calcul peut être repris en remplaçant r par r_1 et a_1 par a_2 et ainsi de suite, d'où un algorithme de calcul des a_i, les nombres cherchés. Si b est la base de représentation des nombres, le produit br correspond bien à l'adjonction d'un zéro à droite. Mais, après tout, rien n'oblige b à être la base dans laquelle on écrit les nombres r et d. Ainsi, si r et d sont écrits en hexadécimal, on peut décider que b sera 2^4, et on obtiendra le développement fractionnaire hexadécimal de r/d. On peut tout aussi bien

décider que b sera 10 (= 2*5) et dans ces conditions, comme $a_i < b$ et $10 < 16 = 2^4$, on obtient *directement* le développement fractionnaire en base dix.

Nous verrons plus loin l'implantation de cet algorithme.

6.3.2. L'Implantation.

a. Le programme de base de la division euclidienne.

Il s'agit à présent de mettre en oeuvre le premier des algorithmes que nous venons d'étudier. Pour ce faire, nous devrions utiliser le principe du 'modèle réduit' pour constituer un premier programme effectuant la division de deux nombres de taille arbitraire, mais que nous testerions sur des nombres 'courts', de quelques mots simplement. Ce programme nous servirait ensuite à modéliser la division de deux registres. En raison de la taille des programmes en assembleur, nous dérogerons, ici, à ce principe. Nous ne considérerons que le programme définitif, mais nous en extrairons ce qui concerne l'implantation directe de l'algorithme mathématique, réservant au paragraphe suivant ce qui concerne la gestion des nombres 'arbitrairement' grands.

Nous traduisons l'algorithme précédent dans un premier programme de contrôle dans lequel les appels correspondent aux différentes étapes de l'algorithme. Voici ce programme dont les commentaires reprennent les notations introduites précédemment :

Après l'élimination des cas particuliers traités par des programmes adéquats (voir en annexe), le programme détaille le cas où $|d| > 1$ et $|D| \geq |d|$. Il commence par installer une réserve sur pile sur le modèle de ce qui a été fait pour la multiplication, puis il appelle le programme d'initialisation :

```
0B03 E84A00          CALL 0B50            ; initialisation
0B06:BOUCLE1 :                            ; Sur le nombre de secteurs de d
0B06 51              PUSH CX              ; on sauve le compteur
0B07 B91000          MOV  CX,0010         ; compteur dans le secteur
0B0A:BOUCLE2 :                            ; intra-secteur
0B0A 51              PUSH CX              ; (010 mots); on le sauve
0B0B E8BA02          CALL 0DC8            ; calcul de u, v et w
0B0E 50              PUSH AX              ; on sauve u (dans AL)
0B0F E8EE02          CALL 0E00            ; division partielle
0B12 D15E0D          RCR  WO [BP+0D],1    ; test sur w (calculé par 0DC8)
0B15 730D            JNB  0B24            ; w=0 : -> suite
0B17 E81602          CALL 0D30            ; w≠0 : vérification du reste
0B1A 58              POP  AX              ; (équilibre de la pile)
0B1B 7207            JB   0B24            ; reste < diviseur : -> suite
0B1D 50              PUSH AX              ; on sauve AX
0B1E E84F02          CALL 0D70            ; on retranche encore d^m b
0B21 58              POP  AX              ; on retrouve u
0B22 40              INC  AX              ; u := u+1
0B23 90              NOP                  ;
0B24 E86902          CALL 0D90            ; u stocké, mise à jour des
0B27 59              POP  CX              ; pointeurs, compteur retrouvé
0B28 E2E0            LOOP 0B0A            ; -> BOUCLE2
0B2A 59              POP  CX              ; compteur retrouvé
0B2B E8D202          CALL 0E00            ; écriture sur le registre q
0B2E E2D6            LOOP 0B06            ; -> BOUCLE1
0B30 8B869200        MOV  AX,[BP+0092]    ; on consulte le signal sur pile
0B34 D1E8            SHR  AX,1            ; de division fractionnaire
0B36 7303            JNB  0B3B            ; on en reste là : -> résultats
0B38 E8AC08          CALL 13E7            ; on 'pousse' après la virgule
0B3B E8F202          CALL 0E30            ; traitement des résultats
```

Le programme se termine par le démantèlement de la réserve constituée sur la pile.

Nous donnons, plus loin, le texte du programme de calcul de u, v, et w, à partir duquel le lecteur établira sans peine ce qu'il doit être dans le cas du 'modèle réduit'. Voici, à présent, l'extrait principal du programme **0E00** qui implante la soustraction/multiplication définie au point 3 de l'algorithme de la division euclidienne.

Après avoir recueilli u dans DL et placé les pointeurs sur D et d en, respectivement D_u et d_u et initialisé le compteur de boucle sur ld (dans la version modèle réduit), on effectue la multiplication et la soustraction dans la même boucle, comme ci-après. Pour ce faire, on utilise BX, mis à zéro au départ, pour stocker les retenues (indépendantes) ADC et SBB :

```
0E16:BOUCLE                             ;
0E16 88E6          MOV  DH,AH           ; DH := rm, retenue MUL précédente
0E18 8A04          MOV  AL,[SI]         ; lire décimale en dc
0E1A F6E2          MUL  DL              ; multiplier par u
0E1C F8            CLC                  ; CF = 0 pour utilisation RCR
0E1D D1DB          RCR  BX,1            ; CF := e, retenue ADC antérieure
0E1F 10F0          ADC  AL,DH           ; AL := AL + rm + e
0E21 D1D3          RCL  BX,1            ; BX0 := e nouveau
0E23 D1D3          RCL  BX,1            ; CF := f, retenue SBB antérieure
0E25 26            ES:                  ; soustraire AL de la décimale vue
0E26 1805          SBB  [DI],AL         ; par Dc en tenant compte de f
0E28 D1DB          RCR  BX,1            ; BX15 := f nouveau
0E2A 46            INC  SI              ; mise à jour de dc
0E2B 47            INC  DI              ; mise à jour de Dc
0E2C E2E8          LOOP 0E16            ; retour à BOUCLE
```

Les retenues engendrées par ADC et SBB, respectivement e et f, sont indépendantes. Dans ce programme, on stocke e en BX_0 et f en BX_{15} : Par une première 'bascule' à droite (rcr), on place e sur CF (f passe alors en BX_{14}). La bascule à gauche suivante (rcl) 'lit' CF et place la nouvelle valeur de e en BX_0, remet f en BX_{15} et l'ancienne valeur de CF sur cet indicateur. Les bascules en ordre inverse (d'abord à gauche puis à droite) permettent de conserver la retenue SBB, compte tenu de ses variations.

La boucle du programme **0E00** peut se terminer, alors qu'une retenue doit être retranchée de la décimale de tête de D qui n'aura pas été lue dans la boucle. C'est ce qui se passe lorsque a, à l'étape considérée, est un nombre à trois chiffres. Par un test adéquat (voir plus loin), on établit la nécessité d'effectuer cette dernière soustraction et on l'exécute :

```
0E82 D1DB          RCR  BX,1            ;
0E84 80D400        ADC  AH,00           ;
0E87 D1D3          RCL  BX,1            ;
0E89 D1D3          RCL  BX,1            ;
0E8B 26            ES:                  ;
0E8C 1825          SBB  [DI],AH         ;
```

On note l'utilisation des retenues par BX ; on observe l'utilisation directe de AH puisque, la multiplication n'ayant plus lieu, les intermédiaires par DH et AL sont devenus inutiles.

C'est le moment d'aborder les problèmes que pose l'exécution, en 'vraie grandeur' de ce programme, ce qui nous permettra, en particulier, de considérer le test du reste trouvé par le programme **0E00**.

b. La gestion des registres dans la division euclidienne.

Le problème 'grandeur nature' ou le problème informatique.

Le programme de division de deux registres (on suppose vérifié par le programme utilisateur que le diviseur n'est pas nul), pose un problème nouveau par rapport à l'adaptation des programmes 'modèle réduit' au programme réel dans le cas des opérations vues précédemment.

En effet, pour les autres opérations, nous avons toujours commencé par le chiffre des unités, ce qui facilite la programmation, basée essentiellement sur la propagation 'vers le haut' d'une retenue accompagnée, dans le cas de la multiplication, d'un décalage. En ce qui concerne la division, on commence par les décimales de tête. Ceci passe pratiquement 'inaperçu' dans le programme 'modèle réduit' dans lequel on suppose les décimales du nombre écrites en mémoire de façon contiguë. L'attention se porte principalement sur le problème du calcul de la décimale de tête du quotient, ce que nous voyons plus loin. Dans le contexte plus général des registres que nous avons définis à l'image des fichiers *DOS*, la difficulté apparaît aussitôt. Il ne suffit pas seulement de trouver la décimale de tête mais encore, après celle-ci, la décimale qui la précède et ainsi de suite. Il faudrait, en quelque sorte, que les secteurs du registre soient reliés entre eux dans l'ordre inverse.

Si nous devions changer l'organisation des registres pour effectuer la division, il nous faudrait alors redéfinir les autres opérations dans le nouveau cadre. Nous pouvons garder le cadre défini jusqu'ici en procédant de la façon suivante :

Sur une zone spécialement allouée à cet effet, le programme relèvera de façon ***contiguë*** les numéros des secteurs composant le dividende, puis, toujours de façon contiguë, les numéros de secteurs du diviseur. Dans les deux cas, l'adresse du début et celle de la fin seront relevées. De sorte que connaissant le dernier secteur du registre, 'la décimale de tête', il sera très facile de revenir en arrière, pas à pas. Il suffira de faire décroître le pointeur des numéros de secteur dans la zone ainsi constituée que nous appellerons, dans ce qui suit, ***zone de chaînage***.

Voici le programme de construction de ce relevé, l'adresse de répertoire du registre concerné étant placée dans *SI* par le programme appelant, *DI* contenant l'adresse du premier octet qui suit le dernier mot relevé au cours de l'opération précédente, donc **0000**, si le programme est appelé pour la première fois :

```
0BB8 2E            CS:                ;
0BB9 8B840040      MOV  AX,[SI+4000]  ; on part du répertoire
0BBD 90            NOP                ;
0BBE:BOUCLE                           ; des relevés selon TAS
0BBE 3D0001        CMP  AX,0100       ; dernier secteur?
0BC1 742D          JZ   0BF0          ; oui : fin
0BC3 26            ES:                ; non :
0BC4 8905          MOV  [DI],AX       ; relever ce numéro
0BC6 50            PUSH AX            ; puis le transformer
0BC7 25FF0F        AND  AX,0FFF       ; en l'adresse TAS du
0BCA 2DC801        SUB  AX,01C8       ; secteur qu'il désigne
0BCD D1E0          SHL  AX,1          ;
0BCF 89C6          MOV  SI,AX         ; SI : déplacement en TAS
0BD1 5B            POP  BX            ;
0BD2 81E300F0      AND  BX,F000       ; recherche du segment par le
0BD6 D1E3          SHL  BX,1          ; quartet le plus fort
0BD8 2E            CS:                ; du numéro de secteur
0BD9 8B870038      MOV  AX,[BX+3800]  ;
0BDD 8ED8          MOV  DS,AX         ; DS : segment de la TAS
0BDF 8B04          MOV  AX,[SI]       ; lecture de cette entrée
0BE1 83FFFE        CMP  DI,-02        ; DI accessible?
0BE4 7206          JB   0BEC          ; oui : suite
0BE6 E8E703        CALL 0FD0          ; non : RAM pleine
0BE9 F9            STC                ;
0BEA C3            RET                ;

0BEC 47            INC  DI            ; préparer DI pour l'inscription
0BED 47            INC  DI            ; suivante
0BEE EBCE          JMP  0BBE          ; retour à BOUCLE
0BF0 C3            RET                ;
```

Les variables partagées de la pile.

Elles ont pour fonction de définir les pointeurs D_u, D_p, D_t, d_t, d_u, d_c et q_c. De plus, pour accélérer l'exécution des utilitaires de gestion des secteurs de chaque registre, un certain nombre d'informations complémentaires sont placées sur la pile.

On a la structure suivante :

D, d et q désignant respectivement le dividende, le diviseur et le quotient, on associe à chacun des pointeurs D_t, D_p, D_u, D_c, d_t, d_u, d_c et q_c trois mots : les deux premiers fournissent l'adresse absolue du pointeur et le troisième, l'adresse, dans la zone de chaînage, où se trouve le numéro *TAS* complet du secteur dans lequel le pointeur se trouve. Par ailleurs, R étant un registre, nous désignons par (R) son adresse dans le répertoire (depuis 4000), par $[R]$ le nombre de ses secteurs et par $\{R\}$ le nombre d'octets du dernier secteur défini, selon ce que nous avons vu au début de ce chapitre par $|R| = 010*([R]-1) + \{R\}$, où $|R|$ est la taille, en octets du registre. De façon précise, les adresses étant relatives à *BP*, on a :

pointeur	adresse absolue	n° chaînage
D_t	18:16	1A
D_p	1E:1C	20
D_u	24:22	26
D_c	2A:28	2C
d_t	38:36	3A
d_u	3E:3C	40
d_c	44:42	46
q_c	58:56	5A

Les informations sur les registres sont :

(D), $[D]$, $\{D\}$: 10, 12, 14
(d), $[d]$, $\{d\}$: 30, 32, 34
(q), $[q]$, $\{q\}$: 50, 52, 54

Les seize premiers octets depuis *BP* sont organisés ainsi :

00 : adresse de segment de la zone de chaînage
02 : numéro de chaînage du premier secteur de D
04 : numéro de chaînage du dernier secteur de D
06 : numéro de chaînage du premier secteur de d
08 : numéro de chaînage du dernier secteur de q
0A : numéro de chaînage du premier secteur de q
0C : 00 si et seulement si $[q] > 1$
0D : valeur de w : 00 si et seulement si $u = v$
0E : place libre

Dans l'implantation de l'algorithme sur des registres, on cherchera à réduire le plus possible le nombre des tests et à préparer les calculs répétitifs de façon à les réduire au strict nécessaire chaque fois qu'il faut les effectuer. Ainsi, dans la multiplication/soustraction, une fois fixés D_u et d_u, on sait que le nombre de multiplications est exactement $|d|$. La boucle sur $|d|$ du 'modèle réduit' est remplacée par deux boucles imbriquées : une boucle extérieure sur $[d]$, le nombre des secteurs, et une boucle intérieure sur les seize octets d'un secteur, les pointeurs d_c et D_c, après avoir été convenablement initialisés étant simplement incrémentés par des instructions **inc si** et **inc di**.

Cependant, le changement de secteur pour d_c et D_c n'est pas, en général, simultané. Cela est dû au fait que le nombre d'octets du dernier secteur n'est pas nécessairement le même pour D que pour d. Il s'introduit un décalage que nous gérerons de la façon suivante. Comme nous nous réglons sur les secteurs de d, on passe d'un secteur de d au suivant dans la boucle extérieure, mais hors de la boucle intérieure.

Par contre, approximativement 15 fois sur 16, on changera de secteur dans D à l'intérieur de la boucle la plus profonde. Or, lorsqu'on calcule la longueur initiale de q, on peut calculer tous les éléments de ce décalage. Cela revient à partager la boucle intérieure sur un secteur de d en deux boucles disjointes : la première boucle, jusqu'à la fin du secteur courant de D, la seconde boucle, depuis le début du secteur suivant de D, le changement de secteur se faisant entre les deux boucles (voir en annexe le texte complet du programme **0E00**). Or, pendant la durée de l'exécution du programme **0E00**, les nombres de passages dans chacune de ces deux sous-boucles sont constants : on les calculera une fois pour toutes, à l'initialisation, et on les actualisera avant chaque nouvel appel du programme **0E00**.

Une exception à ne pas oublier : le cas où l'on trouve $u = 0$ pour la décimale de tête de q. Hormis le cas $q = 0$ traité à part, ceci nécessite de réajuster lql, mais aussi les indices de partage. Ceci étant fait à cette étape, on n'a plus à recalculer ces indices : il suffit de les décaler, comme indiqué ci-dessus.

Afin de gérer ces changements de secteur, on définit sur la pile, en BP+**64**, un mot que l'on décompose en deux :

indices de partage secteur plein pour la multiplication/soustraction :
by 64 : 1ère partie
by 65 : 2ème partie

Cependant, le dernier secteur d'un registre comporte le plus souvent un certain nombre d'octets tous égaux à zéro à partir d'un certain rang : c'est le nombre d'octets du dernier secteur défini dans la table **4040**. De ce fait, pour le dernier secteur de d, les indices de partage seront, en général, différents de ceux qui figurent en BP+**64**. D'où un mot de même nature sur la pile en BP+**66** :

indices de partage dernier secteur pour la multiplication/soustraction :
by 66 : 1ère partie
by 67 : 2ème partie

Ce problème se retrouve lors de la comparaison du reste et du diviseur. Mais là, le calcul est déjà fait : il n'y a lieu de procéder à la comparaison octet par octet (depuis l'octet de tête) que si le reste partiel a la même longueur que le diviseur. Mais alors, les changements de secteur sont les mêmes que précédemment, à l'ordre près : ce qui était auparavant 1ère partie devient 2ème partie et inversement.

Enfin, ce problème du changement de secteur se retrouve également pour définir a lors du calcul de u et v. Il se présente également pour c mais une seule fois car c, remarquons-le, est toujours le même au cours de la division : seul a peut se modifier d'une étape à l'autre. Aussi, pour faciliter les opérations sur a, nous recopions les deux octets de tête de D (donc, après la détermination de la première décimale de q, du reste en cours) dans une 'fenêtre' de quatre octets depuis BP+60. Lors des décalages suivants, cette fenêtre nous permettra de reconnaître si l'on se trouve dans le cas où a s'écrit avec deux chiffres ou bien dans le cas où son écriture en nécessite trois.

Mise en oeuvre.

Le programme d'initialisation joue un rôle fondamental dans le cadre que nous venons de tracer. Après avoir réservé une zone mémoire pour le chaînage, la structure du programme est la suivante (pour le cas où l'allocation a eu lieu) :

recueil des adresses des répertoires des registres ;
0B98 : relevé du chaînage de D et de d ;
0BF8 : calcul de $[q]$ et $\{q\}$;
0CA8 : calcul de c, c+1 ;
0CE8 : calcul de a, D_p et mise en place de la fenêtre BP+**60-62** ;
0D30 : calcul de D_u et D_c ;

0D90 : calcul de d_c, d_u et q_c ;
puis, $[q]$ étant fixé *a priori*, on examine si $[q] = 1$ ou non, et si oui, on réécrit **0AD0** en remplaçant **010** par $\{q\}$ à l'instruction **0B07 mov cx,010**.

Le calcul de $[R]$ et $\{R\}$ pour un registre quelconque est simple : pour $[R]$, on lit la table **4000**. Pour $\{R\}$, la table **4040** donne une valeur en mots. Il faut donc 'aller voir', ce qui est fait, dans le programme **0BF8** de la façon suivante, dans le cas de D, après avoir calculé en BX le double du nombre de mots du dernier secteur et obtenu en $DS{:}SI$ l'adresse du début de ce secteur :

```
0C17 01DE          ADD  SI,BX    ; (BX = 2*[(D)/2])
0C19 4E            DEC  SI       ;
0C1A 4E            DEC  SI       ;
0C1B 8B04          MOV  AX,[SI]  ; AX : dernier mot de D
0C1D 80FC00        CMP  AH,00    ; deux octets?
0C20 7503          JNZ  0C25     ; oui : -> modifier SI, pas BX
0C22 4B            DEC  BX       ; non : modifier (D)
0C23 EB01          JMP  0C26     ;
0C25 46            INC  SI       ;
```

et à présent, $BX = \{D\}$ et SI contient le déplacement de D_l.

Le calcul de $[q]$ et $\{q\}$ repose sur la relation, pour tout registre R : $|R| = 10([R]-1) + \{R\}$, $|R|$ et $\{R\}$ mesurés en octets. Cette formule nous donne, en posant $\mathbf{1}_A = 1$ si la propriété **A** est vraie et $\mathbf{1}_A = 0$ sinon :

$$\{q\} = \{D\} - \{d\} + 1 + 10.\mathbf{1}_{\{d\}>\{D\}}$$
$$[q] = [D] - [d] + \mathbf{1}_{\{d\}\leq\{D\}},$$

auxquelles correspondent les instructions :

```
0C60 8B4614        MOV  AX,[BP+14]     ; AX := {D}
0C63 8B4E34        MOV  CX,[BP+34]     ; CX := {d}
0C66 8B5E12        MOV  BX,[BP+12]     ; BX := [D]
0C69 43            INC  BX             ; BX := [D]+1
0C6A 40            INC  AX             ; AX := {D}+1
0C6B 39C8          CMP  AX,CX          ;
0C6D 7304          JNB  0C73           ; {D}+1 >= {d} : -> soustraction
0C6F 051000        ADD  AX,0010        ; {D}+1 < {d} : ajustement
0C72 4B            DEC  BX             ; et donc BX décrémenté
0C73 29C8          SUB  AX,CX          ; soustraction : AX := {q}
0C75 894654        MOV  [BP+54],AX     ; {q} noté
0C78 2B5E32        SUB  BX,[BP+32]     ; BX := BX - [d] : BX := [q]
0C7B 895E52        MOV  [BP+52],BX     ; [q] noté
0C7E C6460D80      MOV  BY [BP+0D],80  ; signal : |q| provisoire
```

Les indices de partage en découlent presque aussitôt :

$$i_{64} = 10 - \{q\} + 1$$
$$i_{65} = \{q\} - 1$$
$$i_{66} = i_{64}.\mathbf{1}_{\{d\}>\{D\}} + \{d\}.\mathbf{1}_{\{d\}\leq\{D\}}$$
$$i_{67} = (\{d\} - i_{64}).\mathbf{1}_{\{d\}>\{D\}}$$

Ces calculs sont effectués à l'initialisation, dans le programme **0D30** : voir en annexe 2, les instructions **0D55** à **0D8B**.

La décimale de tête du quotient.

Comme nous l'avons vu, cette décimale n'est pas déterminée à coup sûr par la première division. Il faudrait, en toute rigueur, affecter une variable maintenue à 0 tant que cette décimale de tête n'est pas trouvée et fixée à 1, par exemple, dès que la décimale est trouvée et confirmée. Afin d'éviter un test par lequel on repassera ensuite sans cesse, nous avons choisi la solution de la réécriture : le programme d'enregistrement de la décimale du quotient comportera deux entrées. La première sera réservée à l'examen

particulier du résultat du premier calcul, la seconde, plus loin dans le texte, correspondra au cas 'normal' où la décimale trouvée et confirmée est à enregistrer systématiquement. Rappelons que dans le cas de la première division partielle on n'enregistrera la décimale trouvée que si cette dernière est non nulle (le cas $q = 0$ étant déjà 'préparé' à l'initialisation).

Lors du premier passage de l'exécution par ce programme, par la première entrée, on regarde s'il faut ou non enregistrer la décimale. S'il ne le faut pas, il conviendra alors de réajuster la taille de q et de corriger tous les paramètres qui se déduisent de cette taille, en particulier, les indices de partage. S'il le faut, on enregistre la valeur trouvée. Dans les deux cas, on sait qu'au prochain passage par ce programme il faudra enregistrer la décimale, qu'elle soit nulle ou non. Ce prochain passage, ainsi que tous les suivants se feront donc par la seconde entrée. Il suffit donc, dans la partie 'exceptionnelle' du programme considéré (programme **1060** en annexe) d'insérer une inscription de réécriture de l'adresse d'appel. Compte tenu du codage des instructions d'appel intra-segment, il suffit de le faire sous la forme d'une instruction **add** :

```
CS:                                    ;
ADD  WO [0B25],+1A                     ; nouvelle adresse d'appel
```

De la même manière, l'enregistrement du secteur de tête du quotient relèvera de la même technique car, les secteurs suivants, s'ils existent sont tous des secteurs pleins. Voir le programme **11D8** en annexe.

REMARQUE : Afin d'alléger le texte du programme, nous avons choisi la solution qui consiste à réserver un secteur nouveau pour l'enregistrement du quotient au fur et à mesure du calcul et d'écrire les numéros de secteurs du quotient sur la zone de chaînage dans l'ordre inverse de ce qui est fait pour les autres registres puisque, dans le cas du quotient, on ne connaît en premier que la tête.

Une solution plus complexe mais plus efficace consisterait à réaffecter à l'usage du registre quotient les secteurs du dividende qui se trouveraient libérés. On sait qu'à un secteur près, on a $[D] = [q] + [r]$ où r désigne le registre reste (c'est-à-dire D lorsque l'opération est terminée). Compte tenu des variations de la partie 'utile' du dernier secteur d'un registre, il est plus prudent de se réserver une marge de deux secteurs (au cas où l'on sait, *a priori*, que le quotient comporte plusieurs secteurs). Nous laissons à la sagacité du lecteur la modification correspondante du programme proposé en annexe.

Le contrôle du reste.

Comme indiqué dans l'algorithme que nous avons établi au paragraphe 6.3.1.a., on ne procède à cette vérification que si les nombres u et v ont été trouvés différents. Dans ce cas, deux éventualités se présentent : ou bien la taille du reste courant est différente de celle du diviseur, ou bien elle est égale. Lorsque les tailles sont différentes, et d'emblée, on ne peut que constater qu'une taille trop grande du reste, on statue immédiatement : il faut retrancher le diviseur du reste, augmenter u de 1 et cette nouvelle valeur de u est la valeur de la décimale courante de q, ainsi que nous l'avons vu au paragraphe 6.3.1. Si, a priori la taille du reste ne dépasse pas celle du diviseur, il faut comparer, octet par octet les deux nombres en commençant par l'octet de tête, jusqu'à trouver un octet différent ou bien tomber sur l'adresse de départ des deux registres sans avoir trouvé de différence.

Le programme **0F20**, chargé de ce contrôle, se divise donc en deux parties. Pour la première, voir l'annexe 2, l'examen du second mot de la 'fenêtre' *BP*+**60**-62 permet de décider au cas où ce mot ne serait pas vide : il faudrait alors lire sur le 'terrain'. Sinon, le reste n'est pas de taille plus grande que le diviseur et il convient donc de faire la comparaison octet par octet.

La seconde partie de ce programme se situe dans l'hypothèse de la lecture octet par octet. Il suffit alors, comme on l'a déjà noté page 141, de passer en revue le reste et le dividende, à l'envers, depuis la décimale de tête, les indices de décalage étant les

mêmes, mais à considérer dans l'ordre inverse. Ceci se fait dans une boucle sur le nombre de secteurs de [*d*], boucle **0F67** dans l'annexe 2, qui commence par une situation particulière du point de vue des indices de partage : celle du secteur de tête. A chaque fois, on a deux temps dans le parcours du secteur de *d*, selon le secteur de *D* dans lequel on se trouve. On retrouve les particularités déjà indiquées. Cependant, comme il s'agit ici de comparaison, on peut simplifier très sensiblement le code : comme les pointeurs sur *D* et *d* sont, respectivement, *ES:DI* et *DS:SI*, on peut utiliser l'instruction **cmpsb** préfixée par **repe** puisqu'on cherche l'inégalité, avec la valeur *DN* de *DF* puisqu'on part de l'octet de tête. On aura les deux types de séquences suivants :

```
0100 E307    JCXZ 0109          0100 FD      STD
0102 FD      STD                0101 F3      REPZ
0103 F3      REPZ               0102 A6      CMPSB
0104 A6      CMPSB              0103 724B    JB   0150
0105 7249    JB   0150          0105 774B    JA   0152
0107 7749    JA   0152
0109 E8D50E  CALL 0EE0
```

Le programme **0EE0** effectue le changement de secteur dans *D*.

selon que l'indice de partage peut ou non s'avérer égal à zéro.

La division à virgule.

L'algorithme de cette division ne pose pas de problème nouveau. Son implantation n'est pas complètement réalisée dans le programme donné en annexe car elle consisterait à répéter la boucle de base du programme **0AD0** et, en particulier, la soustraction/multiplication du programme **0E00**. Une seule modification importante : remplacer l'appel au programme **01060** de réactualisation des pointeurs par un appel au programme **012A0** qui multiplie le reste trouvé par **0A** (ou **010**, selon l'option choisie : calcul en décimal ou en hexadécimal) puis réactualise les pointeurs.

La première partie du programme **012A0** consiste à multiplier l'entier défini par un registre par la valeur de *BX* (**0A** ou **010** dans notre problème) et figure dans l'annexe 2. Nous reviendrons à ce problème au chapitre 8 pour calculer *n*! plus simplement que nous ne l'avons fait à ce chapitre[1], et nous réutiliserons ce programme dans ce nouveau contexte. En voici la boucle principale, partie gauche ci-après, ainsi que la boucle de multiplication d'un secteur sur la partie droite :

Le multiplicateur est dans *BX* et *ES:DI* pointe sur le premier secteur du registre.

```
12A7 31D2      XOR  DX,DX          1339 B90800    MOV  CX,0008
12A9:BOUCLE                        133C FC        CLD
12A9 51        PUSH CX             133D:BOUCLE
12AA E88B00    CALL 1338           133D 51        PUSH CX
12AD 59        POP  CX             133E 89D1      MOV  CX,DX
12AE 49        DEC  CX             1340 26        ES:
12AF E305      JCXZ 12B6           1341 8B05      MOV  AX,[DI]
12B1 E85C00    CALL 1310           1343 F7E3      MUL  BX
12B4 EBF3      JMP  12A9           1345 01C8      ADD  AX,CX
                                   1347 83D200    ADC  DX,+00
                                   134A AB        STOSW
                                   134B 59        POP  CX
                                   134C E2EF      LOOP 133D
```

Remarquer l'emploi de **stosw** dans la boucle de droite.

L'inconvénient de cette méthode est de remultiplier le reste obtenu à chaque étape. Il en résulte une certaine perte de temps de calcul par rapport à une translation des octets du reste dans le cas où $b = 010$. Cependant, cette méthode est très avantageuse du point de

[1] Mais nous conseillons vivement au lecteur de le faire dès à présent, à titre d'exercice.

vue de l'espace mémoire occupé : on peut même, par cet algorithme, prolonger la division aussi loin que l'on veut, quitte à sauver sur un support de masse les décimales 'certaines' déjà trouvées quand la mémoire vive est pleine, afin de refaire de la place pour les décimales suivantes[1].

Observons, par ailleurs, que si l'on voulait procéder autrement, il faudrait ajouter 'd'un coup' tous les zéros à droite et effectuer la division en base hexadécimale puis reconvertir en décimal. Mais nous n'en serions pas encore prêts à passer au programme de conversion hexadécimal -> décimal que nous connaissons. En effet, si N/D est un nombre fractionnaire hexadécimal, D est donc une puissance de 2, soit $D = 2^k$. Or, un nombre décimal est de la forme M/E où E est une puissance de dix. Il faut donc d'abord multiplier N par 5^k avant d'effectuer la conversion usuelle. Ce qui justifie le choix de la méthode proposée.

Améliorations.

On peut imaginer des raffinements à l'algorithme dont nous donnons le programme intégral en annexe. En particulier, si, lors du calcul de q, la capacité définie par la zone affectée au chaînage est atteinte, on peut réutiliser une partie de l'espace utilisé par D : tout ce qui se trouve au-delà du reste trouvé pour la partie entière (à un octet près pour la multiplication par b). Puis, lorsque cette place est pleine, on peut alors sauver sur le support de masse ce qui a été calculé jusque là et reprendre là où on en était resté. Le programme donné se contente d'afficher un 'dépassement de capacité' lorsque la mémoire associée à la zone de chaînage est saturée.

Dans le programme donné en annexe, le choix de la division 'à virgule' en base seize ou dix est supposé spécifié par une variable placée sur la pile par le programme appelant celui de la division.

Affichage et autres problèmes.

Pour des raisons de place, l'annexe 2 ne contient pas de procédure d'affichage des résultats pour les programmes multi-précision qu'elle fournit. On peut vérifier ces programmes par la méthode des tests différés, comme cela a été détaillé dans le cas de la multiplication. On peut aussi prendre les utilitaires d'affichages des programmes de calcul des nombres de Fibonacci ou de n!. Leur adaptation ne pose pas vraiment de problème et est donc laissée en exercice au lecteur.

Par ailleurs, l'ensemble des programmes de l'annexe 2 est doté d'un programme d'erreur, le programme **0FD0**. Lorsqu'une erreur se produit, le programme affiche simplement l'adresse, dans le code de l'instruction qui l'a appelé et provoque la terminaison du programme en cours par l'appel à la fonction *DOS* **04C** :

Le programme affiche l'annonce de l'adresse par appel de la fonction *DOS* **09** ; puis, il va 'chercher' l'adresse de retour sur la pile et la diminue de **03** pour obtenir l'adresse de l'instruction d'appel de l'utilitaire **0FD0** :

```
0FD7 55        PUSH BP              0FE1 50        PUSH AX
0FD8 89E5      MOV  BP,SP           0FE2 88E0      MOV  AL,AH
0FDA 8B4602    MOV  AX,[BP+02]      0FE4 30E4      XOR  AH,AH
0FDD 2D0300    SUB  AX,0003         0FE6 B310      MOV  BL,10
0FE0 5D        POP  BP              0FE8 F6F3      DIV  BL
```

Cette adresse est ensuite décomposée en ses deux octets grâce à une division octet par div bl, BL initialisé par **010** (à droite ci-dessus) . Puis chaque octet est affiché par le biais de la fonction *DOS* **02**, les arguments ayant été préparés par les instructions ci-après qui

[1] La limite étant alors la masse de mémoire inerte disponible et/ou la patience de l'utilisateur...

différent des instructions indiquées au paragraphe 5.3.4. car il s'agit ici de chiffres hexadécimaux et non simplement décimaux :

```
0FEA 50        PUSH AX          0FFC 58        POP  AX
0FEB 88C2      MOV  DL,AL       0FFD 88E2      MOV  DL,AH
0FED 80C230    ADD  DL,30       0FFF 80C230    ADD  DL,30
0FF0 80FA3A    CMP  DL,3A       1002 80FA3A    CMP  DL,3A
0FF3 7203      JB   0FF8        1005 7203      JB   100A
0FF5 80C207    ADD  DL,07       1007 80C207    ADD  DL,07
0FF8 B402      MOV  AH,02       100A BA02      MOV  AH,02
0FFA CD21      INT  21          100C CD21      INT  21
                                100E 58        POP  AX
```

On observe le parallélisme des deux parties de l'analyse d'un octet. La seconde partie peut être construite à partir de la première sous *DEBUG* par les commandes *m* **FEA** *L* **12 FFD** et *e* **FFE C2** (*AL* transformé en *AH*). A partir de **100F**, après **pop ax**, rappelant l'adresse obtenue en **0FDD**, on a de nouveau la même séquence d'instructions (dans le code), que de **0FEA** à **100E**, ce qu'on peut obtenir par *m* **0FEA** *L* **24 100F**.

Conclusion

L'ensemble de programmes multi-précision présenté ci-dessus est adapté à un espace mémoire de 1 Mo. Les mémoires vives actuelles peuvent atteindre 16 Mo et, à l'avenir, pourront sans doute posséder des capacités bien plus grandes. Comment faire dans ce cas?

On peut prolonger notre méthode pour 16 Mo sans difficulté : on agrandit la taille des secteurs, en passant, par exemple à seize mots, ce qui permet d'agrandir les entrées de la *TAS* à trois octets, par exemple. On aurait un octet pour le numéro du *segment* où se trouve le secteur considéré et toujours deux octets pour l'adresse du secteur dans le segment de 64 Ko considéré. L'agrandissement obtenu par le passage à des secteurs de seize mots permet en fait des entrées de quatre octets, soit deux mots, ce qui permet de réserver un mot à la numérotation des segments et de fonctionner avec toute la mémoire virtuelle du mode protégé du 80286, par exemple.

Pour obtenir un espace de travail encore plus grand, il faudrait soit de nouveau agrandir les secteurs, en passant, par exemple, à la *page* de **0100** octets (256) ou une taille encore plus grande, soit considérer un autre mode de gestion. On pourrait considérer un sur-système du précédent fonctionnant pour un espace de 4 Go que nous appellerions *volume* et définir un numéro de volume enregistré dans une table adéquate.

Le lecteur concevra sans peine que ce système lui-même a ses limites. Il n'est d'ailleurs pas difficile de construire, en théorie, des fonctions telles que pour représenter leurs valeurs, y compris les premières, il faille des numéros de volume plus grands que la taille d'un volume lui-même. Nous connaissons d'ailleurs le prototype d'une telle fonction avec la fonction d'Ackermann qu'on aurait tort de considérer comme la plus terrible, car dans ce domaine également sans limite, on sait faire incomparablement mieux...

CHAPITRE 7

CONSTRUCTEUR ET MAITRE D'OEUVRE

Après l'application de la modélisation réalisée au chapitre 5 à un domaine théorique connaissant d'importantes applications concrètes[1], nous pouvons appliquer les enseignements que nous en avons tirés à deux outils que nous avons déjà vus : le constructeur et le maître d'oeuvre. Nous commencerons par une nouvelle visite à *DEBUG* et ses fonctions d'assemblage, de désassemblage et d'exécution contrôlée.

7.1. *DEBUG.*

Dans la partie analyse de son travail, le simulateur construit au chapitre 5 imite d'assez près ce que fait *DEBUG* quand il assemble du code. Il s'agit en fait de la même chose mais *à une échelle réduite* puisque le nombre des instructions à coder est très petit. La tâche de *DEBUG* est bien plus importante puisqu'il doit coder, et *décoder* un jeu de plusieurs dizaines d'instructions.

Cependant, dans les deux cas, ce code répond aux mêmes principes :

Une instruction est codée par un ensemble d'octets contigus : de un à six dans le cas des instructions du 8086. Le premier octet 'annonce la couleur' et les octets suivants, selon le type de l'instruction, contiennent les informations données par les opérandes de l'instruction : le numéro d'ordre du registre ou l'adresse de l'instruction référencée pour les instructions Minsky.

Dans le cas des instructions du 8086, le codage est plus complexe car il s'agit de concilier deux impératifs qui peuvent entrer en conflit : des principes simples et rigoureux de codage de façon à rendre plus aisée la *programmation* d'outils du type de *DEBUG*, et la recherche du code le plus court possible afin de gagner en vitesse d'exécution. En effet, et nous reviendrons sur ce point au chapitre 9, le processeur 'lit' le code de chaque instruction avant de le traduire en signaux adéquats dans ses circuits, et plus le code est court, plus vite on passe à l'exécution de l'instruction suivante.

Ce second impératif dégage la notion d'instructions *plus fréquentes* dont il importe de minimiser la durée d'exécution. Ceci va faire que, pour de nombreuses opérations, le codage d'une même instruction pourra prendre *deux formes* : une forme 'longue', découlant d'un *paradigme* général et une forme 'courte', réservée à des cas particuliers de l'instruction, qui de ce fait, seront ***privilégiés***.

EXEMPLE :

L'instruction **mov** est une des plus fréquemment utilisées dans les programmes en assembleur. Le processeur est conçu pour favoriser l'utilisation du registre *AX* ou *AL*,

---

[1] Certaines d'entre elles pourraient être qualifiées de *stratégiques*.

selon le type du transfert effectué. Ainsi, l'instruction **mov ax,[1234]** peut être codée de deux façons :

```
8B 06 34 12                    A1 34 12
```

A gauche, le code 'régulier', sur le modèle général, comme par exemple l'instruction **mov cx,[1234]** toujours codée **8B 0E 34 12**, à droite, le code *privilégié*.

Lors de l'assemblage, *DEBUG* choisit systématiquement[1] le code privilégié. Le test sous *DEBUG* montre que le processeur comprend tout aussi bien la forme longue. Ceci peut avoir de l'importance dans des portions de programmes répétées très souvent.

La forme longue présente aussi un intérêt : l'instruction **0B04** du programme de simulation **cmp si,8000** est assemblée sous une forme régulière, tandis que **cmp si,0000** l'est sous forme privilégiée. Comme le mot **0B06** est réécrit par le programme, il importe que le mot **0B04** ne varie pas et donc, que l'instruction soit toujours codée sous sa forme longue. D'où le choix de la valeur **08000** pour la donnée immédiate, ce qui 'force' la forme longue comme nous le verrons plus loin.

Examinons à présent, les principes de ce codage et leur mise en oeuvre sur les instructions que nous avons vues au chapitre 3 sous leur forme *symbolique*.

7.1.1. Le codage des Instructions.

a. Principes de l'assemblage.

Le codage des instructions du 8086 se fonde sur un principe d'économie : si un seul octet suffit pour coder l'instruction, le code machine n'utilisera qu'un octet. Si un ne suffit pas, en raison, par exemple d'informations *explicites* : une donnée immédiate ou une adresse immédiate, il prendra autant d'octets supplémentaires que cette information le demandera.

Une telle information complémentaire ne peut apparaître que dans une instruction admettant des *opérandes* dont nous avons vu qu'ils peuvent prendre la forme d'adresses indirectes. De ce fait, compte tenu du nombre des instructions et du nombre des types d'adresses indirectes, toute cette information ne peut tenir sur un octet. Le codage du 8086 impose qu'elle n'occupe pas plus d'un mot.

Codop et modrm

Pour toutes les instructions, le premier octet du code de l'instruction est appelé ***codop*** (***cod***e d'***op***ération) et correspond à une première classification où entrent en ligne de compte la fonction de l'instruction, son type et la nature de ses opérandes. Si le codop ne suffit pas, le second octet, dit ***modrm*** (***mod***alité des ***r***egistres et de la ***m***émoire), indique quelle formule est utilisée pour le codage de la source et/ou du but, en particulier, lorsque une adresse indirecte est utilisée. Le modrm indique alors, par les registres ou zones mémoires mises en jeu s'il faut ou non des informations complémentaires. Le modrm est également utilisé pour distinguer des instructions de structures voisines, certaines opérations sur les octets et les mots, par exemple, le codop étant alors conçu comme l'identificateur d'un ***groupe d'instructions***.

Cette notion apparaît notamment pour les instructions du co-processeur mathématique 80*x*87 et pour les instructions du mode protégé à partir du 80286. Un tel *codop* a une fonction de *préfixe* du véritable code. Il s'agit de **D8 à DE** pour les instructions du 80*x*87 et de **0F** pour les principales instructions du mode protégé.

---

[1] Bien qu'il puisse quelquefois être pris en défaut.

Ajoutons que du seul point de vue du désassemblage du code, il est nécessaire que le codop indique s'il est ou non suivi d'un modrm et qu'il indique avec ce dernier, lorsque modrm il y a, la taille en octet de l'instruction.

Structure du modrm.

Faisons le compte de toutes les façons possibles pour désigner les opérandes de l'instruction mov. Le codop fixant le type des instructions, nous avons les associations possibles suivantes, en reprenant les notations utilisées par le dictionnaire des pages 59–62 :

```
rm,rm  rm,rs  rs,rm  rm,ai  rs,ai  ai,rm  ai,rs  rm,dm  ai,dm
```

Les registres `rm` entre eux nous donnent 64 couples possibles. La formule générale de l'adressage que nous avons donnée au chapitre 3 comporte 17 formes différentes. Comme le processeur fera la différence entre une information octet et une information mot afin de gagner un octet pour l'exécution, les combinaisons nécessaires passent à 26. Or 192 = 8.24. De ce fait, certains couples de registre, ai,dm, rm,dm rm,rs et rs,rm seront différenciés par le codop. De même, le codop indique l'ordre des opérandes. Les 26 formes déduites précédemment sont ramenées à 24 en assimilant **[bp]** à **[bp+00]** et une adresse immédiate *octet* à une adresse immédiate *mot*.

L'attribution concrète des numéros d'octet à ces formes s'éclaire singulièrement si l'on convient d'écrire le modrm en ***octal***, c'est-à-dire en base huit. C'est ce que montre le tableau suivant des correspondances entre modrm et forme d'adressage indirect. Le chiffre des unités du modrm indique la ligne du tableau, le chiffre de tête indique la colonne et le chiffre des ***octaines*** indique le registre associé à l'adresse indirecte selon la correspondance définie par les deux dernières colonnes du tableau :

Afin de distinguer la représentation hexadécimale et la représentation décimale d'un nombre de sa représentation octale, nous écrirons cette dernière en italique gras.

	0	***1***	***2***	***3***	
00 :	[BX+SI]	[BX+SI+o]	[BX+SI+m]	AX	AL
01 :	[BX+DI]	[BX+DI+o]	[BX+DI+m]	CX	CL
02 :	[BP+SI]	[BP+SI+o]	[BP+SI+m]	DX	DL
03 :	[BP+DI]	[BP+DI+o]	[BP+DI+m]	BX	BL
04 :	[SI]	[SI+o]	[SI+m]	SP	AH
05 :	[DI]	[DI+o]	[DI+m]	BP	CH
06 :	[m]	[BP+o]	[BP+m]	SI	DH
07 :	[BX]	[BX+o]	[BX+m]	DI	BH

Exemple :

les opérandes **dl,[bx+3B]** définissent le codop **57** puisque **3B** peut se coder sur un octet et que **57** = ***127***; d'où la valeur de la taille de l'instruction : 3 octets.

Observons que 191 = ***277***. Les 64 octets restants s'écrivent, en octal de ***300*** à ***377***. Ces octets correspondent aux couples de registres comme l'indique les deux colonnes du tableau rattachées au chiffre ***3***. Le chiffre des octaines indique alors le registre *source*. Ainsi, **mov ax,di** et **mov al,bh** induisent un même codop : **F8** = ***370***.

L'ordre indiqué par les deux dernières colonnes du tableau ci-dessus est souvent appelé ***l'ordre des registres***. On peut noter que dans les trois premières colonnes, les lignes sont placées dans un ordre *induit* par celui des registres.

Le placement des registres généraux avant les registres d'adressage relatif constitue un choix quelque peu arbitraire. Le choix de l'ordre à l'intérieur de ces deux groupes est encore plus arbitraire et donne une couleur 'zoologique' à ce codage, ce qui est, au demeurant, dans la nature des choses.

De la même façon, il existe un ordre des registres de segment :

00	01	02	03
ES	CS	SS	DS

Nous verrons, avec les instruction **mov**, **push** et **pop**, comment est organisé le codage des opérandes où intervient un registre de segment.

Structure du codop.

La définition de la correspondance entre le codop et les instructions qu'il gouverne est moins apparente que celle du modrm.

La parité du codop est significative pour les instructions pouvant fonctionner dans les deux types d'opérandes. Un ***codop pair*** désigne une instruction de ***type octet***, un ***codop impair*** une instruction de ***type mot***.

D'une façon plus générale, une certaine régularité apparaît si l'on représente la correspondance entre codops et instructions sous la forme de tableaux 8*8, ce qui correspond encore à une écriture octale du code. Comme l'indiquent ces tableaux, donnés pages 156 et 157, si le chiffre des octaines définit le numéro de ligne, il apparaît certaines régularités indiquant un regroupement par fonctions. Ainsi, si **inc**, **dec**, **push**, **pop**, *jcond* et certains **mov** présentent une régularité frappante, d'autres correspondances sont moins régulières, ce qui rend toute mnémonique rationnelle simple impossible.

b. Histoire naturelle du 8086.

Voyons, à présent, la correspondance précise des codops et des instructions.

Nous suivons ici une classification beaucoup plus ***syntaxique*** qu'au chapitre 3. Nous regroupons d'abord les instructions en fonction du nombre de leurs opérandes puis, à l'intérieur de chaque groupe, nous les regroupons par similitude des propriétés du code, ce qui parfois recoupe le classement fonctionnel adopté au chapitre 3. Par contre, nous reprendrons la notation des opérandes introduite pour le dictionnaire des instructions, en y ajoutant l'abréviation *AR* pour désigner une adresse indirecte ou un registre (hors segments), le type étant au besoin précisé par les qualifiants **by** ou **wo**..

Instructions à deux opérandes.

Tout d'abord, chef de file d'une famille nombreuse, l'instruction **mov** :

```
MOV :
      089     MOV  AR,RM        088     MOV  AR,RO
      08B     MOV  RM,AR        08A     MOV  RO,AR
      0A1     MOV  AX,[dm]      0A0     MOV  AL,[dm]
      0A3     MOV  [dm],AX      0A2     MOV  [dm],AL
   0BF-0B8    MOV  RM,dm     0B7-0B0    MOV  RO,do
      0C7     MOV  WO AI,dm     0C6     MOV  BY AI,do      0
      08E     MOV  RS,AR
      08C     MOV  AR,RS
```

Remarquer le caractère privilégié des codops **A0-A3**. Ils ne font que doubler avantageusement les formes longues **88-8B** dans le cas des registres *AX* et *AL*, ces formes longues étant comprises par le processeur (et *DEBUG* aussi, accessoirement). Enfin, les codops **B0-BF**, également privilégiés, représentent un cas de codage des registres dans le codop : en effet, le quartet faible correspond au numéro d'ordre du registre : ainsi **mov dx,1234** est codé **BA 34 12**.

EXERCICE :

Donnez les codes des instructions suivantes :

```
MOV AL,[BX+721] ;  MOV  SI,9A8B ;  MOV WO [BP+DI-04],110
```

A quelles instructions correspondent les codes suivants :

```
8E  94  32  ;  C6  46  F4  D8
```

Efforcez-vous de répondre sans utiliser *DEBUG*, dans un premier temps.

Pour les instructions **mov** faisant apparaître les registres de segments, le codage des registres suit la même règle que pour les instructions de codops **88-8B** : le chiffre des octaines fournit le numéro du registre de segment concerné dans l'ordre des registres de segment. Ainsi, **mov ds,[bx+12]** est codé **8E 5F 12** puisque **5F** = *137, 3* correspondant bien à *DS* et *17* à une expression de la forme **[bx+***o***]**, où *o* désigne une donnée immédiate d'un octet.

D'après le tableau ci-dessus, les groupes **88/89** et **8A/8B** ont une 'intersection commune' : en effet, une instruction du type **mov** *RM,RM* peut aussi bien être codée par **89** ou **8B**. Compte tenu du rôle tenu par le registre dans ces deux instructions, le même couple de registres ne sera pas codé de la même façon dans l'un et l'autre cas : les codages seront 'symétriques'. Ainsi **mov ax,dx** est 'normalement' codé **89 D0**, mais aussi **8B C2**, tandis que **89 C2** et **8B D0** codent tous deux **mov dx,ax**.

Enfin, la ligne **C6/C7** comporte un chiffre octal en 'troisième colonne'. Il correspond, en effet, à un *groupement* contenant, avec beaucoup d'autres, l'instruction **mov**. Comme nous l'avons indiqué, la distinction se fait par le modrm puisque toute l'information qu'il contient n'est pas utilisée par l'adressage. La décimale octale non utiliséeest est, en l'occurrence, le chiffre des octaines. Le chiffre 0 définit l'instruction **mov** pour ces deux codops.

La famille MOV.

Dans ce groupe, mais avec moins de codops privilégiés, les instructions d'opération à deux opérandes. Nous reprenons la forme du tableau pour l'instruction **add** :

```
ADD :
      001   ADD  AR,RM        000   ADD  AR,RO
      003   ADD  RM,AR        002   ADD  RO,AR
      005   ADD  AX,dm        004   ADD  AL,do
      081   ADD  AR,dm        080   ADD  AR,do        0
      083   ADD  AR,do        082   ADD  AR,do        0
```

Observer le codop **083** : il présente un caractère exceptionnel puisqu'on affecte un registre *mot* ou un *mot* mémoire par une donnée immédiate *octet*. En fait, la règle des types est respectée dans le fonctionnement de l'instruction. La donnée octet est interprétée comme *entier relatif* et transformée en mot ayant la même valeur : on dit que le signe de l'entier a été ***étendu*** au format mot. C'est ce mot qui est ajouté au but par le processeur.

EXEMPLE :

L'instruction **add si,FFFF** possède de ce fait deux codages :

```
81 C6 FF FF  ADD SI,FFFF      83 C6 FF  ADD SI,-01
```

dont l'action sur le processeur est la même.

Voici le tableau des instructions qui possèdent les mêmes caractéristiques. L'instruction est désignée par la mnémonique **opr** et l'écriture des codes hexadécimaux condensée sur deux chiffres :

```
                groupe MOV :
 add     adc     and     cmp     or      sbb     sub     xor
01 00   11 10   21 20   39 38   09 08   19 18   29 28   31 30    OPR AR,RM/AR,RO
03 02   13 12   23 22   3B 3A   0B 0A   1B 1A   2B 2A   33 32    OPR RM,AR/RO,AR
05 04   15 14   25 24   3D 3C   0D 0C   1D 1C   2D 2C   35 34    OPR AX,dm/AL,do
81 80   81 80   81 80   81 80   81 80   81 80   81 80   81 80    OPR AR,dm/AR,do
83 82   83 82   83 82   83 82   83 82   83 82   83 82   83 82    OPR AR,do
  0       2       4       7       1       3       5       6
```

La règle de formation de ces **codops** apparaît sur ce tableau : si on appelle ***codop de base*** le codop pair de la première ligne, les codops s'écrivent *en octal u0* où *u* est le chiffre des octaines du modrm associé aux codops des lignes 4 et 5 qui sert à distinguer

les instructions relevant d'un même codop de *groupe*. Ce qui apparaît également sur le tableau des pages 156 et 157.

Le groupe TEST.

Parmi les instructions à deux opérandes, une place à part est tenue par les instructions **test** et **xchg** :

```
TEST :
     085     TEST  AR,RM       084     TEST  AR,RO
     085     TEST  RM,AR       084     TEST  RO,AR
     0A9     TEST  AX,dm       0A8     TEST  AL,do
     0F7     TEST  AR,dm       0F6     TEST  AR,do          0
XCHG :
     087     XCHG  AR,RG       086     XCHG  AR,DR
   090-090   XCHG  RM,AX
```

L'assemblage de ces instructions diffère de celui des instructions de la famille **mov** sur le point suivant :

La répartition but/source entre les décimales octales du modrm est le contraire de ce qu'elle est dans le cas **mov** :

```
0100 85 D4    TEST DX,SP          0100 85 E2    TEST SP,DX
0102 89 E2    MOV  DX,SP          0102 89 D4    MOV  SP,DX
0104 87 D4    XCHG DX,SP          0104 87 E2    XCHG SP,DX
```

Cependant, le codage des instructions est le *même* lorsque les deux opérandes sont constitués par un registre et par une adresse indirecte puisque un même codop est associé quelque soit l'ordre des opérandes.

Par ailleurs, le statut privilégié des registres *AX* et *AL* est encore observé. Notons que les instructions de codops **090** à **098** n'occupent qu'un octet et correspondent à l'ordre des registres. L'instruction **nop** est donc un synonyme de l'instruction **xchg ax,ax** qui, effectivement prend un peu de temps sans modifier les résultats.

Une curiosité qui n'apparaît pas immédiatement sur le tableau :

```
0100 93      XCHG  BX,AX
0101 97C3    XCHG  AX,BX
```

Le groupe LES/LDS :

```
Les codops sont donnés ci-après :
     LES :     0C4    LES  RM,AI     LDS :     0C5    LDS  RM,AI
```

Les instructions à opérande unique.

Nous avons vu que les instructions du groupe **mov** d'opérandes *AR,dm* (*AR,do*) possèdent la propriété d'être toutes regroupées sous un même **codop**, la distinction entre *opérations* étant indiquées par le chiffre des octaines du **modrm**. Ceci constitue la règle des instructions à opérande unique puisque dans ce cas le chiffre octal qui n'est plus affecté au codage d'un opérande peut être utilisé dans ce nouveau but.

Instructions de saut.

Les codops sont indiqués sur le tableau de la page ci-contre, le chiffre octal de discrimination des instructions étant mentionné à côté du codop dans le cas d'un codop de groupe.

On remarque que pour le codop **0FF**, la distinction entre **jmp** et **call** d'une part, et entre saut intra- et inter-segment proche et d'autre part, est uniquement assumée par le chiffre des octaines du modrm.

JMP, CALL :

```
      0EB      JMP  dm
      0E9      JMP  dm             0E8      CALL dm
      0EA      JMP  dm1:dm2        09A      CALL dm1:dm2
      0FF  4   JMP  AR             0FF  2   CALL AR
      0FF  5   JMP  FAR AI         0FF  3   CALL FAR AI
```

Autre particularité : l'instruction de saut inconditionnel *proche* n'a pas d'analogue pour l'appel d'un programme, ce qui peut se justifier. Du point de vue du langage symbolique, les deux types de saut inconditionnels ne se distinguent que par le contexte du texte écrit. Au niveau de l'assemblage, la distinction est faite par le codop. Ainsi **0EB** annonce une instruction sur deux octets, l'octet fort contenant le déplacement algébrique à l'instruction but, tandis que **0E9** annonce une instruction sur trois octets, le *mot* fort indiquant un déplacement défini selon le même schéma.

HISTOIRE NATURELLE. Les concepteurs du processeur semblent avoir hésité entre plusieurs façons de distinguer **jmp** de **call**. Ainsi, **0E8** s'oppose-t-il à **0E9**, tandis que **0EA** s'oppose à **09A**. Ceci rappelle les hésitations entre deux paradigmes existants ou possibles que l'on peut observer dans les grammaires de nombreuses langues naturelles.

Nous avions remarqué que les instructions **j***cond*, **loop** et **loop***cond* ont une syntaxe calquée sur celle du **jmp** proche. La raison en est la similitude de codage, ce qu'indique le tableau suivant :

groupe ***Jcond*** :

jo	**jno**	**jb**	**jae**	**jz**	**jnz**	**jbe**	**ja**	**js**	**jns**	
070	071	072	073	074	075	076	077	078	079	OPR dm
07A	07B	07C	07D	07E	07F	0E0	0E1	0E2	0E3	OPR dm
jp	**jnp**	**jl**	**jge**	**jng**	**jg**	**loopne**	**loope**	**loop**	**jcxz**	

Opérations.

Nous regroupons à présent les instructions d'opération à un opérande : syntaxe et logique se retrouvent assez souvent, mais l'aspect histoire naturelle du codage semble prendre le pas.

groupe **RCR** :

rcl	**rcr**	**rol**	**ror**	**shl**	**shr**		
D1 D0	D1 D0	D1 D0	D1 D0	D1 D0	D1 D0	OPR ARM,1	OPR ARO,1
D3 D2	D3 D2	D3 D2	D3 D2	D3 D2	D3 D2	OPR ARM,CL	OPR ARO,CL
2	***3***	***0***	***1***	***4***	***5***		

où ici, et dans les tableaux suivants, ARM indique un registre mot (segments exclus) ou un mot mémoire et ARO un demi-registre ou un octet mémoire.

Noter que la différence du statut de l'opérande *quanta* (*cf.* dictionnaire des instructions du chapitre 3) est prise en charge par le codop.

groupe **DIV** :

not	**neg**	**mul**	**imul**	**div**	**idiv**		
F7 F6	F7 F6	F7 F6	F7 F6	F7 F6	F7 F6	OPR ARM	OPR ARO
2	***3***	***4***	***5***	***6***	***7***		

groupe **INC** :

inc	**dec**	pop	push		
47-40	4F-48	5F-58	57-50	OPR RM	
FF FE	FF FE			OPR ARM	OPR ARO
		8F	FF	OPR ARM	
0	***1***	***0***	***6***		

Dans ce groupe, les instructions **pop** et **push** avec les registres de segments occupent une place à part que partagent les instructions de préfixage :

```
ES      CS      SS      DS
07              17      1F      POP  RS
06      0E      16      1E      PUSH RS
26      2E      36      3E      RS:
```

La structure de ce tableau est encore plus claire si on écrit les codops en octal : le chiffre des unités est toujours ***7*** pour **pop**, ***6*** pour **push** et les instructions de préfixage ; le chiffre des octaines est ***0,1,2,3*** pour **pop** et **push**, ***4,5,6,7*** pour le préfixage. Ainsi, ***76*** = **3E** et ***37*** = **1F**.

Noter l'absence de codop pour **pop cs**. On remarquera que la plupart des versions de *DEBUG* acceptent cette instruction et lui font correspondre l'unique octet **0F** aussi bien à l'assemblage qu'au désassemblage. Ce codop est réservé pour la plupart des instructions du mode protégé des 80x86, $x > 1$, et provoque, en général le plantage du système sur les 8086, car le processeur ne le reconnaît pas. Par contre, **0E/push cs** est parfaitement accepté et même souvent utilisé.

Interruptions et entrées sorties.

Les instructions **int** ont pour **copod 0CD** et, étant de la forme `OPR dm`, elles sont donc codées sur deux octets. Exception, comme nous l'avons vu : l'interruption 03, utilisée par *DEBUG* codée par son seul codop : **0CC**.

Quant aux entrées-sorties, ce sont des instructions de statut *réel* mixte : à un opérande ou sans opérande selon le cas. Ainsi, une instruction **out dx,al** fonctionne en fait avec deux opérandes implicites qu'un seul octet suffit à coder. On a le tableau :

```
 PI       DX
E5 E4    ED EC          IN  AX,PR     IN  AL,PR
E7 E6    EF EE          OUT PR,AX     OUT PR,AL
```

`PI` signifiant un port désigné par une donnée immédiate octet.

Instructions sans opérandes.

Nous avons déjà rencontré quelques instructions sans opérandes : les instructions de préfixage, certaines instructions d'entrée/sortie et l'instruction **int 03**. Nous regroupons les autres instructions de ce type par ordre alphabétique dans le tableau ci-dessous :

```
clc  cld  cli  cmc  iret lahf popf rep  ret  retf sabf stc  std  sti
F8   FC   FA   F5   CF   9F   9D   F3   C3   CB   9E   F9   FD   FB

cmpsb cmpsw lodsb lodsw pushf repne stosb stosw
A6    A7    AC    AD    9C    F2    AA    AB
```

On observe un certain nombre de régularités rapprochant, par exemple, les instructions du type **clf** et du type **stf**, **popf** et **pushf**. On retrouve le lien parité et type octet/mot pour les instructions **cmps**, **lods** et **stos**. Le rapprochement entre **ret** et **retf**, par exemple est plus apparent si l'on écrit ces nombres en octal : ***303*** et ***313*** respectivement.

Notons que les instructions **ret** et **retf** peuvent être employées avec un opérande de type mot. Ceci se traduit par un codop différent :

```
C2    RET  dm          CA    RETF dm
```

le codop entraînant par lui-même que l'instruction doit occuper trois octets.

c. Désassemblage.

Nous donnons, pages 156 et 157 un tableau permettant, en théorie, d'effectuer l'opération inverse : partant du code d'une instruction, retrouver son expression symbolique. Ce tableau est décomposé en quatre parties correspondant au chiffre de tête de la représentation octale d'un octet qui va, en effet, de *000* à *377*. Sur les quatre tableaux, on retrouve les règles de formation des codops des instructions, leur association avec le modrm et les aspects *histoire naturelle* qu'il contient nécessairement.

On en déduit facilement le plan d'un désassembleur à la manière de *DEBUG*.

Avec des tableaux.

En effet, on peut organiser le décodage à partir de trois tableaux :

le tableau 1, de 256 octets, soit une entrée par codop, classe les codops en 11 classes, reprenant les groupes que nous avons énumérés ci-dessus auquel on ajoute un groupe constitué par les codops inconnus du processeur :

```
00 : sans opérande ou instruction déterminée par codop seul,
```

classe qui contient complètement certains des groupes vus au paragraphe *b* et partiellement la plupart des autres, qui constituent les autres classes :

```
01 : instruction mov              06 : groupe rcr
02 : groupe mov (mov exclu)       07 : groupe div
03 : groupe test/xchg             08 : groupe inc
04 : les/lds                      09 : groupe in/out
05 : jmp/call
```

La classe est indiquée par le quartet faible de l'octet fourni par le tableau, le groupe des codops inconnus se distinguant par l'octet **080**. Le quartet fort, quant à lui, indique le nombre d'octets à lire en qualité de données immédiates, les octets entrant éventuellement dans la formule de l'adressage indirect n'étant pas décomptés car indiqués par le modrm.

Le tableau 2, de 256 mots, également une entrée par codop, donne l'adresse, dans le tableau 3 des chaînes de caractères, de celle qui est associée à la mnémonique de l'instruction, lorsque le codop permet de trancher. Sinon, il indique une adresse à laquelle on ajoutera 2**t*, où *t* est le chiffre de tête octal du modrm, le résultat étant un pointeur sur l'adresse de la mnémonique de l'instruction, ce que nous expliquons ci-après, à l'exemple 3.

Le tableau 3 contient tous les mots composants les mnémoniques des instructions, des registres et des qualifiants. Ils sont mis bout à bout par ordre de taille, et, à taille égale, par ordre alphabétique. L'adresse, quant à elle, permet de savoir la taille du composant considéré.

EXEMPLES :

Le début du tableau 1 est donc le suivant :

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14
02 02 02 02 11 21 00 00 02 02 02 02 11 21 00 80 02 02 02 02 11
```

Quant au tableau 2, nous n'en donnons que des extraits, pour chaque groupement par taille (ce qui n'est pas fait dans la transcription dans le code de ce tableau) :

```
al ah ax bl bh bx by cl ch cs cx dl dh ds dx es in ja    ...
aaa aad aam aas adc add and cbw clc cld cli cmc cmp cwd ...
call idiv imul into iret lahf lock loop popf push retf  ...
cmpsb cmpsw lodsb lodsw loopz movsb movsw pushf repnz    ...
loopnz
```

Le mode d'emploi de ces tableaux est expliqué page 158.

Tableaux des codops des instructions.

Les entrées correspondent à la représentation des octets en octal. Le tableau numéroté i correspond au chiffre de tête. Dans chaque entrée, la syntaxe de l'instruction et le codop en hexadécimal.

0	0	1	2	3	4	5	6	7
0	add AR,RO 00	add AR,RM 01	add RO,AR 02	add RM,AR 03	add AL,do 04	add AX,dm 05	push ES 06	pop ES 07
1	or AR,RO 08	or AR,RM 09	or RO,AR 0A	or RM,AR 0B	or AL,do 0C	or AX,dm 0D	push CS 0E	286 0F
2	adc AR,RO 10	adc AR,RM 11	adc RO,AR 12	adc RM,AR 13	adc AL,do 14	adc AX,dm 15	push SS 16	pop SS 17
3	sbb AR,RO 18	sbb AR,RM 19	sbb RO,AR 1A	sbb RM,AR 1B	sbb AL,do 1C	sbb AX,dm 1D	push DS 1E	pop DS 1F
4	and AR,RO 20	and AR,RM 21	and RO,AR 22	and RM,AR 23	and AL,do 24	and AX,dm 25	ES: 26	daa 27
5	sub AR,RO 28	sub AR,RM 29	sub RO,AR 2A	sub RM,AR 2B	sub AL,do 2C	sub AX,dm 2D	CS: 2E	das 2F
6	xor AR,RO 30	xor AR,RM 31	xor RO,AR 32	xor RM,AR 33	xor AL,do 34	xor AX,dm 35	SS: 36	aaa 37
7	cmp AR,RO 38	cmp AR,RM 39	cmp RO,AR 3A	cmp RM,AR 3B	cmp AL,do 3C	cmp AX,dm 3D	DS: 3E	aas 3F

1	0	1	2	3	4	5	6	7
0	inc AX 40	inc CX 41	inc DX 42	inc BX 43	inc SP 44	inc BP 45	inc SI 46	inc DI 47
1	dec AX 48	dec CX 48	dec DX 4A	dec BX 4B	dec SP 4C	dec BP 4D	dec SI 4E	dec DI 4F
2	push AX 50	push CX 51	push DX 52	push BX 53	push SP 54	push BP 55	push SI 56	push DI 57
3	pop AX 58	pop CX 59	pop DX 5A	pop BX 5B	pop SP 5C	pop BP 5D	pop SI 5E	pop DI 5F
4	286 60	286 61	286 62	286 63	286 64	286 65	286 66	286 67
5	286 68	286 69	286 6A	286 6B	286 6C	286 6D	286 6E	286 6F
6	jo dm 70	jno dm 71	jb dm 72	jnb dm 73	jz dm 74	jnz dm 75	jna dm 76	ja dm 77
7	js dm 78	jns dm 79	jp dm 7A	jnp dm 7B	jl dm 7C	jnl dm 7D	jng dm 7E	jg dm 7F

Dans le tableau numéroté 2, nous avons écrit [m] pour [dm] pour des raisons de place. Les cinq codops de groupes apparaissant dans ces tableaux ont été détaillés antérieurement, aussi ne sont-ils pas davantage précisés.

2	0	1	2	3	4	5	6	7
0	opr1 AR,do 80	opr1 AR,dm 81	opr1 do 82	opr1 do 83	test SP 84	test BP 85	xchg SI 86	xchg DI 87
1	mov AR,RO 88	mov AR,RM 88	mov RO,AR 8A	mov RM,AR 8B	mov AR,RS 8C	lea AI 8D	mov RS,AR 8E	pop AR 8F
2	nop 90	xchg CX,AX 91	xchg DX,AX 92	xchg BX,AX 93	xchg SP,AX 94	xchg BP,AX 95	xchg SI,AX 96	xchg DI,AX 97
3	cbw AX 98	cwd CX 99	call dm:dm 9A	wait BX 9B	pushf SP 9C	popf BP 9D	sahf SI 9E	lahf DI 9F
4	mov AL,[m] A0	mov AX,[m] A1	mov [m],AL A2	mov [m],AX A3	movsb A4	movsw A5	cmpsb A6	cmpsw A7
5	test A8	test A9	stosb AA	stosw AB	lodsb AC	lodsw AD	scasb AE	scasw AF
6	mov AL,do B0	mov CL,do B1	mov DL,do B2	mov BL,do B3	mov AH,do B4	mov CH,do B5	mov DH,do B6	mov BH,do B7
7	mov AX,dm B8	mov CX,dm B9	mov DX,dm BA	mov BX,dm BB	mov SP,dm BC	mov BP,dm BD	mov SI,dm BE	mov DI,dm BF

3	0	1	2	3	4	5	6	7
0	opr2 AR,do C0	opr2 AR,dm C1	ret dm C2	ret C3	les AI C4	lds AI C5	mov AI,do C6	mov AI,dm C7
1	286 C8	286 C8	retf dm CA	retf CB	int 3 CC	int do CD	into CE	iret CF
2	opr3 ARO,1 D0	opr3 ARM,1 D1	opr3 ARO,CL D2	opr3 ARM,CL D3	aam D4	aad D5	D6	xlat D7
3	x87 D8	x87 D9	x87 DA	x87 DB	x87 DC	x87 DD	x87 DE	x87 DF
4	loopnz dm E0	loopz dm E1	loop dm E2	jcxz dm E3	in AL,do E4	in AX,do E5	out do,AL E6	out do,AX E7
5	call dm E8	jmp dm E9	jmp dm:dm EA	jmp dm EB	in AL,DX EC	in AX,DX ED	out DX,AL EE	out DX,AX EF
6	lock F0		repnz F2	rep F3	hlt F4	cmc F5	opr4 ARO F6	opr4 ARM F7
7	clc F8	stc F9	cli FA	sti FB	cld FC	std FD	opr5 AR FE	op5 AR FF

Mode d'emploi.

Nous illustrons l'utilisation de ces tableaux sur quatre exemples correspondants à des valeurs particulières du quartet faible des entrées du tableau 1, puisque le cas d'une entrée avec $b_7 = 1$ est immédiatement écarté, le désassembleur indiquant, sous forme conventionnelle, par exemple ??? comme le fait *DEBUG*, que ce codop est inconnu. On supposera que le pointeur de lecture est *DS:SI* et que le mot est lu dans *AX*.

EXEMPLE 1 :

On lit sous *DS:SI* l'octet **0AD**. Le tableau 1 renvoie **00** : instruction d'un octet. Le tableau 2 fournit à l'entrée d'adresse relative **015A**, l'adresse de la mnémonique **lodsw**, (l'adresse indiquant par elle-même que la mnémonique a 5 caractères). Le décodage est terminé et *SI* est incrémenté de 1.

EXEMPLE 2 :

On lit, à présent, **0EA**. Le tableau 1 fournit **45** : instruction de classe 05 dont le codop est suivi de 4 octets de données. Le tableau 2 donne l'adresse de la mnémonique : **jmp**. Le seul cas possible, pour une instruction de classe 5, d'une donnée immédiate sur quatre octets est un **jmp** ou un **call**. Dans les deux cas, la syntaxe de l'instruction après la mnémonique est la même. Ce qui permet de procéder à la transcription et *SI* est augmenté de 5.

EXEMPLE 3 :

On lit **083**. Le tableau 1 indique **012**, d'où classe 02. On lit donc le modrm en *AH*, soit, par exemple, **059**. Le tableau 2 donne l'adresse **02000**, ce qui signifie qu'on se trouve en présence d'un codop de groupe. On écrit le modrm en octal : **059** = ***131***. Le chiffre ***3*** fournit l'adresse **2006** (=2000 + 2*3), où se trouve l'adresse de la mnémonique **sbb** dans le tableau 3. Le 1 de tête indique une déplacement algébrique exprimé sur un octet et l'octet fort de **012** signale un octet de donnée immédiate. L'instruction est donc de taille 4. Supposons que les 4 octets soient : **83 59 F4 12**, on trouve ainsi l'instruction **sbb wo [bx+di–0C],+12.**

EXEMPLE 4 :

On lit **0F6**. Le tableau 1 indique la classe 03. Supposons que le modrm soit **061**. On a : **061** = ***141***. Le 1 de tête signifie qu'il faut lire un octet pour l'adresse indirecte. Le tableau 1 ne signale pas de donnée immédiate. Donc, l'instruction est de taille 3. Si le troisième octet est **024**, comme le chiffre des octaines du modrm, selon le même mécanisme que ci-dessus renvoie, par une seconde adresse à l'adresse de **mul**, on a donc que **F6.61 24** est le code de l'instruction **mul by [bx+di+24]**.

Les lignes principales d'un programme de désassemblage étant ainsi tracées, l'exemple du simulateur donné au chapitre 5 nous paraît suffisant pour que le lecteur puisse de lui-même, écrire un tel programme, s'il en éprouve le désir.

7.1.2. L'exécution contrôlée.

Nous avons donné, au chapitre 4 le principe sur lequel repose le fonctionnement de la commande *g* de *DEBUG*. Nous indiquerons, à présent comment l'implanter. Auparavant, nous reviendrons sur le perfectionnement que l'on peut apporter au programme de simulation d'une machine de Minsky.

Pas à pas et points d'arrêts pour une machine de Minsky.

Le programme donné en annexe 1 du simulateur construit au chapitre 5, comporte deux contrôles de l'exécution : l'accès à l'instruction Minsky HLT implicite et le débordement de la *RAM* lors de l'allocation d'un secteur.

On pourrait aisément apporter des modifications permettant de reproduire, dans le cadre des programmes Minsky, l'action de *DEBUG*.

Il suffit de créer un nouveau programme, disons depuis l'adresse **02000**, contrôlant en fait l'action du programme d'exécution **0B00**. A l'adresse **013A**, on remplacerait **call 0B00** par **call 2000**, et les références **0B05** (ou **0B06** si on utilise le programme du chapitre 5), par la référence **01FFE**.

Le programme **02000** est constitué par une boucle de dialogue. L'utilisateur précise, par une commande 'à la *DEBUG*', une des trois options : *p* pour pas à pas, *g* pour exécution jusqu'au terme du programme, *g nombre* pour l'exécution jusqu'à ce qu'on rencontre l'instruction numérotée *nombre*.

L'implantation des trois commandes repose sur un mécanisme unique : il s'agit en fait de contrôler la boucle **0B03** du programme **0B00**. Or, cette boucle est chapeautée par un test d'entrée en **0B03** par l'instruction **cmp si,***adresse*, le renvoi à **0B03** étant effectué par des sauts inconditionnels. Il suffit donc d'agir sur la donnée immédiate *adresse* de l'instruction **0B03**, le programme **02000** gardant toujours la dernière valeur connue de *SI* après chaque appel à **0B00**, le registre *SI* jouant dans ce programme le rôle de *IP* dans *DEBUG*.

Dans le cas de la commande *g* sans paramètre, le programme **02000** place en **0B05** l'adresse de l'instruction Minsky HLT implicite et appelle le programme **0B00**, puis le programme **0DE0** d'affichage des résultats.

Dans le cas de la commande *p*, il faut simplement exécuter un pas de la boucle du programme **0B00**. Pour cela, le programme **02000** exécute la séquence d'instructions suivante ramenée à l'adresse **0100** :

```
0100 2E          CS:              ;
0101 3B36FE1F    CMP  SI,[1FFE]   ; programme Minsky terminé?
0105 JAE  0120                    ; oui : traitement approprié
0107 2E          CS:              ; non : remplacer
0108 C606030BC3  MOV  BY [0B03],C3; l'instruction 0B03 par RET
010D E8          CALL 0B09        ; appeler 0B00 depuis 0B09
0110 2E          CS:              ; restaurer l'ancienne
0111 C606030B81  MOV  BY [0B03],81; instruction 0B03.
```

Le même jeu de réécriture est utilisé pour exécuter l'instruction *g* avec paramètre. En effet, après vérification avec la valeur du mot en **036C** (*cf.* instruction **031A** dans le programme de transcodage), la valeur du paramètre est convertie en adresse d'instruction et placée comme donnée immédiate de l'instruction **0B03**.

La partie du programme **02000** la moins aisée à écrire est sans doute celle qui concerne le dialogue avec l'utilisateur. La lecture des 'commandes' se fera par la fonction *DOS* **0A**. L'analyse de la commande fait appel aux instructions **rep** et **cmps** et utilise, dans le cas de l'instruction *g* avec paramètres, le programme de conversion d'un nombre entré à l'écran que nous avons vu au chapitre 4.

Implantation de l'exécution contrôlée sous DEBUG.

Dans le cas de *DEBUG*, le contrôle est implanté de façon tout à fait différente pour une raison simple : l'exécution de l'instruction testée est le fait du processeur et

non de *DEBUG*. Il faut donc que *DEBUG* retrouve le contrôle du processeur une fois que celui-ci a exécuté l'instruction ou la séquence d'instructions qui lui est soumise.

Le cas du pas à pas est le plus simple : le processeur a été conçu pour fonctionner en mode pas à pas. Il suffit, pour cela, d'agir sur l'indicateur *TF*. Habituellement, le bit de cet indicateur vaut 0. Quant il vaut 1, le processeur exécute une instruction, remet *TF* à 0 puis appelle l'interruption **01**. Au début de la session, *DEBUG* a naturellement installé un gestionnaire adéquat de cette interruption. De la même manière, comme nous l'avons indiqué au chapitre 4, la commande *g* peut fonctionner grâce au code privilégié de l'interruption **03**, ramené à un octet. Les instructions ci-dessus nous montrent sans peine les jeux de réécritures auxquels *DEBUG* procède pendant l'exécution de cette interruption. La commande *p* est simplement une commande *g* à l'adresse de l'instruction suivante dans les cas de **loop**, **rep**, **call** et **int**, une commande *t* dans tous les autres.

Le fonctionnement du bit *TF* peut se voir grâce à la séquence suivante où (*F*) désigne le registre des indicateurs :

```
0100 PUSHF          ; mettre (F) sur la pile
0101 POP  AX        ; AX := (F)
0102 OR   AH,01     ; mettre TF à 1
0104 PUSH AX        ; nouveau (F) sur la pile
0105 POPF           ; nouveau (F) en place
```

Si on fait suivre cette séquence de trois **mov**, par exemple, et qu'on lance la commande *g* = **100**, le premier **mov** sera exécuté mais pas les suivants.

REMARQUE : avec *TF* = 0 (cas ordinaire), cette dernière commande ferait planter la machine.

Installation d'une interruption.

Le mécanisme d'installation d'une interruption est extrêmement simple. Deux fonctions *DOS* président à cette opération.

La fonction **035** fournit dans *ES:BX* l'adresse du gestionnaire de l'interruption dont le numéro hexadécimal lui a été indiqué dans *AL* telle que cette adresse est indiquée par la table des interruptions. La fonction **025** place à l'entrée de l'interruption numérotée *AL* l'adresse absolue fournie par *DS:DX*. On peut donc, modifier une interruption à sa guise ou en créer une, les interruptions non encore utilisées étant signalées dans la table par une entrée **0000:0000**.

Le mécanisme de réservation et d'appel est illustré par les deux séquences suivantes ramenées à **0100** :

A gauche, la recherche d'une interruption libre, à droite, son installation.

```
0100 B90001    MOV  CX,0100          0130 88164201   MOV  [013E],DL
0103 BA4B00    MOV  DX,004B          0134 88163B01   MOV  [0146],DL
0106:BOUCLE                          0138 90         NOP
0106 B435      MOV  AH,35            0139 90         NOP
0108 88D0      MOV  AL,DL            013A 90         NOP
010A CD21      INT  21               013B B425       MOV  AH,25
010C 8CC0      MOV  AX,ES            013D B000       MOV  AL,00
010E 3D0000    CMP  AX,0000          013F CD21       INT  21
0111 7505      JNZ  0118             0141 58         POP  AX
0113 83FB00    CMP  BX,+00           0142 5B         POP  BX
0116 7410      JZ   0128             0143 59         POP  CX
0118 FEC2      INC  DL               0144 5A         POP  DX
011A E2EA      LOOP 0106             0145 CD00       INT  00
```

Les trois **nop** à droite permettent de ne pas réécrire une instruction que le processeur exécute et donc, a lu avant qu'elle ne soit modifiée. Le nombre de ces **nop**s doit être sensiblement augmenté pour les processeurs 80*x*86, $x \geq 3$.

REMARQUE : On peut gérer soi-même la lecture de la table d'interruption puisque cette structure ne dépend pas du système d'exploitation, mais du processeur, qui l'attend dans la zone **0000:0000-0400**. On multiplie le numéro de l'interruption par 4 pour obtenir l'adresse de l'entrée correspondante dans la table. Comme on le sait, cette entrée contient l'adresse absolue du gestionnaire de l'interruption. Si ce numéro est placé dans *AL*, les instructions de gauche simulent la fonction *DOS* **035** et celles de droite la fonction **025** :

```
0100 MOV  CL,04          0100 MOV  CL,04
0102 MUL  CL             0102 MUL  CL
0104 MOV  SI,AX          0104 MOV  DI,AX
0106 XOR  AX,AX          0106 XOR  AX,AX
0108 MOV  DS,AX          0108 MOV  ES,AX
010A LES  BX,[SI]        010A ES:
                         010B MOV  [DI],DX
                         010D ES:
                         010E MOV  [DI+02],DS
```

7.2. Le maître d'oeuvre.

Le programme de simulation du chapitre 5 réalise en 'modèle réduit' de nombreuses fonctions du maître d'oeuvre d'un système d'exploitation. Si nous avons choisi de reprendre le modèle suivi par *DOS*, c'est pour mieux faire comprendre le fonctionnement des structures utilisées par ce système d'exploitation. Ainsi, la *TAS* des zones de l'espace des registres ressemble beaucoup à la *TAF* d'un support de masse, et il en est de même des répertoires.

Examinons à présent, ce qu'il en est des structures utilisées par *DOS*.

TAF et répertoires.

Secteur d'amorce et secteur de partition.

La *TAF* a une structure qui dépend du volume qu'elle est censée représenter. En outre, pour parer à des incidents d'origines diverses, elle est, en principe, constituée en deux exemplaires. Elle se situe, sur le support de masse, juste après le ***secteur d'amorce***, cas d'une disquette, d'un disque dur 'monolithique' ou de la partie d'un disque partagé portant le système. L'emplacement de la *TAF*, sa taille et le paramètre–clé de sa structure sont indiqués dans le secteur d'amorce.

Sur les disquettes, ce secteur est le premier secteur de la piste 0, face 0. Sur les disques durs, il se trouve sur le premier secteur, non pas du support, mais de la partition définie sur le support, même si le disque n'est pas partagé. Le lieu du premier secteur de la partition est donné dans la ***table de partition*** du disque qui se trouve sur le premier secteur du cylindre 0, tête 0. Il est donné à partir de l'octet **01BF** depuis le début de ce secteur. Ces trois octets donnent, par ordre d'adresses croissantes : la tête sur laquelle commence la partition, puis le cylindre et le secteur du cylindre où elle commence.

Le mot désignant le cylindre et le secteur s'analyse comme suit :
b_0-b_5 : numéro du secteur (de **01** à **011** en général),
b_8-b_{15},b_6-b_7 : numéro du cylindre.
Ainsi, 81 26 désigne le secteur 01 du cylindre **0226** (= 550).

La table de partition ne peut pas être chargée en *RAM* par les commandes de *DEBUG*. Elle ne peut être lue que par l'interruption **013**. A gauche, ci-dessous, lecture par cette interruption et, à droite, écriture :

```
0100 MOV  AX,0201        0100 MOV  AX,0301
0103 MOV  BX,1000        0103 MOV  BX,1000
0106 MOV  CX,0001        0106 MOV  CX,0001
0109 MOV  DX,0000        0109 MOV  DX,0000
010C INT  13             010C INT  13
```

AL indique le nombre de secteurs à lire, *CL* le numéro de secteur (commence à **01**), *CH*, le numéro de piste, *DH* le numéro de face et *DL* le numéro de lecteur (A = **00**, B = **01**, ...). Pour les disques durs, les numéros sont **080** (C), **081**,... et la numérotation des secteurs et des cylindres par *CX* s'effectue selon le schéma donné ci-dessus ; il faut réinitialiser le lecteur (fonction **00** de l'interruption sans autres paramètres) avant la première *écriture* ainsi qu'après toute opération conclue par *CY*.

N'utiliser qu'avec la plus extrême prudence la fonction d'écriture, surtout sur un disque dur. Elle peut cependant être utile pour restaurer des données perdues à l'occasion d'une panne de matériel ou suite à l'action pernicieuse d'un virus.

La TAF.

Dans le secteur d'amorce, le nombre de *TAF* est **by [010]** (adresse depuis le début du secteur). Le nombre de secteurs physiques par *TAF* est dans **wo [016]** et le nombre d'octets d'un secteur physique dans **wo [0B]**.

En outre, chaque entrée de *TAF* est de 12 bits ou 16 bits selon la taille du support. S'il y a plus de 4096 secteurs *DOS*, on aura des *TAF* 16 bits et sinon, des *TAF* 12 bits. Il faut donc diviser le nombre de secteurs physiques (**wo [013]**) par le nombre de secteurs physiques par secteur *DOS* (**by [0D]**) pour le savoir.

La structure de la *TAF* est alors très analogue aux *TAS* des chapitres 5 et 6. L'entrée associée à un secteur *DOS* donne le numéro de l'entrée associé au secteur suivant. Le numéro des entrées correspond à celui des secteurs *DOS* module une translation : les deux premières entrées de la *TAF* ne sont pas utilisées pour représenter les secteurs, car elles représentent la partie du volume occupée par l'amorce, les *TAF* et le répertoire principal. La première entrée porte donc le numéro 2 et correspond, sur une disquette 5"1/4 de 360 Ko au secteur *DOS* N°12.

La lecture d'une *TAF* 16 bits est immédiate sous *DEBUG* avec la commande *l* de syntaxe *l adresse lecteur secteur quantum*. Ainsi pour lire une *TAF* de disque dur de plus de 10 Mo nommé C, :

```
-l 1000 02 01 05
```

Dans le cas d'une *TAF* 12 bits, l'affichage de *DEBUG* ne permet pas de lire ce que contient la *TAF*. Illustrons le sur les premiers octets d'une *TAF* de disquette 5"1/4 de 360 Ko ordinaire :

```
FD FF FF FF FF FF 05 60 00 07 80 00 09 50 01 0B C0 00 0D E0 00 0F 00 01
11 20 01 13 40 01 FF 6F 01 17 80 01 19 A0 01 1B C0 01 1D F0 FF 1F F0 FF
```

Si on applique strictement la convention d'écriture des octets mémoires au sein d'un même octet, on doit inverser les quartets et écrire :

```
DF FF FF FF FF FF 50 06 00 70 08 00 90 05 10 B0 0C 00 D0 0E 00 F0 00 10
11 02 10 31 04 10 FF F6 10 71 08 10 91 0A 10 B1 0C 10 D1 0F FF F1 0F FF
```

Puis on doit grouper les quartets par trois :

```
DFF FFF FFF FFF 500 600 700 800 900 510 B00 C00 D00 E00 F00 010
110 210 310 410 FFF 610 710 810 910 A10 B10 C10 D10 FFF F10 FFF
```

Il ne reste plus qu'à inverser cette écriture pour trouver une présentation tout à fait conforme à nos habitudes :

```
FFD FFF FFF FFF 005 006 007 008 009 015 00B 00C 00D 00E 00F 010
011 012 013 014 FFF 016 017 018 019 01A 01B 01C 01D FFF 01F FFF
```

On observe que le fichier débutant au secteur **04** occupe deux zones contigues distinctes : secteurs **04** à **09** inclus, puis **015** à **01D** inclus.

Dans la *TAF* 12 bits, **FFF** joue le rôle de **0100** dans la *TAS* des chapitres 5 et 6 : cette valeur désigne le dernier secteur *DOS* du fichier.

A partir de cet exemple et des informations ci-dessus, le lecteur intéressé peut écrire un programme permettant d'afficher de façon 'limpide' une *TAF* 12 bits.

En ce qui concerne les répertoires, les informations données au chapitre 3 sont suffisantes, aussi renvoyons-nous le lecteur au paragraphe 3.3.3..

Simulation de la fonction EXEC (1).

Nous avons vu, au chapitre 4, le fonctionnement de cette fonction qu'il s'agit de simuler sans, naturellement, l'utiliser. Nous donnons le texte de ce programme en annexe 6.

Rappelons que le problème se décompose en trois phases : chargement en mémoire, exécution et terminaison.

Nous avons déjà examiné le problème du chargement d'un fichier en mémoire à l'aide des fonctions de *DOS*. Nous examinons donc ici les deux autres phases, principalement la troisième. Il y a essentiellement deux moyens de terminer un programme *.COM* (*cf.* chapitre 4) : l'appel à la fonction *DOS* **04C** ou l'appel à l'interruption **020**. Le but visé étant l'illustration de la méthode, nous prendrons l'hypothèse de la fonction **04C**. La modification du programme pour traiter aussi l'interruption **020** apparaîtra alors immédiatement.

REMARQUE : Il va de soi que les informations dont nous disposons sur les répertoires, la *TAF* et l'interruption **013**, nous permettraient d'écrire des fonctions de lecture (au demeurant nettement plus rapides), mais notre objectif est la simulation de l'exécution.

Principe.

Pour 'récupérer la main' à l'issue de l'exécution du programme, nous utilisons la technique de la redéfinition d'une interruption. Afin d'en comprendre le mécanisme, il faut revenir au fonctionnement de l'instruction **int**. Nous avons vu que son action se décompose en trois **push** : indicateurs, *CS*, *IP* (dans cet ordre pour l'empilement), puis un saut à une adresse absolue contenue dans la table des interruptions. On peut en fait reproduire ce mécanisme avec d'autres instructions du processeur :

Supposons qu'on veuille simuler l'interruption **013**. On peut écrire :

```
0100 INT  13                0100 XOR  AX,AX
                            0102 MOV  DS,AX
sauvegarde de (F) ->        0104 PUSHF
          IF = 0 ->         0105 CLI
sauvegarde de CS:IP et appel -> 0106 CALL FAR [004C]
```

En effet, l'adresse du gestionnaire de l'interruption se trouve à l'adresse **0000:004C** puisque **04*013 = 04C**. En outre, le gestionnaire se termine par un **iret** que l'on peut également, du point de vue du résultat, simuler par les instructions :

```
0100 IRET                   0100 RETF
                            0101 POPF.
```

Les étapes de la redéfinition s'en déduit aussitôt :

on sauvegarde dans le programme l'adresse du gestionnaire courant de l'interruption ;

on définit, dans le programme, un utilitaire constituant le nouveau gestionnaire ;

dans la table des interruptions, on remplace l'adresse du gestionnaire courant par l'adresse absolue de l'utilitaire ;

quand le programme se termine, il replace, dans la table des interruptions l'adresse du gestionnaire courant.

Dans notre cas, l'utilitaire aura pour tâche de 'filtrer' les appels à une fonction. Si *AH* ≠ **04C**, il appelle le gestionnaire habituel de l'interruption **021**. Si *AH* = **04C**, il reprend la main et redonne à l'interruption **021** son adresse habituelle.

Programmation.

Tout d'abord, à gauche ci-après : la sauvegarde de l'ancien gestionnaire et l'installation du nouveau ; à droite : le rétablissement de l'ancien gestionnaire.

```
0278 31C0          XOR  AX,AX
027A 8ED8          MOV  DS,AX
027C C41E8400      LES  BX,[0084]
0280 2E            CS:
0281 891ED203      MOV  [03D2],BX
0285 2E            CS:
0286 8C06D403      MOV  [03D4],ES
028A C7068400A802  MOV  WO [0084],02A8
0290 8C0E8600      MOV  [0086],CS
```

```
02B8 31C0        XOR  AX,AX
02BA 8ED8        MOV  DS,AX
02BC 2E          CS:
02BD C43ED203    LES  DI,[03D2]
02C1 893E8400    MOV  [0084],DI
02C5 8C068600    MOV  [0086],ES
```

Puis le programme du nouvel utilitaire, avec, à gauche, le cas $AH \neq$ **04C**, et à gauche, le cas $AH =$ **04C** :

```
02A8 9C          PUSHF
02A9 80FC4C      CMP  AH,4C
02AC 7540        JNZ  02EE

02EE 9D          POPF
02EF 2E          CS:
02F0 FF2ED203    JMP  FAR [03D2]


les instructions 02C9-02DE
ont pour but de rétablir
la pile, SS:SP ayant été
sauvés avant le transfert
de l'exécution
```

```
02AE 2E          CS:
02AF 8F06A602    POP  [02A6]
02B3 2E          CS:
02B4 893E0003    MOV  [0300],DI
  instructions 02B8-02C9 ci-dessus
02C9 8CC8        MOV  AX,CS
02CB 8ED8        MOV  DS,AX
02CD 8B3E0003    MOV  DI,[0300]
02D1 A1E403      MOV  AX,[03E4]
02D4 FA          CLI
02D5 8ED0        MOV  SS,AX
02D7 FB          STI
02D8 2E          CS:
02D9 A1E203      MOV  AX,[03E2]
02DC 89C4        MOV  SP,AX
02DE E9BEFE      JMP  019F
```

Par ailleurs, il faut veiller à la construction du préfixe et à l'initialisation des registres du processeur, comme pour un *.COM*. Pour la construction du préfixe, on se contentera, dans un premier temps, comme dans le programme de l'annexe 6, de recopier les informations utiles à partir du préfixe du programme de simulation lui-même : voir en annexe les instructions **022F-0245**. On pourra aisément y ajouter un traitement plus complet, comprenant, notamment, celui de la ligne de commande.

Initialisation des registres et transmission de l'exécution par un **jmp far** :

```
0310 2F            CS:                 ; l'adresse de segment de la zone
0312 A1DA03        MOV  AX,[03DA]      ; allouée au programme à exécuter
031E A3E003        MOV  [03E0],AX      ; est placée où il faut pour
0324 C706DE030001  MOV  WO [03DE],0100 ; le JMP FAR qui 'passe la main'
0327 8926E203      MOV  [03E2],SP      ; on note la pile pour le retour
032D 8C16E403      MOV  [03E4],SS      ;
0330 8EC0          MOV  ES,AX          ; ES et DS initialisés sur
0335 8ED8          MOV  DS,AX          ; le futur CS
0333 C706FEFF0000  MOV  WO [FFFE],00   ; 0000 sur la pile
0333 FA            CLI                 ;
0338 BCFEFF        MOV  SP,FFFE        ; SS:SP
0347 8ED0          MOV  SS,AX          ; initialisé
0349 FB            STI                 ; tout est donc prêt
034B 2E            CS:                 ;
034C FF2EDE03      JMP  FAR [03DE]     ; transmission de l'exécution
```

Le **jmp far** renvoie au gestionnaire originel qui contient un **iret**. Lorsque le processeur le rencontrera, la pile contiendra l'adresse de retour installée par l'instruction **int** du programme exécuté. Le retour se fera donc, comme pour l'interruption 'normale' avec la même valeur des registres, y compris celui des indicateurs.

CHAPITRE 8

MACRO-ASSEMBLAGE

Avec *DEBUG*, on peut, si on le veut, tout programmer et contrôler complètement[1] l'ordinateur. Cependant, revers de cette puissance, l'*écriture* des programmes sous *DEBUG*, notamment des *gros* programmes, n'est pas aisée. En particulier, la modification des programmes est alors chose délicate et complexe. Toutefois, contrepoids non négligeable à ce revers, le déboguage sous *DEBUG* de programmes écrits selon les indications données jusqu'à présent est *très* facile à mettre en oeuvre, d'où une efficacité plus grande, pour la raison simple qu'on *sait exactement* où sont les variables, les procédures, les points stratégiques du programme. Si un octet a été mal écrit quelque part, où si l'adresse d'un saut ou d'un appel manque son but d'un octet (et parfois, les choses les plus aberrantes ne tiennent qu'à un octet, effectivement), on finit toujours par localiser l'erreur. Il faut pour cela noter très attentivement, et le plus objectivement possible, le comportement du programme et le contenu des registres. Plus l'analyse présidant à la conception et la rédaction du programme aura été structurée et rigoureuse, moins ce type de problème se trouvera et s'il s'en trouve malgré tout, ce qui est dans la nature des choses humaines[2], plus vite on le débusquera parce qu'on aura dégagé les paramètres pertinents dans le programme testé.

Afin de supprimer le revers de l'écriture, les concepteurs de *constructeurs* pour micro-ordinateurs du type PC compatible ont peu à peu mis au point les ***macro-assembleurs***. L'idée en est fort simple : tout ce qui a été fait dans les chapitres précédents en matière d'écriture du code : *assemblage* de blocs d'instructions, macros et sous-programmes, et surtout, calcul des adresses, tout ceci peut être *automatisé*.

Comme les *constructeurs*, les macros-assembleurs développent un langage que, suivant l'usage, nous appellerons encore *macro-assembleur*. Pour faciliter la conception des programmes, les rendre plus lisibles et offrir les commodités ainsi signalées, ces langages introduisent de façon massive des ***directives*** qui sont des ordres donnés au macro-assembleur et remplacent les adresses *hexadécimales* par des adresses *symboliques* : au lieu de se référer à une adresse par sa valeur hexadécimale, on lui donne un *nom* (de longueur arbitraire) et on écrit ce nom chaque fois qu'il s'agit de donner l'adresse qui lui correspond. Ces facilités, et notamment cette dernière, font ressembler les macro-assembleurs à des langages de programmation de haut niveau. Cependant, comme les macro-assembleurs ont pour but de produire du code machine, leur discours reste encore, fondamentalement, un discours sur la mémoire et les registres du processeur et, sous cet angle, ils ne se distinguent pas des langages d'assemblage.

---

[1] Exception faite du mode protégé des 80x86 pour $x \geq 2$.

[2] On peut dire, sans crainte d'être démenti par les meilleurs programmeurs, qu'il y a toujours au moins une bogue résiduelle qui nécessite le recours aux techniques du déboguage.

Nous considérerons dans ce qui suit le macro-assembleur le plus répandu dans le monde *PC* fonctionnant sous *DOS* : *MASM* de Microsoft. Nous étudierons ensuite le mécanisme de fonctionnement des programmes *.EXE* puis nous indiquerons comment marier les programmes *MASM* et ceux que nous avons construits sous *DEBUG*.

8.1. MASM : règles du jeu.

Il n'est pas question, en quelques pages, de faire le tour de *MASM* où l'aspect histoire naturelle prend une importance sans commune mesure avec ce que nous avons pu voir dans le codage des instructions. De ce fait, nous nous contenterons de décrire ici un *dialecte* de *MASM* tout à fait opérationnel.

Nous commencerons cette présentation par un exemple. Puis nous donnerons les règles de base du langage pour écrire des programmes *.COM*.

8.1.1. Un exemple.

Généralités.

Quand *MASM* transforme le texte d'un programme écrit en assembleur en code machine, le travail qu'il a à faire s'exécute en deux temps.

Tout d'abord, il traduit les instructions machines écrites en code symbolique et il repère l'agencement de ces instructions en fonction des directives rencontrées. Cette première opération ressemble beaucoup à 'la première passe' du simulateur construit au chapitre 5. Là aussi, cette traduction est effectuée à partir d'une première lecture du programme source sur la base d'un compteur de code, ce qui permet à *MASM* d'assembler les instructions une à une. Compte tenu de la structure offerte par le logiciel, cet assemblage n'est pas, en général, fait de façon continue, mais par morceaux. Une deuxième lecture permet à *MASM* de positionner les morceaux les uns par rapport aux autres mais aussi, ce qui est une des raisons d'être de cet outil, par rapport à d'éventuelles sources extérieures, c'est-à-dire d'autres programmes sources. Le résultat de cette analyse est placé dans un fichier *.OBJ*.

Un second logiciel entre alors en action : *LINK*. Comme son nom l'indique, il relie entre eux les différents *.OBJ* concourant à un même programme, calcule les adresses et constitue un unique *.EXE* que l'on pourra appeler directement par *COMMAND.COM*. Dans certaines conditions que nous verrons alors, on peut convertir ce fichier *.EXE* en un fichier *.COM* par un troisième programme : *EXE2BIN*. Ainsi, à partir d'un fichier source **prog.asm**, contenant le texte en assembleur d'un programme destiné à devenir un *.COM*, on entrera successivement à l'écran :

```
C:\>masm prog ;
C:\>link prog ;
C:\>exe2bin prog prog.com
```

Telle est la méthode suivie aujourd'hui par la plupart des programmeurs en assembleur.

L'exemple des tours de Hanoï.

Nous reprenons le jeu des tours de Hanoï pour écrire un programme simple, sans illustration graphique.

Voici, tout d'abord, le texte du programme principal :

```
Début :
      MOV  AX,CS          ; les registres de segment
      MOV  DS,AX          ; sont initialisés
      MOV  ES,AX          ; à partir de CS, ce que,
      CLI                 ; en principe, COMMAND.COM fait.
      MOV  SS,AX          ;
      STI
      CALL CombienDeDisques?
      MOV  AL,CL
      MOV  AH,01
      MOV  CL,02
      MOV  CH,03
      CALL Hanoï          ; appel de la procédure récursive
Terminaison :
      MOV  AX,04C00h
      INT  021h
```

Il comporte deux adresses symboliques : **Début** et **Terminaison** dites en position d'**étiquette** marquée par le symbole : après le nom. La fonction de ces étiquettes est, pour l'instant, de rendre le texte du programme plus clair. Parmi des instructions que nous connaissons bien, deux **call** dont les adresses symboliques nous paraissent suffisamment évocatrices pour que le seul commentaire utile soit l'indication du caractère récursif de la procédure **Hanoï**. Voici, justement, le texte de cette dernière :

```
Hanoï PROC NEAR
      CMP  AL,00          ; N = 0?
      JA   Suite          ; non : -> suite
      RET                 ; oui : c'est fini pour cet appel
Suite :
      PUSH AX             ;
      PUSH CX             ; on sauvegarde N, a, b et c
      CMP  SP,WORD PTR Butoir ; peut-on empiler?
      JBE  Débordement    ; non : la pile déborde
      XCHG CH,CL          ; oui : donc 1) on permute b et c
      DEC  AL             ; 2) N := N-1
      CALL Hanoï          ; 1er appel récursif
      POP  CX             ;
      POP  AX             ; on restaure N, a, b et c
      CALL Ecriture       ;
      PUSH AX             ; inutile ici de vérifier : la pile
      PUSH CX             ; est comme avant l'appel précédent
      XCHG AH,CL          ; on permute a et b
      DEC  AL             ; N := N-1
      CALL Hanoï          ; 2ème appel récursif
      POP  CX             ;
      POP  AX             ;       % dans ce programme et ceux qui suivront,
      RET                 ;       % nous écrirons tout ce qui est instruc-
Débordement :             %       % tion du processeur en MAJUSCULES et ce
      Afficher MessagePile %       % qui est défini par le programmeur en
      JMP  Terminaison    %       % minuscules.
Hanoï ENDP
```

Ce programme nous montre le rôle des étiquettes : ce sont des noms que l'on donne à une ***adresse dans le code***. Comme ce nom est symbolique, il n'est pas nécessaire, *a priori* de se soucier de sa valeur : le compteur de code de l'analyseur de syntaxe mis en oeuvre par *MASM* la calcule pour nous. Dans les instructions de branchement, l'adresse de référence ***doit*** toujours être indiquée sous forme symbolique.

Ce programme comporte aussi un exemple de ***directives*** : en l'occurrence PROC NEAR et ENDP. Elles indiquent que **Hanoï** est le nom d'un programme, on dit ici ***procédure***, dont le premier octet est celui de l'instruction **cmp al,00**. La directive **Hanoï ENDP** signale la fin du texte du programme. Le mot NEAR est un ***qualifiant*** : il précise que le programme sera appelé par un **call** intra-segment et que tout **ret** rencontré dans le texte du programme (d'où la nécessité de ENDP) sera codé **0C3** (ou **0C2**).

Dans les directives PROC et ENDP, ***il faut*** que le nom donné au programme figure sur la *même ligne* que la directive et avant elle. C'est par ce nom que le programme est appelé. Comme dans le cas d'un branchement, *MASM* n'accepte pas d'adresse immédiate. A noter, pour toutes les références, qu'on peut les appeler avant de les avoir définies. Dans le cas d'une étiquette, la définition est marquée par le symbole :, dans le cas d'un programme, par les deux directives PROC/ENDP.

Observons l'instruction **cmp sp,word ptr Butoir**. La présence du qualifiant **word ptr** peut paraître superflue puisque *SP* est un registre 16 bits. Cependant, si on ne précise pas **word ptr**, l'instruction sera correctement codée mais *MASM* donnera un diagnostic d'erreur *légère* qui n'interdit pas l'assemblage ultérieur. Précisons, au passage, que *MASM* tient pour solécismes et barbarismes grossiers les abréviations **wo** et **by** de *DEBUG*. En conséquence, son analyseur de syntaxe les rejette inexorablement[1].

Macros-instructions.

Le texte de la procédure **Hanoï** présente une nouveauté : la ligne **Afficher MessagePile**. Il s'agit là d'une référence à une ***macro-instruction***. On appelle ainsi une *directive* dont le nom se réfère à une séquence d'instruction dûment signalée dans le texte du programme. Quand l'analyseur de *MASM* rencontre cette directive, il assemble à partir de cet endroit du code la séquence d'instructions référencée par la directive et qu'on appelle le ***corps*** de la macro-instruction.

Exemple de macro-instruction (nous dirons par la suite : ***macro***) :

```
Afficher MACRO Chaine
      PUSH AX
      PUSH DX
      MOV  AH,09h
      MOV  DX,offset Chaine ; DX : adresse de Chaine
      INT  21h
      POP  DX
      POP  AX
ENDM
```

Le *nom* de la *macro*, *sur la même ligne* que le mot-clé MACRO est éventuellement suivi, toujours sur la même ligne, d'un ou plusieurs ***paramètres***, ici : le nom **Chaine**. Le texte de la macro se termine par ENDM et *il ne faut pas* répéter le nom de la macro avant ENDM sous-peine d'erreur[2]. Ceci constitue la ***déclaration*** de la macro **Afficher**.

Dans le texte de cette macro, **Chaine** joue le rôle d'adresse symbolique du début d'une zone mémoire. Pour utiliser la macro-instruction, on s'y ***réfère*** en indiquant, comme dans le texte du programme **Hanoï**, le nom de la macro puis, en ***argument*** sur la *même ligne*, le *nom* par lequel il faut remplacer **Chaine**. A l'assemblage, *MASM reproduit* à cet endroit le code du corps de la macro en y remplaçant partout les paramètres de la déclaration par les objets correspond nt aux *noms* qui sont donnés en arguments de la ***référence***. Dans l'exemple, on obtient, à ce même endroit :

```
PUSH AX
PUSH DX
MOV  AH,09h
MOV  DX,offset MessagePile
INT  21h
POP  DX
POP  AX
```

---

[1] Signalons que *DEBUG* et *MASM* sont des logiciels fabriqués par la même société.

[2] On peut considérer ceci comme le moyen de marquer qu'une macro-instruction n'est pas un programme.

exactement comme si au lieu de **Afficher MessagePile**, on avait écrit cette même séquence d'instructions.

On remarque, dans le corps de la macro, l'utilisation d'une ***fonction*** de *MASM*, la fonction *offset*. Cette fonction fournit le *déplacement* de la référence à laquelle elle s'applique depuis le début de la partie concernée du code : ici, depuis le début du code. Ceci correspond aux corrections que nous avons indiquées sous *DEBUG* par le biais de la commande *e* lorsque des séquences d'instructions étaient répétées (voir exemples p. 84 et 146 ainsi qu'en annexe 2, pp.220-221). Même si l'on ne doit pas répéter plusieurs fois la séquence, la solution de *MASM* permet d'écrire des programmes plus lisibles. Contre partie non négligeable : la longueur d'une macro est souvent perdue de vue par le programmeur. Ainsi, si une référence de macro prend place entre une instruction **jnc Suite** et l'étiquette **Suite :**, *MASM* signalera un branchement invalide si le corps de la macro se développe sur plus de 128 octets...

RAPPEL : Par rapport au *code*, la ***référence*** d'une macro est une directive donnée au *constructeur*. Elle n'est pas une instruction du processeur. C'est pourquoi nous réservons le mot *appel* à l'*instruction* **call**.

Macros et programmes : autres exemples.

La procédure **CombienDeDisques?** fournit un exemple de macro plus complexe.

Voici, sur le même listage, ce programme et la macro **Numériser** :

```
CombienDeDisques? PROC NEAR
      Afficher MessageEntrée
      MOV  DX,offset EntréeBrute
      MOV  AH,0Ah
      INT  21h
      Afficher Saut
      Numériser EntréeBrute
      RET
CombienDeDisques? ENDP
```

```
Numériser MACRO Tampon
LOCAL Boucle   ; conversion en base 16
;    de N, résultat placé dans CX :
      XOR  CX,CX
      MOV  CL,BYTE PTR Tampon+1
      XOR  AX,AX
      MOV  BL,0Ah
      MOV  BP,offset Tampon
      ADD  BP,02
      XOR  DX,dx
Boucle :
      MUL  BL
      MOV  DL,[BP]
      SUB  DL,030h
      ADD  AX,DX
      INC  BP
      LOOP Boucle    ; AX := N en hexa
      MOV  CX,AX     ; CX := AX
ENDM
```

Dans le programme, on observe que la macro **Afficher** figure deux fois avec des arguments différents. La seconde fois, le *nom* **Saut** va remplacer **Chaine** dans l'instruction **mov dx,offset Chaine**. La seconde macro de ce programme, **Numériser**, contient, dans sa déclaration une directive LOCAL dont le but est d'indiquer que la liste de *noms* figurant après ce mot-clé, et *sur la même ligne*, n'a de sens que *dans le corps* de la macro. Ceci veut dire que le nom **Boucle** pourrait être utilisé par une autre macro : le *constructeur* associerait alors une autre adresse aux occurrences de **Boucle** dans les appels de l'autre macro. Cela veut également dire que ce nom ne concerne que la portion de code reproduite. En particulier, si on appelle deux fois (ou plus) la macro dans le texte, à chaque fois, l'adresse réelle associée à **Boucle** sera différente. Ce qui correspond tout à fait à l'emploi de la commande *e* sous *DEBUG* pour corriger une portion de code réutilisée dans un fichier *.CSM*.

Nous donnons, page 170, le texte de la procédure **Ecriture**, puis le mode de composition de toutes les parties du programme que nous aurons vues en un ***texte***

source qui puisse être soumis au *constructeur* dè *MASM* avec des chances sérieuses d'être accepté.

Sur le listage ci-après, nous donnons, côte à côte pour des raisons de place, la procédure **Ecriture** et les déclarations des variables utilisées dans le programme :

```
Ecriture PROC NEAR
      push ax
      mov  dl,ah
      add  dl,030h
      mov  ah,02
      int  21h
      Afficher Flèche
      mov  dl,ch
      add  dl,030h
      mov  ah,02
      int  21h
      Afficher Saut
      pop  ax
      ret
Ecriture ENDP
```

```
Butoir : DW  0600h  ; butoir de la
                    ;  pile
EntréeBrute : DB  05,05 DUP (?)
N :       DW  0000   ; N
Flèche : DB  ' -> ','$'
Saut :   DB  0Dh,0Ah,'$'
MessageEntrée : DB  'Entrer le'
          DB  'nombre de disques'
          DB  0Dh,0Ah,'$'
MessagePile :   DB  'Ca débord'
          DB  'e!...',0Dh,0Ah,'$'
```

Variables et constantes.

Elles peuvent se placer n'importe où dans le texte du programme, comme nous l'aurions fait sous *DEBUG*, avec la même contrainte : si la suite des instructions est interrompue par les octets réservés aux variables ou aux constantes, il faut donc prévoir les sauts nécessaires ou utiliser ceux du programme pour placer là ces zones de mémoire réservée. Dans notre exemple, nous les avons regroupées au même endroit, comme indiqué ci-dessus.

Les directives DB et DW de *DEBUG* sont reprises (DW annonce que les deux octets suivants constituent un mot écrit sous forme symbolique : **1234** et non **34 12**). D'autres, propres à *MASM* apparaissent : DUP permet de répéter une même zone (ici un octet) autant de fois que l'indique le nombre placé devant ce mot-clé ; le symbole **?** signifie au *constructeur* que la zone concernée n'est pas à initialiser.

Les variables initialisées utilisent la même notation des nombres que les instructions. Cette notation *diffère* sensiblement de celle de *DEBUG*. Les nombres écrits en données immédiates doivent commencer par un zéro si leur chiffre de tête est une lettre. S'ils sont écrits en hexadécimal, ils doivent être suivis de la lettre **h** et s'ils sont écrits en base dix, ils doivent être suivis de la lettre **d**. Si un nombre est écrit sans indication de base, il est considéré comme écrit en décimal. Ainsi, **0A** seul est considéré comme une faute de syntaxe : c'est un nombre décimal (il ne se termine pas par **d**) et **A** n'est pas un chiffre décimal.

MASM permet de déclarer la base numérique. On utilise à cet effet la directive .RADIX suivie de l'indication de la base (qui doit être écrite en base **dix**) : tous les nombres qui, dans le texte, se situent entre une directive .RADIX et la suivante seront lus comme dans la base indiquée, sauf s'ils sont suivis d'une indication explicite. Ainsi **0D** est-il compris comme **00**, même avec la directive .RADIX 16. Pour comprendre treize il faut écrire 0Dh ou 13d. Pour notre part, nous conseillons d'écrire toujours en hexadécimal *explicitement*. A noter que si aucune directive .RADIX n'apparaît dans le texte, la base par défaut est **dix**. Attention par conséquent aux **interruptions**. L'instruction **int 21** de *DEBUG* doit ici s'écrire **int 21h**. Si on l'oublie, tout peut arriver, car l'interruption **015** (**015** = 21) n'a rien à voir avec les fonctions *DOS*.

Modèle de programme .COM.

On regroupe toutes les parties du programme **Hanoï** en un texte unique suivant l'exemple ci-après :

```
;
; fichier hrec.asm
;
; programmation récursive des tours de Hanoï sous MASM
;
; === MACROS ================================================
  les textes des macros Afficher et Numériser
; === PROGRAMME =============================================
Code  SEGMENT PARA
ORG   100h
ASSUME  cs : Code, ds : Code, es : Code, ss : Code
  le texte de programme principal de la page 167
; === DONNEES ET VARIABLES ==================================
  les variables Butoir, N, la zone d'écriture pour DOS EntréeBrute et les
messages : Flèche, Saut, MessageEntrée et MessagePile.
; === PROCEDURES ============================================
  les procédures Hanoï, Ecriture et CombienDeDisques?
; === FIN ===================================================
Code  ENDS
      END Début
```

Les directives SEGMENT et ENDS délimitent dans un ***segment*** ce que *MASM* doit assembler. Il est d'usage de ne pas y inclure les macros qui, de toute façon, peuvent être placées n'importe où dans le *texte*[1]. Le nom associé au *segment* est fixé par le programmeur. La directive finale END signale la fin du texte : *MASM* ne lit pas ce qui pourrait être écrit après cette ligne. Le paramètre placé après END est le ***point d'entrée*** du programme. Dans un fichier *.COM*, c'est obligatoirement l'adresse **0100** depuis le début du préfixe. C'est ce qui est signifié par la directive ORG 100h et c'est pourquoi l'étiquette **Début** doit être placée après, mais avant la première instruction du programme. Enfin, la directive ASSUME indique à *MASM* quelles sont les valeurs attribuées aux registres *CS*, *DS*, *ES* et *SS* au début du *segment*. *MASM* ne crée aucune instruction pour affecter ces valeurs aux registres. Les indication de l'ASSUME servent à définir les éléments du calcul, par *LINK*, des adresses dans le fichier *.EXE* final.

Le qualifiant PARA demande que le premier octet du code ait une adresse relative multiple de **010**.

Le modèle donné ci-dessus, que nous appellerons *modèle .COM* peut servir de squelette à tous les programmes destinés à être ultérieurement convertis en *.COM* pour être utilisés comme tels.

Listages d'assemblage de *MASM*.

Terminons cet exemple par un *listage* établit par *MASM*. Nous l'avons appliqué à la macro **Numériser** que nous avons insérée dans un *.COM* spécialement conçu à cet effet, et qui se trouve reproduit partiellement page 173.

Pour ces 14 instructions, *MASM* ne produit pas moins de trois pages de compte-rendu. Il reproduit le texte du programme avant de l'assembler, puis il donne une analyse des variables utilisées. Aussi ne donnons-nous que les deux dernières pages de ce listage dans une présentation quelque peu concentrée.

On y remarquera, ligne 58, que l'étiquette **Boucle** n'est pas assemblée puisqu'elle ne correspond pas à une instruction du processeur mais à une facilité offerte

[1] Evidemment pas au milieu du texte d'un message, par exemple...

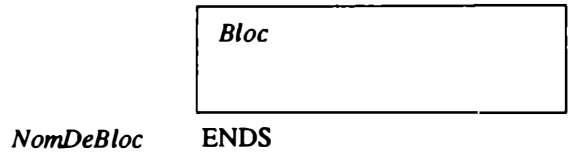
par *MASM*. Comme en témoigne la ligne 64, l'assèmblage de l'instruction **loop Boucle** a cependant été fait complètement puisqu'il s'agit d'un saut proche.

Le code hexadécimal construit par *MASM* est donné à gauche, en regard de chaque instruction. Noter, lignes 52 et 55, la présence de la lettre **R** à la suite d'un mot. On obtient le code réel en lisant **8A 0E 23 01** pour **8A 0E 0123 R**. Noter que l'adresse calculée par la fonction **offset** est bien le déplacement de la variable tampon depuis le début du préfixe que la fonction *EXEC* de *DOS* construit au moment d'exécuter un programme *.COM*. Enfin, dans la liste des symboles, c'est-à-dire des noms des variables, L NEAR signifie ***étiquette proche***.

8.1.2. Des directives et des segments.

Blocs.

La structure fondamentale de *MASM* est le ***bloc***, appelé ***segment*** dans le langage de ce *constructeur*. Il prend l'aspect représenté par la figure ci-dessous :



où l'encadré représente la séquence d'instructions constituant le bloc et que l'on appelle encore ***corps*** du segment. *NomDeBloc* est le nom que le programmeur donne au segment et la première ligne de la figure ci-dessus constitue la ***déclaration du segment***.

Le langage définit sa syntaxe en conformité avec les principes de structuration que nous avons dégagés au chapitre 4, de telle sorte que le programmeur puisse traduire la logique de son programme de la façon la plus apparente possible. Les structures que nous avons vues au paragraphe précédent, PROC et MACRO sont également des blocs sur le plan syntaxique et d'un point de vue fonctionnel. Par rapport aux segments, ils apparaissent comme des sous-blocs pouvant cependant contenir d'autres blocs de même nature qu'eux.

Pour le constructeur, la déclaration d'un segment définit également un *segment* au sens que prend ce terme dans la description de la mémoire vive. En particulier, *MASM* lui associe automatiquement un numéro de paragraphe. C'est dans ce contexte de segments superposés ou emboîtés que la directive ASSUME prend tout son sens. En effet, elle fixe, pour le constructeur, la valeur que prennent les registres de segment à l'intérieur du bloc où la directive apparaît, depuis le point où elle est déclarée jusqu'au point (dans le sens des adresses croissantes) où, éventuellement, une autre directive se présente. De façon précise, sauf mention expresse du contraire, mais que nous n'aborderons pas, le premier segment rencontré définit le premier numéro de paragraphe. Comme, naturellement, *MASM* ne sait pas à quel paragraphe de la mémoire la fonction *EXEC* chargera réellement le programme, ce numéro de paragraphe est conventionnellement fixé à 0000. Mais tous les autres sont automatiquement définis par leur déplacement depuis cette origine 'absolue'. *MASM* mesure ce déplacement au cours de l'analyse syntaxique et le programmeur peut avoir accès à cette valeur (exprimée en paragraphes) par la fonction ***seg***.

ATTENTION : La fonction **seg** ne crée aucune instruction pour le processeur. Elle n'est pas autre chose qu'un instrument de calcul pour permettre au

```
Microsoft (R) Macro Assembler Version 5.10              2/8/91 18:53:46
                                                        Page     1-2


                                Début :

                                Numériser Tampon
0100  33 C9             1      xor       cx,cx
0102  8A 0E 0123 R      1      mov       cl,byte ptr Tampon+1
0106  B8 0000           1      mov       ax,0000
0109  B3 0A             1      mov       bl,0Ah
010B  BD 0122 R          1      mov       bp,offset Tampon
010E  83 C5 02           1      add       bp,02
0111  33 D2             1      xor       dx,dx
0113                    1??0000 :
0113  F6 E3             1      mul       bl
0115  8A 56 00          1      mov       dl,[bp]
0118  80 EA 30           1      sub       dl,030h
011B  03 C2             1      add       ax,dx
011D  45                1      inc       bp
011E  E2 F3             1      loop      ??0000    ;  AX := N en hexa
0120  8B C8             1      mov       cx,ax           ;  CX := AX

0122  05                Tampon :  DB      '  05, 06 DUP (?)
      0006[
      ??
      ]


0129                    Code      ENDS

                                  END       Début
```

```
Microsoft (R) Macro Assembler Version 5.10              2/8/91 18:53:46
                                                        Symbols-1
Macros:
      N a m e                 Lines
Numériser  ...........         18
Segments and Groups:
      N a m e           Length  Align  Combine Class
Code ..............     0129    PARA   NONE
Symbols:
      N a m e           Type    Value  Attr
Début  ..............   L NEAR  0100   Code
Tampon ..............    L NEAR  0122   Code
??0000 ..............    L NEAR  0113   Code
@Cpu ................    TEXT  0101h
@FileName  ..........    TEXT  lireN
@Version ............    TEXT  510
     57 Source  Lines
     75 Total   Lines
     10 Symbols
  47352 + 400228 Bytes symbol space free
      0 Warning Errors
      0 Severe  Errors
```

programmeur de manipuler les adresses des objets de son code sans avoir à en connaître les valeurs réelles. Nous revenons sur ce point p.175.

Directives.

Comme nous l'avons dit, ce sont des ordres donnés au constructeur et, en général, ***les directives ne créent pas de code.*** On veut dire par là qu'elles ne correspondent pas à des instructions du processeur. Cela dit, elles indiquent à *MASM* comment *assembler* le code symbolique qu'il lit.

EXEMPLE : Si une procédure est déclarée avec le qualifiant NEAR, on dira qu'elle est de type ***appel proche***, toute instruction **ret** apparaissant dans le bloc de la procédure est codée **0C3** (ou **0C2**), ce qui correspond au **ret** de *DEBUG*. Si le qualifiant est FAR, l'appel est dit de type ***distant***, la même instruction **ret** est codée **0CB** (ou **0CA**). Au demeurant, la mnémonique **retf** est tenue par *MASM* pour un solécisme inconvenant. Le même traitement est réservé à l'appel lui-même. Dans le cas d'une procédure proche, l'appel est codé par une instruction de codop **0E8**. Si la procédure est distante, l'appel est codé par une instruction de codop **09A**.

Du point de vue formel, ce qui n'est pas commentaire est soit instruction, soit directive. Les directives sont de deux sortes : ***étiquetées*** ou ***anonymes***. Les directives étiquetées sont constituées par l'étiquette, un nom à la convenance du programmeur, suivi *sur la même ligne* d'un mot-clé. Cette forme de directive est souvent appelée ***déclaration***. Lorsqu'elle est anonyme, une directive se rapporte souvent à une directive qui la précède sur la même ligne. On dit alors qu'elle est un ***qualifiant***. Nous avons vu le qualifiant PARA. Nous avons également vu la directive LOCAL, exemple de directive anonyme *déclarative* d'un type particulier : elle ne peut s'employer que dans la déclaration d'une macro-instruction.

Les qualifiants interviennent souvent pour préciser le type d'un objet : variable, procédure, plus généralement, structure définie dans le programme. On peut utiliser, dans ce but la directive LABEL pour fixer le type d'un objet. La syntaxe en est :

nom LABEL **type**

où **type** est un des qualifiants WORD, BYTE, NEAR, FAR. On peut ainsi déclarer une variable d'un certain type dans le code puis, dans une autre partie du code, modifier ce type avec la directive LABEL. Ces qualifiants de type interviennent également dans des instructions, ainsi que nous l'avons vu, mais avec la ***fonction*** **ptr** (voir ci-après).

Une directive importante est la directive EQU qui permet de définir des ***constantes*** dans le programme, comme le montre cet exemple :

```
        MOV  byte ptr [bx+VolumeMots],al
        ....
Lignes EQU 019h
Colonnes EQU 050h
VolumeMots EQU 2*Lignes*Colonnes
```

les constantes étant déclarées après le programme, dans le modèle donné page 171.

REMARQUE : Si l'on préfère placer les variables et/ou les procédures en tête du programme, il faut modifier le schéma proposé page 171 comme indiqué ci-dessous :

```
Début :
        JMP  Départ
(constantes,) variables et (éventuellement) procédures
Départ :
```

Fonctions.

Lors de l'analyse du code, le pointeur de code, dont nous avons vu le rôle au chapitre 5, permet à *MASM* de définir un certain nombre de fonctions qui affranchiront

le programmeur de tout calcul d'adresse par lui-même pour peu qu'il sache les utiliser à bon escient.

Rappelons le rôle des fonctions **offset** et **seg** déjà rencontrées. Quand le pointeur de code désigne l'octet qui suit le modrm (ou le codop) de l'instruction en cours d'assemblage, et qu'il lui correspond dans le programme source la référence d'une variable, par exemple, *Var*, le double mot **seg** *Var*:**offset** *Var* fournit la position de cet octet dans le code. La fonction **seg** mesure la distance, en paragraphes, entre la valeur de *DS* que *suppose* la directive ASSUME du bloc-segment où l'objet est déclaré et le segment de début du code choisi comme origine. La fonction **offset** indique l'adresse de l'objet dans le segment défini par la fonction **seg**.

REMARQUE : la syntaxe du préfixage n'est pas la même que sous *DEBUG* :

```
sous DEBUG :          sous MASM :
     ES:                   MOV  AX,ES:[DI]
     MOV  AX,[DI]
```

La fonction **ptr** affecte à l'objet désigné à sa droite le type indiqué par le qualifiant placé à sa gauche. Ainsi, pour affecter une donnée immédiate à un registre ou à une place mémoire, on doit, indiquer le type du but.

Ainsi, pour affecter l'octet 012 à VariableOctet, on écrira :

```
MOV  byte ptr VariableOctet,012h
```

cependant que cette variable aura été déclarée quelque part dans le programme par :

```
VariableOctet  DB  (?)
```

Une autre application importante de cette fonction concerne les appels de programmes. En effet, selon qu'une procédure a été déclarée on non, dans le *texte* du programme, avant l'instruction qui l'appelle, l'écriture de cette instruction va changer. L'exemple ci-après indique ce qu'il convient de faire :

```
Code1 SEGMENT
      ORG  00h
      ASSUME  cs : Code1, ds : Code1
      ASSUME  es : Code1, ss : Code1
DonnéeMot EQU 08ACh
Utilitaire PROC FAR
      mov  word ptr [bx],DonnéeMot
      ret
Utilitaire ENDP
Début :
      call Utilitaire
      mov  ax,4C00h
      int  021h
Code1 ENDS
      END  Début
```

```
Code1 SEGMENT
      ORG  100h
      ASSUME  cs : Code1
      ASSUME  ds : Code1
      ASSUME  es : Code1
      ASSUME  ss : Code1
Début :
      call far ptr Utilitaire
      mov  ax,4C00h
      int  021h
DonnéeMot EQU 08ACh
Utilitaire PROC FAR
      mov  word ptr [bx],DonnéeMot
      ret
Utilitaire ENDP
Code1 ENDS
      END  Début
```

Les grands services rendus par ce logiciel font regretter plus vivement l'obligation faite au programmeur d'assister le constructeur dans son travail d'assemblage.

Expressions.

Autre facilité apportée par *MASM*, la possibilité d'utiliser des expressions pour éviter au programmeur les calculs d'adresses ou de paramètres. Nous en avons déjà vu un aperçu page précédente. En voici un exemple analogue :

```
MOV  byte ptr [bx+2*Lignes*Colonnes+1],al
     ....
Lignes EQU 019h
Colonnes EQU 050h
```

8.2. Les fichiers .EXE.

La structure par blocs du langage défini par *MASM* a été constituée en même temps que le nouveau type de fichier exécutable introduit alors par les concepteurs de DOS : les fichiers *.EXE*.

Ces fichiers diffèrent fondamentalement des fichiers *.COM* par le fait que leur longueur n'a pas d'autre limite que les possibilités de chargement de la RAM, alors que les fichiers *.COM* doivent avoir une taille inférieure à 64 **Ko**. Cette particularité fait que si la taille d'un code dépasse 64 **Ko**, ce code contient presque sûrement des **jmp** partant d'un segment pour se rendre dans un autre segment, ce qui est appelé ***jmp inter***–segment ou ***long***, par opposition aux ***jmp intra***-segments ou ***courts***. De ce fait, il n'y a plus de raison d'être à imposer l'adresse de la première instruction, et le code peut être organisé sur *plusieurs* segments et non un seul, comme dans le cas des *.COM*.

En contrepartie, la structuration des fichiers *.EXE* est soumise à des règles assez complexes. Nous allons tout d'abord examiner un exemple, puis donner la raison de ces dispositions ce qui nous permettra d'étendre notre simulateur de la fonction *EXEC* au cas des fichiers *.EXE*.

8.2.1. Exemple de démonstration.

Il s'agit du programme ci-contre, appelé **demo.exe** qui a pour seul but de vérifier l'exécution de l'appel à un programme distant sous *MASM*.

Le programme contient quatre segments dont les distances en paragraphes, à l'adresse la plus basse du code prise comme origine sont données par le tableau suivant :

Segment	adresse	taille
Pile	0000	0200
Données	0020	00DF
Code2	002E	0015
Code1	0030	004C

Examinons les instructions ci-après et la façon dont *MASM* les assemble :

```
BA -- --  MOV  DX,seg MessagePré        9A 0D 00 -- --  CALL Distant
52        PUSH DX
1F        POP  DS
```

Dans le bloc où se situent les instructions de gauche, on trouve une directive ASSUME DS : Données. Mais cette directive n'agit sur les instructions produites que par les données immédiates entrant dans le code. Or ici, cette donnée ne peut être indiquée par *MASM* puisque l'adresse de chargement ne peut être connue à l'avance. C'est pourquoi *MASM* désigne cette adresse par des tirets dans le listage d'assemblage. La même remarque vaut pour l'instruction de gauche, où *MASM* n'a inscrit que la seule adresse qu'il a pu calculer : le déplacement de l'adresse absolue, l'appel distant étant traduit par *MASM* par une instruction de codop **09A**.

Quelles sont les valeurs inscrites par *LINK* dans le fichier *.EXE*?

On ne peut les observer sous *DEBUG* en appelant le fichier *.EXE* car *DEBUG*, en le chargeant, le transforme de la même façon que la fonction *EXEC*. Pour observer ces valeurs, on copie, sous *DOS*, le fichier *.EXE* en lui donnant une autre extension et c'est la copie que l'on examine sous *DEBUG*. On obtient ainsi dans notre cas :

```
062F BA2000  MOV  DX,0020        063B 9A0D002E00    CALL 002E:000D
0632 52      PUSH DX
0633 1F      POP  DS
```

```
;
; === CONSTANTES ==========================================
RC  EQU  0Dh
NL  EQU  0Ah
; === PILE ================================================
Pile SEGMENT  PARA  STACK
     ORG  00h
DW   100h DUP (?)
Pile ENDS
; === DONNEES ET VARIABLES ================================
Données SEGMENT
        ORG  84h
MessagePré   DB  RC,NL,RC,NL,"appel du programme distant"
             DB  RC,NL,'$'
MessagePost  DB  RC,NL,RC,NL,'retour au principal'
             DB  RC,NL,'$'
MessageDans  DB  RC,NL,RC,NL,'dans le programme distant'
             DB  RC,NL,'$'
Données ENDS
; === Sous-Programme ======================================
Code2 SEGMENT  PARA
      ORG  0Dh
      ASSUME  cs : Code2, ds : Données, ss : Pile
Distant PROC  FAR                    ; sous-programme Distant
         mov  dx,offset MessageDans   ; affichage
         mov  ah,09h                  ; d'un message
         int  21h
         ret                          ; retour appelant
Distant ENDP
Code2 ENDS
; === PROGRAMME PRINCIPAL ==================================
Code1 SEGMENT  PARA
      ORG  2Fh
      ASSUME  cs : Code1, ds : Données, es : Données
      ASSUME  ss : Pile
Principal PROC  FAR
          mov  dx,seg MessagePré
          push dx
          pop  ds
          mov  dx,offset MessagePré    ; message
          mov  ah,09h                  ; avant appel
          int  21h
          call Distant
          mov  dx,offset MessagePost   ; message
          mov  ah,09h                  ; après appel
          int  21h
          mov  ax,4C00h                ; fin
          int  21h
Principal ENDP
Code1 ENDS
; === FIN =================================================
     END Principal
```

Nous observons que les valeurs données par *LINK* sont les adresses du tableau ci-dessus. Observons à présent ce que l'on obtient lorsque le fichier *.EXE*, est chargé par *DEBUG* et se trouve donc prêt à être exécuté :

```
1589:002F BA7915  MOV  DX,1579     1589:003B 9A0D008715  CALL 1587:000D
1589:0032 52      PUSH DX
1589:0033 1F      POP  DS
```

On constate que les distances entre les trois adresses de segment apparaissant sur ce listage : **01579**, **01587** et **01589** sont liées par les relations **01587 = 01579 + 0E** et **01589 = 01579 + 010**. Elles correspondent exactement aux distances que nous avions indiquées précédemment. Ces inspections sous *DEBUG* font apparaître deux choses :

LINK a calculé des adresses de segment ***relatives*** à une origine absolue conventionnelle et la fonction *EXEC* a corrigé le programme *.EXE* au moment du chargement de façon à ce que la ***copie du programme*** en *RAM* soit pourvue des adresses adéquates en ajoutant toujours une même valeur : l'adresse, après chargement, de l''origine absolue'.

8.2.2. Assemblage et mise en oeuvre.

En-tête des fichiers .EXE.

Lorsque *LINK* recalcule les adresses à partir des informations élaborées par *MASM* dans tous les fichiers *.OBJ* concernés, il prépare également le travail de la fonction *EXEC*. Il le fait sous la forme d'une table, appelée ***table de redressement***, qui contient la liste des ***adresses à redresser***[1] par l'addition du numéro de paragraphe réel du point de référence pris par *LINK*.

Chaque entrée de la table s'étend sur deux mots et définit l'adresse absolue du *mot* à redresser, l'adresse de segment étant en fait la différence, en paragraphes, entre le numéro de paragraphe du bloc considéré et le numéro de paragraphe de l'origine absolue. C'est pourquoi nous appellerons cette adresse ***différentiel du CS local*** puisque le registre *CS* se rapporte au code.

LINK place cette table dans un partie du fichier appelée ***en-tête*** puisqu'elle constitue le début du fichier. L'en-tête ne contient pas tout de suite la table de redressement. Il contient d'abord un certain nombre d'informations structurées de la façon suivante :

A gauche, l'adresse du premier octet de l'information depuis 00, début du fichier.

00 **5A 4D** : signatu e d'un fichier .EXE[2]
02 reste de la longueur du fichier modulo 512
04 quotient entier de la longueur du fichier par 512
06 nombre d'entrées dans la table de redressement
08 taille de l'en-tête en paragraphes
0A nombre minimum de paragraphes requis en supplément
0C nombre maximum de paragraphes requis en supplément
0E valeur de *SS* au lancement du p ogramme
10 valeur de *SP* au lancement du programme
12 complément à 1 de la somme des mots du programme
14 valeur de *IP* au lancement du p ogramme
16 différentiel du *CS* local de la première instruction
18 ad esse, dans le fichier, de la table de redressement
1A nombre des programmes recouvrants
1C réservé par *MS-DOS*

Voici l'en-tête du programme de démonstration donné page 177 :

```
0100   4D 5A 4C 01 03 00 02 00-20 00 00 00 FF FF 00 00
0110   00 02 1E 19 2F 00 30 00-1E 00 00 00 01 00 30 00
0120   30 00 3E 00 30 00 00 00-00 00 00 00 00 00 00 00
```

Nous en déduisons que le fichier occupe **074C** octets, dont **0200** pour l'en-tête, que ce dernier contient une table de redressement débutant à l'adresse **001E**, et que la

---

[1] Le terme de ***relogement*** que l'on rencontre dans la littérature spécialisée de langue française nous paraît une traduction inexacte de l'anglais ***relocation***. En outre, le terme anglais ne nous paraît pas non plus des plus heureux. Il ne s'agit pas de situer quelque chose, mais de corriger des adresses de façon à ce que les branchements soient corrects. Par ailleurs, la correction est toujours *positive* puisque l'origine absolue est l'adresse la plus basse. C'est pourquoi le terme de ***redressement*** nous paraît plus approprié.

[2] Par conséquent, Z.M., soit Mark Zbikowski, selon la rumeur publique.

table possède 2 entrées. On lit ainsi les deux adresses ***différentielles*** suivantes :

```
30 00 30 00     3E 00 30 00      entrées
0030:0030       0030:003E        adresses
```

Exécution des fichiers .EXE.

L'origine absolue est donc définie par la fonction *EXEC* de *DOS*. Lorsque cette fonction charge un fichier *.EXE*, elle charge d'abord les **0200** premiers octets de l'en–tête dont elle vérifie les deux premiers octets. S'il y a lieu, elle charge la suite de l'en-tête. Puis, dans tous les cas, elle construit un préfixe pour le programme *.EXE* à constituer et charge le texte du programme juste après le préfixe qui joue dans ce cas le même rôle que dans le cas des programmes *.COM*. Les registres *ES* et *DS* sont initialisés par le numéro de paragraphe du préfixe et l'origine absolue est fixée au premier octet qui suit le préfixe. Dans notre exemple, le chargement du fichier **demo.exe** sous *DEBUG* montre le début du préfixe à **01549**. L'origine absolue est donc située au paragraphe numéro **01559**. Le tableau suivant résume les opérations de redressement effectuées par la fonction EXEC :

```
table      code       avant :      après :      instruction en RAM
0030:0030  1589:002F  BA2000       BA7915       MOV  DX,1579
0030:003E  1589:003B  9A0D002E00   9A0D008715   CALL 1587:000D
```

On a donc ajouté à chaque fois l'adresse d'exécution de l'origine absolue : **01559**.

Simulation de la fonction EXEC (2).

Ceci nous permet d'ajouter au simulateur de la fonction *EXEC* présenté au chapitre précédent les instructions correspondant au cas des fichiers *.EXE*. Le texte du programme donné en annexe 6 ne concerne que le chargement d'un fichier *.EXE*. Nous donnons ci-après une version plus compacte de la boucle de réécriture des adresses à redresser.

AX est initialisé par l'adresse d'exécution de l'origine absolue et *CX* par le nombre d'octets de la table lue en *DS*:**1006**, la table ayant été chargée depuis *DS*:**1000**. Une instruction **jcxz** chapeaute la boucle au cas où il n'y aurait rien à redresser.

```
0259 BOUCLE
0259 C4BC0010   LES  DI,[SI+1000]; différentiel d'adresse lu
025D 83C604     ADD  SI,+04       ; SI remis à jour
0260 8CC2       MOV  DX,ES        ; DX := ES
0262 01C2       ADD  DX,AX        ; correction
0264 8EC2       MOV  ES,DX        ; ES := CS local
0266 26         ES:               ;
0267 0105       ADD  [DI],AX      ; redressement
0269 E2FE       LOOP 0259         ; retour à BOUCLE
```

La contraction du code est essentiellement dûe à l'utilisation de l'instruction **les**.

L'initialisation des registres du processeur avant le **jmp far** transférant le contrôle de l'exécution au programme *.EXE* ne présente pas de difficulté par rapport à l'initialisation des registres dans le cas d'un programme *.COM* : on sait que *ES* et *DS* prennent l'adresse de segment du préfixe et que les valeurs nécessaires à *CS*, *IP*, *SS* et *SP* figurent dans l'en-tête, modulo le redressement en ce qui concerne les segments. Les instructions correspondantes se trouvent dans l'annexe.

8.3. Association DEBUG-MASM.

Le principe.

Il s'agit d'utiliser des programmes assemblés sous *DEBUG* par la méthode développée dans les chapitres précédents.

Moyennant quelques transformations simples que nous indiquerons un peu plus loin, on peut insérer un code hexadécimal dans un programme source destiné à être assemblé par *MASM* en le déclarant comme donnée. Comme *MASM* ne peut effectuer que des contrôles syntaxiques, il suffit d'écrire en *MASM* des références à ces données pour obtenir l'effet désiré.

Prenons le programme **source.csm** suivant :

```
0100 BA1001          MOV  DX,0110
0103 B409            MOV  AH,09
0105 CD21            INT  21
0107 B8004C          MOV  AX,4C00
010A CD21            INT  21

-d 110 L 0C
0110  0D 0A 6D 65 73 73 61 67-65 0D 0A 24          ..message..$
```

Nous donnons ci-après quatre façons d'inclure ce programme dans un fichier *.ASM* destiné à *MASM* :

```
Code  SEGMENT
      ORG  0100h
      ASSUME  cs : Code, ds : Code
      ASSUME  es : Code, ss : Code
Début :
      DB  0BAh,010h,001h,0B4h,009h
      DB  0CDh,021h,0B8h,000h,04Ch
      DB  0CDh,021h,098h,003h,08Bh
      DB  00Eh,00Dh,00Ah,06Dh,065h
      DB  073h,073h,061h,067h,065h
      DB  00Dh,00Ah,024h,006h,0B3h
      DB  003h,000h
Code  ENDS
      END  Début
```

```
Code  SEGMENT
      ORG  0100h
      ASSUME  cs : Code, ds : Code
      ASSUME  ds : Code, ds : Code
Début :
      DB  0BAh,010h,001h,0B4h,009h
      DB  0CDh,021h,0B8h,000h,04Ch
      DB  0CDh,021h
RC  EQU  0Dh
NL  EQU  0Ah
FM  EQU  024h
      ORG  0110h
      DB  RC,NL,'message',RC,NL,FM
Code  ENDS
      END  Début
```

Ci-dessous, à gauche, utilisation d'une macro et, à droite, d'une procédure :

```
call_104 MACRO
      DB  0EBh,002h
         ENDM
Code  SEGMENT
      ORG  0100h
      ASSUME  cs : Code, ds : Code
      ASSUME  es : Code, ss : Code
Début :
      call_104
      nop
      nop
      DB  0BAh,010h,001h,0B4h,009h
      DB  0CDh,021h,0B8h,000h,04Ch
      DB  0CDh,021h
RC  EQU  0Dh
NL  EQU  0Ah
FM  EQU  024h
      ORG  0110h
      DB  RC,NL,'message',RC,NL,FM
Code  ENDS
      END  Début
```

```
Code  SEGMENT
      ORG  0100h
      ASSUME  cs : Code, ds : Code
      ASSUME  es : Code, ss : Code
Début :
      CALL Programme
      MOV  AX,4C00h
      INT  021h
      ORG  0108h
Programme PROC NEAR
      DB  0BAh,010h,001h,0B4h,009h
      DB  0CDh,021h
      ret
Programme ENDP
RC  EQU  0Dh
NC  EQU  0Ah
FM  EQU  024h
      ORG  0110h
      DB  RC,NL,'message',RC,NL,FM
Code  ENDS
      END  Début
```

L'outil.

La mise en oeuvre de l'une ou l'autre de ces méthodes repose sur la transcription du code hexadécimal sous une forme 'assimilable' par *MASM*. On peut confectionner un utili ire effectuant ce travail à partir des programmes donnés dans le chargement d'un fichier servant de source et l'écriture d'un fichier but recopié ensuite sur disque, ainsi que des programmes d'affichage d'un octet ou d'un mot. Voici, ci-après, la boucle principale d'un programme de transformation :

La boucle est initialisée de la façon suivante : *DS:SI* pointe sur le premier octet du fichier *.COM* chargé, *ES:DI* pointe sur la zone tampon d'écriture et *CX* reçoit la longueur, en octets, du fichier source ; *BL* est initialisé à **010** et *DX* à **00** pour la division apparaissant dans la boucle :

```
02F0:BOUCLE                     ; transformation
02F0 83FA00   CMP  DX,+00       ; début d'une nouvelle ligne?
02F3 7508     JNZ  02FD         ; non : -> suite
02F5 B86462   MOV  AX,6264      ; si :
02F8 AB       STOSW             ; écrire 'DB'
02F9 B82020   MOV  AX,2020      ;
02FC AB       STOSW             ; puis '  '
02FD 30E4     XOR  AH,AH        ; AH := 00
02FF AC       LODSB             ; lire l'octet courant
0300 F6F3     DIV  BL           ; division pour conversion
0302 053030   ADD  AX,3030      ; conversion caractères
0305 3C3A     CMP  AL,3A        ; chiffre décimal?
0307 7202     JB   030B         ; oui : -> test sur AH
0309 0407     ADD  AL,07        ; non : correctif
030B 80FC3A   CMP  AH,3A        ; chiffre décimal?
030E 7203     JB   0313         ; oui : -> suite
0310 80C407   ADD  AH,07        ; non : correctif
0313 26       ES:               ;
0314 C60530   MOV  BY [DI],30   ; écrire '0'
0317 47       INC  DI           ;
0318 AB       STOSW             ; écrire l'octet
0319 B068     MOV  AL,68        ;
031B AA       STOSB             ; écrire 'h'
031C 42       INC  DX           ; un octet de plus transcrit
031D 83FA10   CMP  DX,+10       ; a-t-on écrit 16 octets?
0320 720A     JB   032C         ; non : suite
0322 31D2     XOR  DX,DX        ; si : DX à zéro
0324 B80D0A   MOV  AX,0A0D      ;
0327 AB       STOSW             ; écrire saut à la ligne
0328 E2C6     LOOP 02F0         ; -> BOUCLE
032A C3       RET               ; retour
032B 90       NOP               ;
032C B02C     MOV  AL,2C        ; ligne non terminée :
032E AA       STOSB             ; écrire ','
032F E2BF     LOOP 02F0         ; -> BOUCLE
0331 C3       RET               ;
```

MNEMONIQUE : Noter l'utilisation intensive des instructions **lods** et stos associées, respectivement, à *DS:SI* et *ES:DI*. Pour s'en souvenir : *SI* se réfère à la Source et *DI* à la Destination ; **lods** sert donc à Lire et **stos** à inScrire.

Application : retour au calcul de N!.

Nous allons appliquer cette méthode pour utiliser le programme de multiplication d'un entier naturel représenté par un registre afin de calculer rapidement la valeur de *N*! pour des valeurs de n représentables par un mot. Le programme de calcul est donc mis au point sous *DEBUG*. Nous utilisons *MASM* pour écrire l'***interface*** avec ce programme : ici, la lecture à l'écran de *N*, la mise en place de l'espace des registres avec ses structures de gestion, et l'affichage du résultat.

On trouvera à l'annexe 7 le texte *in extenso* du programme soumis à *MASM*. En particulier, la procédure **Afficheur** de ce programme montre la forme que prend la transcription, sous *MASM*, du code machine élaboré par *DEBUG*. Le premier problème à résoudre, dans la réalisation de l'*interface*, est celui de l'emplacement des programmes fabriqués par *DEBUG*. Dans la mesure où le volume de ces programmes est relativement 'léger', on peut conserver les adresses définies sous *DEBUG* : les directives ORG permettent alors de placer les 'morceaux' de codes ainsi prêts là où il le faut. Ces adresses étant assez lointaines, on peut utiliser le début du code pour la lecture à l'écran de *N* et pour la constitution des structures de gestion des registres.

La partie principale du programme **fact.asm** se présente ainsi :

```
Début :
      MOV  AX,CS              ; initialisations
      MOV  DS,AX              ; des registres de segments
      MOV  ES,AX              ; sur la valeur conférée à CS
      CLI                     ; par le chargeur de la fonction
      MOV  SS,AX              ; EXEC de DOS
      STI                     ;
      CALL LireN              ; CX := N lu à l'écran
      CALL Initialisations    ;
      JCXZ Résultats          ;
Boucle :                      ;
      PUSH CX                 ; compteur (initialement : N)
      PUSH BX                 ; (modifié par Multiplier)
      MOV  CX,[BP]            ;  DX est réservé à (R), ES:DI à R
      CALL Multiplier         ; programme fait sous DEBUG
      MOV  [BP],CX            ;
      POP  BX                 . (BX restauré)
      INC  BX                 ; BX croît de 1 à N
      POP  CX                 ; compteur
      LOOP Boucle             ;
Résultats :                   ;
      CALL Affichage          ; programme fait sous DEBUG
      MOV  AX,4C00h           ;
      INT  021h               ;
```

Le programme **Initialisations** reprend de nombreuses instructions du programme **0750** de l'annexe 4, mais il comporte trop de modifications pour qu'on puisse le reprendre tel qu'il est. En particulier, ce programme initialise *CX*, *DX* et *BX* pour la boucle des multiplications.

Le programme **Affichage** reprend, quant à lui le programme **0670** de l'annexe 4 et réalise l'*interface* avec les extraits '*DEBUG*' que constituent ses sous-programmes. Nous renvoyons à l'annexe pour le texte exact. Dans le listage de l'annexe, nous n'avons pas reproduit la transcription du code correspondant pour la simple raison qu'il se trouve sous forme symbolique dans d'autres programmes de l'annexe. La référence aux programmes concernés est indiquée dans le listage.

Compte tenu de certaines particularités de la syntaxe de *MASM*, le programme **Initialisations** est donné ci-contre avec la déclaration de la macro **EspaceDesRegistres** et des constantes utilisées dans cette dernière.

Dans la procédure **Initialisations**, l'appel au programme **Erreurs** est effectué par un **call**, bien que ce programme ne rende pas 'la main' au programme appelant. La raison en est que la sauvegarde de l'adresse de retour effectuée par le **call** permet de retrouver la cause de l'erreur en revenant à un listage du programme. Nous avons vu ce mécanisme au sujet d'une version simplifiée du programme d'erreur **0FD0** utilisée dans les programmes d'arithmétique multi-précision. Le programme **Erreurs** reprend le code

hexadécimal de ce programme à l'exception de quelques octets, qui, pour des raisons d'adressage, sont remplacés par leur forme selon les règles de *MASM*.

programme **Initialisations** :

```
Initialisations PROC NEAR
      AllocationZone
      JNC  ZoneAllouée
      CALL Erreurs      ; CALL et non JMP à cause de l'adresse!
ZoneAllouée :
      InstallerLaPile
      EspaceDesRegistres
      XOR  BX,BX
      INC  BX
      RET
Initialisations ENDP
```

macro **EspaceDesRegistres** :

```
EspaceDesRegistres MACRO
      PUSH DS
      PUSH CS
      POP  DS
      PUSH CX
      XOR  DI,DI
      XOR  AX,AX
      MOV  CX,8000h
      CLD
      REP  STOSW
      POP  CX
      MOV  word ptr [ValeurDeN],CX
      MOV  word ptr DS:[Répertoire],01C8h
      MOV  word ptr DS:[Répertoire+02],01h
      MOV  word ptr DS:[Répertoire+40h],01h
      MOV  word ptr DS:[TableDesZones],ES
      MOV  word ptr DS:[TableDesZones+20h],0E38h
      MOV  word ptr DS:[TableDesZones+40h],01h
      MOV  word ptr ES:[PremierSecteur],01h
      MOV  word ptr ES:[0000],0100h
      MOV  DI,PremierSecteur
      MOV  DX,0001h
      POP  DS
                  ENDM
```

Les constantes utilisées sont déclarées comme suit :

```
Répertoire EQU 04000h
TableDesZones EQU 03800h
PremierSecteur EQU 01C80h
ValeurDeN DW 00h
```

On observera le préfixage des adresses but dans les instructions **mov** ci-dessus. *MASM* ne supporterait pas qu'on écrive plus simplement :

```
MOV word ptr [Répertoire],01C8h
```

Ce qui est étonnant, pour deux raisons aux moins : dans le listage contenant cette forme, *MASM* la signale comme erreur grave, interdisant de ce fait la confection d'un fichier *.EXE*, alors que son analyseur l'assemble parfaitement ; par ailleurs, le même analyseur accepte sans sourciller l'instruction :

```
MOV word ptr [Répertoire+BX],01C8h
```

CHAPITRE 9

PERSPECTIVES

En conclusion de cette étude, nous proposons un regard sur les directions prises par les *constructeurs* dans l'évolution des processeurs.

Le développement actuel se caractérise par une sophistication accrue de la programmation qui s'accompagne d'une 'compactification' du matériel et des langages machines dans le but d'accroître considérablement la vitesse de calcul.

Dans ce bref aperçu, nous aborderons essentiellement deux directions qui intéressent au premier chef les langages d'assemblage : la protection et l'architecture à jeu d'instructions restreint.

9.1. La protection.

Le principe général.

Le maître mot de toute une conception nouvelle des micro-processeurs que l'on observe aujourd'hui chez les deux principaux fabricants est la ***protection***.

Il s'agit d'introduire, dans le domaine de la micro-informatique, une potentialité qui, jusque là, restait l'apanage des gros systèmes : l'utilisation d'un même système par plusieurs *usagers* et l'exécution par ce système de plusieurs tâches 'en même temps', c'est-à-dire, comme nous l'avons vu au chapitre 3, en *temps partagé*[1], car on reste toujours dans le cadre du principe : une seule instruction en cours à un instant donné d'où une seule tâche active à un instant donné.

Dans le domaine *PC*, ce fonctionnement existe en fait déjà, mais presque exclusivement au niveau de l'électronique, notamment pour la gestion de routine des périphériques. Le seul fonctionnement multi-tâche assuré dans ce cadre était celui de la commande *PRINT* de *DOS*, souvent appelée, pour cette raison, 'tâche d'arrière plan'. Le principe en est le suivant : chaque fois que le processeur n'est pas accaparé par l'utilisateur, *PRINT* met à profit ce 'répit' pour continuer sa tâche en la reprenant là où il l'avait laissée. Le but de la protection est d'étendre ce fonctionnement à tous les niveaux de l'exploitation de façon à ce que plusieurs utilisateurs puissent travailler sur la même machine ou qu'un seul puisse l'occuper simultanément à plusieurs tâches.

---

[1] Il existe depuis quelques années des machines sur lesquelles on peut non pas simuler mais ***réaliser*** des exécutions simultanées : les machines connexionnistes. Bien que ces machines restent dans le cadre théorique de la récursivité, leur conception est totalement différente de ce que nous avons vu ou qui nous reste à voir dans ce livre. Pour l'instant, ces machines sont en dehors du domaine de la micro-informatique.

Lorsque plusieurs usagers utilisent un même appareil en même temps, ce qui est la règle dans les systèmes en 'réseaux', il faut absolument écarter toute possibilité d'entraver une tâche de l'un d'entre eux ou de détériorer ses données du fait d'une autre tâche. Accessoirement, dans le cas d'un seul utilisateur effectuant plusieurs tâches, celui-ci peut être assuré que ces différentes exécutions ne vont pas interférer, sauf, bien entendu, s'il en a décidé le contraire.

L'idée de base est de protéger des zones de la mémoire réservées à différentes tâches (ce qui contient, comme cas particulier, celui de plusieurs usagers). Deux moyens sont mis en oeuvre dans ce but. D'une part la définition de ***niveaux d'action***, qui crée la notion de ***privilège*** (d'action), d'autre part la suppression de tout ***accès direct*** à la mémoire par l'adressage, ce qui rend possible la notion d'***espace privé***.

A partir du 80286, les processeurs *INTEL* de cette famille disposent de deux modes de fonctionnement : le mode réel, qui correspond à ce que nous avons décrit dans ce livre et le mode protégé que nous abordons à présent de façon succincte et qui fonctionne selon ces nouvelles modalités. S'il n'est pas question, en quelques pages, d'entrer dans les détails de ce nouveau mode de fonctionnement, il nous paraît utile de préciser quelques uns de ses principes dans le cas du 80286. En effet, le passage au 80386 et au 80486 n'apporte pas de conception nouvelle par rapport au 80286 mais des améliorations sensibles ou des approfondissements dans la voie ouverte par le 80286.

L'adressage du mode protégé.

L'adressage que nous avons vu jusqu'ici dans l'ouvrage peut-être considéré comme *direct* dans la mesure où, dans tous les cas, le programmeur peut savoir, en définitive à quel emplacement de la mémoire physique de la machine il accède. En effet, toute adresse était jusqu'ici constituée par un numéro de paragraphe déterminant une 'origine' des abscisses locales et par une abscisse depuis l'origine considérée : le déplacement. Ce numéro de paragraphe peut à son tour être considéré comme une abscisse par rapport à une origine absolue, le numéro 0000, après un changement d'échelle. Quand on parlait, dans ce cadre, d'adresse *indirecte*, il s'agissait du mode de calcul, étant entendu qu'une expression XXXX:YYYY constitue, par définition, une adresse directe.

A partir du 80286, le mode protégé supprime l'adressage direct. La mémoire physique de la machine n'est plus considérée comme toute la mémoire vive disponible, mais comme la réalisation *temporaire* et *locale* d'un espace beaucoup plus grand appelé ***espace virtuel***. L'unité de base de cet espace est le ***segment***, toujours de 64 Ko dans le cas du 80286. Il appartient au système d'exploitation de prévoir la gestion du déplacement des segments de la mémoire virtuelle entre la *RAM* et un support de masse. Lorsqu'un accès dans un segment est demandé, le système d'exploitation doit faire en sorte que le segment concerné soit présent en *RAM*.

Le mode d'accès à la mémoire le plus direct se fait en deux temps. Premier temps : on indique le segment visé par le numéro d'une entrée dans une table adéquate, défini par les 14 bits de poids fort d'un registre de segment ; les deux autres bits (b_0 et b_1) définissent le ***privilège requis***, noté PR. Cette entrée est constituée par un ***descripteur*** du segment. Il fournit plusieurs informations sur le segment, dont l'adresse physique (sur 24 bits pour le 80286) du premier octet du segment. Deuxième temps : le processeur ajoute à cette adresse 24 bits une adresse habituelle 16 bits fournie par le programmeur ; le résultat est l'adresse physique du point visé. Mais, rappelons-le, le mode protégé doit garantir le caractère privé des espaces affectés à différentes tâches. C'est le rôle des autres informations contenues dans le descripteur, ce que nous examinons à présent de façon plus précise.

Structure des descripteurs :
Il s'agit d'une zone de quatre mots (huit octets), contenant toutes les caractéristiques nécessaires à l'utilisation du segment considéré. Cette zone est organisée de la façon suivante en sous-zones, les bits étant notés de 00 à 63 :

00-16 : ***limite***, c'est-à-dire taille, en octets, du segment (au plus 64 Ko) ;

16-40 : ***base***, c'est-à-dire adresse physique, sur 24 bits, du premier octet du segment en mémoire vive réelle ;

40-48 : ***droits d'accès*** ; cet octet indique le type du segment (b_{43},b_{44}) : code, données, guichet ou système ; les droits d'accès indiquent aussi si le segment est présent ou non en mémoire physique (b_{47}) et le niveau de privilège du segment (b_{45},b_{46}) ; il indique également des informations propres au type de segment considéré ; ainsi, un descripteur de données indiquera si ces données peuvent être seulement lues ou si elles peuvent être réécrites, tandis que dans un descripteur de code, le même bit indiquera si ce code peut être seulement exécuté ou si l'on peut également le lire ;

48-64 : 0000 sur le 80286 si on veut que le code produit pour ce processeur fonctionne aussi sous 80386 et 80486.

Les segments de la mémoire ayant une taille de 64 Ko, il faut connaître le nombre de segments adressables pour connaître le volume de l'espace adressable virtuel. Les numéros de segment sont contenus dans un mot de 16 bits, mais seuls les quatorze bits de poids fort contribuent à définir ce numéro. Ceci fait donc $2^{14}*2^{16} = 2^{30}$ octets, ou encore un ***gigaoctet***, ce que nous noterons 1 Go[1]. En l'état actuel de la technique, un tel espace est nécessairement un espace virtuel[2].

Afin de permettre à un système d'exploitation et à des tâches de travailler en harmonie, le numéro d'un segment est défini par les treize bits de poids forts du mot. Le quatorzième bit, bit 2 du mot, indique à quelle table se réfère ce numéro. En effet, comme indiqué ci-dessus, les descripteurs se trouvent dans une table. Le processeur en distingue toujours deux : une ***table globale des descripteurs***, que nous noterons ***TGD*** et une ***table locale*** que nous noterons ***TLD***. La table globale, toujours en mémoire physique est accessible à toutes les tâches. Chaque tâche se voit attribuer une table locale. La *TLD* que le processeur connaît est celle de la tâche active à l'instant considéré puisque à chaque instant, une seule tâche est active.

Programmation.

Du point de vue de la programmation, les références à la mémoire utilisent les mêmes instructions que celles que nous avons vues.

Prenons l'exemple de l'instruction **mov ax,[bx]** en mode protégé. Le processeur lit dans les treize octets les plus forts de *DS* le numéro *N* de référence du segment dans lequel *BX* va servir d'abscisse locale ; puis il va chercher dans la table locale ou la table globale selon la valeur du bit 2 de *DS*. A l'entrée définie par *N* il lit, dans le descripteur du segment, l'adresse ***physique*** du premier octet de ce segment. Il obtient donc l'adresse physique du point visé en ajoutant à l'adresse de base du segment la valeur de *BX*. Le contenu du mot mémoire ainsi repéré peut alors être placé dans *AX*.

---

1 2^{10} est un kilo octet, 2^{20}, l'espace adressable du 80x86 en mode réel est de un méga octet, il est donc cohérent que 2^{30} reçoive à son tour un nom...

2 Notons que pour l'heure où 1 Go sera physiquement réalisé, on disposera d'espaces virtuels encore plus grands en concaténant, par exemple, deux 'mots' de 32 bits chacun. Et si cela ne suffit pas, la théorie ne manque pas de systèmes pour aller encore plus loin avec des notations aussi condensées que possibles.

La protection intervient dans ce processus à diverses reprises.

Tout d'abord, quand le processeur lit le descripteur, il lit aussi la limite indiquée. Si *BX* ne dépasse pas cette limite, l'instruction est exécutée. Sinon, le processeur produit une interruption (que le système d'exploitation doit bien entendu gérer) et l'instruction n'est pas exécutée. D'autres vérifications ont lieu. Si, par exemple, on demandait l'exécution de **mov [bx],ax**, le processeur examinerait l'octet d'accès du descripteur. Si ce dernier indiquait que le segment n'est pas inscriptible, l'instruction ne serait pas exécutée et une interruption serait produite.

Mais d'autres vérifications ont eu lieu *avant*, lorsque la valeur actuelle du registre *DS* y a été chargée. Supposons qu'elle ait été effectuée par une instruction **mov ds,ax**. Que s'est-il alors passé? Tout d'abord, le processeur a examiné le bit 2 de AX afin de vérifier si les bits 3-16, formaient un numéro valide d'entrée dans la table désignée. Si ce n'était pas le cas, une interruption spécifique aurait été déclenchée. Cela ne s'étant pas produit, le processeur a consulté le descripteur figurant à l'entrée définie par ce numéro. Il a vérifié les droits d'accès dans les bits 40-48 du descripteur : le segment devait être un segment de données ou bien un segment de code lisible. Sinon, une interruption se serait produite. Dans le même octet, il a comparé le privilège requis par *AX* aux bits 45 et 46 du descripteur. Le privilège requis n'a pas dépassé celui qu'indique le descripteur, sinon l'accès aurait été refusé et une interruption aurait été déclenchée. 'Les papiers d'identité étant en règle', le processeur a procédé au chargement de *AX* dans *DS*.

Toutes les instructions du 8086 s'exécutant en mode protégé sont soumises à ce type de contrôle chaque fois qu'il s'avère, *a priori*, indispensable. Tel n'est pas le cas, par exemple, d'une instruction **add bx,ax** ou **mov bx,ax**. Mais tel est le cas, en particulier, des **call** et des **jmp**.

Le 80286 reprend, pour l'essentiel, les instructions 8086, mais il les ***interprète*** de façon différente. En particulier, la valeur d'un registre de segment n'a pas le même sens qu'en mode réel. En mode protégé, elle est appelée ***sélecteur***, et c'est, comme nous l'avons constaté, un objet *mixte* contenant à la fois un numéro de segment, la référence à une table et l'indication d'un niveau de privilège. Naturellement, des instructions spécifiques permettent de manier les structures propres à la protection.

Les instructions du mode protégé.

Nous les donnons surtout pour que le lecteur puisse les reconnaître s'il a affaire à un code symbolique de 80x86 utilisant ce mode, car la place qui nous est impartie ne nous permet pas les développements importants que leur emploi nécessiterait.

Instructions fonctionnant également en mode réel :

En mode réel ces instructions préparent le passage du processeur au mode protégé qui s'effectue en mettant à 1 le bit 0 d'un registre particulier du processeur appelé ***mot d'état de la machine*** :

La syntaxe ci-après fournit le codop et, le cas échéant, le chiffre clé du modrm.

```
CLTS          0F 06    ; met à zéro l'indicateur de commutation de tâche que
                       le processeur met automatiquement à 1 chaque fois qu'une tâche est commutée ;
LGDT AI       0F 01 2  ; charge 6 octets depuis AI dans le registre de la TGD ;
LIDT AI       0F 01 3  ; charge 6 octets depuis AI dans le registre affecté à une
                       TLD, par définition la TLD de la tâche en cours ;
LMSW AR       0F 00 6  ; charge dans le mot d'état la valeur contenue dans AR :
                       intervient essentiellement à l'initialisation ;
SGDT AI       0F 01 0  ; copie l'adresse (six octets) de la TGD sur AI ;
SIDT AI       0F 01 1  ; copie l'adresse (six octets) de la TLD sur AI ;
SMSW AR       0F 01 4  ; copie le mot d'état dans AR.
```

Instructions non acceptées en mode réel :

Ces instructions utilisent ou manipulent le privilège ; on distingue le privilège requis (PR) que nous avons vu, le privilège ***inscrit*** (PI) et le privilège ***courant*** (PC), égal, par définition, au privilège inscrit de la tâche en cours ; la règle d'accès est la satisfaction de l'inégalité : *max* (PR,PC) ≤ PI. En général, *ZR* indique une vérification des privilèges ayant permis l'exécution de l'instruction et *NZ* signale un non respect de la règle et un rejet de l'action demandée.

ARPL AR,RM 63 ; compare les privilèges source (PR) et but (PI) : si la règle est vérifiée, *ZR* et PI := PR , sinon *NZ* et privilèges inchangés ;
LAR RM,AR 0F 02 ; charge dans l'octet fort de *RM* les droits d'accès du descripteur pointé par le sélecteur défini en *AR* sous bénéfice de *ZR*, sinon *NZ* ;
LLDT AR 0F 00 ***2*** ; *AR* contient un sélecteur pointant, en *TGD*, sur un descripteur de *TLD* ;
LSL RM,AR 0F 03 ; charge la limite du descripteur pointé par *AR* dans *RM* et *ZR* si la règle des privilèges est satisfaite, *NZ* sinon.
LTR AR 0F 00 ***3*** ; charge le registre de tâche depuis *AR* ; le segment d'état de tâche chargé (cf. plus loin) est marqué occupé, mais la commutation n'est pas effectuée ;
SLDT AR 0F 00 ***0*** ; le registre de la *TLD* est copié dans *AR* ;
STR AR 0F 00 ***1*** ; copie le registre de tâche dans *AR* ;
VERR AR 0F 00 ***4*** ; indique si le segment de descripteur pointé par le sélecteur contenu dans *AR* est lisible ou non : *ZR* si lisible, *NZ* sinon ;
VERW AR 0F 00 ***5*** ; même fonctionnement pour le droit d'écriture sur le segment indiqué indirectement par *AR* ;

Signalons que la protection prend du temps. Ainsi, en mode réel, l'exécution de **mov ds,ax** côute deux cycles d'horloge : le minimum. En mode protégé, pour le 80286, la même instruction côute 17 cycles!. Nous reviendrons plus loin sur ce point.

Particularités des 80386 et 80486.

Tout d'abord, l'introduction de registres sur 32 bits de la forme *ERX* (*R* = *A,B,C,D*), *ERI* et *ERP* pour les valeurs de *R* que le lecteur a déjà devinées. A cette introduction s'ajoute celle de registres de segments supplémentaires, *FS* et *GS*. Ces introductions modifient quelque peu le codage. A la formule d'adressage indirect indiquée au chapitre 3 s'ajoute une formule analogue faisant intervenir les registres *ERX* comme *indices*.

Au niveau de la mémoire virtuelle, les 80386 et 80486 se distinguent simplement par un espace physique beaucoup plus grand, lui-même virtuel, dans lequel la taille d'un segment peut atteindre 2^{32} bits d'où un espace adressable total de $2^{14}*2^{32} = 2^{46}$ octets. Du coup, la structure des descripteurs est 'étendue' en occupant la zone 48-64 des descripteurs mise à 0000 dans le 80286. Ces seize bits sont à présent presque complètement utilisés : *INTEL* ne se réserve qu'un seul bit pour la compatibilité avec les processeurs futurs. Dans ces deux octets, on trouve la fin de la limite (4 bits) puisque celle-ci est portée à 2^{20} au plus, la fin de l'adresse de base (8 bits), puisque les adresses physiques sont au format 32 bits dans ces processeurs, et des descripteurs complémentaires pour le système de pagination (gestion de la mémoire virtuelle).

Tâches et commutation des tâches.

Tâche.

Si un utilisateur conçoit sans peine ce qu'est une tâche, comment la définir en termes purement formels, pour qu'une machine puisse en gérer plusieurs? Dans les processeurs 80x86, x ≥ 2, une tâche est une suite d'exécutions référencée par une même valeur d'un registre particulier du processeur, le ***registre de tâche***. Ceci définit en fait la ***tâche active***. Plus généralement, une tâche est donc ce qui peut un jour se trouver actif. C'est pourquoi elle est définie par une référence à un segment d'un type particulier appelé le ***segment d'état de la tâche***, que nous noterons *SET*. Cette référence (un descripteur, donc), pouvant à tout moment être consultée doit se trouver dans la *TGD*.

Le segment d'état d'une tâche comprend exactement 22 mots qui décrivent l'état dans lequel se trouve la tâche. Ainsi, le segment contient-il la copie de l'état des registres du processeur au moment considéré de la tâche, les adresses des piles (autant de piles que de niveaux de privilège) et l'adresse de base du segment lui-même. Il contient aussi un sélecteur, dit ***pointeur de retour***, indiquant, le cas échéant, le segment d'état de la tâche vers laquelle on se dirige quand on quitte celle qui est définie par le segment. Enfin, le segment contient aussi le sélecteur de la table locale des descripteurs propre à la tâche considérée.

Il convient de souligner qu'une tâche n'est pas définie par autre chose que cette référence et le relevé des registres du processeur. En particulier, une tâche n'est pas un programme, bien que son déroulement puisse se limiter à l'exécution d'un seul programme qui ne contiendrait aucun appel. Mais une tâche est toujours plus qu'un programme ou un ensemble de programmes : elle est l'exécution de ces programmes dans un contexte donné. Ainsi, deux tâches différentes peuvent exécuter le même programme, que ce soit de façon identique ou différemment.

Afin de conférer à ces modalités une sécurité suffisante, le pointeur de retour et le sélecteur de la table locale des descripteurs sont considérés comme 'statiques', c'est-à-dire, comme ne pouvant être écrits qu'à l'initialisation. En effet, le niveau de privilège d'un segment d'état de tâche est maximal et il ne peut être modifié que par les mécanismes du processeur (pour l'enregistrement des valeurs actuelles de ses registres, par exemple), ou par un code de même niveau que son PL.

> Remarquons que la protection entraîne des contraintes qui, appliquées strictement, empêcheraient, par exemple, une chose aussi nécessaire que la mise au point d'un programme : supposons qu'il s'agisse d'un code nécessaire au système. Il sera dans le meilleur des cas accessible en lecture, mais en aucun cas, en écriture. La solution consiste à créer un double du descripteur du segment de ce code en déclarant ce descripteur comme descripteur de données réinscriptibles, et rien n'interdit non plus de redéfinir le niveau de privilège du nouveau segment...

Le passage à une nouvelle tâche s'effectue de plusieurs façons : par **call** ou **jmp** vers un descripteur de segment d'état de tâche ; par un call ou jmp vers un guichet de tâche ; par une interruption dont le numéro se réfère en fait à un guichet de tâche ; enfin, par un **iret** avec un indicateur particulier mis à 1. Dans ce dernier cas, le retour est défini par le pointeur de retour de la tâche en cours.

> Un mot sur les guichets. Il s'agit d'un type particulier de descripteur. Le processeur 80286 connaît deux sortes de guichet : les guichets call et les guichets de tâche. Les premiers s'inscrivent dans la conception habituelle des appels de programme. La nouveauté consiste dans le 'confort' et la sécurité apportés au passage des arguments. En effet, un tel guichet ne comporte pas d'adresse de base ni de limite, mais, en leur place, le sélecteur du segment où se trouve le code appelé (bits 16-32) et le déplacement du point d'entrée dans le code à l'intérieur du segment désigné (bits 0-16). Les huit bits 'libérés' par cette référence doublement indirecte sont utilisés pour définir le nombre d'arguments passés (au plus 31). On ne définit pas autre chose : il est sous-entendu que les arguments se trouvent sur la pile de l'appelant ; le processeur met alors sur la pile de l'appelé, et dans cet ordre : l'adresse de retour (*CS:IP*), les paramètres (recopiés depuis la pile de l'appelant) et l'adresse *SS:SP* de la pile de l'appelant. De la sorte, si on veut passer plus de 31 paramètres, la connaissance de l'adresse de la pile appelante permet de lire la suite des arguments s'il y a lieu.
>
> Le guichet tâche, comme son nom l'indique ne concerne pas un programme mais une tâche. Il est beaucoup plus simple qu'un guichet call car la seule chose à connaître, en dehors des 'droits d'accès et privilèges' est le sélecteur qui pointe sur un descripteur de *SET*. Ce sélecteur est placé aux bits 16-32 et, dans un descripteur de guichet tâche, les bits 0-16 et 32-40 ne sont pas utilisés.

Enfin, au niveau des instructions machine, les différents types d'appel ne se distinguent pas des instructions **call**, **jmp** et **iret** du 8086. La différence se fait d'abord par le mode du processeur, ce que le processeur sait, bien évidemment, à chaque instant. Ensuite, elle est donnée par les valeurs des arguments. Il n'y a pas lieu de créer une instruction pour guichet call et une instruction pour guichet tâche puisque la différence porte sur le type du guichet qui est inscrit dans le descripteur pointé par l'instruction.

Commutation de tâche et fonctionnement multi-tâche.

La commutation de tâche est le passage d'une tâche à une autre. Il s'effectue selon l'une des quatre modalités de passage d'une tâche à une autre que nous avons vues. Mais cette description n'indique pas immédiatement comment on peut, de cette façon faire fonctionner plusieurs tâches en même temps.

Le principe de fonctionnement multi-tâche dans le contexte d'une tâche active unique à chaque instant est nécessairement celui du temps partagé. Le maître d'oeuvre installé à l'initialisation par le système d'exploitation est couplé à une interruption prévue à cet effet. Chaque fois que l'interruption est appelée, le maître d'oeuvre consulte la liste des tâches 'en souffrance', c'est-à-dire qui ont été commencées mais qui sont interrompues sans être terminées, et, en fonction de critères dont nous ne discuterons pas ici, indique la tâche que le processeur va reprendre, au point où elle avait été abandonnée, et ce jusqu'à ce que lui, maître d'oeuvre, soit de nouveau appelé.

Comme nous l'avons vu au chapitre 3 avec la tâche d'entretien de l'écran, pour mener à bien cette tâche de 'distribution des rôles', le maître d'oeuvre doit être appelé de l'extérieur par une interruption provoquée par un événement matériel se produisant régulièrement. On va donc coupler le maître d'oeuvre avec, par exemple, l'interruption de l'horloge et décider que tous les x coups d'horloge (tous les 0100 cycles, par exemple), l'interruption l'appellera pour qu'il accomplisse le programme de distribution que nous avons esquissé. Les instructions que nous avons décrites page 188 ont été conçues dans ce but.

Notons qu'un autre aspect de la protection est l'existence de plusieurs instructions que le processeur n'exécutera qu'avec un privilège courant nul qui sont : **clts**, **lgdt**, **lldt**, **lidt** **lmsw** et **ltr**. Si le programme d'initialisation installe le système d'exploitation et le maître d'oeuvre au niveau de privilège nul, il suffit ensuite au maître d'oeuvre d'affecter à toute tâche un niveau au moins égal à un pour se garantir contre toute tentative de 'coup d'état' de la part d'une tâche. En effet, la création de segment étant ultimement de son ressort, il pourra fixer tout privilège inscrit à une valeur plus grande que 1 et interdire, de ce fait, à tout autre code l'accès aux instructions permettant la commutation des tâches.

Il convient de noter que cette 'règle du jeu' général pourrait se faire par voie logicielle en mode réel : il suffirait de placer un gestionnaire adéquat dans l'interruption **08** qui reçoit les impulsions de l'horloge. On conçoit sans peine que cela se traduirait par un travail fort lent des tâches ainsi exécutées 'simultanément'[1]. L'implantation de cette nouvelle 'donne' dans le matériel transforme radicalement la perspective et rend possible le fonctionnement multi-tâche dans des conditions satisfaisantes, pourvu que le nombre de tâches concurrentes ne soit pas trop élevé. Cette facilité paie cependant un tribu relativement élevé au temps d'exécution, précisément, et à la mémoire. Nous l'avons vu sur l'exemple de l'instruction **mov ds,ax**. Il y a 'mieux' : un **call** en mode réel, tout 'périlleux' qu'il soit est plus rapide qu'un **call** en mode protégé : ainsi l'appel par un guichet de tâche demande plus de *quatorze fois* le temps d'exécution d'un **call far** en mode réel. Quant au délai le plus court, il n'est celui du mode réel que pour les **call** intra-segment les plus ordinaires, certes, les plus nombreux.

---

[1] C'est cependant, plus ou moins, ce que font les logiciels 'fenêtres multiples' fonctionnant sur les ordinateurs équipés de 8086).

9.2. Architectures à jeu d'instructions restreint.

On constate ainsi que la sophistication conduisant à inclure davantage de logiciel dans le matériel aboutit, à conception de processeur égal, à un accroissement des délais d'exécution qui ne peut être contre-balancé que par l'accélération 'intrinsèque' des horloges et du rendement des circuits électroniques. Néanmoins, les progrès dans cette voie ne paraissent pas infinis. Quand bien même le seraient-ils, pourquoi ne pas chercher une conception différente susceptible d'accélérer la vitesse d'exécution de façon spectaculaire et ce, à conditions technologiques égales?

C'est la question que se sont posés plusieurs équipes de recherche dans des universités américaines au début des années 80. Ces recherches ont conduit à la notion d'architecture à jeu d'instructions restreint, en anglais *RISC* (*R*estricted *I*nstruction *S*et *C*omputer), qui s'est peu à peu répandue. Aujourd'hui, tous les constructeurs proposent leurs processeurs *RISC*, dont certains intégrent également le fonctionnement de la protection et la gestion du mécanisme de mémoire virtuelle.

Pour comprendre ce dont il s'agit, rappelons-nous le codage des instructions du 8086. Il conduit, d'un point de vue purement syntaxique, à des instructions se présentant sur un nombre *variable* d'octets : de un à six, sur le 8086, voir un nombre plus grand pour les instructions du co-processeur mathématique ou les instructions du mode protégé dans le cas des 80x86, $x \geq 2$. Ceci concourt, comme nous l'avons vu, si l'on complexifie encore les instructions, à *ralentir* la durée moyenne d'exécution d'une instruction.

Sans entrer dans les détails plus fins des circuits électroniques, l'exécution d'une commande par le processeur s'effectue selon le schéma suivant :

Le processeur charge l'instruction qui vient à son tour. Puis il la décode. En fonction de ce décodage, il prend éventuellement connaissances des *données* que contient l'instruction (indication de déplacement dans un adressage indirect ou donnée immédiate), puis il exécute l'instruction quand tous les circuits concernés ont été préparés à cette tâche. Chaque instant, dans cette décomposition prend un temps plus ou moins long mesuré par les cycles produits par l'horloge du système. Afin de gagner du temps, le processeur fait en réalité plusieurs choses en même temps, car à chacune de ces étapes correspond un organe spécialisé du processeur. En effet, pendant que le décodeur s'occupe d'une instruction, soit *A*, le chargeur recherche déjà la suivante, soit *B*. Puis, lorsque le lecteur prend connaissance des données de *A* en les transférant aux circuits concernés (par l'adressage ou les calculs), le décodeur s'occupe de *B* tandis que le chargeur repère déjà l'instruction qui suit *B*, soit *C*. Et pendant que l'exécuteur est aux ordres de *A*, le lecteur étudie *B*, le décodeur déchiffre *C* et le chargeur à déjà en vue une quatrième instruction, *D*. C'est pourquoi les instructions sont lues par groupe de plusieurs octets à la fois (six dans le cas du 8086) et placées en attente dans des registres internes au processeur[1] et qui constituent ce qu'on appelle le ***pipeline***.

Ce schéma idéal est fort malmené dans le cas des codes traditionnels (du 8086 au 80486 inclus[2]). Deux raisons y contribuent : la longueur inégale des instructions et les branchement. Ces derniers, en effet, viennent interrompre l'empilement de la file d'attente : bien pire, ils provoquent son vidage dans les modèles 'traditionnels').

La solution imaginée par les créateurs de l'architecture *RISC* consiste à 'standardiser' les instructions qui devront être codées par un mot unique (de 32 bits). Il

---

[1] Ils ne sont pas accessibles au programmeur.

[2] Ce qui vaut aussi pour les 68000, 68020 et 68030 de Motorola pour ne faire de peine à personne) et ne retire rien, bien entendu, à l'excellence de tous ces produits.

en résulte une refonte du jeu des instructions qui devra être plus *réduit*, d'où le nom de ce type d'architecture.

La conception de l'adressage doit être simplifiée et, d'une façon générale, le code produit pour un même programme sera plus long. En quelque sorte, un certain pas vers la machine théorique que nous avons décrite au chapitre 2 qui ne comprend, elle que quatre instructions...

Cette standardisation satisfait donc à la première condition de fonctionnement du pipeline 'sans à-coups'. La seconde condition concerne les branchements : dans le 'moteur' à quatre temps que nous avons vu ci-dessus, si *A* est un branchement vers une instruction qui n'est pas *B*, on se heurte à une difficulté. Le moyen de la résoudre, est d'intercaler des **nop** après chaque branchement : un examen simple permet de se convaincre que deux suffisent (à la lecture, le processeur sait qu'il devra dérouter le chargeur et il n'a rien de particulier à exécuter) cela retarde un peu l'exécution mais évite de 'déstabiliser' le chargeur du processeur.

Dernier point à signaler :

Le choix du jeu des instructions. Il est remarquable de constater que tous les projets se sont fondés sur une approche *statistique* du problème. Le raisonnement étant : regardons quelles sont les instructions machine les plus fréquentes dans les programmes couramment exécutés, quels sont les modes d'adressage, les appels, et ainsi de suite, et faisons un processeur reprenant les instructions qui reviennent le plus souvent. Cette démarche n'est pas dépourvue d'intérêt et conduit à des résultats remarquables, même si quelques uns des résultats des observations se trouvent plus que biaisés par les contraintes a priori qu'imposent les langages de programmation les plus massivement utilisés et qui se trouvent être les plus rigides...

Nous n'entrerons pas dans le détail de ces choix. Dans ce bref aperçu, nous nous contenterons de signaler l'estimation des performances : l'uniformisation des instructions et leur réduction par rapport aux plus longues rendent possible la réalisation (ou presque) 'en régime de croisière' de l'objectif de ce type d'architecture qu'exprime l'équation :

délai d'exécution d'une instruction = un cycle de temps machine

Avec l'augmentation impressionnante des cadences d'horloges de ces dernières années, le chiffre de dix millions d'instructions exécutées par seconde (MIPS) est un chiffre couramment réalisé par ce type de processeurs. Aussi, l'accélération ainsi obtenue compense-t-elle largement la perte due à l'allongement du code consécutive à la restriction du jeu des instructions et aux retards imposés par les branchements.

Les progrès paraissent devoir continuer dans cette direction comme en témoigne l'intégration, par l'architecture *RISC* du mode protégé. Nous permettra-t-on de suggérer que l'étude critique des modèles théoriques est susceptible d'apporter des éléments d'innovation intéressants?

ANNEXES

Dans cette partie, le lecteur trouvera les listages des principaux programmes illustrant l'ouvrage. Même confectionnés sous *DEBUG*, les listages d'assembleur sont volumineux. Aussi les avons-nous reproduits ici de façon condensée sur le modèle suivant :

```
  0D30: Décrementer le secteur courant          Adresse d'appel et nom du programme
(0D08).                                         adresse du (des) programme(s) appelant(s)
0D30 PUSH CX          ; compteur
0D31 MOV  CX,0008     ;
0D34:BOUCLE           ; buit mots               le haut d'une boucle est indiqu  ainsi
0D34 INC  DX          ; vérif. |R|              R pour registre, |R| taille (ici en mots)
0D35 ES:              ;
0D36 MOV  AX,[DI]     ; lire mot
0D38 SUB  AX,0001     ; -1
0D3B ES:              ; écrire
0D3C MOV  [DI],AX     ; résultat
0D3E JNB  0D46        ; r=0: fini               r : retenue
0D40 INC  DI          ; r=1: mot
0D41 INC  DI          ; suivant
0D42 LOOP 0D34        ; -> BOUCLE               le bas d'une boucle est signalé ainsi
0D44 POP  CX          ; sortie CY               sortie avec indicateur CF à CY (1)
0D45 RET              ;
0D46 POP  CX          ;
0D47 CLC              ; R.A.S.                  sortie normale
0D48 RET              ;
```

La réduction des listages à ce format nécessite des abréviations. Souvent, le contexte permet de les comprendre sans risque d'erreur. Nous signalons cependant les plus fréquentes :

```
fdf : fin de fichier                M : multiplicande
st  : saut                          m : multiplicateur
lgn : ligne                         D : dividende
R.A.S. conditions remplies          d : diviseur
R   : registre                      q : quotient
S   : secteur                       r : retenue
|R| : taille d'un registre
[R] : nombre de secteurs d'un registre
{R} : nombre de mots (d'octets) du dernier secteur
```

Afin de lire plus facilement le listage, les références de saut ou d'appel sont en caractères gras et dans la colonne des adresses, une adresse référencée l'est de même. Les mnémoniques suivantes sont également en caractère gras : **ret**, **retf**, **repz**, **repnz** et **loop**. L'instruction **jmp** ou *jcond* l'est aussi lorsque le saut correspondant définit une boucle. Les adresses de variables ou de tableaux sont en italiques gras.

Au début de chaque annexe, on donne, très succinctement, la liste des ressources mises en oeuvre. L'indication succincte du rôle de ces ressources dans le programme utilise les abréviations ci-dessus et les notations de l'ouvrage sans autres références : se reporter aux paragraphes concernés.

ANNEXE 1 : simulation d'une machine de Minsky

Ressources utilisées par le programme :

```
0166  segment zone allouée
0232  taille fichier (octets)
0358  table débuts de mnémoniques
0360  tampon d'écriture
0364  adresses d'appel (02D8)
036C  nombre d'instructions
0486  table des mnémoniques
04AE  compteur de lignes
0588  nombre de registres
0764  paramètres mode graphique
0770  table des chiffres décimaux
077A  tamon d'écriture décimale
07E4  tampon de transcription
086C  nombre d'essais de lecture
0A20  adresse S courant
0A22  n° de secteur de S
0A24  adresse TAS de S
0A26  taille de R en mots : |R|
0A28  adresse de R en répertoire
0A2A  adresse de 1er S de R
0A2C  n° de 1er S de R
0A30  tampon de lecture à l'écran
0B05  adr. code HLT implicte
1070  nombre d'erreurs
1072  pointeur pile des erreurs
1074  pointeur dernière erreur
2000  table des noms de registres
4000  répertoire des registres
6000  table adr. instructions
8000  zone transcodage
```

0100: Programme principal.

```
0100 MOV  AX,CS        ; CS placé
0102 MOV  DS,AX        ; dans les
0104 MOV  ES,AX        ; registres
0106 MOV  SS,AX        ; de segments
0108 NOP               ; (déboguage)
0109 CALL 0150         ; rest. 64 Ko
010C CALL 0158         ; alloc. 64 K
010F JNB  0120         ; fait: suite
0111 MOV  DX,0170      ; msg. échec
0114 MOV  AH,09        ;
0116 INT  21           ;
0118 MOV  AX,4C00      ; TERMINAISON
011B INT  21           ; retour DOS
0120 CALL 0190         ; lire source
0123 JNB  0130         ; fait: suite
0125 JMP  0118         ; non: -> fin
0130 CALL 02D8         ; TRANSCODAGE
0133 JB   013F         ; si erreur
0135 CALL 06A0         ; ARGUMENTS
0138 JB   013F         ; si erreur
013A CALL 0B00         ; EXECUTION
013D JNB  014B         ; sans erreur
013F CALL 0FD0         ; ERREURS
0142 JMP  0118         ;
014B CALL 0DE0         ; RESULTATS
014E JMP  0118         ; -> fin
```

0150: Restriction mémoire (0100).

```
0150 MOV  BX,1000      ; taille par.
0153 MOV  AH,4A        ; restriction
0155 INT  21           ; allocation
0157 RET               ;
```

0158: Allocation zone RAM (0100)

```
0158 MOV  BX,1000      ; taille par.
015B MOV  AX,4800      ; allocation
015E INT  21           ; en RAM
0160 JB   0165         ; échec
0162 MOV  [0166],AX    ; alloué
0165 RET               ;
```

adresse segment zone allouée

```
0166 00 00                ..
```

message

```
0170 0D 0A 'espace non alloué'
0183 0D 0A 24
```

0190: Chargement fichier source (0100).

```
0190 MOV  DX,0298      ; message:
0193 MOV  AH,09        ; nom du
0195 INT  21           ; fichier
0197 MOV  DX,025E      ; réception
019A MOV  AH,0A        ; du nom
019C INT  21           ;
019E MOV  CL,[025F]    ; taille nom
01A2 XOR  CH,CH        ;
01A4 MOV  DI,CX        ; on écrit
01A6 MOV  BY [DI+0260],00; 00 après
01AB MOV  DX,0260      ; adr. du nom
01AE MOV  AX,3D00      ; pour fonc.
01B1 INT  21           ; ouverture
01B3 JNB  01BE         ; sans erreur
01B5 MOV  DX,0210      ;
01B8 MOV  AH,09        ; fichier non
01BA INT  21           ; trouvé
01BC STC               ;
01BD RET               ;
01BE PUSH AX           ; AX = sésame
01BF MOV  BX,AX        ;
01C1 MOV  AX,4202      ; ptr. fich.
01C4 MOV  CX,0000      ; mis à fdf
01C7 MOV  DX,0000      ; (taille du
01CA INT  21           ; fichier)
01CC MOV  [0232],AX    ; oct. faible
01CF MOV  [0234],DX    ; idem, fort
01D3 CMP  DX,+00       ; > 64 Ko?
01D6 JZ   01E7         ; non: suite
01D8 MOV  AH,02        ; si: refusé
01DA MOV  DL,07        ; 'biiip'
01DC INT  21           ; puis
01DE MOV  DX,0238      ; message
01E1 MOV  AH,09        ;
01E3 INT  21           ;
01E5 STC               ;
01E6 RET               ;
```

```
01E7 MOV  AX,4200      ; ptr. fich.
01EA MOV  CX,0000      ; remis au
01ED MOV  DX,0000      ; début
01F0 INT  21           ;
01F2 PUSH [0166]       ; ZA, zone
01F6 POP  DS           ; allouée: DS
01F7 MOV  DX,0000      ; depuis 0000
01FA CS:               ; code par CS
01FB MOV  CX,[0232]    ; taille fic.
01FF MOV  AH,3F        ; fonc. DOS
0201 INT  21           ; de lecture
0203 POP  BX           ; BX:=sésame
0204 MOV  AH,3E        ; fermer
0206 INT  21           ; fichier
0208 PUSH CS           ;
0209 POP  DS           ; DS sur code
020A CALL 02C2         ; correction
020D CLC               ; sans erreur
020E RET               ;
```

Variables et données du programme 0190.

message d'erreur

```
0210 0D 0A 'fichier non trouvé,'
0225 ' arrêt...' 0D 0A 24
```

taille du fichier

```
0232 00 00 00 00
```

message d'erreur

```
0238 0D 0A 'fichier trop volumi'
024B 'neux, arrêt...' 0D 0A 24
```

tampon de réception du nom du fichier

```
025E 40 00
0260 à 0298: 038 octets à 00
```

message

```
0298 0D 0A 'entrer le nom du fi'
02AB 'chier source' 0D 0A '   >'
02BD ' ' 24
```

02C2: Correction possible de la taille du fichier (0190).

```
02C2 MOV  BX,[0232]    ; BX:=taille
02C6 PUSH [0166]       ; ES pointe
02CA POP  ES           ; sur ZA
02CB ES:               ; AL:=dernier
02CC MOV  AL,[BX-01]   ; octet fich.
02CF CMP  AL,1A        ; Ctrl-Z?
02D1 JNZ  02D7         ; non: retour
02D3 DEC  WO [0232]    ; oui: taille
02D7 RET               ; corrigée
```

02D8: TRANSCODAGE DU FICHIER SOURCE (0100).

```
02D8 PUSH [0166]       ; 1ère passe
02DC POP  ES           ; ES sur ZA
02DD XOR  DI,DI        ; ES:DI: lire
02DF XOR  SI,SI        ; DS:8000+SI:
02E1 ;                 ; code instr.
02E1 XOR  DX,DX        ; DX: n°inst.
02E3:BOUCLE1           ;
02E3 CALL 0370         ; début suiv.
02E6 CALL 0660         ; stock. adr.
02E9 ;                 ; code inst.
02E9 JB   030D         ; si erreur
02EB CMP  DI,[0232]    ; fdf?
02EF JNB  0301         ; oui: pas. 2
02F1 XOR  BX,BX        ;
02F3 MOV  BL,AL        ; type instr.
02F5 ADD  BL,BL        ; cv adresse
02F7 CALL [BX+0364]    ; pour appel
02FB CALL 0590         ; fin de lgn.
02FE INC  DX           ; instr. sv.
02FF JMP  02E3         ; -> BOUCLE1
0301 CMP  BY [1070],00 ; erreurs?
0306 JZ   0313         ; non: suite
0308 STC               ; oui: sortie
0309 RET               ;
030D CALL 0496         ; st.l.
0310 INC  DX           ; compteur
0311 JMP  02E3         ; -> BOUCLE
0313 CALL 0660         ; adr. dern.
0316 MOV  [0B05],SI    ; instruction
031A MOV  [036C],DX    ; nbre instr.
031E XCHG BX,BX        ; remplissage
0320                   ;
0320 XOR  DI,DI        ; 2ème passe
0322 XOR  SI,SI        ;
0324:BOUCLE2           ;
0324 CMP  SI,[0B05]    ; fini?
0328 JNB  0354         ; oui: -> fin
032A MOV  AL,[SI+8000] ; non: lire
032E CMP  AL,02        ; NUL ou INC?
0330 JB   034E         ; oui: suiv.
0332 CMP  AL,02        ; instr. ST?
0334 JNZ  0339         ; non: DSZ
0336 INC  SI           ; oui: lire
0337 JMP  033C         ;
0339 ADD  SI,+03       ; sauter nom
033C MOV  BX,[SI+8000] ; lire  n°
0340 ADD  BX,BX        ; cv adresse
0342 MOV  AX,[BX+6000] ; dépl.instr.
0346 MOV  [SI+8000],AX ; adresse/n°
034A INC  SI           ; au début
034B INC  SI           ; ins. suiv.
034C JMP  0324         ; -> BOUCLE2
034E ADD  SI,+03       ; passer
0351 JMP  0324         ; -> BOUCLE2
0353 NOP               ;
0354 CLC               ; sans erreur
0355 RET               ;
```

Tableau des débuts d'instruction.

```
0358 4E 6E 49 69 53 73    NnIiSs
035E 44 64                Dd
```

Zone pour écrire le nom présumé d'instruction.

```
0360 00 00 00 00
```

Adresses des programmes de traduction des instructions.

```
0364 E0 03 E0 03 C0 03 00 04
```

Nombre des instructions.

```
036C 00 00
```

0370: Détection du début de l'instruction suivante (02D8).

```
0370:BOUCLE
0370 CMP  DI,[0232]    ; fdf?
0374 JB   0378         ; non: suite
0376 CLC               ;
0377 RET               ;
0378 ES:               ; ES sur ZA
0379 MOV  AL,[DI]      ; lire octet
```

```
037B CMP  AL,3B       ; comment.?
037D JNZ  0386        ; non: suite
037F CLC              ; oui:
0380 INC  DI          ; octet suiv.
0381 CALL 0496        ; st.l.
0384 JMP  0370        ; -> BOUCLE
0386 CMP  AL,20       ; blanc?
0388 JNZ  038D        ; non: tests
038A INC  DI          ; octet suiv.
038B JMP  0370        ; -> BOUCLE
038D PUSH ES          ;
038E PUSH DI          ;
038F PUSH DS          ; init. comp.
0390 POP  ES          ; par SCASB
0391 MOV  DI,0358     ; tableau
0394 MOV  CX,0008     ; sa taille
0397 CLD              ; -> en haut
0398 REPNZ            ; différence
0399 SCASB            ; AL,tableau
039A JZ   03A7        ; trouvé
039C CALL 0FD0        ; non (pb: ;)
039F STC              ; (erreur)
03A0 POP  DI          ; équilibrage
03A1 POP  ES          ; de la pile
03A2 RET              ;
03A3 XCHG BX,BX       ; remplissage
03A5 XCHG BX,BX       ;
03A7 CMP  AL,60       ; minuscule?
03A9 JA   03AD        ; oui: écrire
03AB ADD  AL,20       ; non: cv min
03AD MOV  [0360],AL   ; lère lettre
03B0 SUB  DI,0358     ; DI converti
03B4 DEC  DI          ; en type
03B5 SHR  DI,1        ; instruction
03B7 MOV  AX,DI       ; mis dans AL
03B9 CLC              ;
03BA POP  DI          ; (lecture)
03BB INC  DI          ; mise à jour
03BC POP  ES          ; ES retouvé
03BD RET              ;
```

03C0: Traduction de ST *n* (02D8).

```
03C0 MOV  CX,0002     ;
03C3 CALL 0428        ; VERIFICAT.
03C6 JNB  03C9        ; sans erreur
03C8 RET              ;
03C9 MOV  [SI+8000],AL; type stocké
03CD INC  SI          ; mise à jour
03CE CALL 05E8        ; AX:=réf.
03D1 JB   03D9        ;
03D3 MOV  [SI+8000],AX; réf.stockée
03D7 INC  SI          ; SI
03D8 INC  SI          ; mis à jour
03D9 RET              ;
```

03E0: Traduction de NUL *RX* et de INC *RX* (lère passe) (02D8)

```
03E0 MOV  CX,0003     ;
03E3 CALL 0428        ; VERIFICAT.
03E6 JNB  03E9        ; sans erreur
03E8 RET              ;
03E9 MOV  [SI+8000],AL; type stocké
03ED INC  SI          ; mise à jour
03EE CALL 04B8        ; nom regist.
03F1 JB   03F9        ; erreur
03F3 MOV  [SI+8000],AX; nom stocké
03F7 INC  SI          ; SI
03F8 INC  SI          ; mis à jour
03F9 RET              ;
```

0400 : Traduction de DSZ *RX,n* (lère passe) (02D8)

```
0400 MOV  CX,0003     ;
0403 CALL 0428        ; VERIFICAT.
0406 JNB  0409        ; sans erreur
0408 RET              ;
0409 MOV  [SI+8000],AL; type stocké
040D INC  SI          ; mise à jour
040E CALL 04B8        ; nom regist.
0411 JNB  0414        ;
0413 RET              ;
0414 MOV  [SI+8000],AX; nom stocké
0418 INC  SI          ; SI
0419 INC  SI          ; mis à jour
041A CALL 05E8        ; AX:=réf.
041D JNB  0420        ; sans erreur
041F RET              ;
0420 MOV  [SI+8000],AX; réf.stockée
0424 INC  SI          ; SI
0425 INC  SI          ; mis à jour
0426 RET              ;
```

0428: Vérification des noms d'instruction (03C0,03E0,0400).

```
0428 PUSH AX          ; (AL type)
0429 MOV  BX,0009     ; pointeur
042C:BOUCLE           ;
042C CMP  DI,[0232]   ; fdf?
0430 JNB  0445        ; oui: erreur
0432 ES:              ; non:
0433 MOV  AL,[DI]     ; lire
0435 CMP  AL,60       ; minuscule?
0437 JA   043B        ; oui: écrire
0439 ADD  AL,20       ; non: corr.
043B MOV  [BX+0358],AL; écrire
043F INC  DI          ; mise à jour
0440 INC  BX          ; mise à jour
0441 LOOP 042C        ; -> BOUCLE
0443 JMP  044B        ; -> vérifier
0445 POP  AX          ;
0446 CALL 0FD0        ; (pb: fdf)
0449 STC              ;
044A RET              ;
044B POP  AX          ; AL:=type
044C PUSH AX          ; resauver
044D CMP  AL,02       ; instr. ST?
044F JNZ  0456        ; non: tests
0451 MOV  BY [BX+0358],20; oui
0456 PUSH ES          ; préparer
0457 PUSH DI          ; test
0458 PUSH SI          ; CMPSB
0459 PUSH DS          ;
045A POP  ES          ;
045B MOV  DI,0486     ; tableau
045E XOR  AH,AH       ; réajuster
0460 MOV  CL,02       ; DI
0462 SHL  AX,CL       ; pour entrée
0464 ADD  DI,AX       ; idoine du
0466 MOV  SI,0360     ; tableau
0469 MOV  CX,0004     ; 4 lettres
046C CLD              ;
046D REPZ             ; (égalité)
046E CMPSB            ;
046F JZ   047A        ; c'est bon
0471 CALL 0FD0        ; erreur
0474 STC              ;
```

```
0475 POP  SI          ; restitution
0476 POP  DI          ; des
0477 POP  ES          ; registres
0478 POP  AX          ;
0479 RET              ;
047A CLC              ; sans erreur
047B JMP  0475        ; -> sortie
```

tableau des noms d'instructions.

```
0486 6E 75 6C 40 69 6E    nul@in
048C 63 40 73 74 40 20    c@st@
0492 64 73 7A 40          dsz@
```

0496: Saut à la ligne (02D8, 0590).

```
0496:BOUCLE
0496 CMP  DI,[0232]   ; fdf?
049A JNB  04A7        ; oui: fini
049C ES:              ;
049D MOV  AX,[DI]     ; lire
049F CMP  AX,0A0D     ; test RC,NL
04A2 JZ   04A7        ; oui: retour
04A4 INC  DI          ; mise à jour
04A5 JMP  0496        ; -> relire
04A7 INC  DI          ; DI
04A8 INC  DI          ; mis à jour
04A9 INC  WO [04AE]   ; compteur
04AD RET              ;
```

compteur de lignes

```
04AE 01 00            ; au début: 1
```

04B8: Traitement des noms de registre (03E0,0400).

```
04B8 PUSH AX          ; AL sauvé
04B9:BOUCLE
04B9 CMP  DI,[0232]   ; fdf?
04BD JNB  04C9        ; oui: erreur
04BF ES:              ; non:
04C0 MOV  AL,[DI]     ; lecture
04C2 CMP  AL,20       ; blanc?
04C4 JNZ  04D1        ; non: tests
04C6 INC  DI          ; relire
04C7 JMP  04B9        ; -> BOUCLE
04C9 POP  AX          ; équilibre
04CA CALL 0FD0        ; (pb: fdf)
04CD STC              ;
04CE RET              ;
04D1 CMP  AL,60       ; minuscule?
04D3 JA   04D7        ; oui: suite
04D5 ADD  AL,20       ; non: corr.
04D7 CMP  AL,72       ; début: 'r'?
04D9 JZ   04E7        ; oui: suite
04DB POP  AX          ; équilibre
04DC CALL 0FD0        ; (faute)
04DF STC              ;
04E0 RET              ;
04E1 NOP              ;
04E2 CALL 0FD0        ; (faute)
04E5 STC              ;
04E6 RET              ;
04E7 ;                ; vérificat.
04E7 INC  DI          ; de syntaxe
04E8 CMP  DI,[0232]   ; fdf?
04EC JB   04F0        ; non: suite
04EE JMP  04C9        ; oui: erreur
04F0 ES:              ;
04F1 MOV  AL,[DI]     ; lecture
04F3 CMP  AL,30       ; correct?
04F5 JB   04DB        ; non: erreur
04F7 CMP  AL,39       ; chiffre?
04F9 JA   0503        ; non: test
04FB CALL 060E        ; nombre?
04FE POP  AX          ; équilibre
04FF JNB  0540        ; oui: tests
0501 JMP  04E2        ; non: erreur
0503 CMP  AL,60       ; minuscule?
0505 JNB  0509        ; oui: suite
0507 ADD  AL,20       ; non: corr.
0509 CMP  AL,7A       ; de a à z?
050B JA   04DB        ; non: erreur
050D MOV  BH,AL       ; 2ème lettre
050F INC  DI          ; voir suite
0510 CMP  DI,[0232]   ; fdf?
0514 JB   051E        ; non: tests
0516 POP  AX          ; AL retrouvé
0517 CMP  AL,03       ; instr. DSZ?
0519 JZ   04CA        ; oui: erreur
051B JMP  053C        ; non: reg.
051D NOP              ; connu?
051E ES:              ;
051F MOV  AL,[DI]     ; lecture
0521 MOV  BL,AL       ; garder lu
0523 POP  AX          ; AL retrouvé
0524 CMP  AL,03       ; instr. DSZ?
0526 JNZ  0532        ; non: tests
0528 CMP  BL,2C       ; virgule?
052B JZ   053B        ; oui: suite
052D CALL 0FD0        ; (pb: ,)
0530 STC              ;
0531 RET              ;
0532 MOV  AL,BL       ; cas NUL,INC
0534 CALL 05C8        ; VER. DELIM.
0537 JNB  053C        ; sans erreur
0539 JMP  04E2        ;
053B INC  DI          ; , passée
053C XOR  BL,BL       ;
053E MOV  AX,BX       ; AX = nom
0540 PUSH ES          ; préparat.
0541 PUSH DI          ; SCASW
0542 MOV  CX,[0588]   ; nbre reg.
0546 CMP  CX,+00      ; 1er reg?
0549 JZ   056D        ; oui: alloc.
054B PUSH DS          ; non:
054C POP  ES          ; recherche
054D MOV  DI,2000     ; dans liste
0550 CLD              ; depuis 2000
0551 REPNZ            ; (différent)
0552 SCASW            ;
0553 JNZ  055D        ; allouer
0555 DEC  DI          ;
0556 DEC  DI          ; adr. reg.
0557 MOV  AX,[DI+2000]; n° regist.
055B JMP  0583        ; -> retour
055D CMP  WO [0588],1000; place?
0563 JB   056D        ; oui: suite
0565 CALL 0FD0        ; (dépass.)
0568 POP  DI          ;
0569 POP  ES          ;
056A STC              ;
056B RET              ;
056C NOP              ;
056D MOV  BX,[0588]   ;
0571 SHL  BX,1        ;
0573 MOV  [BX+2000],AX; stocker nom
0577 MOV  AX,BX       ;
```

```
0579 SHR  AX,1          ; inscript.
057B MOV  [BX+4000],AX; répertoire
057F INC  WO [0588]     ; compteur
0583 CLC                ;
0584 POP  DI            ;
0585 POP  ES            ;
0586 RET                ;
```

nombre des registres enregistrés:

```
0588 00 00
```

0590: Contrôle de la ligne après l'instruction (02D8).

```
0590 JNB  0596          ; sans erreur
0592 CALL 0496          ; st.l.
0595 RET                ;
0596:BOUCLE             ;
0596 CMP  DI,[0232]     ; fdf?
059A JB   059D          ; non: lire
059C RET                ; (fdf)
059D ES:                ;
059E MOV  AL,[DI]       ; lecture
05A0 CMP  AL,20         ; blanc?
05A2 JNZ  05A7          ; non: tests
05A4 INC  DI            ; oui: relire
05A5 JMP  0596          ; -> BOUCLE
05A7 CMP  AL,3B         ; ;?
05A9 JNZ  05AD          ; non: tests
05AB JMP  0592          ; -> st.l.
05AD ES:                ;
05AE MOV  AX,[DI]       ; lire mot
05B0 CMP  AX,0A0D       ; RC,NL?
05B3 JNZ  05BC          ; non: erreur
05B5 INC  DI            ; DI
05B6 INC  DI            ; mis à jour
05B7 INC  WO [04AE]     ; compteur
05BB RET                ; retour
05BC CALL 0FD0          ;
05BF JMP  0592          ;
```

05C8: Contrôle de la délimitation d'un nom (04B8, 05E8).

```
05C8 CMP  AL,20         ; AL = ' '?
05CA JNZ  05CE          ; non : suite
05CC CLC                ; si : bien
05CD RET                ; retour
05CE CMP  AL,3B         ; AL = ';'?
05D0 JNZ  05D4          ; non : suite
05D2 CLC                ; si : bien
05D3 RET                ; retour
05D4 CMP  AL,0D         ; AL = RC?
05D6 JNZ  05E0          ; non : mal
05D8 ES:                ; si :
05D9 MOV  AX,[DI]       ; lire AX
05DB CMP  AX,0A0D       ; vrai RC?
05DE JZ   05D2          ; oui : bien
05E0 STC                ; non : mal
05E1 RET                ;
```

05E8: Traitement d'une référence (03C0,0400).

```
05E8:BOUCLE1            ; de lecture
05E8 CMP  DI,[0232]     ; fdf?
05EC JNB  05F8          ; oui : err.
05EE ES:                ; non:
05EF MOV  AL,[DI]       ; lire en AL
05F1 CMP  AL,20         ; ' '?
05F3 JNZ  05FE          ; non : tests
05F5 INC  DI            ; oui :
05F6 JMP  05E8          ; -> BOUCLE1
05F8 CALL 0FD0          ;
05FB STC                ;
05FC RET                ;
05FD NOP                ; 1er test :
05FE CMP  AL,2F         ; AL >= 30?
0600 JA   0608          ; oui : tests
0602 CALL 0FD0          ; non :
0605 STC                ; erreur
0606 RET                ;
0607 NOP                ; 2ème test:
0608 CMP  AL,3A         ; AL < 3A?
060A JB   060E          ; oui : tests
060C JMP  0602          ; non : err.
060E;                   ; vérifier
060E MOV  CX,0003       ; si nombre
0611 PUSH BP            ;
0612 SUB  AL,30         ; AL numérisé
0614 XOR  AH,AH         ; AX = S
0616 XOR  BH,BH         ; (Hörner)
0618 MOV  BL,0A         ; BX = dix
061A:BOUCLE2            ; lecture
061A INC  DI            ;
061B CMP  DI,[0232]     ; fdf?
061F JNB  0650          ; oui : fin
0621 PUSH AX            ; sauver S
0622 ES:                ;
0623 MOV  AL,[DI]       ; lire en AL
0625 CMP  AL,30         ; chiffre?
0627 JB   0654          ; non : test
0629 CMP  AL,39         ; chiffre?
062B JA   0654          ; non : test
062D SUB  AL,30         ; oui:
062F MOV  BP,AX         ; sauver AL
0631 POP  AX            ; rappel S
0632 PUSH DX            ; DX sauvé
0633 MUL  BX            ;
0635 POP  DX            ; DX retrouvé
0636 ADD  AX,BP         ; S:=S*BX+BP
0638 LOOP 061A          ; -> BOUCLE2
063A MOV  BP,AX         ; BP := S
063C INC  DI            ;
063D CMP  DI,[0232]     ; fdf?
0641 JNB  064E          ; oui : fin
0643 ES:                ; non :
0644 MOV  AL,[DI]       ; lire en AL
0646 CALL 05C8          ; délim.?
0649 JNB  064E          ; oui : fin
064B POP  BP            ; non :
064C JMP  0602          ; erreur
064E MOV  AX,BP         ; S retrouvé
0650 POP  BP            ;
0651 CLC                ; calcul
0652 RET                ; terminé
0653 NOP                ;
0654 POP  BP            ;
0655 JMP  0646          ;
```

0660: Adresse, dans le code, de l'instruction lue. (0370).

```
0660 PUSHF              ; sauver (F)
0661 PUSH BX            ; id. BX
0662 MOV  BX,DX         ; conversion
0664 SHL  BX,1          ; n° instr.
0666 CMP  BX,2000       ; en adresse
066A JB   0672          ; daccord
```

```
066C CALL 0FD0          ; trop grand
066F JMP  0676          ; erreur
0671 NOP                ;
0672 MOV  [BX+6000],SI; adresse en
0676 POP  BX            ; table
0677 POPF               ; registres
0678 RET                ; retrouvés
```

06A0: LECTURE DES ARGUMENTS (0100).

```
06A0 MOV  CX,8000       ; remplir la
06A3 XOR  AX,AX         ; zone par
06A5 XOR  DI,DI         ; des zéros
06A7 CLD                ; (8000 mots)
06A8 REPZ               ;
06A9 STOSW              ;
06AA CALL 0700          ; effacer
06AD MOV  DX,0505       ; l'écran,
06B0 MOV  BX,0000       ; position
06B3 MOV  AH,02         ; du curseur
06B5 INT  10            ;
06B7 MOV  CX,[0588]     ; nbre reg.
06BB PUSH CX            ; soit S
06BC MOV  DX,0A80       ; 'il y a '
06BF MOV  AH,09         ;
06C1 INT  21            ;
06C3 CALL 07A0          ; afficher S
06C6 MOV  DX,0A8A       ; 'registres'
06C9 MOV  AH,09         ;
06CB INT  21            ;
06CD POP  CX            ; S retrouvé
06CE MOV  DX,0705       ; (DL,DH) :
06D1 XOR  SI,SI         ; curseur
06D3:BOUCLE             ; d'affichage
06D3 PUSH DX            ; curseur
06D4 MOV  BX,0000       ; sauvé
06D7 MOV  AH,02         ; positionné
06D9 INT  10            ;
06DB MOV  DX,0A98       ; 'entrer la
06DE MOV  AH,09         ; valeur du
06E0 INT  21            ; registre'
06E2 CALL 07F0          ; nom du reg.
06E5 CALL 0820          ; lire valeur
06E8 JB   06F4          ; dépassement
06EA POP  DX            ; rappel cur-
06EB INC  DH            ; -seur, re-
06ED INC  DH            ; position
06EF INC  SI            ; n° registre
06F0 LOOP 06D3          ; -> BOUCLE
06F2 CLC                ; R.A.S.
06F3 RET                ; retour
06F4 POP  DX            ;
06F5 CALL 0FD0          ; erreur :
06F8 STC                ; dépassement
06F9 RET                ; de capacité
```

0700: Effacer l'écran (06A0, 0DE0).

```
0700 PUSH AX            ; sauver les
0701 PUSH CX            ; registres
0702 PUSH DI            ;
0703 PUSH SI            ;
0704 PUSH ES            ;
0705 PUSHF              ; sauver (F)
0706 CMP  BY [0764],00; mode graph.
070B JNZ  0742          ; choisi
070D MOV  AH,0F         ; non : qui
070F INT  10            ; en service?
0711 MOV  [0765],AL     ; garder
0714 INC  BY [0764]     ; signal mode
0718 PUSH DS            ; choisi
0719 PUSH BX            ;
071A MOV  BX,0040       ; lire type
071D PUSH BX            ; carte écran
071E POP  DS            ; en
071F MOV  BX,0063       ; 0040:0063
0722 MOV  AL,[BX]       ;
0724 POP  BX            ;
0725 POP  DS            ;
0726 CMP  AL,B4         ; monochrome?
0728 JZ   072C          ; oui : suite
072A JMP  0738          ; non : voir
072C MOV  AX,0007       ; fixer mode
072F INT  10            ; monochrome
0731 MOV  BY [0766],07; sauver mode
0736 JMP  0742          ; adresse RAM
0738 MOV  AX,0002       ; fixer mode
073B INT  10            ; écran n°2
073D MOV  BY [0766],02; sauver mode
0742 CMP  BY [0766],07; monochrome?
0747 JZ   074E          ; plus loin
0749 MOV  AX,B800       ; RAM de n°2
074C JMP  0751          ;
074E MOV  AX,B000       ; adresse RAM
0751 PUSH AX            ; monochrome
0752 POP  ES            ;
0753 XOR  DI,DI         ; effacer :
0755 MOV  AX,0720       ; on écrit
0758 MOV  CX,07D0       ; une page de
075B REPZ               ; 'blancs'
075C STOSW              ; d'où écran
075D POPF               ; noir
075E POP  ES            ; registres
075F POP  SI            ; restitués
0760 POP  DI            ;
0761 POP  CX            ;
0762 POP  AX            ;
0763 RET                ;
```

variables du programme 07A0

```
0764 00 00 00 00
```

07A0: Conversion hexa -> décimal (06A0,07F0,0DE0).

tableau des chiffres décimaux

```
0770 30 31 32 33 34 35    012345
0776 36 37 38 39          6789
```

tampon d'écriture des chiffres

```
077A 00 00 00 00 00 00
```

sous-programme 0780:

```
0780:BOUCLE             ; selon déc.
0780 MOV  AX,[077A]     ; programme
0783 DIV  BX            ; analogu à
0785 MOV  [077A],AX     ; celui des
0788 MOV  AX,[077C]     ; pages 75-76
078B DIV  BX            ; mais
078D MOV  [077C],AX     ; prévu pour
0790 MOV  [BP+SI],DL    ; un double
0792 XOR  DX,DX         ; mot dont
0794 CMP  AX,0000       ; la partie
0797 JZ   079C          ; forte est
0799 INC  SI            ; mise à 0000
079A JMP  0780          ; -> BOUCLE
079C RET                ;
```

programme principal: 07A0

```
07A0 MOV  AX,0000      ; mot fort
07A3 MOV  [077A],AX    ; mis à 0000
07A6 MOV  AX,CX        ;
07A8 MOV  [077C],AX    ;
07AB MOV  SI,0000      ;
07AE MOV  BP,07E4      ;
07B1 MOV  BX,000A      ;
07B4 XOR  DX,DX        ;
07B6 CALL 0780         ;
07B9 MOV  BX,0770      ;
07BC MOV  CX,SI        ;
07BE CMP  CX,+00       ;
07C1 JZ   07D5         ;
07C4:BOUCLE            ; d'affichage
07C4 XOR  AX,AX        ;
07C6 XOR  DX,DX        ;
07C8 MOV  AL,[BP+SI]   ; position
07CA MOV  DI,AX        ;
07CC MOV  DL,[BX+DI]   ; chiffre
07CE MOV  AH,02        ; affichage
07D0 INT  21           ;
07D2 DEC  SI           ;
07D3 LOOP 07C4         ; -> BOUCLE
07D5 XOR  AX,AX        ;
07D7 MOV  AL,[BP+00]   ;
07DA MOV  DI,AX        ;
07DC MOV  DL,[BX+DI]   ;
07DE MOV  AH,02        ;
07E0 INT  21           ;
07E2 RET               ;
```

tampon d'écriture du résultat

```
07E4 00 00 00 00
```

07F0: Afficher le nom du registre. (06A0,0DE0).

```
07F0 PUSH SI           ; SI : n°reg.
07F1 PUSH CX           ;
07F2 MOV  DL,52        ; DL = 'R'
07F4 MOV  AH,02        ;
07F6 INT  21           ; puis SI
07F8 SHL  SI,1         ; converti en
07FA MOV  AL,[SI+2001] ; adresse
07FE CMP  AL,60        ; nom?
0800 JA   0804         ; oui : suite
0802 JMP  080E         ; non, nombre
0804 SUB  AL,20        ; conversion
0806 MOV  DL,AL        ; majuscule
0808 MOV  AH,02        ; affichage
080A INT  21           ; (nom : RX)
080C JMP  0815         ; -> finir
080E MOV  CX,[SI+2000] ; cas nombre
0812 CALL 07A0         ; affichage
0815 MOV  DX,0AB8      ; st lgn
0818 MOV  AH,09        ;
081A INT  21           ;
081C POP  CX           ;
081D POP  SI           ;
081E RET               ;
```

0820: Lecture du nombre entré à l'écran et son enregistrement (06A0).

```
0820 PUSH SI           ;
0821 PUSH CX           ;
0822 MOV  BY [086C],00 ; 3 essais
0827:BOUCLE1           ;
0827 CALL 0870         ; lecture du
082A JNB  0840         ; nombre
082C INC  BY [086C]    ; raté : on
0830 CMP  BY [086C],03 ; recommence?
0835 JA   0839         ; trop : non
0837 JMP  0827         ; -> BOUCLE1
0839 POP  CX           ;
083A POP  SI           ;
083B CALL 0FD0         ; erreur
083E STC               ;
083F RET               ;
0840 XOR  CX,CX        ; nombre lu
0842 MOV  CL,[0A31]    ; fixer sa
0846 PUSH CX           ; longueur
0847 SHR  CX,1         ; en mots
0849 POP  BX           ;
084A JNB  0850         ; pair : OK
084C CALL 0AC8         ; ajuster
084F INC  CX           ;
0850 CALL 0898         ; init. reg.
0853 NOP               ; (CX prêt)
0854:BOUCLE2           ; Hörner
0854 CALL 0908         ; R:=R*064
0857 JB   0866         ; erreur: fin
0859 CALL 0970         ; R:=R+déc.
085C JB   0866         ; erreur: fin
085E LOOP 0854         ; -> BOUCLE2
0860 CALL 09F0         ; réinitial.
0863 JB   0866         ;
0865 CLC               ; sans erreur
0866 POP  CX           ;
0867 POP  SI           ;
0868 RET               ;
```

variable

```
086C 00 00
```

0870: Lecture à l'écran et contrôle (0820).

```
0870 MOV  DX,0A30      ; tampon de
0873 MOV  AH,0A        ; lecture par
0875 INT  21           ; DOS
0877 PUSH CX           ;
0878 PUSH DI           ;
0879 XOR  CX,CX        ; nombre
087B MOV  CL,[0A31]    ; d'octets
087F MOV  DI,0A32      ;
0882:BOUCLE            ; de contrôle
0882 MOV  AL,[DI]      ; il faut
0884 CMP  AL,3A        ;
0886 JB   088C         ; 30<=AL<=39
0888 POP  DI           ;
0889 POP  CX           ;
088A STC               ; sortie en
088B RET               ; erreur
088C CMP  AL,30        ;
088E JB   0888         ;
0890 INC  DI           ;
0891 LOOP 0882         ; -> BOUCLE
0893 CLC               ; sortie sans
0894 POP  DI           ; erreur
0895 POP  CX           ;
0896 RET               ;
```

0898: Initialisation du registre (0820).

```
0898 CMP  SI,+00       ; R = 1er?
089B JA   08BD         ; non : suite
089D MOV  DI,1C80      ; initialiser
```

```
08A0 MOV  SI,01C8      ; gestion :
08A3 MOV  [0A20],DI    ; adresse S
08A7 MOV  [0A22],SI    ; n°S sauvé
08AB MOV  WO [0A28],0000; adr. R en
08B1 MOV  [4000],SI    ; répertoire
08B5 MOV  WO [4002],0000; sans S
08BB JMP  08D8         ;
08BD MOV  DI,[0A20]    ; même chose
08C1 MOV  SI,[0A22]    ; pour R
08C5 ADD  WO [0A28],+04; suivants
08CA MOV  BX,[0A28]    ; BX : adr.
08CE MOV  [BX+4000],SI; R en réper-
08D2 MOV  WO [BX+4002],0000; toire
08D8 MOV  [0A2A],DI    ;
08DC MOV  [0A2C],SI    ;
08E0 PUSH SI           ;
08E1 SUB  SI,01C8      ; n°TAS
08E5 SHL  SI,1         ; converti en
08E7 MOV  [0A24],SI    ; adresse
08EB MOV  BX,SI        ;
08ED POP  SI           ;
08EE MOV  WO [0A26],0001; |R| = 1
08F4 ES:               ; marque de
08F5 MOV  [BX],0100    ; fin de R
08F9 PUSH DI           ; S
08FA PUSH CX           ; initialisé
08FB XOR  AX,AX        ; à 00
08FD MOV  CX,0008      ; (huit mots
0900 REPZ              ; à 0000)
0901 STOSW             ;
0902 POP  CX           ;
0903 POP  DI           ; puis BP:
0904 MOV  BP,0A32      ; ptr lect.
0907 RET               ;
```

0908: Multiplication par cent (064) (0820).

```
0908 PUSH DI           ;
0909 PUSH SI           ;
090A PUSH CX           ;
090B PUSH DX           ;
090C MOV  BX,0064      ;
090F MOV  SI,[0A2C]    ; n°S
0913 MOV  DI,[0A2A]    ; adresse S
0917 MOV  CX,[0A26]    ; |R| (mots).
091B XOR  DX,DX        ; retenue = 0
091D:BOUCLE            ; sur |R|
091D PUSH CX           ; compteur
091E PUSH DX           ; ret. MUL
091F ES:               ;
0920 MOV  AX,[DI]      ; lire décim.
0922 MUL  BX           ; AL*064
0924 POP  CX           ; ex-ret. MUL
0925 ADD  AX,CX        ; ajoutée
0927 JNB  092A         ; sans ret.
0929 INC  DX           ; si : +1
092A ES:               ; écrire
092B MOV  [DI],AX      ; nouv. déc.
092D INC  DI           ; à la
092E INC  DI           ; suivante
092F POP  CX           ; compteur
0930 LOOP 091D         ; -> BOUCLE
0932 CMP  DX,+00       ; ret. > 0?
0935 JZ   0948         ; non : fini
0937 MOV  AX,[0A20]    ; si : 'nou-
093A ADD  AX,0010      ; veau S'
093D CMP  AX,DI        ; sortie?
093F JBE  094B         ; oui : alloc
0941 ES:               ; non : écr.
0942 MOV  [DI],DX      ; retenue
0944 INC  WO [0A26]    ; |R| : +1
0948 CLC               ; R.A.S.
0949 JMP  095E         ; -> fin
094B CMP  SI,0FFF      ; hors zone?
094F JA   095A         ; oui: erreur
0951 MOV  [0A20],DI    ; non : noter
0955 CALL 09CC         ; actual. TAS
0958 JMP  0941         ; -> écrire
095A CALL 0FD0         ; erreur :
095D STC               ; dépassement
095E POP  DX           ;
095F POP  CX           ;
0960 POP  SI           ;
0961 POP  DI           ;
0962 RET               ;
```

0970: AX, décimale courante, ajoutée à la valeur du registre (0820).

```
0970 MOV  BL,0A        ; BL = dix
0972 MOV  AL,[BP+00]   ; AL = décim.
0975 SUB  AL,30        ; numériser
0977 MUL  BL           ; cv hexa
0979 XOR  BX,BX        ;
097B INC  BP           ;
097C MOV  BL,[BP+00]   ;
097F SUB  BL,30        ;
0982 ADD  AX,BX        ;
0984 INC  BP           ; ci-après :
0985 PUSH DI           ; r =
0986 PUSH SI           ; retenue
0987 PUSH CX           ; d'addition
0988 PUSH DX           ;
0989 MOV  CX,[0A26]    ;
098D ES:               ; AX ajouté
098E ADD  [DI],AX      ; au ler ch.
0990:BOUCLE            ; si retenue
0990 JNB  09C5         ; r=0 : fini
0992 INC  DI           ;
0993 INC  DI           ;
0994 ES:               ; on ajoute r
0995 ADD  WO [DI],+01  ;
0998 LOOP 0990         ; -> BOUCLE
099A JNB  09C5         ; r=0: fini
099C MOV  AX,[0A20]    ; r=1 :
099F ADD  AX,0010      ; sort-on de
09A2 CMP  AX,DI        ; S?
09A4 JBE  09B2         ; oui: alloc.
09A6 ES:               ; non: écrire
09A7 MOV  WO [DI],0001; r et incr.
09AB INC  WO [0A26]    ; |R|
09AF CLC               ; R.A.S.
09B0 JMP  09C5         ; fini
09B2 CMP  SI,0FFF      ; hors zone?
09B6 JNB  09C1         ; oui: erreur
09B8 MOV  [0A20],DI    ; non:
09BC CALL 09CC         ; actual. TAS
09BF JMP  09A6         ; écrire
09C1 CALL 0FD0         ; dépassement
09C4 STC               ; de capacité
```

```
09C5 POP  DX          ;
09C6 POP  CX          ;
09C7 POP  SI          ;
09C8 POP  DI          ;
09C9 RET              ;
```

09CC: Marquages en TAS (0908, 0970).

```
09CC MOV  BX,[0A24]   ; adr. en TAS
09D0 ES:              ; écrire
09D1 MOV  [BX],SI     ; n°S
09D3 ES:              ; y ajouter 1
09D4 INC  WO [BX]     ; écrire
09D6 INC  BX          ; BX entrée
09D7 INC  BX          ; suivante
09D8 ES:              ; marquée
09D9 MOV  WO [BX],0100; fin de reg.
09DD INC  SI          ; n°S actuel
09DE MOV  [0A22],SI   ; noté
09E2 MOV  [0A24],BX   ; et adr. TAS
09E6 RET              ;
```

09F0: REinitialisation des paramètres (0820).

```
09F0 MOV  SI,[0A22]   ; n°S
09F4 INC  SI          ; incrémenté
09F5 CMP  SI,0FFF     ; hors zone?
09F9 JB   0A00        ; non : suite
09FB CALL 0FD0        ; si: erreur
09FE STC              ; zone pleine
09FF RET              ;
0A00 ADD  WO [0A20],+10; nouv. adr.
0A05 MOV  [0A22],SI   ; nouv. n°
0A09 MOV  BX,[0A28]   ; adr. rép.
0A0D MOV  AX,[0A26]   ; |R| lu puis
0A10 MOV  [BX+4002],AX; noté rép.
0A14 CLC              ;
0A15 RET              ;
```

variables de contrôle de l'initialisation des registres

```
0A20 8F 8F 8F 8F 8F 8F 8F 8F
0A28 8F 8F 8F 8F 8F 8F 8F 8F
```

zone de lecture des arguments (on admet 040 chiffres au plus)

```
0A30 40 00
0A32 à 0A80: 4E octets à 00
```

messages de l'initialisation des registres

```
0A80 0D 0A 'il y a ' 24 ' regis'
0A90 'tres' 0D 0A 24
0A98 0D 0A 'entrer la valeur du'
0AAD ' registre ' 24
0AB8 ':' 0D 0A '   > ' 24
```

0AC8: Décalage et correction de l'entrée (0820).

```
0AC8 PUSH CX          ;
0AC9 MOV  CX,BX       ;
0ACB MOV  AH,30       ;
0ACD MOV  BX,0A32     ;
0AD0:BOUCLE           ; de décalage
0AD0 MOV  AL,[BX]     ; glissement
0AD2 MOV  [BX],AH     ; via
0AD4 MOV  AH,AL       ; AH,AL
0AD6 INC  BX          ;
0AD7 LOOP 0AD0        ; -> BOUCLE
0AD9 MOV  [BX],AH     ;
0ADB POP  CX          ;
0ADC RET              ;
```

0B00: EXECUTION DU PROGRAMME MINSKY (0100).

```
0B00 MOV  SI,0000     ;
0B03:BOUCLE           ; d'exécution
0B03 CMP  SI,8000     ; fini?
0B07 JNB  0B5E        ; oui : fin
0B09 MOV  AL,[SI+8000]; lire type
0B0D CMP  AL,00       ; instr. NUL?
0B0F JNZ  0B1D        ; non: tests
0B11 MOV  AX,[SI+8001]; si : réf. R
0B15 CALL 0C90        ; R := 0
0B18 ADD  SI,+03      ; inst. suiv.
0B1B JMP  0B03        ; -> BOUCLE
0B1D CMP  AL,01       ; instr. INC?
0B1F JNZ  0B35        ; non : tests
0B21 MOV  AX,[SI+8001]; si : réf. R
0B25 CALL 0BC8        ; R := R+1
0B28 JNB  0B30        ; erreur?
0B2A CALL 0FD0        ; oui :
0B2D STC              ; dépassement
0B2E RET              ;
0B2F NOP              ; non :
0B30 ADD  SI,+03      ; inst. suiv.
0B33 JMP  0B03        ; -> BOUCLE
0B35 CMP  AL,02       ; instr. ST?
0B37 JNZ  0B3F        ; non : tests
0B39 MOV  SI,[SI+8001]; exéc. saut
0B3D JMP  0B03        ; -> BOUCLE
0B3F CMP  AL,03       ; instr. DSZ?
0B41 JZ   0B48        ; oui : suite
0B43 CALL 0FD0        ; erreur du
0B46 STC              ; simulateur
0B47 RET              ;
0B48 MOV  AX,[SI+8001]; réf. R
0B4C CALL 0DB0        ; R = 0?
0B4F JB   0B5F        ; erreur: fin
0B51 JNZ  0B59        ; R<>0: suite
0B53 MOV  SI,[SI+8003]; R=0: saut
0B57 JMP  0B03        ; -> BOUCLE
0B59 ADD  SI,+05      ; inst. suiv.
0B5C JMP  0B03        ; -> BOUCLE
0B5E CLC              ; R.A.S.
0B5F RET              ; fin exéc.
```

0B70: Repérage du registre dans le répertoire (0BC8,0C90,0DB0).

```
0B70 SHL  AX,1        ; AX : n°reg.
0B72 SHL  AX,1        ; AX := AX*4
0B74 MOV  BX,AX       ; adr. rép.
0B76 ADD  BX,4000     ; adaptée
0B7A MOV  AX,[BX]     ; entrée :
0B7C MOV  SI,AX       ; 1er S
0B7E MOV  CX,0004     ; n°S
0B81 SHL  AX,CL       ; converti en
0B83 MOV  DI,AX       ; adresse S
0B85 MOV  [0A20],AX   ; noté
0B88 MOV  AX,[BX+02]  ; |R| relevé
0B8B MOV  [0A26],AX   ; et noté
0B8E XOR  CX,CX       ; [R] := 0
0B90 MOV  AX,SI       ; AX: n°S
0B92:BOUCLE           ; lecture TAS
0B92 CMP  AX,1C70     ; hors TAS?
0B95 JB   0B9C        ; non : suite
```

```
0B97 CALL 0FD0          ; erreur
0B9A STC                ; logiciel
0B9B RET                ;
0B9C INC  CX            ; [R] : +1
0B9D SUB  AX,01C8       ; n°S
0BA0 SHL  AX,1          ; converti en
0BA2 MOV  BP,AX         ; adr. TAS
0BA4 ES:                ; lecture
0BA5 MOV  AX,[BP+00]    ; entrée TAS
0BA8 CMP  AX,0100       ; fin de R?
0BAB JBE  0BB0          ; peut-être
0BAD JMP  0B92          ; -> BOUCLE
0BAF NOP                ;
0BB0 CMP  AX,0100       ; vrai?
0BB3 JZ   0BBA          ; oui : fin
0BB5 CALL 0FD0          ; erreur du
0BB8 STC                ; simulateur
0BB9 RET                ;
0BBA CLC                ; R.A.S.
0BBB RET                ;
```

0BC8: EXECUTION de INC. (B00)

```
0BC8 PUSH SI            ; sauver SI
0BC9 CALL 0B70          ; CX := [R]
0BCC XOR  DX,DX         ; vérif. |R|
0BCE:BOUCLE             ; incrémenter
0BCE CALL 0C10          ; S := S+1
0BD1 ES:                ;
0BD2 MOV  AX,[BP+00]    ; entrée TAS
0BD5 JNB  0BFC          ; r=0 : fin
0BD7 CALL 0C28          ; sect. suiv.
0BDA LOOP 0BCE          ; -> BOUCLE
0BDC PUSHF              ; vérifier
0BDD CMP  AX,0100       ; S dernier?
0BE0 JZ   0BE8          ; oui : suite
0BE2 POPF               ; non:
0BE3 CALL 0FD0          ; erreur du
0BE6 STC                ; simulateur
0BE7 RET                ;
0BE8 POPF               ;
0BE9 JNB  0BFC          ; r=0: fin
0BEB CALL 0C48          ; r=1: alloc.
0BEE JNB  0BF6          ; pas de pb.
0BF0 CALL 0FD0          ; si: erreur
0BF3 STC                ; dépassement
0BF4 RET                ;
0BF5 NOP                ;
0BF6 ES:                ; écriture
0BF7 MOV  WO [DI],0001; retenue (r)
0BFB INC  DX            ; actual. DX
0BFC CMP  DX,[0A26]     ; |R| changé?
0C00 JBE  0C05          ; non : fin
0C02 MOV  [BX+02],DX    ; oui : corr.
0C05 POP  SI            ;
0C06 CLC                ; R.A.S.
0C07 RET                ;
```

0C10: Incrémenter le secteur courant. (BC8)

```
0C10 PUSH CX            ;
0C11 MOV  CX,0008       ;
0C14:BOUCLE             ; huit mots
0C14 INC  DX            ;
0C15 ES:                ;
0C16 MOV  AX,[DI]       ;
0C18 ADD  AX,0001       ; incrémenter
0C1B ES:                ;
0C1C MOV  [DI],AX       ; noter
0C1E JNB  0C24          ; r=0: fin
0C20 INC  DI            ;
0C21 INC  DI            ;
0C22 LOOP 0C14          ; -> BOUCLE
0C24 POP  CX            ;
0C25 RET                ;
```

0C28: Adresse du secteur suivant (0BC8).

```
0C28 PUSHF              ; AX: n°S
0C29 CMP  AX,0100       ; S dernier?
0C2C JZ   0C42          ; oui : fin
0C2E PUSH CX            ; non :
0C2F PUSH AX            ; AX,CX pile
0C30 MOV  CX,0004       ; AX converti
0C33 SHL  AX,CL         ; en
0C35 MOV  DI,AX         ; adresse
0C37 POP  AX            ; puis, en
0C38 PUSH AX            ; adresse TAS
0C39 SUB  AX,01C8       ; du même
0C3C SHL  AX,1          ; secteur
0C3E MOV  BP,AX         ; BP: en TAS
0C40 POP  AX            ; registres
0C41 POP  CX            ; restitués
0C42 POPF               ; (F) rendu
0C43 RET                ;
```

0C48: Allouer un nouveau secteur (0BC8).

```
0C48 CLD                ;
0C49 MOV  DI,BP         ; adresse TAS
0C4B MOV  AX,DI         ;
0C4D SHR  AX,1          ;
0C4F ADD  AX,01C8       ;
0C52 MOV  CX,1000       ; on cherche
0C55 SUB  CX,AX         ; d'abord
0C57 PUSH AX            ; plus haut
0C58 XOR  AX,AX         ; que S
0C5A REPNZ              ; actuel (BP)
0C5B SCASW              ;
0C5C POP  AX            ;
0C5D JZ   0C6D          ; trouvé
0C5F XOR  DI,DI         ; non :
0C61 MOV  CX,AX         ; on cherche
0C63 SUB  CX,01C8       ; depuis 0000
0C67 REPNZ              ; à S actuel.
0C68 SCASW              ;
0C69 JZ   0C6D          ; trouvé
0C6B STC                ; non: signe
0C6C RET                ;
0C6D DEC  DI            ;
0C6E DEC  DI            ; adresse TAS
0C6F PUSH DI            ; sauvée
0C70 MOV  AX,DI         ; convertie
0C72 SHR  AX,1          ; en n° S
0C74 ADD  AX,01C8       ;
0C77 MOV  SI,AX         ; dans SI
0C79 MOV  CX,0004       ; AX converti
0C7C SHL  AX,CL         ; en adresse
0C7E MOV  DI,AX         ; dans DI
0C80 ES:                ; marquage
0C81 MOV  [BP+00],SI    ; S suivant
0C84 POP  BP            ; puis S al-
0C85 ES:                ; loué comme
0C86 MOV  WO [BP+00],0100; dernier
0C8B CLC                ;
0C8C RET                ;
```

0C90: EXECUTION de NUL (0B00).

```
0C90 PUSH SI           ; registre R
0C91 CALL 0B70         ; CX := [R]
0C94 CALL 0CF8         ; 1erS := 0
0C97 MOV  AX,SI        ; AX := n°S
0C99 PUSH AX           ; converti
0C9A SUB  AX,01C8      ; en adresse
0C9D SHL  AX,1         ; TAS de S
0C9F MOV  BP,AX        ; mise en BP
0CA1 POP  AX           ;
0CA2 ES:               ; marquage de
0CA3 MOV  AX,[BP+00]   ; S comme
0CA6 ES:               ; dernier :
0CA7 MOV  WO [BP+00],0100;
0CAC MOV  WO [BX+02],0001; |R|=1
0CB1 NOP               ; noté en rép
0CB2:BOUCLE            ; libération
0CB2 LOOP 0CB6         ; (boucle
0CB4 JMP  0CE3         ; WHILE CX>1)
0CB6 CMP  AX,0100      ; S dernier?
0CB9 JNZ  0CC2         ; non: suite
0CBB CALL 0FD0         ; si: erreur
0CBE STC               ; simulateur
0CBF POP  SI           ;
0CC0 RET               ;
0CC1 NOP               ;
0CC2 PUSH CX           ; compteur
0CC3 PUSH AX           ; AX = n°S
0CC4 MOV  CX,0004      ; converti
0CC7 SHL  AX,CL        ; en adresse
0CC9 MOV  DI,AX        ;
0CCB CALL 0CF8         ; mettre à 00
0CCE POP  AX           ; puis
0CCF SUB  AX,01C8      ; désallouer:
0CD2 SHL  AX,1         ;
0CD4 MOV  BP,AX        ; BP = adr.S
0CD6 ES:               ; en TAS
0CD7 MOV  AX,[BP+00]   ; AX = n°S
0CDA ES:               ; suivant
0CDB MOV  WO [BP+00],0000; libérer
0CE0 POP  CX           ; compteur
0CE1 JMP  0CB2         ; -> BOUCLE
0CE3 CMP  AX,0100      ; S dernier?
0CE6 JZ   0CED         ; oui : suite
0CE8 CALL 0FD0         ; erreur du
0CEB STC               ; simulateur
0CEC RET               ;
0CED POP  SI           ;
0CEE CLC               ; R.A.S.
0CEF RET               ;
```

0CF8: Mettre un secteur à zéro (0C90).

```
0CF8 PUSH CX           ; sauver
0CF9 PUSH DI           ; CX,DX
0CFA MOV  CX,0008      ; huit mots
0CFD XOR  AX,AX        ; à 0000
0CFF CLD               ;
0D00 REPZ              ;
0D01 STOSW             ;
0D02 POP  DI           ;
0D03 POP  CX           ;
0D04 RET               ;
```

0D08: Diminuer un registre de 1 (0DB0)

```
0D08 XOR  DX,DX        ; DX pour |R|
0D0A:BOUCLE            ; décrémenter
0D0A CALL 0D30         ; S := S-1
0D0D ES:               ; lire
0D0E MOV  AX,[BP+00]   ; entrée TAS
0D11 JNB  0D18         ; r=0: sortie
0D13 CALL 0C28         ; S suivant
0D16 LOOP 0D0A         ; -> BOUCLE
0D18 CALL 0D50         ; désallouer
0D1B RET               ; S dernier
```

0D30: Décrementer le secteur courant (0D08).

```
0D30 PUSH CX           ; compeur
0D31 MOV  CX,0008      ;
0D34:BOUCLE            ; huit mots
0D34 INC  DX           ; vérif. |R|
0D35 ES:               ;
0D36 MOV  AX,[DI]      ; lire mot
0D38 SUB  AX,0001      ; -1
0D3B ES:               ; écrire
0D3C MOV  [DI],AX      ; résultat
0D3E JNB  0D46         ; r=0: fini
0D40 INC  DI           ; r=1: mot
0D41 INC  DI           ; suivant
0D42 LOOP 0D34         ; -> BOUCLE
0D44 POP  CX           ; sortie CY
0D45 RET               ;
0D46 POP  CX           ;
0D47 CLC               ; R.A.S.
0D48 RET               ;
```

0D50: Désallouer le dernier secteur si nécessaire (0D08).

```
0D50 CMP  DX,[0A26]    ; déc. tête?
0D54 JB   0DA6         ; non: fini
0D56 CMP  AX,0100      ; si: vérif.
0D59 JZ   0D60         ; oui : suite
0D5B CALL 0FD0         ; non :
0D5E STC               ; erreur
0D5F RET               ; simulateur
0D60 ES:               ;
0D61 MOV  AX,[DI]      ; lire AX
0D63 CMP  AX,0000      ; annulé
0D66 JNZ  0DA6         ; non : fini
0D68 CMP  DX,+01       ; si : fini
0D6B JZ   0DA6         ; si 1 mot
0D6D DEC  DX           ; sinon
0D6E MOV  [BX+02],DX   ; |R|:=|R|-1
0D71 INC  DX           ; |R| initial
0D72:BOUCLE1           ; réduction
0D72 CMP  DX,+08       ; de DX
0D75 JBE  0D7D         ; modulo 8
0D77 SUB  DX,+08       ; dans [1,8]
0D7A JMP  0D72         ; -> BOUCLE1
0D7C NOP               ;
0D7D CMP  DX,+01       ; DX > 1?
0D80 JNZ  0DA6         ; oui : fini
0D82 MOV  AX,[BX]      ; non: S 1er.
0D84:BOUCLE2           ;
0D84 SUB  AX,01C8      ; convertir
0D87 SHL  AX,1         ; adr. TAS
0D89 MOV  BP,AX        ;
0D8B ES:               ; lire entrée
0D8C MOV  AX,[BP+00]   ; TAS
0D8F CMP  AX,0100      ; dernier?
0D92 JZ   0D98         ; oui: sortie
0D94 MOV  DX,BP        ; sauver adr.
0D96 JMP  0D84         ; -> BOUCLE2
```

```
0D98 ES:                ; dernier
0D99 MOV  WO [BP+00],0000; libéré
0D9E MOV  BP,DX         ; avant lui :
0DA0 ES:                ; marqué
0DA1 MOV  WO [BP+00],0100; dernier
0DA6 CLC                ;
0DA7 RET                ;
```

0DB0: EXECUTION de DSZ (0B00).

```
0DB0 PUSH SI            ;
0DB1 CALL 0B70          ; CX := [R]
0DB4 CMP  WO [0A26],+01; |R| = 1?
0DB9 JNZ  0DC4          ; non : décr.
0DBB ES:                ; oui :
0DBC CMP  WO [DI],+00 ; R = 0?
0DBF JNZ  0DC4          ; non : décr.
0DC1 POP  SI            ;
0DC2 RET                ; (ici NC)
0DC3 NOP                ;
0DC4 PUSHF              ;
0DC5 CALL 0D08          ; décrémenter
0DC8 JNB  0DCE          ; pas de pb.
0DCA POPF               ; si :
0DCB POP  SI            ; sortie
0DCC STC                ; en erreur
0DCD RET                ;
0DCE POPF               ;
0DCF POP  SI            ; sortie
0DD0 CLC                ; R.A.S.
0DD1 RET                ;
```

0DE0: AFFICHAGE DES RESULTATS (0100).

```
0DE0 CALL 0700          ; écran noir
0DE3 MOV  DX,0505       ; curseur
0DE6 MOV  BX,0000       ; positionné
0DE9 MOV  AH,02         ; (DL,DH)
0DEB INT  10            ;
0DED MOV  CX,[0588]     ; nbre de R
0DF1 PUSH CX            ; sauver CX
0DF2 MOV  DX,0E38       ; 'état des '
0DF5 MOV  AH,09         ;
0DF7 INT  21            ;
0DF9 CALL 07A0          ; afficher CX
0DFC MOV  DX,0E42       ; 'registres'
0DFF MOV  AH,09         ;
0E01 INT  21            ;
0E03 POP  CX            ;
0E04 MOV  DX,0705       ; curseur
0E07 XOR  SI,SI         ; n° 1er R
0E09:BOUCLE             ; AFFICHAGE
0E09 PUSH DX            ;
0E0A MOV  BX,0000       ; curseur po-
0E0D MOV  AH,02         ; sitionné
0E0F INT  10            ;
0E11 MOV  DX,0E4D       ; 'valeur du
0E14 MOV  AH,09         ; registre '
0E16 INT  21            ;
0E18 CALL 07F0          ; nom de R
0E1B CALL 0E70          ; afficher R
0E1E JB   0E2A          ; impossible
0E20 POP  DX            ; si :
0E21 INC  DH            ; curseur
0E23 INC  DH            ; reposition
0E25 INC  SI            ;
0E26 LOOP 0E09          ; -> BOUCLE
0E28 CLC                ; R.A.S.
0E29 RET                ; (fin)
0E2A POP  DX            ;
0E2B CALL 0FD0          ; erreur :
0E2E STC                ; dépassement
0E2F RET                ;
```

0E38: messages du programme 0DE0.

```
0E30 'état des ' 24
0E42 ' registres' 24
0E4D 'valeur du registre ' 24
```

0E70: Affichage d'un registre (0DE0).

```
0E70 PUSH SI            ; SI et CX
0E71 PUSH CX            ; utilisés
0E72 MOV  BX,SI         ;
0E74 SHL  BX,1          ; adresse de
0E76 SHL  BX,1          ; R dans
0E78 ADD  BX,4000       ; répertoire
0E7C MOV  AX,[BX+02]    ; AX = |R|
0E7F CMP  AX,001C       ; tester |R|
0E82 JB   0E8F          ; autre test
0E84 MOV  DX,0F50       ; 'trop
0E87 MOV  AH,09         ; grand'
0E89 INT  21            ;
0E8B POP  CX            ;
0E8C POP  SI            ;
0E8D CLC                ;
0E8E RET                ;
0E8F MOV  AX,[BX+02]    ; AX = |R|
0E92 MOV  [0A26],AX     ; |R| sauvé
0E95 MOV  CX,FFF8       ; CX := -8
0E98 MOV  BP,7F02       ; depuis
0E9B MOV  AX,[BX]       ; CS:7F02
0E9D NOP                ; copie
0E9E:BOUCLE1            ; test, copie
0E9E MOV  SI,AX         ; n°1er S
0EA0 SUB  AX,01C8       ; converti
0EA3 SHL  AX,1          ; en adr. TAS
0EA5 CMP  AX,1C70       ; en TAS?
0EA8 JB   0EB1          ; oui : suite
0EAA CALL 0FD0          ; non :
0EAD STC                ; erreur
0EAE POP  CX            ; simulateur
0EAF POP  SI            ;
0EB0 RET                ;
0EB1 CALL 0F68          ; copier
0EB4 ADD  BP,+10        ; secteur
0EB7 ADD  CX,+08        ; nouveau
0EBA PUSH BX            ;
0EBB MOV  BX,AX         ; lire entrée
0EBD ES:                ; TAS (S
0EBE MOV  AX,[BX]       ; suivant)
0EC0 POP  BX            ;
0EC1 CMP  AX,0100       ; dernier?
0EC4 JZ   0EC8          ; oui : fini
0EC6 JMP  0E9E          ; -> BOUCLE1
0EC8 PUSH DI            ; suite :
0EC9 PUSH ES            ; conversion
0ECA PUSH DS            ; en
0ECB POP  ES            ; représenta-
0ECC DEC  DI            ; tion
0ECD DEC  DI            ; décimale
0ECE PUSH CX            ; on vérifie
0ECF XOR  AX,AX         ; d'abord |R|
0ED1 MOV  CX,0008       ; calcul de
0ED4 STD                ; (R) (mots)
```

```
0ED5 REPE              ;
0ED6 SCASW             ;
0ED7 POP  CX           ;
0ED8 POP  ES           ;
0ED9 POP  AX           ;
0EDA INC  DI           ;
0EDB INC  DI           ;
0EDC PUSH DI           ;
0EDD SUB  AX,DI        ; addition
0EDF SHR  AX,1         ; avec valeur
0EE1 SUB  AX,0008      ; précédente
0EE4 NEG  AX           ; de CX
0EE6 INC  AX           ;
0EE7 ADD  AX,CX        ;
0EE9 CMP  AX,[0A26]    ; vérifier
0EED JZ   0EF7         ; =: suite
0EEF CALL 0FD0         ; erreur
0EF2 STC               ; simulateur
0EF3 POP  DI           ;
0EF4 POP  CX           ;
0EF5 POP  SI           ;
0EF6 RET               ;
0EF7 MOV  CX,[0A26]    ; CX = |R|
0EFB SHL  CX,1         ; conversion
0EFD POP  DI           ; en octets
0EFE MOV  AX,[DI]      ; lire déc.
0F00 CMP  AH,00        ; de tête
0F03 JNZ  0F06         ;
0F05 DEC  CX           ; correctif
0F06 MOV  AX,CX        ;
0F08 CMP  AX,0055      ; test |R|
0F0B JB   0F10         ; a priori OK
0F0D JMP  0E84         ; trop grand
0F10 MOV  BP,7F52      ; tampon
0F13 MOV  [7F00],AX    ; longueur
0F16:BOUCLE2           ; divisions
0F16 CALL 0F88         ; AX:=Rmod064
0F19 MOV  [BP+00],AX   ; écrire
0F1C INC  BP           ; aux
0F1D INC  BP           ;suivants
0F1E CMP  BY [7F00],00 ; longueur?
0F23 JZ   0F27         ; si 0, fini
0F25 JMP  0F16         ; -> BOUCLE2
0F27 MOV  CX,BP        ; nombre
0F29 SUB  CX,7F52      ; d'octets à
0F2D DEC  BP           ; afficher
0F2E DEC  BP           ;
0F2F MOV  AX,[BP+00]   ; mot de tête
0F32 INC  BP           ;
0F33 CMP  AH,30        ;
0F36 JNZ  0F3A         ;
0F38 DEC  BP           ; correctif
0F39 DEC  CX           ; éventuel
0F3A:BOUCLE3           ; affichage
0F3A MOV  DL,[BP+00]   ; 'lire'
0F3D MOV  AH,02        ; afficher
0F3F INT  21           ;
0F41 DEC  BP           ; suivant
0F42 LOOP 0F3A         ; -> BOUCLE3
0F44 POP  CX           ;
0F45 POP  SI           ; fin
0F46 RET               ;
```

0F50: message du programme 0E70.

```
0F50 'valeur trop grande ' 24
```

0F68: Copie d'un secteur (0E70).

```
0F68 PUSH CX           ; sauvegarde
0F69 PUSH ES           ; des
0F6A PUSH AX           ; registres
0F6B PUSH ES           ;
0F6C POP  DS           ; DS = zone
0F6D PUSH CS           ; ES = CS
0F6E POP  ES           ;
0F6F MOV  AX,SI        ; converion
0F71 MOV  CX,0004      ; n°S en
0F74 SHL  AX,CL        ; adresse
0F76 MOV  SI,AX        ; (en DS:SI)
0F78 MOV  CX,0008      ; écrire
0F7B MOV  DI,BP        ; sur CS:BP
0F7D CLD               ;
0F7E REPZ              ; huit mots
0F7F MOVSW             ;
0F80 PUSH CS           ; registres
0F81 POP  DS           ; restaurés
0F82 POP  AX           ;
0F83 POP  ES           ;
0F84 POP  CX           ;
0F85 RET               ;
```

0F88: Division par cent (0E70).

```
0F88 PUSH BX           ;
0F89 PUSH CX           ;
0F8A MOV  BX,0064      ; diviseur
0F8D MOV  DI,7F02      ;
0F90 MOV  CX,[7F00]    ; taille
0F94 ADD  DI,CX        ; départ
0F96 PUSH DI           ;
0F97 XOR  AH,AH        ; AH = 0
0F99:BOUCLE            ; div. octale
0F99 DEC  DI           ;
0F9A MOV  AL,[DI]      ; lire
0F9C DIV  BL           ; diviser
0F9E MOV  [DI],AL      ; écrire q
0FA0 LOOP 0F99         ; -> BOUCLE
0FA2 XOR  AL,AL        ;
0FA4 XCHG AH,AL        ; AX = reste
0FA6 MOV  BL,0A        ; division
0FA8 DIV  BL           ; par dix
0FAA XCHG AH,AL        ; permuter
0FAC ADD  AX,3030      ; conversion
0FAF POP  DI           ; en chiffres
0FB0 PUSH AX           ;
0FB1 DEC  DI           ;
0FB2 MOV  AL,[DI]      ; regarder
0FB4 CMP  AL,00        ; chiffre de
0FB6 JNZ  0FBC         ; tête
0FB8 DEC  WO [7F00]    ; décroître
0FBC POP  AX           ; si nul
0FBD POP  CX           ;
0FBE POP  BX           ;
0FBF RET               ;
```

0FD0: GESTIONNAIRE DES ERREURS.

```
0FD0 PUSH CX           ; sauvegarde
0FD1 PUSH BX           ; des
0FD2 PUSH BP           ; registres
0FD3 PUSH DI           ;
0FD4 PUSH SI           ;
0FD5 PUSH ES           ;
0FD6 MOV  BP,SP        ; recueil
0FD8 ADD  BP,+0A       ; adresse
0FDB MOV  AX,[BP+00]   ; retour
0FDE MOV  BX,[1072]    ; BX sur pile
0FE2 INC  BX           ; puis actua-
0FE3 INC  BX           ; lisé pour
```

```
0FE4 MOV  [BX],AX      ; empiler AX
0FE6 MOV  [1074],BX    ; BX noté
0FEA CMP  AX,0660      ; après 0660?
0FED JNB  0FFB         ; oui : suite
0FEF CMP  AX,0DE0      ; syntaxe?
0FF2 JB   0FFB         ; non : suite
0FF4 MOV  AX,[04AE]    ; n°ligne
0FF7 INC  BX           ; BX réactua-
0FF8 INC  BX           ; lisé pour
0FF9 MOV  [BX],AX      ; empiler AX
0FFB MOV  [1072],BX    ; et 'rangé'
0FFF MOV  AX,[1070]    ; |erreurs|
1002 INC  AX           ; +1
1003 MOV  [1070],AX    ; en compteur
1006 CMP  AX,000A      ; >= dix?
1009 JNB  1017         ; oui : suite
100B CALL 1100         ; fatale?
100E JB   1017         ; oui : suite
1010 POP  ES           ; non :
1011 POP  SI           ; retour au
1012 POP  DI           ; transcodeur
1013 POP  BP           ;
1014 POP  BX           ;
1015 POP  CX           ;
1016 RET               ;
1017 CALL 0700         ; écran noir
101A MOV  DX,0505      ; curseur
101D XOR  BX,BX        ; positionné
101F MOV  AH,02        ;
1021 INT  10           ;
1023 MOV  CX,[1070]    ; |erreurs|
1027 PUSH CX           ; sauver CX
1028 CALL 07A0         ; afficher CX
102B MOV  DX,105A      ; 'erreurs
102E MOV  AH,09        ; relevées'
1030 INT  21           ;
1032 MOV  DX,0705      ; curseur
1035 XOR  BX,BX        ; plus bas
1037 MOV  AH,02        ;
1039 INT  10           ;
103B MOV  BX,1078      ; 1ère erreur
103E POP  CX           ; compteur
103F:BOUCLE            ; d'affichage
103F CALL 10A0         ; DI := adr.
1042 MOV  DX,[DI+1160] ; adr. mess.
1046 CALL 10B8         ; n° ligne
1049 MOV  AH,09        ; puis
104B INT  21           ; message
104D INC  BX           ; réactuali-
104E INC  BX           ; ser BX
104F LOOP 0103F        ; -> BOUCLE
1051 POP  ES           ; restaurer
1052 POP  SI           ; les
1053 POP  DI           ; registres
1054 POP  BP           ;
1055 POP  BX           ;
1056 POP  CX           ;
1057 JMP  0118         ; terminaison
```

105A messages du programme 0FD0

```
105A 'erreurs relevées:' 24
```

1070 variables du programme 0FD0

```
1070-1078
1078-10A0 pile des erreurs
```

10A0 : Adresse de l'entrée du tableau associée à l'erreur (0FD0).

```
10A0 PUSH CX           ; compteur
10A1 MOV  AX,[BX]      ; 'n°' erreur
10A3 PUSH CS           ;
10A4 POP  ES           ; ES=CS
10A5 MOV  DI,1120      ;
10A8 MOV  CX,001E      ;
10AB CLD               ;
10AC REPNZ             ;
10AD SCASW             ;
10AE DEC  DI           ;
10AF DEC  DI           ; adresse
10B0 SUB  DI,1120      ; obtenue
10B4 POP  CX           ; compteur
10B5 RET               ;
```

10B8 : Affichage éventuel du numéro de ligne (0FD0).

```
10B8 MOV  AX,[DI+1120] ; id. erreur
10BC CMP  AX,06A0      ; syntaxe?
10BF JA   10C8         ; non: sortie
10C1 CMP  AX,02D8      ; syntaxe?
10C4 JB   10C8         ; non: sortie
10C6 JMP  10CA         ; suite
10C8 RET               ; sortie
10C9 NOP               ;
10CA PUSH DX           ;
10CB PUSH CX           ;
10CC MOV  DX,10F0      ; 'ligne n° '
10CF MOV  AH,09        ;
10D1 INT  21           ;
10D3 INC  BX           ; incrémenter
10D4 INC  BX           ; BX (mot)
10D5 MOV  CX,[BX]      ; n° de ligne
10D7 CALL 07A0         ; afficher CX
10DA MOV  DX,10FA      ; ' : '
10DD MOV  AH,09        ;
10DF INT  21           ;
10E1 POP  CX           ;
10E2 POP  DX           ;
10E3 RET               ;
```

10F0 : messages du programme 10B8.

```
10F0 'ligne n° ' 24
10FA ' : ' 24
```

1100 : Erreur fatale? (0FD0).

```
1100 MOV  BX,[1074]    ; lire ptr
1104 MOV  AX,[BX]      ; AX = erreur
1106 PUSH DS           ;
1107 POP  ES           ;
1108 MOV  DI,11A0      ; adr. Tbl.3
110B MOV  CX,0013      ; nb. entrées
110E CLD               ; recherche
110F REPNZ             ;
1110 SCASW             ;
1111 JZ     1115       ; trouvé
1113 CLC               ; non
1114 RET               ;
1115 STC               ; oui
1116 RET               ;
```

1120 : Table I : les erreurs.

```
1120  42 01  9F 03  49 04  74 04
1128  CD 04  DF 04  E5 04  30 05
1130  68 05  BF 05  FB 05  05 06
1138  6F 06  F8 06  3E 08  5D 09
1140  C4 09  FE 09  2D 0B  46 0B
1148  9A 0B  B8 0B  E6 0B  F3 0B
```

```
1150  BE 0C  EB 0C  5E 0D  2E 0E
1158  AD 0E  F2 0E
```

1160 : Table II : adresses des messages d'erreur.

```
1160  00 12  16 12  48 12  66 12
1168  48 12  84 12  9A 12  AD 12
1170  BD 12  DC 12  48 12  F9 12
1178  15 13  2D 13  5A 13  86 13
1180  86 13  86 13  86 13  B5 13
1188  B5 13  B5 13  B5 13  86 13
1190  B5 13  B5 13  B5 13  2D 13
1190  B5 13  B5 13
```

11A0 : Table III : les erreurs fatales.

```
11A0  42 01  6F 06  F8 06  3E 08
11A8  5D 09  C4 09  FE 09  2D 0B
11B0  46 0B  9A 0B  B8 0B  E6 0B
11B8  F3 0B  BE 0C  EB 0C  5E 0D
11C0  2E 0E  AD 0E  F2 0E
```

1200 : messages d'erreur.

```
1200 0D 0A 'erreur de syntaxe'
1213 0D 0A 24
1216 0D 0A 'début d',27,'instru'
1226 'ction non rencontré : ; a'
123F 'ttendu' 0D 0A 24
1248 0D 0A 'fin de fichier inat'
125D 'tendue' 0D 0A 24
1266 0D 0A 'nom d',27,'instruct'
1276 'ion inconnu' 0D 0A 24
1284 0D 0A 'symbole incorrect'
1297 0D 0A 24
129A 0D 0A 'nombre attendu' 0D
12AB 0A 24
12AD 0D 0A ' , attendue' 0D 0A 24
12BD 0D 0A 'trop de registres d'
12D2 'éclarés' 0D 0A 24
12DC 0D 0A 'retour chariot mal '
12F1 'écrit' 0D 0A 24
12F9 0D 0A 'chiffre décimal att'
130E 'endu' 0D 0A 24
1315 0D 0A 'trop d',27,'instruc'
1325 'tions' 0D 0A 24
132D 0D 0A 'valeur trop grande,'
1342 ' affichage impossible' 0D
1358 0A 24
135A 0D 0A 'pas plus de trois e'
136F 'ssais permis, désolé' 0D
1384 0A 24
1386 0D 0A 'dépassement de capa'
139B 'cité de l',27,'espace all'
13AF 'oué',0D 0A 24
13B5 0D 0A 'erreur de programma'
13CA 'tion du simulateur' 0D 0A
13DE 24
```

Ressources du programme d'arithmétique multi-précission.

ressources globales

```
3800  table des segments alloués
3820  table des taux d'occupation
        des segments alloués
3840  nombre des segments alloués
4000  répertoire des registres
  entrées de 4 octets :
  +00  n° complet premier secteur
  +02  nombre de secteurs de R, [R]
4040  répertoire, 2ème partie
  entrées de 2 octets :
  nombre de mots du dernier secteur
    de R, {R}
```

ressources de l'addition

```
0A00  taille du registre source
0A02  taille du registre but
0A04  n° TAS strict 1er S source
0A06  n° TAS strict 1er S but
0A08  segment zone R but
0A0A  adresse répertoire R but
```

ressources de la multiplication

```
[0A26]:[0A24]  S source de M
[0A2A]:[0A28]  S source de m
[0A2E]:[0A2C]  S but faible
[0A32]:[0A30]  S but fort
utilisation d'une zone de variables
sur la pile :
  00    adresse répert. R auxiliaire
04:02   1er S de R auxiliaire
08:06   S dernier de R auxiliaire
0C:0A   adresse TAS de ce dernier
10:0E   S suppl. de R auxiliaire
14:12   son adresse TAS
  16    son n° complet en TAS
1A:18   1er S en décalage
1E:1C   son adresse TAS
  20    son n° complet en TAS
24:22   S courant de m
28:26   son adresse TAS
  2A    DI initial
  2C    BX initial
  2E    SI initial
0700  décimale de tête du produit
09FC  compteur de décalage
09FE  indicateur modification 04A0
```

ressources de la division

```
zone de variables sur la pile :
  00  segment zone de chaînage
  02  n° chaînage de u0 de D
  04  n° chaînage de Dt
  06  n° chaînage de u0 de d
  08  n° chaînage de u0 de q
  0A  n° chaînage de qt
  0C  00 ssi [q] > 1
  0D  w : 00 ssi u = v
  10  adresse répertoire de D
  12  [D]
  14  {D} (en octets)
18-16 Dt (décimale de tête de D)
  1A  n° chaînage de Dt
1E:1C Dp (décimale de prédivision)
  20  n° chaînage de Dp
```

ANNEXE 2 : Programmes d'opérations arithmétiques multi-précision

ressources de la division (suite)

```
24:22 Du (1er S de D div. partlle)
  26  n° chaînage de Du
2A:28 Dc (S courant de D)
  2C  n° chaînage de Dc
  30  adresse répertoire de d
  32  [d]
  34  {d} (en octets)
38:36 dt
  3A  n° chaînage de dt
3E:3C du
  40  n° chaînage du
44:42 dc
  46  n° chaînage dc
  50  q
  52  [q] (calcul en marchant)
  54  {q} (en octets)
58:56 qc
  5A  n° chaînage qc
60-62 fenêtre de prédivision
  64  indices de partage secteur
  plein pour mult/soustr:
    by 64: 1ère partie
    by 65: 2ème partie
  66:  indices de partage dernier
  secteur pour mult/soustr
    by 66: 1ère partie
    by 67: 2ème partie
```

Dans le programme de division 0AD0, R désigne le registre reste et r la retenue concernée par les instructions commentées. Les adresses en répertoire de D, d et q sont, respectivement, dans SI, BX et DI à l'entrée du programme.

0200: ADC deux secteurs.

```
0200 MOV  CX,0008      ; huit mots
0203:BOUCLE
0203 MOV  AX,[SI]      ; (DS = seg.)
0205 ES:               ;
0206 ADC  AX,[DI]      ; addition
0208 ES:               ; (ret. ant.)
0209 MOV  [DI],AX      ; écriture
020B INC  DI           ; mise
020C INC  DI           ; à jour DI
020D INC  SI           ; mise
020E INC  SI           ; à jour SI
020F LOOP 0203         ; -> BOUCLE
0211 RET               ;
```

0220: Incrémenter un secteur.

```
0220 MOV    CX,0008    ; huit mots
0223:BOUCLE
0223 ES:               ;
0224 MOV  AX,[DI]      ; lecture
0226 ADD  AX,0001      ; ADD pour CF
0229 ES:               ;
022A MOV  [DI],AX      ; écriture
022C JNB  0232         ; plus de ret
022E INC  DI           ; mise
022F INC  DI           ; à jour DI
0230 LOOP 0223         ; -> BOUCLE
0232 RET
```

0240 Initialiser les secteurs courants.

```
0240 CS:               ; sauvegarde:
0241 MOV  AX,[SI+4002] ;
0245 CS:               ;
0246 MOV  [0A00],AX    ; taille (SI)
0249 CS:               ;
024A MOV  AX,[DI+4002] ;
024E CS:               ;
024F MOV  [0A02],AX    ; taille (DI)
0252 CS:               ;
0253 MOV  [0A0A],DI    ; n°(DI)
0257 CS:               ;
0258 MOV  AX,[SI+4000] ; conversion
025C MOV  BX,1000      ; du n° de
025F XOR  DX,DX        ; secteur
0261 DIV  BX           ; en adresse
0263 SHL  AX,1         ; absolue du
0265 MOV  BX,AX        ; secteur
0267 CS:               ; placée
0268 MOV  AX,[BX+3800] ; dans DS:SI
026C MOV  DS,AX        ; DS prêt
026E MOV  AX,DX        ;
0270 CS:               ;
0271 MOV  [0A04],AX    ;
0274 MOV  CL,04        ;
0276 SHL  AX,CL        ;
0278 MOV  SI,AX        ; SI prêt
027A CS:               ;
027B MOV  AX,[DI+4000] ; puis même
027F MOV  BX,1000      ; opération
0282 XOR  DX,DX        ; avec (DI)
0284 DIV  BX           ;
0286 SHL  AX,1         ;
0288 CS:               ;
0289 MOV  [0A08],AX    ;
028C MOV  BX,AX        ;
028E CS:               ;
028F MOV  AX,[BX+3800] ;
0293 MOV  ES,AX        ; ES prêt
0295 MOV  AX,DX        ;
0297 CS:               ;
0298 MOV  [0A06],AX    ;
029B MOV  CL,04        ;
029D SHL  AX,CL        ;
029F MOV  DI,AX        ; DI prêt
02A1 XOR  DX,DX        ; DX prêt
02A3 CLC               ; CF prêt
02A4 RET               ;
```

02B0 Passer aux secteurs suivants.

```
02B0 PUSH DX           ;
02B1 CS:               ; d'abord
02B2 MOV  BX,[0A04]    ; DS:SI
02B6 SUB  BX,01C8      ;
02BA SHL  BX,1         ;
```

```
02BC DS:                ;
02BD MOV  AX,[BX]       ;
02BF MOV  BX,1000       ;
02C2 XOR  DX,DX         ;
02C4 DIV  BX            ; (DX reste)
02C6 CMP  DX,0100       ; fin source?
02CA JZ   0322          ; oui: AL=00
02CC SHL  AX,1          ; (AX quot.)
02CE MOV  BX,AX         ; BX := 2*AX
02D0 CS:                ;
02D1 MOV  AX,[BX+3800]; seg. zone
02D5 MOV  DS,AX         ; DS prêt
02D7 MOV  AX,DX         ;
02D9 CS:                ; enregistrer
02DA MOV  [0A04],AX     ; n° secteur.
02DD MOV  CL,04         ; puis
02DF SHL  AX,CL         ; 'l'étendre'
02E1 MOV  SI,AX         ; SI prêt
02E3 CS:                ; à présent,
02E4 MOV  BX,[0A06]     ; idem ES:DI
02E8 SUB  BX,01C8       ;
02EC SHL  BX,1          ;
02EE ES:                ;
02EF MOV  AX,[BX]       ;
02F1 MOV  BX,1000       ;
02F4 XOR  DX,DX         ;
02F6 DIV  BX            ;
02F8 CMP  DX,0100       ; fin du but?
02FC JZ   031C          ; oui: AL=01
02FE SHL  AX,1          ; non:
0300 CS:                ; enregistrer
0301 MOV  [0A08],AX     ; n° secteur
0304 MOV  BX,AX         ; puis
0306 CS:                ; conversions
0307 MOV  AX,[BX+3800];
030B MOV  ES,AX         ; ES prêt
030D MOV  AX,DX         ;
030F CS:                ; enregistrer
0310 MOV  [0A06],AX     ; n°secteur
0313 MOV  CL,04         ; puis
0315 SHL  AX,CL         ; 'l'étendre'
0317 MOV  DI,AX         ; DI prêt
0319 POP  DX            ;
031A CLC                ; R.A.S.
031B RET                ; retour
031C MOV  AL,01         ; but terminé
031E POP  DX            ; source non
031F STC                ; signaler
0320 RET                ; retour
0321 NOP                ;
0322 XOR  AL,AL         ; source fini
0324 POP  DX            ;
0325 STC                ; signaler
0326 RET                ; retour
```

0330 Fixer le secteur but suivant.

```
0330 PUSH DX            ;
0331 CS:                ; pour lire
0332 MOV  BX,[0A06]     ; entrée TAS
0336 SUB  BX,01C8       ; n°sect.
033A SHL  BX,1          ; devenu adr.
033C ES:                ;
033D MOV  AX,[BX]       ; TAS lue
033F MOV  DX,AX         ; extraction
0341 AND  AX,0FFF       ; n° secteur
0344 MOV  CL,0B         ;
0346 SHR  DX,CL         ; DX:= 2*zone
0348 CMP  AX,0100       ; fin de but?
034B JZ   0366          ; oui: alloc.
034D MOV  BX,DX         ; non:
034F PUSH AX            ; préparation
0350 CS:                ; secteur
0351 MOV  AX,[BX+3800]; suivant
0355 MOV  ES,AX         ; ES prêt
0357 POP  AX            ;
0358 CS:                ;
0359 MOV  [0A06],AX     ;
035C MOV  CL,04         ;
035E SHL  AX,CL         ;
0360 MOV  DI,AX         ; DI prêt
0362 POP  DX            ;
0363 CLC                ; R.A.S.
0364 RET                ; retour
0365 NOP                ;
0366 PUSH ES            ; allouer
0367 PUSH BX            ; un secteur
0368 CS:                ;
0369 MOV  BX,[0A08]     ; n°zone même
036D CS:                ; secteur
036E CMP  WO [BX+3820],+00; pleine?
0373 JZ   03AE          ; oui: voir
0375 MOV  CX,0E38       ; non:
0378 XOR  DI,DI         ; chercher
037A XOR  AX,AX         ; sec. libre
037C CLD                ; (0000) via
037D REPNZ              ; SCASW
037E SCASW              ;
037F JZ   0389          ; trouvé
0381 POP  BX            ; non:
0382 POP  ES            ; équilibrer
0383 POP  DX            ; pile
0384 CALL 0FD0          ; erreur
0387 STC                ; le signaler
0388 RET                ;
0389 DEC  DI            ; réajuster
038A DEC  DI            ; DI (adresse
038B ES:                ; en TAS)
038C MOV  WO [DI],0100; marquage
0390 CS:                ; fin
0391 DEC  WO [BX+3820]; mise à jour
0395 MOV  AX,DI         ; compteur
0397 SHR  AX,1          ; confection
0399 ADD  AX,01C8       ; n°TAS du
039C CS:                ; secteur
039D MOV  DX,[0A08]     ; trouvé
03A1 MOV  CL,0B         ;
03A3 SHL  DX,CL         ;
03A5 OR   AX,DX         ;
03A7 POP  BX            ; retrouver
03A8 POP  ES            ; ex-secteur
03A9 ES:                ; y inscrire
03AA MOV  [BX],AX       ; le nouveau
03AC JMP  033F          ; -> v.sortie
03AE CS:                ; chercher
03AF MOV  CX,[3840]     ; une zone
03B3 MOV  DI,3820       ; non pleine
03B6 PUSH CS            ; par SCASW
03B7 POP  ES            ; dans le
03B8 XOR  AX,AX         ; tableau
03BA CLD                ; 3840 taux
03BB REPZ               ; de rempl.
03BC SCASW              ;
03BD JZ   03D4          ; non trouvé
```

```
03BF DEC  DI          ; si: noter
03C0 DEC  DI          ; le segment
03C1 SUB  DI,+20      ; dans ES
03C4 CS:              ;
03C5 MOV  AX,[DI]     ;
03C7 MOV  ES,AX       ; ES prêt
03C9 SUB  DI,3800     ; noter n°de
03CD CS:              ; zone
03CE MOV  [0A08],DI   ; chercher en
03D2 JMP  0375        ; zone sect.
03D4 MOV  BX,1000     ; allouer
03D7 MOV  AH,48       ; zone 64 Ko
03D9 INT  21          ;
03DB JB   040A        ; impossible
03DD CS:              ; si:
03DE MOV  BX,[3840]   ; enregistrer
03E2 SHL  BX,1        ; segment
03E4 CS:              ; alloué tab.
03E5 MOV  [BX+3800],AX; 3800
03E9 CS:              ; initaliser
03EA MOV  WO [BX+3820],0E38; taux
03F0 CS:              ; remplissage
03F1 MOV  [0A08],BX   ;
03F5 CS:              ; mise à jour
03F6 INC  WO [3840]   ; compteur
03FA PUSH AX          ;
03FB POP  ES          ; ES prêt
03FC XOR  DI,DI       ;
03FE MOV  CX,8000     ; initialiser
0401 XOR  AX,AX       ; la zone
0403 CLD              ; à 00
0404 REPZ             ;
0405 STOSW            ;
0406 XOR  DI,DI       ; DI prêt
0408 JMP  038B        ; -> v.sortie
040A POP  BX          ; échec alloc
040B POP  ES          ; équilibrage
040C POP  DX          ; de la pile
040D CALL FD0         ; erreur
0410 STC              ; le signaler
0411 RET              ;
```

0420 Secteur source suivant

```
0420 PUSH DX          ;
0421 CS:              ; n° secteur
0422 MOV  BX,[0A04]   ; courant
0426 SUB  BX,01C8     ; transformé
042A SHL  BX,1        ; en adresse
042C DS:              ; entrée TAS
042D MOV  AX,[BX]     ; entrée lue
042F MOV  DX,AX       ; on sauve AX
0431 AND  AX,0FFF     ; DX converti
0434 MOV  CL,0B       ; en adresse
0436 SHR  DX,CL       ; tabl. 3800
0438 CMP  AX,0100     ; fin source?
043B JZ   0456        ; oui: -> fin
043D MOV  BX,DX       ;
043F PUSH AX          ; AX sauvé
0440 CS:              ; lecture
0441 MOV  AX,[BX+3800]; tabl.3800
0445 MOV  DS,AX       ; DS prêt
0447 POP  AX          ; AX retrouvé
0448 CS:              ;
0449 MOV  [0A04],AX   ; enregistré
044C MOV  CL,04       ; et converti
044E SHL  AX,CL       ; en dépl.
0450 MOV  SI,AX       ; SI prêt
0452 POP  DX          ;
0453 CLC              ; R.A.S.
0454 RET              ;
0455 NOP              ;
0456 POP  DX          ;
0457 STC              ; pas suivant
0458 RET              ;
```

0460 copier secteur source sur secteur but

```
0460 PUSH DI          ;
0461 MOV  CX,0008     ; utilisation
0464 CLD              ; classique
0465 REPZ             ; de MOVSW
0466 MOVSW            ;
0467 POP  DI          ;
0468 RET              ;
```

0470 Determiner la longueur du registre.

```
0470 CS:              ;
0471 MOV  DI,[0A06]   ; DI lu est
0475 MOV  CL,04       ; converti
0477 SHL  DI,CL       ; en adresse
0479 ADD  DI,+0E      ; fin du sec.
047C MOV  CX,0008     ;
047F XOR  AX,AX       ;
0481 STD              ; on descend
0482 REPZ             ;
0483 SCASW            ;
0484 JNZ  048C        ; -> suite
0486 CALL 0FD0        ; erreur
0489 STC              ; signaler
048A RET              ;
048B NOP              ;
048C MOV  AX,CX       ; nombre mots
048E INC  AX          ; correction
048F RET              ;
```

04A0: ADDITION DE DEUX REGISTRES.

```
04A0 PUSH CX          ;
04A1 CALL 0240        ; initialiser
04A4 PUSHF            ; sauver ret.
04A5:BOUCLE1          ; ADC sect.
04A5 POPF             ; retenue
04A6 INC  DX          ; compter
04A7 CALL 0200        ; ADC sect.
04AA PUSHF            ; ret. sauvée
04AB CALL 02B0        ; sect.suiv.
04AE JNB  04A5        ; -> BOUCLE1
04B0 CMP  AL,00       ; test
04B2 JNZ  04DF        ; cas AL=01
04B4 POPF             ; retenue
04B5:BOUCLE2          incr. but
04B5 JNB  04C8        ; fin: taille
04B7 INC  DX          ; compter
04B8 CALL 0330        ; sect.suiv.
04BB JB   04C2        ; (dépassem.)
04BD CALL 0220        ; +1 au rés.
04C0 JMP  04B5        ; -> BOUCLE2
04C2 POP  CX          ; équil. pile
04C3 CALL 0FD0        ; erreur
04C6 STC              ;
04C7 RET              ;
04C8 CALL 0470        ; mots du der
```

```
04CB CS:                ; enregistrer
04CC MOV  BX,[0A0A]     ; la taille
04D0 CS:                ; du registre
04D1 MOV  [BX+4002],DX  ; nb. sect.
04D5 SHR  BX,1          ; et
04D7 CS:                ; de der sec.
04D8 MOV  [BX+4040],AL  ; nb. mots
04DC POP  CX            ;
04DD CLC                ; R.A.S.
04DE RET                ; sortie
04DF:BOUCLE3            ; cas AL=01
04DF INC  DX            ; compter
04E0 CALL 0330          ; nouveau but
04E3 JB   04C2          ; (dépassem.)
04E5 CALL 0460          ; copie
04E8 POPF               ; rappel ret.
04E9 JNB  04EE          ; nulle: suiv
04EB CALL 0220          ; +1 résult.
04EE PUSHF              ; ret. sauvée
04EF CALL 0420          ; source suiv
04F2 JB   04F6          ; (terminé)
04F4 JMP  04DF          ; -> BOUCLE3
04F6 POPF               ; rappel ret.
04F7 JNB  04C8          ; fin: taille
04F9 INC  DX            ; compter
04FA CALL 0330          ; ult. sect.
04FD JB   04C2          ; (dépassem.)
04FF CALL 0220          ; +1 final
0502 JMP  04C8          ; ajus.taille
```

0510 Multiplication de deux secteurs.

```
0510 CS:                ; sect. buts
0511 LES  DI,[0A30]     ; à 00:
0515 MOV  CX,0008       ; fort en
0518 XOR  AX,AX         ; [A32][A30]
051A CLD                ;
051B REPZ               ;
051C STOSW              ;
051D CS:                ; faible en
051E LES  DI,[0A2C]     ; [A2E][A2C]
0522 MOV  CX,0008       ;
0525 XOR  AX,AX         ;
0527 CLD                ;
0528 REPZ               ;
0529 STOSW              ;
052A CS:                ; comparaison
052B LES  DI,[0A2C]     ; à 00
052F CS:                ; de
0530 LDS  SI,[0A24]     ; mult.cande
0534 MOV  CX,0008       ;
0537 CLD                ;
0538 REPZ               ;
0539 CMPSW              ;
053A JNZ  053E          ; non nul
053C RET                ; si: fin
053D NOP                ;
053E CS:                ; comparaison
053F LES  DI,[0A2C]     ; à 00
0543 CS:                ; de
0544 LDS  SI,[0A28]     ; mult.cateur
0548 MOV  CX,0008       ;
054B CLD                ;
054C REPZ               ;
054D CMPSW              ;
054E JNZ  0552          ; non nul
0550 RET                ; si: fin
0551 NOP                ;
0552 XOR  BX,BX         ; décalage
0554:BOUCLE1            ; mult.simple
0554 CS:                ;
0555 MOV  AX,[0A2A]     ; ES: segment
0558 MOV  ES,AX         ; mult.cateur
055A CS:                ; DS:SI:
055B LDS  SI,[0A24]     ; mult.cande
055F XOR  DX,DX         ; retenue MUL
0561 CS:                ; DI: début
0562 MOV  DI,[0A28]     ; mult.cateur
0566 ES:                ;
0567 MOV  BP,[BX+DI]    ; BP:décimale
0569 MOV  DI,05F8       ; tampon
056C MOV  CX,0008       ; sur 8 mots
056F PUSH BX            ; sauv.décal.
0570 CLC                ; ret. init.
0571:BOUCLE2            ; mult.chiff.
0571 MOV  AX,[SI]       ;
0573 MOV  BX,DX         ; BX: ex-ret.
0575 PUSHF              ;
0576 MUL  BP            ; (DX<=FFFE)
0578 POPF               ;
0579 ADC  AX,BX         ;
057B CS:                ;
057C MOV  [DI],AX       ; écriture
057E INC  SI            ;
057F INC  SI            ;
0580 INC  DI            ;
0581 INC  DI            ;
0582 LOOP 0571          ; -> BOUCLE2
0584 ADC  DX,+00        ; report ret.
0587 POP  BX            ; BX: décal.
0588 MOV  CX,0010       ; addition
058B SUB  CX,BX         ; sur but
058D SHR  CX,1          ; faible
058F PUSH CX            ;
0590 CS:                ; initialiser
0591 LES  DI,[0A2C]     ;
0595 ADD  DI,BX         ; DI décalé
0597 MOV  SI,05F8       ;
059A CLC                ;
059B:BOUCLE3            ; ADC sur
059B CS:                ; but faible
059C MOV  AX,[SI]       ;
059E ES:                ;
059F ADC  [DI],AX       ;
05A1 INC  SI            ;
05A2 INC  SI            ;
05A3 INC  DI            ;
05A4 INC  DI            ;
05A5 LOOP 059B          ; -> BOUCLE3
05A7 POP  CX            ; faut-il
05A8 PUSHF              ; ADC sur
05A9 MOV  AX,0008       ; but fort?
05AC SUB  AX,CX         ; calcul
05AE CMP  AX,0000       ; test
05B1 JZ   05E3          ; non: suite
05B3 MOV  CX,AX         ; oui:
05B5 CS:                ; initaliser
05B6 LES  DI,[0A30]     ;
05BA POPF               ;
05BB:BOUCLE4            ; ADC sur
05BB CS:                ; but fort
05BC MOV  AX,[SI]       ;
```

```
05BE ES:                ;
05BF ADC  [DI],AX       ;
05C1 INC  SI            ;
05C2 INC  SI            ;
05C3 INC  DI            ;
05C4 INC  DI            ;
05C5 LOOP 05BB          ; -> BOUCLE4
05C7 ADC  DX,+00        ; report ret.
05CA CMP  DX,+00        ; retenue?
05CD JZ   05D9          ; non: suite
05CF ES:                ;
05D0 ADD  [DI],DX       ; (pas ADC)
05D2 JNB  05D9          ; -> suite
05D4 INC  DI            ; encore ret.
05D5 INC  DI            ;
05D6 ES:                ;
05D7 INC  WO [DI]       ; +1
05D9 INC  BX            ;
05DA INC  BX            ;
05DB CMP  BX,+10        ; tout fini?
05DE JZ   05F0          ; oui: sortie
05E0 JMP  0554          ; -> BOUCLE1
05E3 POPF               ;
05E4 CMP  DX,+00        ; retenue?
05E7 JZ   05D9          ; non : fin
05E9 CS:                ; oui : ES:DI
05EA LES  DI,[0A30]     ; s. but fort
05EE JMP  05CF          ;
05F0 RET                ;
```

0610 Multiplication d'un registre par un secteur.

```
0610 PUSH SI            ; SI et DI
0611 PUSH DI            ; pour la fin
0612 MOV  AX,CS         ; initaliser
0614 CS:                ; adresses
0615 MOV  [0A32],AX     ; de retenue
0618 CS:                ; MULT (2 s.)
0619 MOV  WO [0A30],0700;
061F PUSH DI            ; ret. à 00:
0620 CS:                ;
0621 MOV  DI,[0A30]     ; ES:DI
0625 MOV  ES,AX         ; initialisé
0627 MOV  CX,0008       ; et CX, DF
062A XOR  AX,AX         ; pour STOSW
062C CLD                ;
062D REPZ               ;
062E STOSW              ;
062F POP  DI            ; DI retrouvé
0630 CS:                ; init. MULT
0631 MOV  AX,[SI+4000]; n° 1er s.
0635 PUSH AX            ;
0636 AND  AX,0FFF       ; numéro
0639 MOV  CL,04         ; converti
063B SHL  AX,CL         ; en adresse
063D CS:                ; absolue
063E MOV  [0A24],AX     ; via la
0641 POP  BX            ; table 3800
0642 MOV  CL,0B         ; et
0644 SHR  BX,CL         ; enregistré
0646 CS:                ; en
0647 MOV  AX,[BX+3800]; [A26]:[A24]
064B CS:                ;
064C MOV  [0A26],AX     ;
064F CS:                ;
0650 MOV  AX,[DI+4000]; n° 1er s.
0654 PUSH AX            ; but faible
0655 AND  AX,0FFF       ;
0658 MOV  CL,04         ; même
065A SHL  AX,CL         ; conversion
065C CS:                ; stockée
065D MOV  [0A2C],AX     ; en
0660 POP  BX            ; [A2E]:[A2C]
0661 MOV  CL,0B         ;
0663 SHR  BX,CL         ;
0665 CS:                ;
0666 MOV  AX,[BX+3800]; 
066A CS:                ;
066B MOV  [0A2E],AX     ;
066E CLC                ; ret. ADC
066F NOP                ;
0670:BOUCLE             ; s. mltcande
0670 PUSHF              ; r. ADC sauv
0671 CS:                ;
0672 LDS  SI,[0A30]     ; r. MULT
0676 CS:                ; recopiée
0677 LES  DI,[0A30]     ; sur tampon:
067B ADD  DI,+10        ;
067E MOV  CX,0008       ;
0681 CLD                ;
0682 REPZ               ;
0683 MOVSW              ;
0684 CALL 0510          ; MULT
0687 CS:                ; ES:DI:
0688 LES  DI,[0A2C]     ; s.but faib.
068C CS:                ; DS:SI:
068D LDS  SI,[0A30]     ; ret. MULT
0691 ADD  SI,+10        ; précédente
0694 POPF               ; r. pour ADC
0695 CALL 0200          ; ADC
0698 PUSHF              ; ret. sauvée
0699 CS:                ; nouveaux
069A LDS  SI,[0A24]     ; secteurs
069E MOV  CL,04         ; mult.cande
06A0 SHR  SI,CL         ; mult.cateur
06A2 SUB  SI,01C8       ;
06A6 SHL  SI,1          ;
06A8 MOV  AX,[SI]       ;
06AA CMP  AX,0100       ; (SI) fini?
06AD JZ   06F8          ; oui: fin
06AF PUSH AX            ; non:
06B0 AND  AX,0FFF       ; au tour du
06B3 MOV  CL,04         ; secteur
06B5 SHL  AX,CL         ; but faible
06B7 CS:                ;
06B8 MOV  [0A24],AX     ;
06BB POP  BX            ;
06BC MOV  CL,0B         ;
06BE SHR  BX,CL         ;
06C0 CS:                ;
06C1 MOV  AX,[BX+3800]; 
06C5 CS:                ;
06C6 MOV  [0A26],AX     ;
06C9 CS:                ;
06CA LDS  SI,[0A2C]     ;
06CE MOV  CL,04         ;
06D0 SHR  SI,CL         ;
06D2 SUB  SI,01C8       ;
06D6 SHL  SI,1          ;
06D8 MOV  AX,[SI]       ;
06DA PUSH AX            ;
06DB AND  AX,0FFF       ;
06DE MOV  CL,04         ;
```

```
06E0 SHL  AX,CL        ;
06E2 CS:               ;
06E3 MOV  [0A2C],AX    ;
06E6 POP  BX           ;
06E7 MOV  CL,0B        ;
06E9 SHR  BX,CL        ;
06EB CS:               ;
06EC MOV  AX,[BX+3800];
06F0 CS:               ;
06F1 MOV  [0A2E],AX    ;
06F4 POPF              ; r. retrouv.
06F5 JMP  0670         ; -> BOUCLE
06F8 POPF              ; équilibre
06F9 POP  DI           ; pile
06FA POP  SI           ;
06FB CLC               ; R.A.S.
06FC RET               ; retour
```

0728: Réservation d'un nouveau secteur.

en sortie de cette procédure:
ES:DI: adresse absolue secteur
AX: son numéro TAS complet
BX: son numéro de zone

```
0728 XOR  BX,BX        ;
072A CS:               ;
072B MOV  CX,[3840]    ;
072F:BOUCLE            ; recherche
072F CS:               ; zone libre
0730 MOV  AX,[BX+3820]; AX := taux
0734 CMP  AX,0000      ; taux nul?
0737 JNZ  076C         ; oui: suite
0739 INC  BX           ; au
073A INC  BX           ; suivant
073B LOOP 072F         ; -> BOUCLE
073D MOV  AH,48        ; taux nul:
073F MOV  BX,1000      ; allocation
0742 INT  21           ; demandée
0744 JNB  0748         ; obtenue
0746 STC               ; refusée
0747 RET               ;
0748 CS:               ; trouver
0749 MOV  BX,[3840]    ; le numéro
074D SHL  BX,1         ; adéquat
074F CS:               ; et inscrip.
0750 MOV  [BX+3800],AX; seg. zone
0754 CS:               ; inscript.
0755 MOV  WO [BX+3820],0E38; taux
075B CS:               ; mise à jour
075C INC  WO [3840]    ; compteur
0760 MOV  ES,AX        ; ES sur zone
0762 XOR  DI,DI        ; initialiser
0764 XOR  AX,AX        ; la zone
0766 MOV  CX,8000      ; à 00
0769 CLD               ; (8000 mots)
076A REPZ              ;
076B STOSW             ;
076C CS:               ; recherche
076D MOV  AX,[BX+3800]; sect. libre
0771 MOV  ES,AX        ; ES sur zone
0773 XOR  DI,DI        ; DI: déb.TAS
0775 MOV  CX,0E38      ; on cherche
0778 XOR  AX,AX        ; marque 0000
077A CLD               ;
077B REPNZ             ;
077C SCASW             ;
077D JZ   0782         ; trouvé
077F STC               ; non: erreur
0780 RET               ;
0781 NOP               ;
0782 DEC  DI           ; corriger
0783 DEC  DI           ; DI
0784 MOV  AX,DI        ; transformé
0786 SHR  AX,1         ; en n° TAS
0788 ADD  AX,01C8      ;
078B PUSH AX           ; sauver AX
078C MOV  CL,0B        ; adjonction
078E PUSH BX           ; du n° zone
078F SHL  BX,CL        ;
0791 OR   AX,BX        ; n°TAS comp.
0793 POP  BX           ; BX retrouvé
0794 POP  SI           ;
0795 MOV  CL,04        ;
0797 SHL  SI,CL        ; ES:DI prêt
0799 CLC               ; R.A.S.
079A RET               ;
```

07A0: Produit de deux registres.

```
07A0 PUSH SI           ; sauver les
07A1 PUSH BX           ; adr. rép.
07A2 PUSH DI           ; des reg.
07A3 MOV  BP,SP        ; fixation
07A5 SUB  BP,+24       ; réserve
07A8 CLI               ; sur pile
07A9 MOV  SP,BP        ;
07AB STI               ;
07AC CALL 07F0         ; init. AUX
07AF MOV  BX,[BP+2C]   ; ad. rép. m
07B2 CS:               ;
07B3 MOV  CX,[BX+4002]; CX := |m|
07B7 MOV  SI,[BP+2E]   ; ad. rép. M
07BA MOV  DI,[BP+00]   ; ad.rép. AUX
07BD:BOUCLE            ; sur sec. m
07BD PUSH SI           ; sauver
07BE PUSH DI           ; les
07BF PUSH CX           ; paramètres
07C0 MOV  AX,[BP+22]   ; adresse
07C3 CS:               ; absolue du
07C4 MOV  [0A28],AX    ; secteur
07C7 MOV  AX,[BP+24]   ; courant
07CA CS:               ; mult.cateur
07CB MOV  [0A2A],AX    ; soit s
07CE PUSH BP           ;
07CF CALL 0610         ; AUX := M*s
07D2 POP  BP           ;
07D3 CALL 0958         ; P := P+AUXd
07D6 CALL 0A40         ; s,d nouv.
07D9 POP  CX           ; récuperer
07DA POP  DI           ; les
07DB POP  SI           ; paramètres
07DC LOOP 07BD         ; -> BOUCLE
07DE MOV  BP,SP        ; annuler
07E0 ADD  BP,+2A       ; réservat.
07E3 CLI               ; sur pile
07E4 MOV  SP,BP        ;
07E6 STI               ;
07E7 POP  DI           ; adr. rép.
07E8 POP  BX           ; registres
07E9 POP  SI           ; retrouvées
07EA RET               ;
```

07F0: Initialisation du registre auxiliaire.

```
07F0 PUSH CS           ;
07F1 POP  DS           ; DS := CS
```

```
07F2 MOV  DI,4000      ; cherche
07F5 MOV  CX,0010      ; entrée
07F8:BOUCLE1           ; répertoire
07F8 MOV  AX,[DI]      ; lire entrée
07FA CMP  AX,0000      ; libre?
07FD JZ   080A         ; oui: suite
07FF ADD  DI,+04       ; non:
0802 LOOP 07F8         ; -> BOUCLE1
0804 CALL 0FD0         ; non trouvé:
0807 STC               ; erreur
0808 RET               ;
0809 NOP               ;
080A SUB  DI,4000      ; adr. entrée
080E MOV  [BP+00],DI   ; AUX
0811 CALL 0728         ; réser. sec.
0814 JNB  081C         ; oui: suite
0816 CALL 0FD0         ; RAM pleine
0819 STC               ;
081A RET               ;
081B NOP               ;
081C ES:               ; marquer s.
081D MOV  WO [DI],0100; comme der
0821 MOV  [BP+0A],DI   ; dépl. TAS
0824 PUSH AX           ;
0825 MOV  AX,ES        ;
0827 MOV  [BP+0C],AX   ; seg. TAS
082A MOV  [BP+08],AX   ; seg. sect.
082D MOV  [BP+04],AX   ; seg. sect.
0830 CS:               ; décrémenter
0831 DEC  WO [BX+3820]; le taux
0835 POP  AX           ; (n° TAS)
0836 MOV  BX,[BP+00]   ; adr. rép.
0839 CS:               ; n° TAS sec.
083A MOV  [BX+4000],AX; dans rép.
083E MOV  AX,0001      ; initialiser
0841 CS:               ; longueurs
0842 MOV  [BX+4002],AX; à 1 en rép.
0846 CS:               ;
0847 SHR  BX,1         ; idem. pour
0849 MOV  [BX+4040],AX; mots der.s.
084D MOV  [BP+02],SI   ; dépl. sect.
0850 MOV  [BP+06],SI   ; dépl. sect.
0853 CALL 0728         ; réserver
0856 JNB  085E         ; oui: suite
0858 CALL 0FD0         ; non: RAM
085B STC               ; pleine
085C RET               ;
085D NOP               ;
085E ES:               ; marquage
085F MOV  WO [DI],0100; comme der.
0863 MOV  [BP+12],DI   ; (s. suppl.)
0866 PUSH AX           ;
0867 MOV  AX,ES        ;
0869 MOV  [BP+14],AX   ; seg. icelui
086C MOV  [BP+10],AX   ; idem.
086F CS:               ; décompté
0870 DEC  WO [BX+3820]; du total
0874 POP  AX           ;
0875 MOV  [BP+16],AX   ; n°TAS comp.
0878 MOV  [BP+0E],SI   ; dépl.s.spp.
087B MOV  SI,[BP+2E]   ; ad. rép. M
087E CS:               ;
087F MOV  SI,[SI+4000]; n° 1er sec.
0883:BOUCLE2           ; copie à 00
0883 PUSH SI           ;
0884 AND  SI,0FFF      ; SI devient
0888 SUB  SI,01C8      ; l'adresse
088C SHL  SI,1         ; TAS du sec.
088E POP  BX           ; BX := ex-SI
088F AND  BX,F000      ; BX devient
0893 MOV  CL,0B        ; adresse
0895 SHR  BX,CL        ; de zone
0897 CS:               ; dont le
0898 MOV  AX,[BX+3800]; segment va
089C MOV  DS,AX        ; dans DS
089E MOV  AX,[SI]      ; entrée TAS
08A0 CMP  AX,0100      ; der. sect.?
08A3 JZ   08DE         ; oui: fin
08A5 PUSH AX           ; non:
08A6 CALL 0728         ; réser.sect.
08A9 JNB  08B0         ; obtenu
08AB CALL 0FD0         ; refusé: RAM
08AE STC               ; pleine
08AF RET               ;
08B0 PUSH AX           ; paramètres
08B1 PUSH DI           ; 0728
08B2 PUSH ES           ; sur pile
08B3 LES  DI,[BP+0A]   ; ad. TAS AUX
08B6 ES:               ; inscript.
08B7 MOV  [DI],AX      ; n° du suiv.
08B9 POP  AX           ; (=ES 0728)
08BA MOV  [BP+08],AX   ; seg. rangé
08BD MOV  [BP+0C],AX   ; idem
08C0 MOV  ES,AX        ; ES sur zone
08C2 POP  DI           ; adr. TAS
08C3 ES:               ; marqué
08C4 MOV  WO [DI],0100; comme der.
08C8 CS:               ; mise à jour
08C9 DEC  WO [BX+3820]; compteur
08CD MOV  [BP+0A],DI   ; dépl.TAS
08D0 POP  AX           ; (n°complet)
08D1 AND  AX,0FFF      ; transformé
08D4 MOV  CL,04        ; en
08D6 SHL  AX,CL        ; déplacement
08D8 MOV  [BP+06],AX   ; noté
08DB POP  SI           ; n° secteur
08DC JMP  0883         ; -> BOUCLE2
08DE MOV  SI,[BP+2A]   ; DI initial
08E1 CS:               ;
08E2 MOV  SI,[SI+4000]; n° secteur
08E6 PUSH SI           ; ce secteur
08E7 AND  SI,0FFF      ; est le
08EB MOV  D1,SI        ; premier du
08ED SUB  SI,01C8      ; décalage au
08F1 SHL  SI,1         ; départ
08F3 MOV  [BP+1C],SI   ; dépl. noté
08F6 POP  BX           ; (n° TAS)
08F7 MOV  [BP+20],BX   ; noté
08FA AND  BX,F000      ;
08FE MOV  CL,0B        ;
0900 SHR  BX,CL        ;
0902 CS:               ;
0903 MOV  AX,[BX+3800]; seg. zone
0907 MOV  [BP+1A],AX   ; noté
090A MOV  [BP+1E],AX   ; idem
090D MOV  CL,04        ;
090F SHL  DI,CL        ; déplacement
0911 MOV  [BP+18],DI   ; noté
0914 MOV  SI,[BP+2C]   ; BX initial
0917 CS:               ;
0918 MOV  SI,[SI+4000]; n° secteur
091C PUSH SI           ; ce secteur
091D AND  SI,0FFF      ; est le
0921 MOV  DI,SI        ; premier
```

```
0923 SUB  SI,01C8      ; secteur
0927 SHL  SI,1         ; mult.cateur
0929 MOV  [BP+26],SI   ; dép. TAS
092C POP  BX           ;
092D AND  BX,F000      ;
0931 MOV  CL,0B        ;
0933 SHR  BX,CL        ;
0935 CS:               ;
0936 MOV  AX,[BX+3800]; seg. lu
093A MOV  [BP+24],AX   ; et noté
093D MOV  [BP+28],AX   ; idem
0940 MOV  CL,04        ;
0942 SHL  DI,CL        ; déplacement
0944 MOV  [BP+22],DI   ; noté
0947 CS:               ; réécriture
0948 MOV  WO [0486],04EB; 0470
094E CS:               ; initialiser
094F MOV  WO [09FC],0000; décal. DX
0955 CLC               ;
0956 RET               ;
```

0958: Appel à l'addition avec décalage.

```
0958 CS:               ;
0959 LES  DI,[0A30]    ; ret. MUL
095D ADD  DI,+0E       ; DI en fin
0960 MOV  CX,0008      ; 08 mots
0963 XOR  AX,AX        ;
0965 STD               ; vers le bas
0966 REPZ              ; chercher
0967 SCASW             ; non nul
0968 JZ   0987         ; ret. nulle
096A CS:               ; ret. > 0
096B LES  DI,[BP+0E]   ; sect. supp.
096E CS:               ;
096F LDS  SI,[0A30]    ; ret. MUL
0973 MOV  CX,0008      ;
0976 CLD               ; la ret. MUL
0977 REPZ              ; est copiée
0978 MOVSW             ; sur sect.
0979 CS:               ; supplém.
097A INC  BY [09FE]    ; on le note
097E LES  DI,[BP+0A]   ; le suppl.
0981 MOV  AX,[BP+16]   ; devient
0984 ES:               ; le der. s.
0985 MOV  [DI],AX      ; de AUX
0987 MOV  BX,[BP+00]   ; ad.rép. AUX
098A CS:               ;
098B MOV  AX,[BX+4002]; |AUX|
098F CS:               ; noté
0990 MOV  [0A00],AX    ; dans [A00]
0993 MOV  AX,[BP+02]   ; s. source
0996 MOV  CL,04        ; on fait
0998 SHR  AX,CL        ; son n° TAS
099A CS:               ; 'aux 3/4'
099B MOV  [0A04],AX    ; noté (ADC)
099E MOV  AX,[BP+18]   ; idem, but
09A1 MOV  CL,04        ;
09A3 SHR  AX,CL        ; 3/4 n° TAS
09A5 CS:               ;
09A6 MOV  [0A06],AX    ; noté
09A9 MOV  AX,[BP+20]   ; son n° zone
09AC AND  AX,F000      ; (extrait du
09AF SHL  AX,1         ; n° TAS
09B1 CS:               ; complet)
09B2 MOV  [0A08],AX    ; noté
09B5 MOV  AX,[BP+2A]   ; DI initial
09B8 CS:               ;
09B9 MOV  [0A0A],AX    ; noté
09BC LDS  SI,[BP+02]   ; source ADC
09BF LES  DI,[BP+18]   ; but ADC
09C2 MOV  AX,09D0      ; adr. retour
09C5 PUSH AX           ; sur pile
09C6 PUSH CX           ; équil. pile
09C7 CS:               ; initialiser
09C8 MOV  DX,[09FC]    ; DX
09CC CLC               ; puis CF
09CD JMP  04A4         ; appel ADC
09D0 JNB  09D8         ; ADC faite
09D2 CALL 0FD0         ; non: RAM
09D5 STC               ; pleine
09D6 RET               ;
09D7 NOP               ;
09D8 CS:               ; s. suppl.
09D9 CMP  BY [09FE],00; utilisé?
09DE JZ   09F9         ; non: fini
09E0 LES  DI,[BP+0A]   ; oui:
09E3 ES:               ; remettre
09E4 MOV  WO [DI],0100; der AUX
09E8 LES  DI,[BP+0E]   ; comme
09EB MOV  CX,0008      ; auparavant
09EE XOR  AX,AX        ; et suppl.
09F0 CLD               ; à 00
09F1 REPZ              ;
09F2 STOSW             ;
09F3 CS:               ; annuler
09F4 MOV  BY [09FE],00; témoin
09F9 CLC               ;
09FA RET               ;
```

variable du programme 0958

```
09FC 00 00
09FE 00
```

0A40: Passage au secteur suivant du multiplicateur.

```
0A40 LES  DI,[BP+26]   ; TAS de m
0A43 ES:               ;
0A44 MOV  AX,[DI]      ; lire entrée
0A46 CMP  AX,0100      ; dernier s.?
0A49 JNZ  0A5E         ; non: suite
0A4B CS:               ; oui: réécr.
0A4C MOV  WO [0486],47E8; en 0470
0A52 CMP  WO [BP-06],+01; vrai?
0A56 JZ   0AC1         ; oui: fin
0A58 CALL 0FD0         ; non: erreur
0A5B STC               ;
0A5C RET               ;
0A5D NOP               ;
0A5E PUSH AX           ; extraction
0A5F AND  AX,0FFF      ; adr. sect.
0A62 PUSH AX           ; suivant
0A63 MOV  CL,04        ; de m
0A65 SHL  AX,CL        ;
0A67 MOV  [BP+22],AX   ; dépl. noté
0A6A POP  AX           ;
0A6B SUB  AX,01C8      ;
0A6E SHL  AX,1         ;
0A70 MOV  [BP+26],AX   ; dépl. TAS
0A73 POP  BX           ;
0A74 AND  BX,F000      ;
0A78 MOV  CL,0B        ;
0A7A SHR  BX,CL        ;
0A7C CS:               ;
0A7D MOV  AX,[BX+3800]; seg. trouvé
```

```
0A81 MOV  [BP+24],AX  ; noté (sec.)
0A84 MOV  [BP+28],AX  ; noté (TAS)
0A87 LES  DI,[BP+1C]  ; idem sect.
0A8A ES:              ; décalage:
0A8B MOV  AX,[DI]     ;
0A8D CMP  AX,0100     ; ∃ suivant?
0A90 JNZ  0A98        ; oui: suite
0A92 CALL 0FD0        ; non: erreur
0A95 STC              ;
0A96 RET              ;
0A97 NOP              ;
0A98 PUSH AX          ;
0A99 AND  AX,0FFF     ;
0A9C PUSH AX          ;
0A9D MOV  CL,04       ; d'abord
0A9F SHL  AX,CL       ; dépl. sect.
0AA1 MOV  [BP+18],AX  ; noté
0AA4 POP  AX          ;
0AA5 SUB  AX,01C8     ; puis
0AA8 SHL  AX,1        ; dépl. TAS
0AAA MOV  [BP+1C],AX  ; noté
0AAD POP  BX          ;
0AAE AND  BX,F000     ;
0AB2 MOV  CL,0B       ;
0AB4 SHR  BX,CL       ;
0AB6 CS:              ;
0AB7 MOV  AX,[BX+3800]; seg. trouvé
0ABB MOV  [BP+1A],AX  ; noté (sec.)
0ABE MOV  [BP+1E],AX  ; noté (TAS)
0AC1 CS:              ; incrémenter
0AC2 INC  WO [09FC]   ; décalage DX
0AC6 CLC              ;
0AC7 RET              ;
```

DIVISION EUCLIDIENNE DE DEUX REGISTRES.

0AD0 : Programme de contrôle.

```
0AD0 CS:              ; lire
0AD1 MOV  DX,[BX+4002]; [d]
0AD5 CMP  DX,+01      ; [d] > 1?
0AD8 JA   0AEA        ; oui: suite
0ADA SHR  BX,1        ; non:
0ADC CS:              ; lire nombre
0ADD MOV  AX,[BX+4040]; de mots
0AE1 CMP  AX,0001     ; nbre > 1?
0AE4 JA   0AEA        ; oui: suite
0AE6 CALL 13A0        ; non: DIV
0AE9 RET              ; type mot
0AEA MOV  BP,SP       ;
0AEC SUB  BP,0090     ; installer
0AF0 CLI              ; une réserve
0AF1 MOV  SP,BP       ; sur la pile
0AF3 STI              ; (090 mots)
0AF4 CS:              ; lire
0AF5 MOV  CX,[SI+4002]; [D]
0AF9 CMP  CX,DX       ; [D] >= [d]?
0AFB JNB  0B03        ; oui: suite
0AFD CALL 1259        ; non: cas
0B00 JMP  0B30        ; q=0, r=D
0B02 NOP              ;
0B03 CALL 0B50        ; initialiser
0B06:BOUCLE1          ; selon [q]
0B06 PUSH CX          ; (compteur)
0B07 MOV  CX,0010     ; un secteur
0B0A:BOUCLE2          ; seize oct.
0B0A PUSH CX          ; (compteur)
0B0B CALL 0DC8        ; u, v, w
0B0E PUSH AX          ;
0B0F CALL 0E00        ; D:=D-u_m b^m d
0B12 SHR  BY [BP+0D],1; w = 0?
0B15 JNB  0B24        ; oui: suite
0B17 CALL 0F20        ; non: vérif.
0B1A POP  AX          ;
0B1B JB   0B24        ; correct: ->
0B1D PUSH AX          ; non:
0B1E CALL 0E90        ; corriger
0B21 POP  AX          ;
0B22 INC  AX          ; u_n = u_n+1
0B23 NOP              ;
0B24 POP  CX          ; (compteur)
0B25 CALL 1060        ; mise à jour
0B28 LOOP 0B0A        ; -> BOUCLE2
0B2A POP  CX          ; (compteur)
0B2B CALL 11D8        ;
0B2E LOOP 0B06        ; -> BOUCLE1
0B30 MOV  AX,[BP+0092]; signal DIV
0B34 SHR  AX,1        ; à virgule
0B36 JNB  0B3B        ; non: suite
0B38 CALL 13E7        ; DIV virgule
0B3B CALL 0E30        ; résultat
0B3E MOV  BP,SP       ;
0B40 ADD  BP,0090     ; démanteler
0B44 CLI              ; la réserve
0B45 MOV  SP,BP       ; sur pile
0B47 STI              ;
0B48 RET              ;
```

0B50: Initialisation.

```
0B50 MOV  AH,48       ; réserver
0B52 MOV  BX,1000     ; 01000 para.
0B55 INT  21          ;
0B57 JNB  0B5E        ; oui: suite
0B59 CALL 0FD0        ; RAM pleine
0B5C STC              ;
0B5D RET              ;
0B5E MOV  [BP+00],AX  ; seg. zone
0B61 MOV  [BP+10],SI  ; noter les
0B64 MOV  [BP+30],BX  ; adr. rép.
0B67 MOV  [BP+50],DI  ; de D, d, q
0B6A CALL 0B94        ; relevé D, d
0B6D CALL 0BF8        ; calcul |q|
0B70 CALL 0CA8        ; trouver c
0B73 CALL 0CE8        ; trouver a
0B76 CALL 0D30        ; calculer Du
0B79 CALL 0D90        ; d_u, d_c, q_c
0B7C MOV  CX,[BP+52]  ; compteur
0B7F MOV  BY [BP+0C],00; cas normal
0B83 CMP  CX,+01      ; [q] > 1?
0B86 JA   0B92        ; oui: suite
0B88 INC  BY [BP+0C]  ; non: signal
0B8B MOV  AX,[BP+54]  ; AX := {q}
0B8E CS:              ; réécrire
0B8F MOV  [0B08],AX   ; en 0AD0
0B92 RET              ;
```

0B94: Relevés de D et d.

```
0B94 MOV  ES,AX       ; ES sur zone
0B96 XOR  DI,DI       ; débute à 00
0B98 MOV  [BP+02],DI  ; noter début
0B9B CALL 0BB8        ; cas de D
0B9E DEC  DI          ; précédent:
0B9F DEC  DI          ; fin de D
0BA0 MOV  [BP+1A],DI  ; noter
0BA3 INC  DI          ;
0BA4 INC  DI          ;
```

```
0BA5 MOV  [BP+06],DI  ; idem
0BA8 MOV  [BP+40],DI  ; n°ch. du
0BAB MOV  SI,[BP+30]  ; pour 0BB8
0BAE CALL 0BB8        ; cas de d
0BB1 DEC  DI          ;
0BB2 DEC  DI          ;
0BB3 MOV  [BP+3A],DI  ; n°ch. dt
0BB6 RET              ;
```

0BB8: Relevé des secteurs d'un registre.

```
0BB8 CS:              ; SI: adresse
0BB9 MOV  AX,[SI+4000]; répertoire
0BBD NOP              ; de R
0BBE:BOUCLE           ; selon TAS
0BBE CMP  AX,0100     ; sect. der.?
0BC1 JZ   0BF0        ; oui: -> fin
0BC3 ES:              ; non: inscr.
0BC4 MOV  [DI],AX     ; n°TAS plein
0BC6 PUSH AX          ; retenir
0BC7 AND  AX,0FFF     ; pour
0BCA SUB  AX,01C8     ; conversion
0BCD SHL  AX,1        ; adresse TAS
0BCF MOV  SI,AX       ; déplacement
0BD1 POP  BX          ; (AX repris)
0BD2 AND  BX,F000     ; puis
0BD6 SHL  BX,1        ; segment
0BD8 CS:              ; cherché en
0BD9 MOV  AX,[BX+3800]; table 3800
0BDD MOV  DS,AX       ; segment
0BDF MOV  AX,[SI]     ; lire n°TAS
0BE1 CMP  DI,-02      ; z. pleine?
0BE4 JB   0BEC        ; non: suite
0BE6 CALL 0FD0        ; oui: contr.
0BE9 STC              ; logiciel
0BEA RET              ;
0BEB NOP              ;
0BEC INC  DI          ; mise à jour
0BED INC  DI          ; de DI
0BEE JMP  0BBE        ; -> BOUCLE
0BF0 RET              ; retour
```

0BF8: Longueur a priori du quotient.

```
0BF8 MOV  BX,[BP+10]  ; rappel adr.
0BFB CS:              ; répertoire
0BFC MOV  AX,[BX+4002]; lire [D]
0C00 MOV  [BP+12],AX  ; noté
0C03 SHR  BX,1        ; ajust. 4040
0C05 CS:              ; calcul de
0C06 MOV  BX,[BX+4040]; {D} depuis
0C0A SHL  BX,1        ; la taille
0C0C PUSH BX          ; mots du
0C0D MOV  SI,[BP+1A]  ; dernier
0C10 CALL 0C88        ; lire (CALL)
0C13 MOV  [BP+18],DS  ; seg. noté
0C16 POP  BX          ;
0C17 ADD  SI,BX       ; SI sur le
0C19 DEC  SI          ; mot
0C1A DEC  SI          ; dernier
0C1B MOV  AX,[SI]     ; lu en AX
0C1D CMP  AH,00       ; 2 octets?
0C20 JNZ  0C25        ; oui: suite
0C22 DEC  BX          ; non: corr.
0C23 JMP  0C26        ; -> suite
0C25 INC  SI          ; 1 octet
0C26 MOV  [BP+14],BX  ; noter {D}
0C29 MOV  [BP+16],SI  ; dépl. Dt
0C2C MOV  BX,[BP+30]  ; 0C2C à 0C60
0C2F CS:              ; même chose
0C30 MOV  AX,[BX+4002]; pour d
0C34 MOV  [BP+32],AX  ;
0C37 SHR  BX,1        ; ajust. 4040
0C39 CS:              ;
0C3A MOV  BX,[BX+4040];
0C3E SHL  BX,1        ;
0C40 PUSH BX          ;
0C41 MOV  SI,[BP+3A]  ;
0C44 CALL 0C88        ;
0C47 MOV  [BP+38],DS  ; seg. dt
0C4A POP  BX          ;
0C4B ADD  SI,BX       ;
0C4D DEC  SI          ;
0C4E DEC  SI          ;
0C4F MOV  AX,[SI]     ;
0C51 CMP  AH,00       ;
0C54 JNZ  0C59        ;
0C56 DEC  BX          ;
0C57 JMP  0C5A        ;
0C59 INC  SI          ;
0C5A MOV  [BP+34],BX  ; {d} noté
0C5D MOV  [BP+36],SI  ; dépl. dt
0C60 MOV  AX,[BP+14]  ; AX := {D}
0C63 MOV  CX,[BP+34]  ; CX := {d}
0C66 MOV  BX,[BP+12]  ; BX := [D]
0C69 INC  BX          ; corr. BX
0C6A INC  AX          ; corr. AX
0C6B CMP  AX,CX       ; retenue?
0C6D JNB  0C73        ; non: suite
0C6F ADD  AX,0010     ; oui:
0C72 DEC  BX          ;
0C73 SUB  AX,CX       ; {q} calculé
0C75 MOV  [BP+54],AX  ; et noté
0C78 SUB  BX,[BP+32]  ; [q] calculé
0C7B MOV  [BP+52],BX  ; et noté
0C7E RET              ;
```

0C88: adresse absolue d'un secteur (0BF8, 0CA8, 0CE8, 0D30, 0D90, 0EA0, 0EC0, 0EE0, 0F00).

```
0C88 ES:              ; lire n°TAS
0C89 MOV  BX,[SI]     ; (SI n° ch)
0C8B: autre entrée (13D0)
0C8B PUSH BX          ; converti en
0C8C AND  BX,F000     ; adresse
0C90 MOV  CL,0B       ; absolue du
0C92 SHR  BX,CL       ; secteur
0C94 CS:              ;
0C95 MOV  AX,[BX+3800];
0C99 MOV  DS,AX       ; DS prêt
0C9B POP  SI          ;
0C9C AND  SI,0FFF     ;
0CA0 MOV  CL,04       ;
0CA2 SHL  SI,CL       ; SI prêt
0CA4 RET              ;
```

0CA8: Calcul de c.

```
0CA8 LES  DI,[BP+36]  ; ES:DI en dt
0CAB ES:              ; lire cet
0CAC MOV  CH,[DI]     ; octet
0CAE TEST DI,000F     ; dernier?
0CB2 JNZ  0CCA        ; non: suite
0CB4 PUSH CX          ; si:
0CB5 MOV  ES,[BP+00]  ; chercher le
0CB8 MOV  SI,[BP+08]  ; secteur
0CBB SUB  SI,+04      ; précédent
```

```
0CBE CALL 0C88          ; adr. abs.
0CC1 ADD  SI,+0F        ; octet dern.
0CC4 POP  CX            ; CH restauré
0CC5 MOV  CL,[SI]       ; noté en CL
0CC7 JMP  0CCE          ; -> suite
0CC9 NOP                ; cas simple:
0CCA DEC  DI            ; octet
0CCB ES:                ; précédent
0CCC MOV  CL,[DI]       ; noté en CL
0CCE CS:                ; réécrire
0CCF MOV  [0DEA],CX     ; 2ème place
0CD3 CMP  CX,-01        ; c = 0FFFF?
0CD6 JNZ  0CE0          ; non: suite
0CD8 CS:                ; si: changer
0CD9 ADD  WO [0B0C],+0E; adr. CALL
0CDE RET                ; terminé
0CDF NOP                ; cas usuel:
0CE0 INC  CX            ; c+1
0CE1 CS:                ; réécrit
0CE2 MOV  [0DCF],CX     ; 1ère place
0CE6 RET                ; terminé
```

0CE8: Calcul de a.

```
0CE8 LES  DI,[BP+16]    ; ES:DI en Dt
0CEB ES:                ; lire cet
0CEC MOV  AH,[DI]       ; octet
0CEE TEST DI,000F       ; dernier?
0CF2 JNZ  0D12          ; non: suite
0CF4 PUSH AX            ; si:
0CF5 MOV  ES,[BP+00]    ; lire
0CF8 MOV  SI,[BP+06]    ; le secteur
0CFB SUB  SI,+04        ; précédent
0CFE MOV  [BP+20],SI    ; ch. noté
0D01 CALL 0C88          ;
0D04 MOV  [BP+1E],DS    ; seg. noté
0D07 ADD  SI,+0F        ; SI sur
0D0A MOV  [BP+1C],SI    ; oct. der.
0D0D POP  AX            ; AH retrouvé
0D0E MOV  AL,[SI]       ; AL lu
0D10 JMP  0D22          ; -> suite
0D12 DEC  DI            ; cas usuel:
0D13 ES:                ;
0D14 MOV  AL,[DI]       ; AL lu
0D16 MOV  [BP+1E],ES    ; seg. Dp
0D19 MOV  [BP+1C],DI    ; dépl. Dp
0D1C PUSH [BP+04]       ;
0D1F POP  [BP+20]       ; n°ch. Dp
0D22 MOV  [BP+60],AX    ; fenêtre
0D25 MOV  WO [BP+62],0000; (suite)
0D2A RET                ;
```

0D30: Calcul de D_u.

```
0D30 MOV  SI,[BP+52]    ; SI := [q]
0D33 PUSH SI            ; [q] gardé
0D34 DEC  SI            ; donc SI ≥ 0
0D35 SHL  SI,1          ; (SI n°ch.)
0D37 MOV  [BP+26],SI    ; n°ch. noté
0D3A MOV  [BP+2C],SI    ; pour Dc
0D3D MOV  ES,[BP+00]    ; prép. 0C88
0D40 CALL 0C88          ; adr. abs.
0D43 MOV  [BP+24],DS    ; seg. Du
0D46 MOV  [BP+2A],DS    ; seg. Dc
0D49 MOV  AX,[BP+54]    ; AX := [q]
0D4C DEC  AX            ; -> adresse
0D4D ADD  SI,AX         ; corr. SI
0D4F MOV  [BP+22],SI    ; dépl. Du
0D52 MOV  [BP+28],SI    ; dépl. Dc
0D55 MOV  AX,0010       ; indices de
0D58 AND  SI,000F       ; partage
0D5C MOV  [BP+65],SI    ; i65=10-i64
0D5F SUB  AX,SI         ;
0D61 MOV  [BP+64],AL    ; i64 noté
0D64 PUSH AX            ;
0D65 MOV  AX,[BP+14]    ; AX := [D]
0D68 MOV  CX,[BP+34]    ; CX := [d]
0D6B CMP  AX,CX         ; (dernier
0D6D POP  AX            ; secteur)
0D6E JB   0D76          ; 1er cas:
0D70 MOV  [BP+66],CX    ; i66 (i67=0)
0D73 JMP  0D7E          ; -> suite
0D75 NOP                ; 2ème cas:
0D76 MOV  [BP+66],AL    ; i66
0D79 SUB  CL,AL         ; puis
0D7B MOV  [BP+67],CL    ; i67 (≠ 0)
0D7E POP  SI            ; [q] repris
0D7F CMP  SI,+01        ; 1 secteur?
0D82 JA   0D8A          ; non: suite
0D84 PUSH [BP+66]       ; si: [BP+66]
0D87 POP  [BP+64]       ; sur [BP+64]
0D8A RET                ; terminé
```

0D90: Calcul de d_u, d_c et q_c.

```
0D90 MOV  AL,[BP+54]    ;
0D93 MOV  [BP+56],AL    ;
0D96 MOV  ES,[BP+00]    ;
0D99 MOV  SI,[BP+06]    ;
0D9C MOV  [BP+40],SI    ;
0D9F MOV  [BP+46],SI    ;
0DA2 CALL 0C88          ;
0DA5 MOV  [BP+3E],DS    ;
0DA8 MOV  [BP+44],DS    ;
0DAB ADD  SI,[BP+34]    ;
0DAE DEC  SI            ;
0DAF MOV  [BP+3C],SI    ;
0DB2 MOV  [BP+42],SI    ;
0DB5 RET                ;
```

0DC8: PRE-DIVISION (usuel).

```
0DC8 MOV  DX,[BP+62]    ; DX:AX := la
0DCB MOV  AX,[BP+60]    ; fenêtre
0DCE MOV  CX,FFFF       ; c+1 réécrit
0DD1 DIV  CX            ; DIV mot
0DD3 PUSH AX            ; u sauvé
0DD4 JMP  0DDF          ; -> vers v
```

0DD6: PRE-DIVISION (exception).

```
0DD6 MOV  AL,[BP+62]    ; DIV mot
0DD9 MOV  DX,[BP+60]    ; anticipée
0DDC XOR  AH,AH         ;
0DDE PUSH AX            ;
0DDF MOV  DX,[BP+62]    ; calcul de
0DE2 MOV  AX,[BP+60]    ; DX:AX := la
0DE5 INC  AX            ; fenêtre + 1
0DE6 JNZ  0DE9          ; retenue non
0DE8 INC  DX            ; si: corr.
0DE9 MOV  CX,FFFF       ; c réécrit
0DEC DIV  CX            ; DIV mot
0DEE POP  CX            ; CX := u
0DEF SHR  BY [BP+0D],1; sauver bit7
0DF2 CLC                ; prép. w
0DF3 CMP  CX,AX         ; comp. u, v
0DF5 RCL  BY [BP+0D],1; noter w
0DF8 MOV  AX,CX         ; AX := u
0DFA RET                ; terminé
```

0E00: DIVISION PARTIELLE.

```
0E00 MOV  DL,AL        ; sauver u
0E02 PUSH [BP+40]      ; initaliser
0E05 POP  [BP+46]      ; dc
0E08 XOR  BX,BX        ;
0E0A LES  DI,[BP+22]   ; ES:DI: Du
0E0D LDS  SI,[BP+3C]   ; DS:SI: du
0E10:BOUCLE1           ; selon [q]
0E10 PUSH CX           ; (compteur)
0E11 XOR  CH,CH        ;
0E13 MOV  CL,[BP+64]   ; CX := i64
0E16:BOUCLE21          ; lère partie
0E16 MOV  DH,AH        ;
0E18 MOV  AL,[SI]      ; multiplic.
0E1A MUL  DL           ;
0E1C CLC               ;
0E1D RCR  BX,1         ;
0E1F ADC  AL,DH        ;
0E21 RCL  BX,1         ;
0E23 RCL  BX,1         ; soustract.
0E25 ES:               ;
0E26 SBB  [DI],AL      ;
0E28 RCR  BX,1         ;
0E2A INC  SI           ;
0E2B INC  DI           ;
0E2C LOOP 0E16         ; -> BOUCLE21
0E2E MOV  CL,[BP+65]   ; CX := i65
0E31 JCXZ 0E4E         ; (si i65=0)
0E33 CALL 0EA0         ; nouv. s. Dc
0E36:BOUCLE22          ; 2ème partie
0E36 MOV  DH,AH        ;
0E38 MOV  AL,[SI]      ; multiplic.
0E3A MUL  DL           ;
0E3C CLC               ;
0E3D RCR  BX,1         ;
0E3F ADC  AL,DH        ;
0E41 RCL  BX,1         ;
0E43 RCL  BX,1         ; soustract.
0E45 ES:               ;
0E46 SBB  [DI],AL      ;
0E48 RCR  BX,1         ;
0E4A INC  SI           ;
0E4B INC  DI           ;
0E4C LOOP 0E36         ; -> BOUCLE22
0E4E POP  CX           ; (compteur)
0E4F CMP  CX,+01       ; fini?
0E52 JZ   0E6F         ; oui: -> fin
0E54 PUSH CX           ; non:
0E55 CALL 0EC0         ; nouv. s. dc
0E58 CMP  CX,+02       ; av.-der.?
0E5B JNZ  0E69         ; non: suite
0E5D PUSH [BP+64]      ; si: écahnge
0E60 PUSH [BP+66]      ; [BP+64] et
0E63 POP  [BP+64]      ; [BP+66]
0E66 POP  [BP+66]      ; (réservent)
0E69 POP  CX           ; (compteur)
0E6A LOOP 0E10         ; -> BOUCLE1
0E6C PUSH AX           ;
0E6D MOV  AX,[BP+62]   ; fenêtre:
0E70 CMP  AX,0000      ; 2 octets?
0E73 POP  AX           ;
0E74 JZ   0E8B         ; oui: suite
0E76 TEST DI,000F      ; 3: même s.?
0E7A JNZ  0E7F         ; oui: suite
0E7C CALL 0EA0         ; non: Dc
0E7F RCR  BX,1         ; traitement
0E81 ADC  AH,00        ; dernière
0E84 RCL  BX,1         ; décimale
0E86 RCL  BX,1         ;
0E88 ES:               ;
0E89 SBB  [DI],AH      ;
0E8B RET               ; terminé
```

0E90 : Correction du reste excédentaire (0AD0).

```
0E90 MOV  AL,01        ; initialiser
0E92 CALL 0E00         ; progr. 0E00
0E95 RET               ;
```

0EA0 : Secteur suivant de D.

```
0EA0 PUSH AX           ; on sauve
0EA1 PUSH CX           ; les
0EA2 PUSH DX           ; registres
0EA3 PUSH BX           ; généraux
0EA4 PUSH DS           ; pointeur de
0EA5 PUSH SI           ; dc sauvé
0EA6 MOV  ES,[BP+00]   ; préparation
0EA9 MOV  SI,[BP+2C]   ; de l'appel
0EAC INC  SI           ; à 0C88
0EAD INC  SI           ;
0EAE MOV  [BP+2C],SI   ; Dc à jour
0EB1 CALL 0C88         ; DS:SI
0EB4 PUSH DS           ; mis
0EB5 POP  ES           ; sur
0EB6 MOV  DI,SI        ; ES:DI
0EB8 POP  SI           ; pointeur dc
0EB9 POP  DS           ; restauré
0EBA POP  BX           ; registres
0EBB POP  DX           ; généraux
0EBC POP  CX           ; rétablis
0EBD POP  AX           ;
0EBE RET               ;
```

0EC0: Secteur suivant de d.

```
0EC0 PUSH AX           ; registres
0EC1 PUSH CX           ; généraux
0EC2 PUSH DX           ; sauvés
0EC3 PUSH BX           ;
0EC4 PUSH ES           ; pointeur Dc
0EC5 PUSH DI           ; sauvé
0EC6 MOV  ES,[BP+00]   ; préparation
0EC9 MOV  SI,[BP+46]   ; appel 0C88
0ECC INC  SI           ;
0ECD INC  SI           ;
0ECE MOV  [BP+46],SI   ; dc à jour
0ED1 CALL 0C88         ; DS:SI
0ED4 POP  DI           ; pointeur Dc
0ED5 POP  ES           ; restauré
0ED6 POP  BX           ;
0ED7 POP  DX           ; registres
0ED8 POP  CX           ; généraux
0ED9 POP  AX           ; rétablis
0EDA RET               ;
```

0EE0: Secteur précédent de D.

on recopie 0EA0-0EBF en modifiant le mot 0EEC (remplacer INC SI par DEC SI) sous DEBUG :

```
-m 0EA0 L 1F 0EE0
-e 0EEC 4E 4E
-e 0EF2 94 FD
```

0F00: Secteur précédent de d.

même opération avec 0EC0 :

```
-m 0EC0 L 1B 0F00
```

```
-e 0F0C 4E 4E
-e 0F12 74 FD
```

0F20: CONTROLE DU RESTE.

```
0F20 LES  DI,[BP+16]  ; ES:DI en Dt
0F23 MOV  AX,[BP+62]  ; fenêtre lue
0F26 CMP  AX,0000     ; 2 décim.?
0F29 JZ   0F40        ; oui : suite
0F2B ES:              ; non : 3 d.
0F2C MOV  AL,[DI]     ; lire en Dt
0F2E CMP  AL,00       ; 00 en Dt?
0F30 JZ   0F34        ; oui : suite
0F32 JMP  0F86        ; non : sous.
0F34 TEST DI,000F     ; nouv. sec.?
0F38 JZ   0F3D        ; oui : faire
0F3A DEC  DI          ; non : décr.
0F3B JMP  0F40        ; -> suite
0F3D CALL 0EE0        ; nouv. sect.
0F40 LDS  SI,[BP+36]  ; DS:SI en dt
0F43 XOR  CH,CH       ; dernier
0F45 MOV  CL,[BP+67]  ; secteur :
0F48 JCXZ 0F51        ; I vide ->
0F4A STD              ; non :
0F4B REPZ             ; comparer
0F4C CMPSB            ; Dc avec dc
0F4D JB   0F88        ; r < d, bien
0F4F JA   0F86        ; r > d, pb.
0F51 MOV  CH,[BP+66]  ; à présent
0F54 CALL 0EE0        ; partie II
0F57 STD              ; comparer
0F58 REPZ             ; Dc avec dc
0F59 CMPSB            ; tant que =
0F5A JB   0F88        ; r < d, bien
0F5C JA   0F86        ; r > d, pb.
0F5E MOV  CX,[BP+32]  ; lire [d]
0F61 DEC  CX          ; 1 sec. fait
0F62 JCXZ 0F86        ; (= -> fin)
0F64:BOUCLE           ; selon [d]-1
0F64 PUSH CX          ; compteur
0F65 CALL 0F00        ; nv.sect. d
0F68 XOR  CH,CH       ; partie I
0F6A MOV  CL,[BP+65]  ; non vide?
0F6D JCXZ 0F76        ; si oui, ->
0F7F STD              ; sinon:
0F70 REPZ             ; comparer
0F71 CMPSB            ; dc et Dc
0F72 JB   0F88        ; r < d, bien
0F74 JA   0F86        ; r > d, pb.
0F76 CALL 0EE0        ; partie II
0F79 MOV  CL,[BP+64]  ; taille
0F7C STD              ; comparer
0F7D REPZ             ; dc avec Dc
0F7E CMPSB            ;
0F7F JB   0F88        ; r < d, bien
0F81 JA   0F86        ; r > d, pb.
0F83 POP  CX          ; (compteur)
0F84 LOOP 0F64        ; -> BOUCLE
0F86 CLC              ; soustraire
0F87 RET              ;
0F88 STC              ; R.A.S.
0F89 RET              ;
```

0FD0: GESTIONNAIRE DES ERREURS.

```
0FD0 MOV  DX,1048     ; 'erreur en'
0FD3 MOV  AH,09       ;
0FD5 INT  21          ;
0FD7 PUSH BP          ; lire
0FD8 MOV  BP,SP       ; l'adresse
0FDA MOV  AX,[BP+02]  ; de retour
0FDD SUB  AX,0003     ; adr. appel
0FE0 POP  BP          ; (formel)
0FE1 PUSH AX          ; afficher:
0FE2 MOV  AL,AH       ; dans
0FE4 XOR  AH,AH       ; octet fort
0FE6 MOV  BL,10       ;
0FE8 DIV  BL          ;
0FEA PUSH AX          ;
0FEB MOV  DL,AL       ; 1/2 faible
0FED ADD  DL,30       ;
0FF0 CMP  DL,3A       ; chiffre?
0FF3 JB   0FF8        ; oui: suite
0FF5 ADD  DL,07       ; add. lettre
0FF8 MOV  AH,02       ; afficher
0FFA INT  21          ;
0FFC POP  AX          ; puis
0FFD MOV  DL,AH       ; 1/2 fort
0FFF ADD  DL,30       ;
1002 CMP  DL,3A       ;
1005 JB   100A        ;
1007 ADD  DL,07       ;
100A MOV  AH,02       ; afficher
100C INT  21          ;
100E POP  AX          ; adr. revue
100F XOR  AH,AH       ;
1011 MOV  BL,10       ; oct. faible
1013 DIV  BL          ;
1015 PUSH AX          ;
1016 MOV  DL,AL       ; 1/2 faible
1018 ADD  DL,30       ;
101B CMP  DL,3A       ;
101E JB   1023        ;
1020 ADD  DL,07       ;
1023 MOV  AH,02       ; afficher
1025 INT  21          ;
1027 POP  AX          ;
1028 MOV  DL,AH       ; 1/2 fort
102A ADD  DL,30       ;
102D CMP  DL,3A       ;
1030 JB   1035        ;
1032 ADD  DL,07       ;
1035 MOV  AH,02       ; afficher
1037 INT  21          ;
1039 MOV  DX,105B     ; stl
103C MOV  AH,09       ;
103E INT  21          ;
1040 MOV  AX,4C00     ; arrêt
1043 INT  21          ; retour DOS
1045 RET              ; (<-inutile)
```

1048 messages du programme 0FD0

```
1048 'erreur trouvée en ' 24
105B 0D,0A,24
```

1060: ENREGISTREMENT DE u ET REACTUALISATION DES POINTEURS.

```
1060 MOV  BX,BP       ; initialiser
1062 ADD  BX,008F     ; le pointeur
1066 CS:              ; intérimaire
1067 MOV  [1092],BX   ; de qc
106B CS:              ; nouv. adr.
106C ADD  WO [0B25],+1A; appel
1071 CMP  AL,00       ; u = 0?
1073 JNZ  107A        ; non: suite
1075 CALL 1098        ; si: |q|-1
1078 JMP  1088        ; -> suite
```

```
107A CS:                ; enregistrer
107B MOV  BX,[1092]     ; lire le
107F SS:                ; pointeur
1080 MOV  [BX],AL       ; noter
1082 DEC  BX            ; mise à jour
1083 CS:                ; conservée
1084 MOV  [1092],BX     ;
1088 CMP  CX,+01        ; tout dern.?
108B JBE  1090          ; oui: -> fin
108D CALL 10E0          ; mise à jour
1090 RET                ;
```

variable pour le programme 1060.

```
1092 00 00              ;
```

1098: Réduire |q| de 1 et redéfinir les indices de partage.

```
1098 MOV  AL,[BP+54]    ; AL := (q)
109B CMP  AL,01         ; AL > 1?
109D JA   10A9          ; oui: suite
109F MOV  AL,10         ; non:
10A1 MOV  [BP+54],AL    ; (q):=010
10A4 DEC  WO [BP+52]    ; (q):=(q)-1
10A7 JMP  10AC          ; -> suite
10A9 DEC  BY [BP+54]    ; (q):=(q)-1
10AC DEC  CX            ; modif. CX
10AD CMP  WO [BP+52],+00; q = 0?
10B1 JZ   10D7          ; oui: fin
10B3 MOV  AH,[BP+54]    ; non:
10B6 DEC  AH            ; AH:=(q)-1
10B8 MOV  AL,10         ; AL:=i64:
10BA SUB  AL,AH         ;
10BC MOV  [BP+64],AX    ; i64 et i65
10BF MOV  BL,[BP+14]    ; BL:=(D)
10C2 MOV  BH,[BP+34]    ; BH:=(d)
10C5 CMP  BL,BH         ; (D)>(d)?
10C7 JNB  10D0          ; oui: suite
10C9 MOV  AH,BH         ; non:
10CB SUB  AH,AL         ; AH:=(d)-i64
10CD JMP  10D4          ; -> suite
10CF NOP                ;
10D0 MOV  AL,BH         ; i66:=(d)
10D2 XOR  AH,AH         ; i67:=0
10D4 MOV  [BP+66],AX    ; noté
10D7 RET                ;
```

10E0: Réactualisation de D_u, D_p, D_t et de la fenêtre.

```
10E0 MOV  AX,[BP+62]    ; fenêtre
10E3 CMP  AX,0000       ; 2 décim.?
10E6 JZ   111D          ; oui: suite
10E8 LES  DI,[BP+16]    ; non:
10EB ES:                ; 3 decimales
10EC MOV  AL,[DI]       ; vérifier
10EE CMP  AL,00         ; 00 sous Dt
10F0 JZ   10F8          ; oui: suite
10F2 CALL 0FD0          ; non: erreur
10F5 STC                ;
10F6 RET                ;
10F7 NOP                ;
10F8 TEST DI,000F       ; Dt-1
10FC JNZ  110D          ; -> décrém.
10FE PUSH [BP+20]       ; changer de
1101 POP  [BP+1A]       ; secteur
1104 LES  DI,[BP+1C]    ; (Dt:=Dp+1)
1107 MOV  [BP+18],ES    ; seg. Dt
110A INC  DI            ;
110B JMP  110E          ; -> suite
110D DEC  DI            ; (cas usuel)
110E MOV  [BP+16],DI    ; dépl. Dt
1111 CALL 11B8          ; |D|:=|D|-1
1114 ES:                ;
1115 MOV  AL,[DI]       ;
1117 MOV  [BP+62],AL    ; oct. 02 mis
111A DEC  DI            ;
111B JMP  113C          ; suite 2 & 3
111D LES  DI,[BP+16]    ; 2 décimales
1120 ES:                ;
1121 MOV  AL,[DI]       ; lu sous Dt
1123 MOV  [BP+62],AL    ; oct. 02 mis
1126 LES  DI,[BP+1C]    ; pointer Dp
1129 CMP  AL,00         ; modif. Dt?
112B JNZ  113C          ; non: suite
112D MOV  [BP+16],DI    ; si: Dt:=Dp
1130 MOV  [BP+18],ES    ;
1133 PUSH [BP+20]       ;
1136 POP  [BP+1A]       ;
1139 CALL 11B8          ; |D|:=|D|-1
113C ES:                ; cas 2 & 3
113D MOV  AL,[DI]       ;
113F MOV  [BP+61],AL    ; oct. 01 mis
1142 TEST DI,000F       ; ch. sect.?
1146 JNZ  115C          ; non: suite
1148 PUSH [BP+20]       ; si:
114B POP  [BP+2C]       ; préparer
114E CALL 0EE0          ; appel EE0
1151 PUSH [BP+2C]       ; nouveau
1154 POP  [BP+20]       ; n° ch. noté
1157 MOV  [BP+1E],ES    ; seg. noté
115A JMP  115D          ; -> suite
115C DEC  DI            ; décr.
115D ES:                ;
115E MOV  AL,[DI]       ;
1160 MOV  [BP+60],AL    ; oct. 00 mis
1163 MOV  [BP+1C],DI    ; dépl. Dp
1166 LES  DI,[BP+21]    ; Du actuel
1169 TEST DI,000F       ; ch. sect?
116D JNZ  1183          ; non: (décr)
116F PUSH [BP+26]       ; si:
1172 POP  [BP+2C]       ; préparer
1175 CALL 0EE0          ; appel EE0
1178 PUSH [BP+2C]       ;
117B POP  [BP+26]       ; n° ch. noté
117E MOV  [BP+24],ES    ; seg. noté
1181 JMP  1184          ;
1183 DEC  DI            ;
1184 MOV  [BP+22],DI    ; dépl. noté
1187 MOV  AL,[BP+64]    ; indices i64
118A DEC  AL            ;
118C CMP  AL,00         ; i64=0?
118E JNZ  1192          ; non: suite
1190 MOV  AL,10         ; si: AL:=10
1192 MOV  AH,10         ;
1194 SUB  AH,AL         ; i65:=10-i64
1196 MOV  [BP+64],AX    ; i64,5 notés
1199 MOV  AL,[BP+14]    ; AL:=(D)2
119C MOV  AH,[BP+34]    ; AH:=(d)
119F CMP  AL,AH         ; comparer
11A1 JB   11AA          ; -> (D)>(d)
11A3 MOV  AL,AH         ; i66:=(d)
11A5 XOR  AH,AH         ; i67:=00
11A7 JMP  11AC          ;
11A9 NOP                ; cas (D)>(d)
```

```
11AA SUB  AH,AL        ; i67=(d)-i66
11AC MOV  [BP+66],AX   ; i66,7 notés
11AF RET               ;
```

11B8: Réduire |D| de 1.

```
11B8 MOV  AL,[BP+14]   ; AX := (D)
11BB DEC  AL           ; réduction
11BD CMP  AL,00        ; AL = 00?
11BF JA   11C6         ; non : suite
11C1 MOV  AL,10        ; si : (D)=10
11C3 DEC  WO [BP+12]   ; [D]:=[D]-1
11C6 MOV  [BP+14],AL   ; (D) noté
11C9 RET               ;
```

11D8: TRANSCRIPTION D'UN SECTEUR DU QUOTIENT q.

11D8:ENTREE initiale (q_t)

```
11D8 CMP  WO [BP+52],+00; q =00?
11DC JNZ  11E0         ; non : suite
11DE RET               ; si : retour
11DF NOP               ;
11E0 MOV  AL,[BP+56]   ; (q) initial
11E3 MOV  AH,[BP+54]   ; (q) final
11E6 CMP  AL,AH        ;
11E8 JZ   11F1         ; sans corr.
11EA CMP  AL,01        ; si: 1?
11EC JBE  1238         ; oui : suite
11EE DEC  BY [BP+56]   ; (q) corrigé
11F1 MOV  CX,[BP+54]   ;
11F4 MOV  SI,[BP+3A]   ; n°ch. dt
11F7 INC  SI           ;
11F8 INC  SI           ; d'où ch. qt
11F9 PUSH SI           ;
11FA CALL 0C88         ; adresse qt
11FD PUSH DS           ;
11FE POP  ES           ;
11FF MOV  DI,SI        ; ES:DI en qt
1201 PUSH ES           ; adresse
1202 PUSH DI           ; sauvée
1203 PUSH CS           ;
1204 POP  DS           ; DS := CS
1205 CS:               ;
1206 MOV  SI,[1092]    ; qc provis.
120A INC  SI           ;
120B CLD               ;
120C REPZ              ;
120D MOVSB             ; copie
120E POP  DI           ; effectuée
120F POP  ES           ; ES:DI : qt
1210 MOV  CL,04        ; conversion
1212 SHR  DI,CL        ; en
1214 SUB  DI,01C8      ; adresse TAS
1218 SHL  DI,1         ;
121A MOV  [BP+56],DI   ; adr. TAS
122D MOV  [BP+58],ES   ; notée
1220 ES:               ; marquage qt
1221 MOV  WO [DI],0100; dernier
1225 POP  SI           ; n°ch. qt
1226 ES:               ;
1227 PUSH [SI]         ; ce n° est
1229 POP  [BP+5A]      ; gardé (qc)
122C INC  SI           ; n° ch.
122D INC  SI           ; actualisé
122E MOV  [BP+5C],SI   ; noté (pour
1231 NOP               ; q, ordre S
1232 CS:               ; inverse)
1233 ADD  WO [0B2C],+60; réécriture
1233                   ; appel
```

1238:ENTREE USUELLE

```
1238 CMP  CX,+01       ; DIV finie?
123B JZ   1240         ; oui : suite
123D CALL 10E0         ; act. point.
1240 CALL 0728         ; alloc. S
1243 JNB  124A         ; obtenu
1245 CALL 0FD0         ; non: RAM
1248 STC               ; pleine
1249 RET               ;
124A PUSH ES           ; ES:DI : TAS
124B PUSH DI           ; ES:SI : S
124C MOV  DI,SI        ; à présent,
124E MOV  CX,0010      ; ES:DI sur S
1251 CS:               ; pour copie
1252 MOV  SI,[1092]    ; du tampon
1256 INC  SI           ; sur S
1257 PUSH CS           ;
1258 POP  DS           ;
1259 STD               ; à reculons
125A REPZ              ;
125B MOVSB             ; effectué
125C MOV  SI,[BP+5C]   ; n° ch. qc
125F MOV  DS,[BP+00]   ; DS: zone ch
1262 MOV  [SI],AX      ; n°TAS S sur
1264 PUSH AX           ; zone
1265 LES  DI,[BP+56]   ; ES:DI TAS
1268 MOV  AX,[BP+5A]   ; S suivant
126B ES:               ; AX n°TAS
126C MOV  [DI],AX      ; noté en TAS
126E ADD  WO [BP+5C],+02; n°ch qc
1272 POP  [BP+5A]      ; n°TAS noté
1275 POP  DI           ; rappel adr.
1276 POP  ES           ; TAS de S
1277 MOV  [BP+56],DI   ; notée
127A MOV  [BP+58],ES   ;
127D RET               ;
```

12A0 : Multiplication d'un registre par un mot.

```
12A0 PUSH DI           ; ES:SI sur
12A1 PUSH ES           ; R but (=M)
12A2 PUSH AX           ; CX = (R)
12A3 PUSH DX           ; DX = (R)
12A4 PUSH CX           ; BX = m
12A5 MOV  BP,SP        ;
12A7 XOR  DX,DX        ; r = 0
12A9:BOUCLE            ; multiplier
12A9 PUSH CX           ; compteur
12AA CALL 1338         ; S*m
12AD POP  CX           ; ES:DI : TAS
12AE DEC  CX           ;
12AF JCXZ 12B6         ; év. fini
12B1 CALL 1310         ; S suivant
12B4 JMP  12A9         ; -> BOUCLE
12B6 CMP  DX,+00       ; r <> 0?
12B9 JZ   12D0         ; non : DX
12BB CALL 12E0         ; si : écrire
12BE JNB  12C5         ; corr. DX
12C0 CALL 0FD0         ; erreur
12C3 STC               ;
12C4 RET               ;
12C5 POP  CX           ; CX et DX
12C6 INC  CX           ; modifiés,
12C7 POP  DX           ; car (R) et
12C8 MOV  DX,0001      ; (R) aussi
12CB POP  AX           ;
12CC POP  ES           ;
```

```
12CD POP  DI          ;
12CE CLC              ; R.A.S.
12CF RET              ;
12D0 POP  CX          ;
12D1 POP  DX          ;
12D2 CALL 1360        ; modif DX
12D5 JMP  12CB        ; -> sortie
```

12E0 : Ecrire la dernière retenue MUL (12A0).

```
12E0 PUSH DI          ;
12E1 PUSH ES          ;
12E2 CALL 0728        ; alloc. S
12E5 JNB  12EA        ; obtenu
12E7 STC              ; non
12E8 RET              ; sortie
12E9 NOP              ;
12EA ES:              ; écriture
12EB MOV  [SI],DX     ; de r
12ED ES:              ; marquage de
12EE MOV  WO [DI],0100; S dernier
12F2 POP  ES          ; adresse TAS
12F3 POP  DI          ; ex-S
12F4 ES:              ; 'raccord'
12F5 MOV  [DI],AX     ; effectué
12F7 RET              ;
```

1310 : Détermination du secteur suivant du Registre (12A0).

```
1310 PUSH CX          ; sauver
1311 PUSH BX          ; registres
1312 ES:              ; lire
1313 MOV  BX,[DI]     ; entrée TAS
1315 PUSH BX          ;
1316 AND  BX,F000     ; obtention
131A MOV  CL,0B       ; du n°zone
131C SHR  BX,CL       ;
131E CS:              ;
131F MOV  AX,[BX+3800];
1323 MOV  ES,AX       ; segment S
1325 POP  DI          ; n°TAS chan-
1326 AND  DI,0FFF     ; gé en
132A MOV  CL,04       ; adresse de
132C SHL  DI,CL       ; S
132E POP  BX          ;
132F POP  CX          ;
1330 RET              ;
```

1338 : Multiplication du secteur courant par un mot (12A0).

```
1338 PUSH DI          ; ES:DI : S
1339 MOV  CX,0008     ; DX=r, BX=m
133C CLD              ;
133D:BOUCLE           ; huit mots
133D PUSH CX          ; compteur
133E MOV  CX,DX       ; CX :=  ex-r
1340 ES:              ; lire
1341 MOV  AX,[DI]     ; décimale
1343 MUL  BX          ; multiplier
1345 ADD  AX,CX       ; add. ex-r
1347 ADC  DX,+00      ; corr. r
134A STOSW            ; écrire
134B POP  CX          ; compteur
134C LOOP 133D        ; -> BOUCLE
134E POP  DI          ; adresse S
134F MOV  CL,04       ; convertie
1351 SHR  DI,CL       ; en
1353 SUB  DI,01C8     ; adresse TAS
1357 SHL  DI,1        ; de S
1359 RET              ;
```

1360 : Ajustement de (R) (12A0).

```
1360 PUSH CX          ; à sauver
1361 SHR  DI,1        ; DI converti
1363 ADD  DI,01C8     ; en adresse
1367 MOV  CL,04       ; de S, mais
1369 SHL  DI,CL       ; sur dernier
136B ADD  DI,+0F      ; octet de S
136E XOR  AL,AL       ;
1370 MOV  CL,10       ; seize oct.
1372 STD              ;
1373 REPZ             ;
1374 SCASB            ;
1375 JNZ  137C        ; trouvé
1377 CALL 0FD0        ; S = 0 :
137A STC              ; erreur
137B RET              ;
137C MOV  DX,DI       ; calcul
137E INC  DX          ; de (R)
137F INC  DX          ;
1380 AND  DX,000F     ;
1384 CMP  DX,+00      ;
1387 JNZ  138B        ;
1389 MOV  DL,10       ;
138B POP  CX          ;
138C RET              ;
```

13A0 : DIVISION D'UN REGISTRE PAR UN MOT (0AD0).

zone de variables propre à ce programme, installée sur la pile :

```
  00  segment zone de chaînage
  02  adresse répertoire de D
  04  [D]
  06  (D)  (en mots)
  08  n° chaînage de u0 de D
  0A  n° chaînage de Dt
0E-0C adresse absolue de Dc
  10  n° chaînage de Dc
  12  adresse répertoire de d
  14  adresse répertoire de r
13A0 MOV  BP,SP       ; réserve
13A2 SUB  BP,+20      ; sur pile
13A5 CLI              ;
13A6 MOV  SP,BP       ;
13A8 STI              ;
13A9 CALL 13D0        ; initialiser
13AC CALL 1428        ; lère décim.
13AF MOV  CX,[BP+04]  ; CX := [D]
13B2 DEC  CX          ;
13B3 JCXZ 13BF        ; (DIV finie)
13B5:BOUCLE           ; selon [D]-1
13B5 PUSH CX          ; compteur
13B6 CALL 0EE0        ; S suivant
13B9 CALL 14B0        ; S DIV BX
13BC POP  CX          ; compteur
13BD LOOP 13B5        ; -> BOUCLE
13BF CALL 14D0        ; résultat
13C2 MOV  BP,SP       ; réserve
13C4 ADD  BP,+20      ; sur pile
13C7 CLI              ; démantelée
13C8 MOV  SP,BP       ;
13CA STI              ;
13CB RET              ;
```

```
   13D0 : Initialisation (13A0).
13D0 MOV  AH,48          ; allocation
13D2 MOV  BX,1000        ; zone de
13D5 INT  21             ; chaînage D
13D7 JNB  13DE           ; obtenue
13D9 CALL 0FD0           ; non :
13DC STC                 ; RAM pleine
13DD RET                 ;
13DE MOV  [BP+00],AX     ; seg zone,
13E1 MOV  [BP+02],SI     ; adr. rép. D
13E4 CS:                 ; sur pile
13E5 MOV  AX,[SI+4002]   ; puis :
13E9 MOV  [BP+04],AX     ; [D]
13EC SHR  SI,1           ; (convertir
13EE CS:                 ; pour 4040)
13EF MOV  AX,[SI+4040]   ;
13F3 MOV  [BP+06],AX     ; (D) (mots)
13F6 MOV  [BP+12],BX     ; adr. rép. d
13F9 MOV  [BP+14],DI     ; adr. rép. r
13FC MOV  ES,AX          ; ES sur zone
13FE XOR  DI,DI          ; DI := 00
1400 MOV  [BP+08],DI     ; n°ch. u0 D
1403 CALL 0BB8           ; chaînage
1406 DEC  DI             ;
1407 DEC  DI             ;
1408 MOV  [BP+0A],DI     ; n°ch. Dt
140B CS:                 ;
140C MOV  BX,[BX+4000]   ; n°TAS de d
1410 CALL 0C8B           ; adr. abs.
1413 MOV  BX,[SI]        ; BX := d
1415 CS:                 ; réécriture
1416 MOV  BY [0EEB],10   ; de 0EE0
141B CS:                 ; pour ce
141C MOV  BY [0EF0],10   ; programme
1421 RET                 ;
   1428 : Première décimale (13A0).
1428 MOV  SI,[BP+0A]     ; n°ch. Dt
142B MOV  [BP+10],SI     ; n°ch. Dc
142E PUSH BX             ; d sauvé
142F CALL 0C88           ; ES:DI adr.
1432 PUSH DS             ; absolue
1433 POP  ES             ; de Dc
1434 MOV  DI,SI          ; (ici Dt)
1436 POP  BX             ;
1437 MOV  CX,[BP+06]     ; CX := (D)
143A PUSH CX             ;
143B DEC  CX             ;
143C DEC  CX             ;
143D ADD  DI,CX          ; DI : Dt mot
143F XOR  DX,DX          ; retenue 00
1441 ES:                 ;
1442 MOV  AX,[DI]        ; lire AX
1444 DIV  BX             ; division
1446 STOSW               ; quot. écrit
1447 POP  CX             ; CX repris
1448 CMP  AX,0000        ; quot. nul?
144B JZ   145A           ; oui : voir
144D DEC  CX             ; non
144E JCXZ 1459           ; ((D) = 1)
1450 STD                 ; (pour STOS)
1451:BOUCLE              ; selon (D)-1
1451 ES:                 ;
1452 MOV  AX,[DI]        ; lire AX
1454 DIV  BX             ; diviser
1456 STOSW               ; quot. écrit
1457 LOOP 1451           ; -> BOUCLE
1459 RET                 ; fin
145A MOV  AL,[BP+06]     ; cas nul :
145D DEC  AL             ; |D|: |D|-1
145F CMP  AL,00          ;
1461 JA   1468           ;
1463 MOV  AL,08          ;
1465 DEC  WO [BP+04]     ; nouveau |D|
1468 MOV  [BP+06],AL     ; noté puis
146B DEC  CX             ; action sur
146C JCXZ 1470           ; CX
146E JMP  1450           ; cas (D)>1
1470 PUSH DX             ; cas (D)=1 :
1471 PUSH CX             ; on libère
1472 MOV  SI,[BP+0A]     ; S dernier
1475 CMP  SI,+00         ; un seul S?
1478 JZ   14A8           ; oui : fin
147A MOV  CL,04          ; non :
147C SHR  DI,CL          ; ES:DI
147E SUB  DI,01C8        ; converti
1482 SHL  DI,1           ; en adresse
1484 ES:                 ; TAS de S
1485 MOV  WO [DI],0000   ; S libéré
1489 DEC  SI             ; SI: n°ch.
148A DEC  SI             ; n°ch. pré.
148B MOV  AX,[BP+00]     ;
148E MOV  ES,AX          ; ES: zone
1490 PUSH BX             ;
1491 CALL 0C88           ; DS:SI adr.
1494 PUSH DS             ; absolue
1495 POP  ES             ; portée sur
1496 MOV  DI,SI          ; ES:DI
1498 MOV  CL,04          ; DI converti
149A SHR  DI,CL          ; en adresse
149C SUB  DI,01C8        ; TAS
14A0 SHL  DI,1           ;
14A2 ES:                 ; S marqué
14A3 MOV  WO [DI],0100   ; dernier
14A7 POP  BX             ; d retrouvé
14A8 POP  CX             ;
14A9 POP  DX             ;
14AA RET                 ; retour
   14B0 : Division d'un secteur
(13A0).
14B0 ADD  DI,+0E         ; sur dernier
14B3 MOV  CX,0008        ; huit mots
14B6 STD                 ; vers le bas
14B7:BOUCLE              ;
14B7 ES:                 ;
14B8 MOV  AX,[DI]        ; lire
14BA DIV  BX             ; diviser
14BC STOSW               ; écrire
14BD LOOP 14B7           ; -> BOUCLE
14BF RET                 ;
   14D0 : Résultat (13A0).
14D0 CS:                 ; restaurer
14D1 MOV  BY [0EEB],2C   ; le code
14D6 CS:                 ; initial de
14D7 MOV  BY [0EF0],2C   ; 0EE0
14DC MOV  AX,[BP+24]     ; signal de
14DF SHR  AX,1           ; DIV virg.?
14E1 JNB  14E6           ; non : suite
14E3 CALL 1520           ; si: virgule
14E6 MOV  SI,[BP+02]     ; SI restauré
14E9 PUSH SI             ; sauvé
14EA PUSH [BP+04]        ; (q) (ex(D))
```

```
14ED CS:                ; placé dans
14EE POP  [SI+4002]     ; répertoire
14F2 PUSH [BP+06]       ; même chose
14F5 SHR  SI,1          ; pour
14F7 CS:                ; (q)
14F8 POP  [SI+4040]     ; puis on
14FC CS:                ; écrit le
14FD MOV  SI,[BP+14]    ; reste (DX)
1500 PUSH SI            ; de la
1501 MOV  BX,[SI+4000]; division :
1505 CALL 0C8B          ; adr. de d
1508 MOV  [SI],DX       ; écriture
150A POP  DI            ; DI restauré
150B POP  SI            ; SI restauré
150C MOV  BX,[BP+12]    ; BX restauré
150F RET                ; fin
   1520 : CAS q = 0 et r = D
(0AD0).
1520 MOV  AH,48         ; réserver
1522 MOV  BX,1000       ; une zone
1525 INT  21            ;
1527 JNB  152E          ;
1529 CALL 0FD0          ;
152C STC                ;
152D RET                ;
152E MOV  ES,AX         ;
1530 MOV  [BP+00],AX    ;
1533 MOV  [BP+10],SI    ;
1536 MOV  [BP+30],BX    ;
1539 MOV  [BP+50],DI    ;
153C CALL 0B94          ; relevé D, d
153F MOV  AL,01         ;
1541 MOV  [BP+0C],AL    ; (q) = 1
1544 MOV  [BP+52],AL    ; idem
1547 MOV  [BP+54],AL    ; (q) = 1
154A CS:                ;
154B MOV  AX,[SI+4000];
154F MOV  [BP+12],AX    ; [D] noté
1552 CS:                ;
1553 MOV  AX,[DI+4000];
1557 MOV  [BP+32],AX    ; (d) noté
155A MOV  SI,[BP+1A]    ; n°ch Dt
155D CALL 0C88          ; DS:SI : Dt
1560 MOV  DI,SI         ; calcul de
1562 ADD  SI,+0F        ; (D) octets
1565 MOV  CX,0010       ; 010 octets
1568 MOV  AL,00         ;
156A STD                ; vers le bas
156B REPZ               ; tant que =
156C SCASB              ;
156D JNZ  1574          ; trouvé
156F CALL 0FD0          ; non :
1572 STC                ; erreur
1573 RET                ;
1574 INC  SI            ; correction
1575 SUB  SI,DI         ;
1577 AND  SI,000F       ; (D)
157B MOV  [BP+14],SI    ; noté
157E MOV  SI,[BP+3A]    ; même calcul
1581 CALL 0C88          ; pour (d)
1584 MOV  DI,SI         ;
1586 ADD  SI,+0F        ;
1589 MOV  CX,0010       ;
158C MOV  AL,00         ;
158E STD                ;
158F REPZ               ;
1590 SCASB              ;
1591 JNZ  1598          ;
1593 CALL 0FD0          ;
1596 STC                ;
1597 RET                ;
1598 INC  SI            ;
1599 SUB  SI,DI         ;
159B AND  SI,000F       ;
159F MOV  [BP+34],SI    ; (d) noté
15A2 RET                ;
   15B0 : RESULTATS (0AD0).
15B0 MOV  SI,[BP+10]    ; SI retrouvé
15B3 MOV  AX,[BP+12]    ; AX := [r]
15B6 CS:                ; inscrit en
15B7 MOV  [SI+4000],AX; répertoire
15BB MOV  AX,[BP+14]    ; puis (r)
15BE SHR  AL,1          ; converti de
15C0 ADC  AL,00         ; octets en
15C2 MOV  BX,SI         ; mots
15C4 SHL  BX,1          ; et noté en
15C6 CS:                ; table 4040
15C7 MOV  [BX+4040],AL;
15CB MOV  DI,[BP+50]    ; même opér.
15CE MOV  AX,[BP+52]    ; pour DI
15D1 CS:                ; avec q
15D2 MOV  [DI+4000],AX; (q) noté en
15D6 MOV  AX,[BP+54]    ; répertoire
15D9 SHR  AL,1          ; puis
15DB ADC  AL,00         ; (q) mots
15DD MOV  BX,DI         ; calculé
15DF SHL  BX,1          ; et noté
15E1 CS:                ; en
15E2 MOV  [BX+4040],AL; table 4040
15E6 MOV  BX,[BP+30]    ; BX retrouvé
15E9 RET                ;
```

ANNEXE 3 : Programme non récursif de la fonction d'Ackermann

```
   0100: PROGRAMME PRINCIPAL.
0100 MOV  AX,CS        ; initialisa-
0102 MOV  DS,AX        ; tion des
0104 MOV  ES,AX        ; registres
0106 MOV  SS,AX        ; de segment
0108 NOP               ; (DEBUG)
0109 CALL 0200         ; DONNEES
010C MOV  BX,0290      ; dest.chrono
010F CALL 01F0         ; top chrono
0112 CALL 0300         ; ACKERMANN
0115 PUSH AX           ; AX = psi
0116 MOV  BX,0294      ; dest.chrono
0119 CALL 01F0         ; top chrono
011C POP  CX           ; résultat
011D CALL 0310         ; AFFICHAGE
0120 CALL 03A0         ; TEMPS MIS
0123 MOV  AX,4C00      ; TERMINAISON
0126 INT  21           ;
   0130: Conversion bexa ->
décimal.
   tableau de chiffres
0130 '0123456789
   variables
013A 000000000000
   sous programme 0140
0140:BOUCLE
0140 MOV  AX,[010A]    ;
0143 DIV  BX           ;
0145 MOV  [010A],AX    ;
0148 MOV  AX,[010C]    ;
014B DIV  BX           ;
014D MOV  [010C],AX    ; quotient
0150 DS:               ;
0151 MOV  [BP+SI],DL   ; reste
0153 XOR  DX,DX        ;
0155 CMP  AX,0000      ; quot. nul?
0158 JZ   015D         ; oui: -> fin
015A INC  SI           ;
015B JMP  0140         ; -> BOUCLE
015D RET               ;
   programme
0160 PUSH DS           ;
0161 PUSH ES           ;
0162 POP  DS           ;
0163 MOV  AX,0000      ; init. des
0166 MOV  [010A],AX    ; variables
0169 MOV  AX,CX        ;
016B MOV  [010C],AX    ;
016E MOV  SI,0000      ; init. ptr.
0171 MOV  BP,0179      ; lect./écr.
0174 MOV  BX,000A      ; 0A = dix
0177 XOR  DX,DX        ; DX=0 (DIV)
0179 CALL 0140         ; décomposer
017C MOV  BX,0100      ;
017F MOV  CX,SI        ; reste-t-il
0181 CMP  CX,+00       ; encore?
0184 JZ   0199         ; non: -> fin
0186 NOP               ;
0187:BOUCLE
0187 XOR  AX,AX        ;
0189 XOR  DX,DX        ;
018B DS:               ;
018C MOV  AL,[BP+SI]   ;
018E MOV  DI,AX        ;
0190 MOV  DL,[BX+DI]   ;
0192 MOV  AH,02        ; affichage
0194 INT  21           ; une lettre
0196 DEC  SI           ;
0197 LOOP 0187         ; -> BOUCLE
0199 XOR  AX,AX        ; dernière
019B DS:               ; décimale
019C MOV  AL,[BP+00]   ;
019F MOV  DI,AX        ;
01A1 MOV  DL,[BX+DI]   ;
01A3 MOV  AH,02        ; affichage
01A5 INT  21           ; une lettre
01A7 POP  DS           ;
01A8 RET               ;
   01B8: Lire un nombre à l'écran.
01B8 MOV  DX,0208      ; msg. écran
01BB MOV  AH,09        ;
01BD INT  21           ;
01BF PUSH DS           ;
01C0 PUSH ES           ;
01C1 POP  DS           ;
01C2 MOV  DX,0100      ; tampon de
01C5 MOV  AH,0A        ; réception
01C7 INT  21           ;
01C9 XOR  CX,CX        ;
01CB MOV  CL,[0101]    ; nbre chiff.
01CF MOV  AX,0000      ; init. tot.
01D2 MOV  BL,0A        ;
01D4 MOV  BP,0102      ; init. ptr.
01D7 XOR  DX,DX        ; retenue = 0
01D9:BOUCLE            ; Horner
01D9 MUL  BL           ;
01DB DS:               ;
01DC MOV  DL,[BP+00]   ; octet lu
01DF SUB  DL,30        ; converti
01E2 ADD  AX,DX        ; ajouté à AX
01E4 INC  BP           ; mise à jour
01E5 LOOP 01D9         ; -> BOUCLE
01E7 MOV  CX,AX        ; CX: résult.
01E9 POP  DS           ;
01EA RET               ;
   01F0: Top chrono
01F0 PUSH CX           ; CX et DX:
01F1 PUSH DX           ; arguments
01F2 MOV  AH,2C        ; fonction
01F4 INT  21           ; horloge
01F6 MOV  [BX+02],DX   ; DH:s,DL:100
01F9 MOV  [BX],CX      ; CH:h,CL:mn
01FB POP  DX           ; arguments
01FC POP  CX           ; retrouvés
01FD RET               ;
   0200: Lecture des données
0200 MOV  AX,CS        ; préparation
0202 ADD  AX,000B      ; appel de
0205 PUSH AX           ; 01B8
0206 POP  ES           ;
0207 MOV  WO [01B9],0330; adr.msg.
```

```
020D CALL 01B8          ; lire X
0210 PUSH CX            ; (CX = X)
0211 MOV  WO [01B9],0358; adr.msg.
0217 CALL 01B8          ; lire Y
021A MOV  DX,0368       ; st. 1.
021D MOV  AH,09         ;
021F INT  21            ;
0221 PUSH CX            ; (CX = Y)
0222 POP  DX            ; DX := Y
0223 POP  CX            ; CX := X
0224 RET                ;
```

0230: Fonction d'Ackermann.

variable

```
0230 0000
```

programme

```
0232 MOV  SI,SP         ; butoirs
0234 MOV  BX,0600       ; de la pile
0237:BOUCLE1
0237 CMP  SP,SI         ; tt. dépilé?
0239 JB   0240          ; non: suite1
023B CMP  CX,+00        ; X = 0?
023E JZ   0261          ; oui: -> fin
0240 CMP  SP,BX         ; au butoir?
0242 JA   0246          ; non: suite2
0244 JMP  0270          ; -> débord.
0246:BOUCLE2            ; tant q. X>0
0246 CMP  CX,+00        ; X = 0?
0249 JZ   025A          ; -> sortie
024B CMP  DX,+00        ; Y = 0?
024E JA   0256          ; non: -> Y>0
0250 DEC  CX            ; X := X-1
0251 MOV  DX,0001       ; Y := 1
0254 JMP  0246          ; -> BOUCLE2
0256 PUSH CX            ; empiler X
0257 DEC  DX            ; Y := Y-1
0258 JMP  0246          ; -> BOUCLE2
025A INC  DX            ; Y := Y+1
025B MOV  AX,DX         ; psi := Y
025D POP  CX            ; dépiler X
025E DEC  CX            ; X := X-1
025F JMP  0237          ; -> BOUCLE1
0261 INC  DX            ; Y := Y+1
0262 MOV  AX,DX         ; psi := Y
0264 RET                ;
0270 MOV  DX,0370       ; msg. débord
0273 MOV  AH,09         ;
0275 INT  21            ;
0277 JMP  DI            ; issue sec.
```

0290: Recueil des chronos

```
0290 00000000
0294 00000000
```

02A0: Mesure de la durée du calcul (03A0).

```
02A0 PUSH DI            ; sauver les
02A1 PUSH SI            ; registres
02A2 PUSH DS            ; DI,SI,DS
02A3 PUSH CS            ;
02A4 POP  DS            ; DS := CS
02A5 MOV  DI,0294       ; h. fin
02A8 MOV  SI,0290       ; h. début
02AB MOV  BX,0298       ; correctifs
02AE MOV  CX,0004       ; 4 octets
02B1 CLC                ; pas de ret.
02B2:BOUCLE             ;
02B2 PUSH CX            ; compteur
02B3 MOV  AL,[DI]       ; lire fin
02B5 MOV  AH,[SI]       ; lire début
02B7 MOV  CL,[BX]       ; correctif
02B9 CALL 02D8          ; soustraire
02BC MOV  [DI+08],AL    ; écrire f-d
02BF INC  DI            ; mise à jour
02C0 INC  SI            ; pour le
02C1 INC  BX            ; prochain
02C2 POP  CX            ; compteur
02C3 LOOP 02B2          ; -> BOUCLE
02C5 MOV  CX,[029E]     ; résultat
02C9 MOV  AX,[029C]     ; dans CX:AX
02CC POP  DS            ; registres
02CD POP  SI            ; restaurés
02CE POP  DI            ;
02CF RET                ;
```

02D8 : Soustraction.

```
02D8 ADC  AH,00         ; retenue
02DB CMP  AL,AH         ; AL >= AH?
02DD JNB  02E6          ; oui : suite
02DF ADD  AL,CL         ; non : corr.
02E1 SUB  AL,AH         ; soustraire
02E3 STC                ; retenue
02E4 RET                ;
02E5 NOP                ;
02E6 SUB  AL,AH         ; soustraire
02E8 CLC                ; sans reten.
02E9 RET                ;
```

0300: Appel de la fonction d'Ackermann

```
0300 PUSH DS            ; ajustement
0301 MOV  AX,CS         ; de DS
0303 ADD  AX,0013       ; pour le
0306 PUSH AX            ; programme
0307 POP  DS            ; récursif
0308 MOV  DI,0114       ; adr. issue
030B CALL 0232          ; ACKERMANN
030E POP  DS            ; DS restauré
030F RET                ;
```

0310: Affichage du résultat

```
0310 MOV  DX,0390       ; msg. écran
0313 MOV  AH,09         ;
0315 INT  21            ;
0317 MOV  AX,CS         ; adaptation
0319 ADD  AX,0003       ; ES pour
031C PUSH AX            ; programme
031D POP  ES            ; 0160
031E CALL 0160          ; conversion
0321 MOV  DX,0368       ; st. 1.
0324 MOV  AH,09         ;
0326 INT  21            ;
0328 RET                ;
```

0330: Messages.

```
0330 0D 0A 'calcul de psi (x,y)'
0346 ':' 0D 0A 0D 0A '   X = '
0352 24
0358 0D 0A '   Y = ' 24
0368 0D 0A 24
0370 0D 0A 'débordement de la p'
0385 'ile' 0D 0A 24
0390 0D 0A psi = ' 24
```

03A0: Affichage du temps de calcul.

```
03A0 MOV  DX,0410      ; msg. écran
03A3 MOV  AH,09        ;
03A5 INT  21           ;
03A7 CALL 02A0         ; CX:AX=temps
03AA PUSH AX           ; AX sauvé
03AB MOV  AX,CS        ;
03AD ADD  AX,0003      ; préparation
03B0 PUSH AX           ; appel conv.
03B1 POP  ES           ; heures
03B2 CMP  CH,00        ; zéro heure?
03B5 JZ   03C9         ; oui: -> mn.
03B7 PUSH CX           ; sauver CX
03B8 XCHG CH,CL        ; CH <-> CL
03BA XOR  CH,CH        ; CH := 0
03BC CALL 0160         ; afficher
03BF MOV  DL,68        ; 'h'
03C1 MOV  AH,02        ;
03C3 INT  21           ;
03C5 CALL 0408         ; un blanc
03C8 POP  CX           ; CX retrouvé
03C9 CMP  CL,00        ; zéro mn.?
03CC JZ   03E2         ; oui: -> s.
03CE XOR  CH,CH        ; CH := 0
03D0 CALL 0160         ; afficher
03D3 MOV  DL,6D        ; 'm'
03D5 MOV  AH,02        ;
03D7 INT  21           ;
03D9 MOV  DL,6E        ; 'n'
03DB MOV  AH,02        ;
03DD INT  21           ;
03DF CALL 0408         ; un blanc
03E2 POP  AX           ; AX retrouvé
03E3 PUSH AX           ; AX resauvé
03E4 XOR  CH,CH        ; CH := 0
03E6 MOV  CL,AH        ; CL := s.
03E8 CALL 0160         ; afficher
03EB MOV  DL,73        ; 's'
03ED MOV  AH,02        ;
03EF INT  21           ;
03F1 CALL 0408         ; un blanc
03F4 POP  AX           ; AX retrouvé
03F5 XOR  CH,CH        ; CH := 0
03F7 MOV  CL,AL        ; CL := 100e
03F9 CALL 0160         ; afficher
03FC MOV  DX,0368      ; st. 1.
03FF MOV  AH,09        ;
0401 INT  21           ;
0403 RET               ;
```

0408: Affichage d'un blanc

```
0408 MOV  DL,20        ;
040A MOV  AH,02        ;
040C INT  21           ;
040E RET               ;
```

message temps de calcul

```
0410 0D 0A 'temps de calcul: '
0423 24
```

copyright

```
0428 '(C) 1991 Maurice MARGENST'
0441 'ERN '
```

ANNEXE 4 : NOMBRES DE FIBONACCI

Programme principal.

```
0100 MOV  AX,CS        ;
0102 MOV  DS,AX        ;
0104 MOV  ES,AX        ;
0106 MOV  SS,AX        ;
0108 NOP               ;
0109 CALL 0120         ; DONNEES
010C CALL 0150         ; CALCUL
010F CALL 0670         ; AFFICHAGE
0112 MOV  AX,4C00      ; terminaison
0115 INT  21           ;
```

0120: Entrée de N à l'écran.

```
0120 MOV  DX,01B0      ; msg. entrée
0123 MOV  AH,09        ;
0125 INT  21           ;
0127 MOV  DX,0118      ; tampon de
012A MOV  AH,0A        ; lecture
012C INT  21           ;
012E XOR  CX,CX        ; initiali-
0130 MOV  CL,[0119]    ; -sations
0134 MOV  AX,0000      ;
0137 MOV  BL,0A        ;
0139 MOV  BP,011A      ;
013C XOR  DX,DX        ;
013E:BOUCLE            ; (Hörner)
013E MUL  BX           ;
0140 MOV  DL,[BP+00]   ;
0143 SUB  DL,30        ;
0146 ADD  AX,DX        ;
0148 INC  BP           ;
0149 LOOP 013E         ; -> BOUCLE
014B MOV  CX,AX        ;
014D RET               ;
```

0150 : Fonction de Fibonacci.

```
0150 CALL 0510         ; Initialiser
0153 XOR  DI,DI        ; 1er R.
0155 MOV  SI,0004      ; 2ème R.
0158 JCXZ 0170         ; si N=0, fin
015A CALL 01A0         ; +1 à (DI)
015D CMP  CX,+01       ; N<=1?
0160 JBE  0170         ; oui : fin
0162 DEC  CX           ; N := N-1
0163 PUSH DI           ;
0164 PUSH SI           ;
0165:BOUCLE            ; sur N-1
0165 POP  DI           ;
0166 POP  SI           ; (SI)<->(DI)
0167 PUSH DI           ; et
0168 PUSH SI           ; sauvegarde
0169 CALL 04A0         ; addition
016C LOOP 0165         ; -> BOUCLE
016E POP  SI           ;
016F POP  DI           ;
0170 RET               ;
```

01A0 : Incrémentation de (DI).

```
01A0 PUSH CX           ;
01A1 PUSH DI           ;
01A2 MOV  DI,1C80      ; 1er S.
01A5 CALL 0220         ; +1 à (DI)
01A8 POP  DI           ;
01A9 POP  CX           ;
01AA RET               ;
```

01B0 : messages du programme.

```
01B0 0D 0A 'Entrer N' 0D 0A
01BC '  >' 24
01C8 0D 'Fib (N) = ' 24
```

0200-0504 : programmes d'addition.

voir les instructions (mêmes adresses) de l'annexe 2.

0510 : Initialisation des registres.

```
0510 MOV  AH,4A        ; restriction
0512 MOV  BX,1000      ; du progr.
0515 INT  21           ; à 64 Ko
0517 JNB  051E         ; oui : suite
0519 CALL 0FD0         ; échec :
051C STC               ; problème
051D RET               ; système
051E MOV  AH,48        ; réserver
0520 MOV  BX,1000      ; 64 Ko (=
0523 INT  21           ; 01000 para)
0525 JNB  052C         ; oui : suite
0527 CALL 0FD0         ; non: RAM
052A STC               ; pleine
052B RET               ;
052C PUSH CX           ; (CX = N)
052D PUSH AX           ;
052E XOR  AX,AX        ; AX := 00
0530 MOV  DI,3800      ; table 3800
0533 MOV  CX,0021      ; 021 mots
0536 XOR  AX,AX        ; initialisés
0538 CLD               ; à 0000
0539 REPZ              ;
053A STOSW             ;
053B POP  AX           ; (adr. zone)
053C MOV  DI,3800      ; seg zone
053F MOV  [DI],AX      ; noté
0541 ADD  DI,+20       ; puis taux
0544 MOV  WO [DI],0E36; d'occupat.
0548 ADD  DI,+20       ; puis une
054B INC  WO [DI]      ; zone
054D PUSH [3800]       ; ES := seg
0551 POP  ES           ; zone
0552 XOR  AX,AX        ; puis
0554 MOV  CX,8000      ; initialiser
0557 XOR  DI,DI        ; la zone
0559 CLD               ; à 0000
055A REPZ              ; (8000 mots)
055B STOSW             ;
055C ES:               ; initialiser
055D MOV  WO [0000],0100; la TAS
0563 ES:               ; par 2 R de
0564 MOV  WO [0002],0100; 1 S.
056A MOV  BX,4000      ; initialiser
056D MOV  WO [BX],01C8; le répert.
0571 INC  BX           ; (depuis
0572 INC  BX           ; CS:4000)
0573 MOV  WO [BX],0001; pour 2 R
0577 INC  BX           ; de 1 S
0578 INC  BX           ; chacun
0579 MOV  WO [BX],01C9;
057D INC  BX           ;
057E INC  BX           ;
057F MOV  WO [BX],0001;
0583 POP  CX           ;
0584 MOV  [0700],CX    ; noter N
0588 MOV  BX,4040      ; initialiser
058B XOR  AX,AX        ; la table
058D INC  AX           ; 04040
058E MOV  [BX],AX      ;
0590 MOV  [BX+02],AX   ;
0593 RET               ;
```

05A0: Copie du registre sur le tampon d'affichage.

```
05A0 CS:               ; DI : adr.
05A1 MOV  AX,[DI+4000]; rép. de R
05A5 XOR  BP,BP        ;
05A7 MOV  DI,8000      ; ES:DI
05AA PUSH CS           ; initialisé
05AB POP  ES           ; à CS:8000
05AC:BOUCLE            ; selon [R]
05AC MOV  DX,AX        ; AX = N°S
05AE AND  DX,F000      ; converti
05B2 AND  AX,0FFF      ; en adresse
05B5 PUSH AX           ; abs. de S :
05B6 MOV  CL,0B        ; le dernier
05B8 SHR  DX,CL        ; 1/4 de AX
05BA MOV  BX,DX        ; donne la
05BC CS:               ; zone via
05BD MOV  AX,[BX+3800]; table 3800
05C1 MOV  DS,AX        ; d'où DS
05C3 POP  AX           ; puis les
05C4 PUSH AX           ; 3/4 de AX
05C5 MOV  CL,04        ; convertis
05C7 SHL  AX,CL        ; en
05C9 MOV  SI,AX        ; déplacement
05CB MOV  CX,0008      ; d'où DS:SI
05CE CLD               ; S copié
05CF REPZ              ; (huit mots)
05D0 MOVSW             ;
05D1 INC  BP           ; mise à jour
05D2 POP  AX           ; puis AX
05D3 SUB  AX,01C8      ; converti
05D6 SHL  AX,1         ; en adresse
05D8 MOV  SI,AX        ; TAS de S.
05DA DS:               ; lire N° du
05DB MOV  AX,[SI]      ; S suivant :
05DD CMP  AX,0100      ; S dernier?
05E0 JZ   05E5         ; oui : fini
05E2 JMP  05AC         ; -> BOUCLE
05E4 NOP               ;
05E5 RET               ; (BP = [R])
```

05F0 : Conversion à la représentation décimale.

```
05F0 PUSH CS           ;
05F1 POP  DS           ; DS := CS
05F2 MOV  AX,BP        ; (BP = [R])
05F4 MOV  BP,SP        ;
05F6 INC  BP           ;
05F7 INC  BP           ;
05F8 MOV  BX,[BP+00]   ; BX := (R)
05FB CMP  AX,[BX+4002]; vérif. [R]
05FF JZ   0605         ; accord ->
0601 CALL 0FD0         ; non : err.
0604 RET               ; logiciel
0605 SUB  DI,+10       ; calcul
0608 SHR  BX,1         ; de |R|
060A MOV  AX,[BX+4040]; en octets
060E SHL  AX,1         ; placé
0610 ADD  DI,AX        ; dans DI
0612 DEC  DI           ;
0613 DEC  DI           ;
0614 MOV  AX,[DI]      ;
```

```
0616 CMP  AH,00        ; |DI| pair?
0619 JZ   061C         ; non : suite
061B INC  DI           ; si : +1
061C INC  DI           ;
061D MOV  BP,F7FE      ;
0620 SUB  DI,8000      ; DI := |R|
0624 MOV  [0646],DI    ;
0628 MOV  WO [BP+00],2024; fin
062D DEC  BP           ; d'affichage
062E DEC  BP           ;
062F MOV  WO [BP+00],0A0D; st lgn.
0634:BOUCLE            ; divisions
0634 DEC  BP           ;
0635 DEC  BP           ; BP := BP-2
0636 CALL 0880         ; DIV 064
0639 MOV  [BP+00],AX   ; noter reste
063C CMP  WO [0646],+00; q = 0?
0641 JZ   0645         ; oui : fini
0643 JMP  0634         ; -> BOUCLE
0645 RET               ;
```

0646 : variable du programme 05F0.

```
0646 00 00 00 00
```

0658 : Affichage à l'écran.

```
0658 MOV  AX,[BP+00]   ;
065B CMP  AL,30        ; positionne-
065D JA   0660         ; ment sur
065F INC  BP           ; 1er octet
0660 MOV  DX,BP        ; à afficher
0662 MOV  AH,09        ;
0664 INT  21           ;
0666 RET               ;
```

0670 : AFFICHAGE DU RESULTAT.

```
0670 PUSH DI           ;
0671 PUSH DS           ;
0672 PUSH ES           ;
0673 PUSH CS           ;
0674 POP  DS           ; DS := CS
0675 MOV  DX,08C0      ; 'Fib ('
0678 MOV  AH,09        ;
067A INT  21           ;
067C CS:               ;
067D MOV  CX,[0700]    ; lire N
0681 CALL 07A0         ; l'afficher
0684 MOV  DX,08C8      ; ') = '
0687 MOV  AH,09        ; et st lgn.
0689 INT  21           ;
068B POP  ES           ;
068C POP  DS           ;
068D POP  DI           ; DI retrouvé
068E PUSH DI           ; et resauvé
068F CS:               ;
0690 MOV  AX,[DI+4000] ;
0694 AND  AX,F000      ; test pour
0697 CMP  AX,0000      ; versions
069A JNZ  06A9         ; futures
069C POP  DI           ;
069D PUSH DI           ;
069E CALL 05A0         ; copie 8000
06A1 CALL 05F0         ; conversion
06A4 CALL 0658         ; affichage
06A7 POP  DI           ;
06A8 RET               ;
06A9 CLC               ;
06AA RET               ;
```

0700 : variable (argument de la fonction Fib).

```
0700 00 00
```

07A0 : Concersion hexadécimal -> décimal et affichage.

0770-07E3 : identique aux instructions 0130-01A9 de l'annexe 3 (fonction d'Ackermnan)

0880 : division par cent.

```
0880 PUSH BX           ; BX et CX
0881 PUSH CX           ; sauvés
0882 MOV  BX,0064      ; BX : d
0885 MOV  DI,8000      ;
0888 MOV  CX,[0646]    ; nbre de
088C ADD  DI,CX        ; divisions
088E PUSH DI           ; DI : tête
088F XOR  AH,AH        ; r := 0
0891:BOUCLE            ;
0891 DEC  DI           ; suivant
0892 MOV  AL,[DI]      ; lire
0894 DIV  BL           ; diviser
0896 MOV  [DI],AL      ; écrire
0898 LOOP 0891         ; -> BOUCLE
089A XOR  AL,AL        ;
089C XCHG AH,AL        ; AL := r
089E MOV  BL,0A        ; conversion
08A0 DIV  BL           ; décimale de
08A2 XCHG AH,AH        ; AL
08A4 ADD  AX,3030      ; chiffres
08A7 POP  DI           ;
08A8 PUSH AX           ;
08A9 DEC  DI           ; lire décim.
08AA MOV  AL,[DI]      ; de tête :
08AC CMP  AL,00        ; nulle?
08AE JNZ  08B4         ; non : suite
08B0 DEC  WO [0646]    ; |q|:=|q|-1
08B4 POP  AX           ;
08B5 POP  CX           ;
08B6 POP  BX           ;
08B7 RET               ;
```

08C0 : message du programme d'affichage.

```
08C0 0D 0A 'Fib (' 24
08C8 ') = ' 0D 0A 24
```

08E0 : copyright

cf. 0428 à l'annexe 3.

0FD0 : gestionnaire des erreurs.

identique au programme de même adresse de l'annexe 2

ANNEXE 5 : factorielle N.

```
   0100: PROGRAMME PRINCIPAL.
0100 MOV  AX,CS          ;
0102 MOV  DS,AX          ;
0104 MOV  ES,AX          ;
0106 MOV  SS,AX          ;
0108 NOP                 ;
0109 CALL 0120           ; CX := X
010C CALL 0150           ; (DI) = X!
010F CALL 0AB0           ; afficher X!
0112 MOV  AX,4C00        ;
0115 INT  21             ;
   0150: Factorielle de CX.
0150 CALL 0740           ; pile et rg.
0153 MOV  SI,0004        ; initialiser
0156 MOV  DI,0008        ; DI et SI
0159 INC  CX             ; init. CX
015A:BOUCLE              ; sur X
015A LOOP 015E           ; (loop en
015C JMP  0184           ; tête)
015E PUSH CX             ; sauvegarde
015F PUSH SI             ; de
0160 PUSH DI             ; registres
0161 PUSH BP             ;
0162 CALL 0610           ; (SI)*k
0165 POP  BP             ;
0166 CALL 0848           ; ret. mul?
0169 JZ   0176           ; non: suite
016B CALL 0860           ; oui: report
016E JNB  0176           ; fait: suite
0170 POP  CX             ; dépassement
0171 CALL 0FD0           ; de capacité
0174 STC                 ; erreur
0175 RET                 ;
0176 CS:                 ;
0177 LES  DI,[0A28]      ; ES:DI = k
017B CALL 0220           ; k := k+1
017E NOP                 ;
017F POP  SI             ; on retrouve
0180 POP  DI             ; les
0181 POP  CX             ; registres
0182 JMP  015A           ; -> BOUCLE
0184 MOV  DI,SI          ; DI: résult.
0186 PUSH DI             ;
0187 CALL 0980           ; taille der.
018A POP  DI             ;
018B CALL 0834           ; ôter pile
018E CLC                 ;
018F RET                 ; retour
   message du programme 0120
0190 0D 0A 'Entrer N' 0D 0A '  '
019E ' > ' 24'
   0200-06FD :
```

Voir les instructions de mêmes adresses de l'annexe 2.

```
   0740: Initialisation des
registres.
0740 MOV  AH,4A          ; retr. zone
0742 MOV  BX,1000        ; actuel pr.
0745 INT  21             ;
0747 JNB  074E           ; fait: suite
0749 CALL 0FD0           ; erreur
074C STC                 ;
074D RET                 ;

074E MOV  AH,48          ; allouer
0750 MOV  BX,1000        ; 1ère zone
0753 INT  21             ; de 64 Ko
0755 JNB  075C           ; fait: suite
0757 CALL 0FD0           ; non: RAM
075A STC                 ; pleine
075B RET                 ;
075C MOV  ES,AX          ; ES sur zone
075E MOV  BP,SP          ; réservation
0760 SUB  BP,+18         ; espace
0763 CLI                 ; sur pile:
0764 MOV  SP,BP          ; SP corrigé
0766 STI                 ;
0767 MOV  AX,[BP+18]     ; déplacer
076A MOV  [BP+00],AX     ; adr. retour
076D INC  BP             ; désormais
076E INC  BP             ; BP fixe
076F PUSH CX             ; sauver X
0770 PUSH ES             ; préparer
0771 PUSH CS             ; STOSW
0772 POP  ES             ; pour initia
0773 XOR  AX,AX          ; liser la
0775 MOV  DI,3800        ; table 3800
0778 MOV  CX,0021        ; (021 mots)
077B XOR  AX,AX          ;
077D CLD                 ;
077E REPZ                ;
077F STOSW               ;
0780 POP  AX             ; adr. zone
0781 MOV  [BP+16],AX     ; mise sur
0784 MOV  [BP+12],AX     ; pile aux
0787 MOV  [BP+0A],AX     ; emplacem.
078A MOV  [BP+06],AX     ; voulus
078D CS:                 ;
078E MOV  [0A2A],AX      ; init. adr.
0791 CS:                 ; absolue
0792 MOV  WO [0A28],1C80;  de k
0798 MOV  DI,3800        ; 1ère entrée
079B MOV  [DI],AX        ; table 3800
079D ADD  DI,+20         ; puis taux
07A0 MOV  WO [DI],0E35; de rempl.
07A4 ADD  DI,+20         ; puis
07A7 INC  WO [DI]        ; 1*64 Ko
07A9 PUSH AX             ; (DS = CS)
07AA POP  ES             ; ES sur zone
07AB XOR  AX,AX          ; zone mise à
07AD MOV  CX,8000        ; zéro par
07B0 XOR  DI,DI          ; STOSW
07B2 CLD                 ; (8000 mots)
07B3 REPZ                ;
07B4 STOSW               ;
07B5 ES:                 ; initialiser
07B6 MOV  WO [0000],0100; la TAS
07BC ES:                 ; pour trois
07BD MOV  WO [0002],0100; secteurs
07C3 ES:                 ; (k, (SI)
07C4 MOV  WO [0004],0100; et (DI))
07CA MOV  BX,4000        ; répertoire:
07CD MOV  WO [BX],01C8; (DS = CS)
```

```
07D1 INC  BX          ; initialiser
07D2 INC  BX          ; trois
07D3 MOV  WO [BX],0001; entrées
07D7 INC  BX          ; table 4000
07D8 INC  BX          ; (n° complet
07D9 MOV  WO [BX],01C9; de secteur
07DD INC  BX          ; TAS et
07DE INC  BX          ; nombre de
07DF MOV  WO [BX],0001; secteurs
07E3 INC  BX          ; formant le
07E4 INC  BX          ; registre)
07E5 MOV  WO [BX],01CA;
07E9 INC  BX          ; puis, sur
07EA INC  BX          ; pile,
07EB MOV  WO [BX],0001; déplacement
07EF MOV  WO [BP+14],0002; pour
07F4 MOV  WO [BP+10],1C90; chaque
07F9 MOV  WO [BP+08],0004; adresse
07FE MOV  WO [BP+04],1CA0; absolue
0803 POP  CX          ; X retrouvé
0804 MOV  [0AF0],CX   ; garder X
0808 MOV  BX,4040     ;
080B XOR  AX,AX       ;
080D MOV  [BP+02],AX  ;
0810 MOV  [BP+0E],AX  ;
0813 MOV  WO [BP+00],0008;
0818 MOV  WO [BP+0C],0004;
081D INC  AX          ; trois fois
081E MOV  [BX],AX     ; 1 dans
0820 MOV  [BX+02],AX  ; la table
0823 MOV  [BX+04],AX  ; 4040
0826 ES:              ;
0827 MOV  [1C90],AX   ; initialiser
082A ES:              ; les trois
082B MOV  [1CA0],AX   ; registres
082E ES:              ; à 1
082F MOV  [1C80],AX   ;
0832 RET              ;
```

0834: Réaménagement de la pile.

```
0834 DEC  BP          ; inversion
0835 DEC  BP          ; stricte des
0836 MOV  AX,[BP+00]  ; instruct.
0839 MOV  [BP+18],AX  ; 075E-076F
083C MOV  BP,SP       ; (notamment,
083E ADD  BP,+18      ; adr. retour
0841 CLI              ; du progr.
0842 MOV  SP,BP       ; placée
0844 STI              ; sous SP)
0845 RET              ;
```

0848: Taille du dernier secteur.

```
0848 CS:              ; ES:DI mis
0849 LES  DI,[0A30]   ; sur la fin
084D ADD  DI,+0E      ; du secteur
0850 MOV  CX,0008     ; 08 mots
0853 XOR  AX,AX       ; chercher
0855 STD              ; 1er mot
0856 REPZ             ; non zéro
0857 SCASW            ;
0858 RET              ;
```

0860: Adjonction de la retenue au produit.

```
0860 MOV  AX,[BP-06]  ; lire DI
0863 CMP  AX,0008     ; DI = [BP]?
0866 JZ   0890        ; oui: suite
0868 CS:              ; non:
0869 MOV  AL,[0864]   ; réécrire
086C CS:              ; instruction
086D MOV  AH,[097E]   ; 0863 et 97E
0871 CS:              ; pour
0872 MOV  [0864],AH   ; prochain
0876 CS:              ; test
0877 MOV  [097E],AL   ;
087A PUSH BP          ; sauver BP
087B MOV  CX,0006     ;
087E:BOUCLE           ; permuter
087E MOV  BX,[BP+00]  ; [BP] et
0881 MOV  AX,[BP+0C]  ; [BP+0C]
0884 MOV  [BP+00],AX  ; sur 06 mots
0887 MOV  [BP+0C],BX  ;
088A INC  BP          ;
088B INC  BP          ;
088C LOOP 087E        ; -> BOUCLE
088E POP  BP          ; BP retrouvé
088F NOP              ;
0890 MOV  BX,[BP+02]  ; lire taux
0893 SHL  BX,1        ; zone dans
0895 CS:              ; table 3820
0896 CMP  WO [BX+3820],+00; plein?
089B JNZ  08A8        ; non suite
089D CALL 09B8        ; oui: alloc.
08A0 JNB  08A8        ; fait: suite
08A2 CALL 0FD0        ; échec: RAM
08A5 STC              ; pleine
08A6 RET              ;
08A7 NOP              ;
08A8 CS:              ; mettre en
08A9 MOV  AX,[BX+3800]; ES adr. z.
08AD MOV  ES,AX       ; (t. 3800)
08AF XOR  DI,DI       ; chercher
08B1 XOR  AX,AX       ; secteur
08B3 MOV  CX,0E38     ; libre (1er
08B6 CLD              ; mot 00 en
08B7 REPNZ            ; TAS)
08B8 SCASW            ;
08B9 JZ   08C0        ; trouvé
08BB CALL 0FD0        ; non: erreur
08BE STC              ;
08BF RET              ;
08C0 DEC  DI          ; corriger
08C1 DEC  DI          ; DI (SCASW)
08C2 ES:              ; marquer
08C3 MOV  WO [DI],0100; der. sect.
08C7 CS:              ; décrément.
08C8 DEC  WO [BX+3820]; compteur
08CC MOV  AX,DI       ; transfo DI
08CE SHR  AX,1        ; en n° de
08D0 ADD  AX,01C8     ; secteur
08D3 PUSH AX          ; pour TAS
08D4 MOV  CL,0B       ;
08D6 SHL  BX,CL       ;
08D8 OR   AX,BX       ; n° prêt
08DA LDS  SI,[BP+08]  ; adr. TAS
08DD MOV  [SI],AX     ; inscript.
08DF POP  AX          ; transfo
08E0 MOV  CL,04       ; demi n°
08E2 SHL  AX,CL       ; en adresse
08E4 MOV  [BP+04],AX  ; rangée
08E7 MOV  AX,ES       ; adr. seg.
08E9 MOV  [BP+06],AX  ; placée sur
08EC MOV  [BP+0A],AX  ; pile
08EF MOV  [BP+08],DI  ; dépl. rangé
08F2 LES  DI,[BP+04]  ; copier
```

```
08F5 CS:                ; la retenue
08F6 LDS  SI,[0A30]     ; mul sur le
08FA MOV  CX,0008       ; secteur
08FD CLD                ; réservé
08FE REPZ               ; (MOVSW
08FF MOVSW              ; 08 mots)
0900 MOV  BX,[BP+00]    ; noter
0903 CS:                ; nouv. tail.
0904 INC  WO [BX+4002]; en répert.
0908 MOV  AX,[BP-02]    ; lire 'CX'
090B CMP  AX,0001       ; terminé?
090E JA   0912          ; non: (SI)
0910 CLC                ; oui:
0911 RET                ; retour
0912 MOV  BX,[BP+0E]    ; au tour
0915 SHL  BX,1          ; de (SI):
0917 CS:                ; on réserve
0918 CMP  WO [BX+3820],+00; un
091D JNZ  092A          ; secteur
091F CALL 09B8          ; qu'on lui
0922 JNB  092A          ; adjoint
0924 CALL 0FD0          ; car ce
0927 STC                ; registre
0928 RET                ; recevra le
0929 NOP                ; produit
092A CS:                ; par k au
092B MOV  AX,[BX+3800]; prochain
092F MOV  ES,AX         ; tour de
0931 XOR  DI,DI         ; boucle;
0933 XOR  AX,AX         ; on effectue
0935 MOV  CX,0E38       ; les mêmes
0938 CLD                ; opérations
0939 REPNZ              ; que pour
093A SCASW              ; (DI);
093B JZ   0942          ; ces instr.
093D CALL 0FD0          ; ont été
0940 STC                ; écrites
0941 RET                ; sous DEBUG
0942 DEC  DI            ; par copie
0943 DEC  DI            ; via m des
0944 ES:                ; instruct.
0945 MOV  WO [DI],0100; 0890-0908
0949 CS:                ; corrigées
094A DEC  WO [BX+3820]; par e
094E MOV  AX,DI         ; aux adr.
0950 SHR  AX,1          ; ad hoc:
0952 ADD  AX,01C8       ; e 912 0E
0955 PUSH AX            ; e 95E 14
0956 MOV  CL,0B         ; e 968 10
0958 SHL  BX,CL         ; e 96D 12
095A OR   AX,BX         ; e 970 16
095C LDS  SI,[BP+14]    ; e 973 14
095F MOV  [SI],AX       ;
0961 POP  AX            ;
0962 MOV  CL,04         ;
0964 SHL  AX,CL         ;
0966 MOV  [BP+10],AX    ;
0969 MOV  AX,ES         ;
096B MOV  [BP+12],AX    ;
096E MOV  [BP+16],AX    ;
0971 MOV  [BP+14],DI    ;
0974 MOV  BX,[BP+0C]    ;
0977 CS:                ;
0978 INC  WO [BX+4002];
097C CLC                ;
097D RET                ;
```

variable du programme 0860

```
097E 04 00
```

0980: Taille du dernier secteur (cas sans extension).

```
0980 PUSH BP            ; sauver BP
0981 CMP  DI,[BP+00]    ; DI = [BP]?
0984 JZ   098A          ; oui: suite
0986 ADD  BP,+0C        ; non: donc
0989 NOP                ; tester +0C
098A LES  DI,[BP+04]    ; ES:DI fin
098D ADD  DI,+0E        ; du secteur
0990 XOR  AX,AX         ; recherche
0992 MOV  CX,0008       ; du 1er mot
0995 STD                ; non zéro
0996 REPZ               ;
0997 SCASW              ;
0998 JNZ  09A0          ; trouvé
099A POP  BP            ; sinon
099B CALL 0FD0          ; erreur
099E STC                ;
099F RET                ;
09A0 INC  CX            ; nombre de
09A1 MOV  BX,[BP+00]    ; mots
09A4 SHR  BX,1          ; à écrire
09A6 CS:                ; en répert.
09A7 MOV  [BX+4040],CX; (t. 4040)
09AB POP  BP            ; BP retrouvé
09AC CLC                ;
09AD RET                ;
```

09B8 : Allouer une nouvelle zone.

```
09B8 PUSH BX            ; sauver BX
09B9 MOV  AH,48         ; allouer
09BB MOV  BX,1000       ; 64 Ko
09BE INT  21            ;
09C0 POP  BX            ; rappeler BX
09C1 JNB  09C6          ; oui : suite
09C3 STC                ; RAM pleine
09C4 RET                ;
09C5 NOP                ;
09C6 INC  BX            ; incrémenter
09C7 INC  BX            ; BX
09C8 CS:                ;
09C9 MOV  [BX+3800],AX; noter seg
09CD CS:                ;   zone
09CE MOV  WO [BX+3820],0E38; taux
09D4 CS:                ; noter :
09D5 INC  WO [3840]     ; +1 zone
09D9 CLC                ; fait
09DA RET                ;
```

0AB0 : AFFICHAGE.

0AB0-0E0D : Identique aux instructions 05A0-08FD de l'annexe 4 (nombres de Fibonacci).

0FD0 : gestionnaire des erreurs.

identique au programme de même adresse de l'annexe 2

ANNEXE 6 : Simulation de la fonction EXEC (.EXE seuls)

```
   0100: Programme principal.
0100 MOV  AX,CS       ;
0102 MOV  DS,AX       ;
0104 MOV  ES,AX       ;
0106 MOV  SS,AX       ;
0108 NOP              ;
0109 MOV  DX,0392     ; nom fichier
010C MOV  AX,3D00     ; supposé
010F INT  21          ; installé
0111 MOV  [03DC],AX   ; sésame
0114 MOV  BX,AX       ; mis en BX
0116 MOV  AX,4202     ; mesure de
0119 MOV  CX,0000     ; la taille
011C MOV  DX,0000     ; du fichier
011F INT  21          ;
0121 MOV  [03D6],AX   ; sauver la
0124 MOV  [03D8],DX   ; taille
0128 MOV  AX,4200     ; remise du
012B MOV  CX,0000     ; pointeur
012E MOV  DX,0000     ; fichier
0131 INT  21          ; à 0000
0133 MOV  DX,1000     ; lecture
0136 MOV  CX,0100     ; du début de
0139 MOV  BX,[03DC]   ; l'en-tête
013D MOV  AH,3F       ; sur CS:1000
013F INT  21          ;
0141 MOV  AX,[03D6]   ; calcul de
0144 PUSH AX          ; la taille
0145 MOV  CL,04       ; nécessaire
0147 SHR  AX,CL       ; au fichier
0149 POP  BX          ; en
014A PUSH AX          ; paragraphes
014B SHL  AX,CL       ; soit T
014D CMP  AX,BX       ;
014F POP  AX          ;
0150 JZ   0153        ;
0152 INC  AX          ;
0153 PUSH AX          ;
0154 MOV  AX,[03D8]   ;
0157 MOV  CL,0C       ;
0159 SHL  AX,CL       ;
015B POP  BX          ;
015C ADD  AX,BX       ;
015E SUB  AX,[1008]   ;
0162 MOV  [03E6],AX   ; T noté
0165 CLC              ;
0166 ADD  AX,0010     ; T+10 (PSP)
0169 ADD  AX,[100C]   ; T maxi
016D JNB  0172        ; T <= FFFF
016F MOV  AX,FFFF     ; non? : si!
0172 PUSH AX          ; sauver T
0173 MOV  BX,1000     ; espace
0176 MOV  AH,4A       ; chargeur
0178 INT  21          ; à 64 Ko
017A POP  BX          ; retrouver T
017B MOV  AX,4800     ; allouer
017E INT  21          ; T paragr.
0180 JNB  01B4        ; oui : suite
0182 MOV  AX,[03E6]   ; comparer
0185 ADD  AX,0010     ; avec le
0188 ADD  AX,[100A]   ; minimum
018C CMP  AX,BX       ;
018E JBE  01A4        ; -> allouer
0190 MOV  DX,0404     ; 'alloc.
0193 MOV  AH,09       ; impossible'
0195 INT  21          ;
0197 MOV  BX,[03DC]   ; fermer le
019B MOV  AH,3E       ; fichier
019D INT  21          ;
019F MOV  AX,4C00     ; terminaison
01A2 INT  21          ;
01A4 MOV  AX,4800     ; alloc. mini
01A7 INT  21          ;
01A9 JNB  01B4        ; oui : suite
01AB MOV  DX,0480     ; non:
01AE MOV  AH,09       ; message
01B0 INT  21          ; RAM pleine
01B2 JMP  019F        ; -> fin
01B4 MOV  [03DA],AX   ; lecture du
01B7 MOV  [03E8],BX   ; fichier :
01BB MOV  AX,[1008]   ; toute
01BE MOV  CL,04       ; l'en-tête
01C0 SHL  AX,CL       ;
01C2 PUSH AX          ;
01C3 MOV  AX,4200     ; pointeur
01C6 MOV  CX,0000     ; à zéro
01C9 MOV  DX,0000     ;
01CC MOV  BX,[03DC]   ;
01D0 INT  21          ;
01D2 POP  CX          ; |en-tête|
01D3 MOV  DX,1000     ; CS:1000
01D6 MOV  AH,3F       ; pour
01D8 INT  21          ; l'en-tête
01DA MOV  AX,[03D6]   ; puis
01DD CMP  AX,CX       ; calcul de
01DF JNB  01F1        ; la taille
01E1 DEC  WO [03D8]   ; du progr.
01E5 PUSH AX          ;
01E6 MOV  AX,FFFF     ;
01E9 SUB  AX,CX       ;
01EB INC  AX          ;
01EC POP  DX          ;
01ED ADD  AX,DX       ;
01EF JMP  01F3        ;
01F1 SUB  AX,CX       ;
01F3 PUSH [03DA]      ;
01F7 POP  DS          ;
01F8 MOV  DX,0100     ; (préfixe)
01FB MOV  CX,AX       ;
01FD MOV  AH,3F       ; puis lire
01FF INT  21          ;
0201 CS:              ;
0202 MOV  AX,[03D8]   ;
0205 CMP  AX,0000     ; tout lu?
0208 JZ   0225        ; oui : suite
020A CS:              ; non :
020B MOV  CX,[03D8]   ; (fich.
020F SHL  CX,1        ; > 64 Ko)
```

```
0211:BOUCLE             ; chargement
0211 MOV  AX,DS         ; du fichier
0213 ADD  AX,0800       ;
0216 PUSH AX            ;
0217 POP  DS            ;
0218 PUSH CX            ;
0219 MOV  CX,8000       ; par 32 Ko
021C XOR  DX,DX         ; à la fois
021E MOV  AH,3F         ; lire
0220 INT  21            ;
0222 POP  CX            ;
0223 LOOP 0211          ; -> BOUCLE
0225 PUSH CS            ;
0226 POP  DS            ;
0227 MOV  BX,[03DC]     ; sésame
022B MOV  AH,3E         ; fermer le
022D INT  21            ; fichier
022F MOV  AX,[03DA]     ; fabrication
0232 PUSH AX            ; d'un
0233 ADD  AX,[03E8]     ; préfixe
0237 POP  ES            ; minimum :
0238 ES:                ;
0239 MOV  [0002],AX     ;
023C MOV  AX,[002C]     ;
023F ES:                ;
0240 MOV  [002C],AX     ;
0243 JMP  0246          ; -> suite
```

0246 : Corrections des adresses à redresser.

```
0246 MOV  AX,[03DA]     ; zone
0249 ADD  AX,0010       ; + préfixe
024C MOV  CX,[1006]     ; nombre
0250 JCXZ 0278          ; si aucun ->
0252 MOV  SI,0018       ; adresse du
0255 MOV  SI,[SI+1000]; premier
0259:BOUCLE             ;
0259 LES  DI,[SI+1000]; lieu
025D ADD  SI,+04        ; prochain
0260 MOV  DX,ES         ; calcul
0262 ADD  DX,AX         ; du
0264 MOV  ES,DX         ; CS local
0266 ES:                ; correction
0267 ADD  [DI],AX       ; faite
0269 LOOP 0259          ; -> BOUCLE
026B JMP  0278          ; -> suite
```

0278 : Redéfinition de l'interruption DOS 021.

```
0278 XOR  AX,AX         ; l'entrée
027A MOV  DS,AX         ; ad hoc est
027C LES  BX,[0084]     ; en 000:0084
0280 CS:                ; on sauve
0281 MOV  [03D2],BX     ; le vrai
0285 CS:                ; gestion-
0286 MOV  [03D4],ES     ; naire
028A MOV  WO [0084],02A8; nouveau :
0290 MOV  [0086],CS     ; CS:02A8
0294 JMP  0310          ; -> suite
```

variable de l'interruption détournée

```
02A6 0000               ;
```

02A8 : la nouvelle interruption.

```
02A8 PUSHF              ; sauver (F)
02A9 CMP  AH,4C         ; terminer?
02AC JNZ  02EE          ; non : INT
02AE CS:                ; si:
02AF POP  [02A6]        ; 02A6 := (F)
02B3 CS:                ;
02B4 MOV  [0300],DI     ; sauver DI
02B8 XOR  AX,AX         ;
02BA MOV  DS,AX         ; DS := 0000
02BC CS:                ; lire vraie
02BD LES  DI,[03D2]     ; adresse
02C1 MOV  [0084],DI     ; la réins-
02C5 MOV  [0086],ES     ; taller
02C9 MOV  AX,CS         ;
02CB MOV  DS,AX         ; DS := CS
02CD MOV  DI,[0300]     ; DI restauré
02D1 MOV  AX,[03E4]     ; restaurer
02D4 CLI                ; la pile du
02D5 MOV  SS,AX         ; présent
02D7 STI                ; programme
02D8 CS:                ;
02D9 MOV  AX,[03E2]     ;
02DC MOV  SP,AX         ;
02DE MOV  DX,03F0       ; message de
02E1 MOV  AH,09         ; programme
02E3 INT  21            ; terminé
02E5 PUSH [02A6]        ;
02E9 POPF               ; (F) rétabli
02EA JMP  019F          ; -> suite
02ED NOP                ; (ici = fin)
02EE POPF               ; (F) rétabli
02EF CS:                ; saut à la
02F0 JMP  FAR [03D2]    ; vraie INT
02F4                    ; (-> IRET)
```

0310 : Lancer l'exécution.

```
0310 PUSH CS            ;
0311 POP  DS            ; DS := CS
0312 MOV  AX,[03DA]     ; calcul
0315 ADD  AX,0010       ; du CS
0318 MOV  BX,1000       ; initial
031B ADD  AX,[BX+16]    ; [03E0] :=
031E MOV  [03E0],AX     ; CS initial
0321 MOV  AX,[BX+14]    ; [03DE] :=
0324 MOV  [03DE],AX     ; IP initial
0327 MOV  [03E2],SP     ; sauver
032B MOV  AX,SS         ; SS:SP
032D MOV  [03E4],AX     ; actuels
0330 PUSH [03DA]        ;
0334 POP  ES            ; ES, DS
0335 PUSH ES            ; fixés
0336 POP  DS            ;
0337 CS:                ;
0338 MOV  SP,[BX+10]    ; SP fixé
033B CS:                ;
033C MOV  AX,[BX+0E]    ; calcul de
033F CS:                ; SS
0340 ADD  AX,[03DA]     ;
0344 ADD  AX,0010       ; (préfixe)
0347 MOV  SS,AX         ; SS fixé
0349 NOP                ;
034A NOP                ;
034B CS:                ; on passe
034C JMP  FAR [03DE]    ; 'la main'
```

zone de réception du nom de fichier

```
0390 3E 00
```

puis 036 octets de 03A0 à 03D6

```
   variables
03D6 00 00 00 00 00 00 00 00
   adresse absolue du JMP FAR
03DE 00 00 00 00
   variables
03E2 00 00 00 00 00 00 00 00 00
   messages
03F0 0D 0A '.EXE terminé' 0D 0A
0400 24
0404 0D 0A 'allocation impossib'
0419 'le,' 0D 0A 'exécution ajo'
042B 'urnée...' 0D 0A 24
0480 0D 0A 'erreur système,' 0D
0492 0A 'exécution ajournée...'
04A8 0D 0A 24
   copyright.
04B0 '(C) 1991 Maurice MARGENST'
04C8 'ERN '
```

ANNEXE 7 : FACTORIELLE N (2)

```
;
; == MACROS ====================
Afficher MACRO Chaine
  PUSH AX
  PUSH DX
  MOV  AH,09h
  MOV  DX,offset Chaine
  INT  21h
  POP  DX
  POP  AX
       ENDM
Numériser MACRO Tampon
LOCAL Boucle
;    conversion en base 16 de N,
;    résultat placé dans CX:
  XOR  CX,CX
  MOV  CL,byte ptr Tampon+1
  MOV  AX,0000
  MOV  BL,0Ah
  MOV  BP,offset Tampon
  ADD  BP,02
  XOR  DX,DX
Boucle:
  MUL  BL
  MOV  DL,[BP]
  SUB  DL,030h
  ADD  AX,DX
  INC  BP
  LOOP Boucle  ; AX := N en hexa
  MOV  CX,AX   ; CX := AX
       ENDM
; == PROGRAMME ================
Code SEGMENT
  ORG 0100h
  ASSUME  cs : Code, ds : Code
  ASSUME  es : Code, ss : Code
Début :
  MOV  AX,CS
  MOV  DS,AX
  MOV  ES,AX
  CLI
  MOV  SS,AX
  STI
  CALL LireN
  CALL Initialisations
  JCXZ Résultats
Boucle :
  PUSH CX
  PUSH BX
  MOV  CX,[BP] ; en DX (R),
;              ; ES:DI sur R
  CALL Multiplier
  MOV  [BP],CX
  POP  BX
  INC  BX
  POP  CX
  LOOP Boucle
Résultats :
  CALL Affichage
  MOV  AX,4C00h
  INT  021h
```

```
; == VARIABLES ET CONSTANTES ======
Répert EQU 04000h   ; répertoire
TDZ EQU 03800h ; table des zones
PremSect EQU 01C80h ; 1er secteur
RC EQU 0Dh      ; retour chariot
NL EQU 0Ah      ; nouvelle ligne
FM EQU 024h     ; fin de chaîne DOS
MessageEntrée  DB  RC,NL,'Entre'
   DB  'r N : ',RC,NL,FM
MessagedErreur  DB  RC,NL,'erre'
   DB  'ur trouvée en  ',FM
EntréeBrute  DB  06h,
             DB  00h, 06h DUP (?)
Saut  DB  RC,NL,FM
ValeurDeN  DW  00h
; == MACROS ======================
AllocationZone MACRO
LOCAL Fin
  MOV  AH,4Ah  ; on restreint la
  MOV  BX,1000h; place occupée par
  INT  021h    ; le programme
  JC   Fin     ;
  MOV  AH,48h  ; puis on réserve
  MOV  BX,1000h; l'espace des
  INT  021h    ; registres
Fin:
               ENDM
InstallerLaPile MACRO
  MOV  BP,SP
  SUB  BP,08
  CLI
  MOV  SP,BP
  STI
  MOV  AX,[BP+08]
  MOV  [BP],AX
  INC  BP
  INC  BP
  MOV  word ptr [BP],0001h
               ENDM
EspaceDesRegistres MACRO
  PUSH DS
  PUSH CS
  POP  DS
  PUSH CX
  XOR  DI,DI   ; on remplit la zone
  XOR  AX,AX   ; par zéro
  MOV  CX,8000h; (ES a la valeur
  CLD          ; requise)
  REP  STOSW   ; % respecter cette
               ; zone initialisée
  POP  CX
  MOV  word ptr [ValeurDeN],CX
  MOV  word ptr DS:[Répert],01C8h
  MOV  word ptr DS:[Répert+02],01h
  MOV  word ptr DS:[Répert+40h],01h
  MOV  word ptr DS:[TDZ],ES
  MOV  word ptr DS:[TDZ+20h],0E38h
  MOV  word ptr DS:[TDZ+40h],01h
  MOV  word ptr ES:[PremSect],01h
  MOV  word ptr ES:[0000],0100h
  MOV  DI,PremSect
  MOV  DX,0001h
  POP  DS
                    ENDM
; == PROCEDURES ==================
Initialisations PROC NEAR
  AllocationZone
  JNC  ZoneAllouée
  CALL Erreurs ; CALL et non JMP du
;              ; fait de l'adresse!
ZoneAllouée :
  MOV  ES,AX
  InstallerLaPile
  EspaceDesRegistres
  XOR  BX,BX
  INC  BX
  RET
Initialisations ENDP
LireN PROC NEAR
  Afficher  MessageEntrée
  MOV  DX,offset EntréeBrute
  MOV  AH,0Ah
  INT  021h
  Afficher  Saut
  Numériser EntréeBrute
  RET
LireN   ENDP
  ORG  0220h
  ; programme de conversion d'hexa-
  ; décimal en décimal (mot)
  DB  '0123456789' 0000 0000 0000
```

code hexadécimal, sous forme de données, de la procédure 0140 du programme Ackermann

```
       ORG  0250h
Conversion PROC NEAR
```

code hexadécimal, sous forme de données, de la procédure 0160 du programme Ackermann (avec redressement d'adresses)

```
Conversion ENDP
  ORG  0560h
  ;  programmes d'affichage
```

code hexadécimal, sous forme de données, de la procédure 0880 du programme Fibonacci

```
  ORG  05A0h
Copieur  PROC NEAR
```

code hexadécimal, sous forme de données, de la procédure 05A0 du programme Fibonacci

```
Copieur  ENDP
  ORG  05F0h
Convertisseur  PROC NEAR
```

code hexadécimal, sous forme de données, de la procédure 05F0 du programme Fibonacci

```
Convertisseur  ENDP
  ORG  0658h
Afficheur  PROC NEAR
  DB  08Bh,046h,000h,03Ch,030h,077h
  DB  001h,045h,089h,0EAh,0B4h,009h
  DB  0CDh,021h,0C3h
Afficheur  ENDP
Interface  MACRO
LOCAL  Noter
  MOV  AX,[BP]
  MOV  CS:[Répert+02h],AX
  SHR  DX,1
  JNC  Noter
  INC  DX
```

```
Noter :
  MOV  CS:[Répert+40h],DX
          ENDM
Affichage PROC NEAR
  Interface    ; mise  en conformité
;              ; du répertoire
  MOV  CX,CS:[ValeurDeN]
  PUSH CS
  POP  DS
;              ; écriture de N
  CALL Conversion
;              ; puis de '! = '
  Afficher Factorielle
  XOR  DI,DI
  MOV  AX,CS:[4000h+DI]
  AND  AX,0F000h
  CMP  AX,00h
  JNE  Impossible
  PUSH DI
  CALL Copieur
  CALL Convertisseur
  CALL Afficheur
  POP  DI
  RET
Impossible :
  Afficher  PasPossible
  RET
Affichage ENDP
Factorielle  DB  '! = ',0Dh,0Ah
             DB  024h
PasPossible  DB  'impossible'
             DB  0Dh,0Ah,24h
  ORG  0728h
Reserver PROC NEAR
```

code hexadécimal, sous forme de données, de la procédure 0728 du programme de multi-précision

```
Reserver ENDP
      ORG  0FD0h
Erreurs PROC NEAR
      MOV  DX,offset MessagedErreur
      MOV  AH,09h
      INT  021h
```

code hexadécimal, sous forme de données, de la procédure d'erreurs du programme de multi-précision

```
  MOV  DX,offset Saut
  MOV  AH,09h
  INT  021h
  DB  0B8h,000h,04Ch,0CDh,021h,0C3h
Erreurs ENDP
  ORG  12A0h
Multiplier PROC NEAR
```

code hexadécimal, sous forme de données, des procédures depuis 12A0 du programme de multi-précision

```
Multiplier ENDP
Code ENDS
     END  Début
```

ANNEXE 8 :
TOURS DE HANOI (extraits).

0358: Tracer d'un disque.

```
0358 PUSH BP          ; d : disque
0359 PUSH DI          ;
035A PUSH CX          ;
035B CALL SI          ; adresse RAM
035D MOV  AL,DA       ; coin HG
035F ES:              ; écrire en
0360 MOV  [BX],AX     ; RAM vidéo
0362 PUSH BX          ; replacer
0363 INC  BX          ; BX
0364 INC  BX          ;
0365 PUSH DX          ;
0366 DEC  DX          ;
0367 DEC  DX          ;
0368 MOV  CX,DX       ; largeur d
036A MOV  DI,BX       ;
036C MOV  AL,C4       ; trait large
036E CLD              ;
036F REPZ STOSW       ; tracer
0371 ADD  BX,DX       ; décalage
0373 ADD  BX,DX       ;
0375 MOV  AL,BF       ; coin HD
0377 ES:              ;
0378 MOV  [BX],AX     ; l'écrire
037A POP  DX          ;
037B POP  BX          ;
037C ADD  BX,00A0     ; ligne suiv.
0380 MOV  AL,C1       ; raccord
0382 ES:              ; 'support'
0383 MOV  [BX],AX     ; à gauche
0385 DEC  DX          ;
0386 ADD  BX,DX       ;
0388 ADD  BX,DX       ;
038A MOV  AL,C1       ; puis
038C ES:              ; à droite
038D MOV  [BX],AX     ;
038F POP  CX          ;
0390 POP  DI          ;
0391 POP  BP          ;
0392 RET              ;
```

0394: Effacer un disque.

```
0394 PUSH BP          ;
0395 PUSH DI          ;
0396 PUSH CX          ;
0397 CALL SI          ;
0399 MOV  CX,DX       ; on efface
039B MOV  DI,BX       ; avec
039D MOV  AL,20       ; ' '
039F CLD              ; répété sur
03A0 REPZ STOSW       ; la largeur
03A2 ADD  BX,00A0     ; du disque
03A6 MOV  AL,C4       ; 'raccords'
03A8 ES:              ; remplacés
03A9 MOV  [BX],AX     ; par trait
03AB DEC  DX          ; large
03AC ADD  BX,DX       ;
03AE ADD  BX,DX       ;
03B0 ES:              ;
03B1 MOV  [BX],AX     ;
03B3 POP  CX          ;
03B4 POP  DI          ;
03B5 POP  BP          ;
03B6 RET              ;
```

0580 : Déplacement d'un disque.
ressources programme :

```
action définie p. 88 ; notations :
S : plot de départ (source)
D : plot de destination
I : plot intermédiaire
d : disque à déplacer
06A0 : adresse de base de S en 0240
06A2 : adresse de d sur S en 0240
06A4 : adresse de base de D en 0240
06A6 : adresse de d sur D en 0240
06A8 : DX (tracer et effacer)
06AA : n° de d
06AC : n° de S, n° de I
rappel : (BL,BH), coordonnées écran
```

programme

```
0580 PUSH AX          ; registres
0581 PUSH CX          ; sauvés
0582 PUSH DS          ; (AX,CX :
0583 PUSH SI          ; contexte)
0584 PUSH CS          ;
0585 POP  DS          ; DS := CS
0586 MOV  [06AC],AH   ; S puis D
058A MOV  [06AD],CH   ; notés
058E SUB  WO [06AC],0101; décalage
0594 XOR  AH,AH       ;
0596 MOV  AL,[06AC]   ; abscisse
0599 MOV  BL,0C       ; liée à S
059B MUL  BL          ; dans la
059D MOV  BP,024A     ; table 0240
05A0 ADD  BP,AX       ;
05A2 MOV  [06A0],BP   ; sauvegarde
05A6 MOV  AX,BP       ; calcul de
05A8 ADD  AL,[BP+01]  ; l'adresse
05AB ADD  AL,02       ; de d
05AD MOV  [06A2],AX   ; noté
05B0 XOR  AH,AH       ; puis mêmes
05B2 MOV  AL,[06AD]   ; calculs
05B5 MOV  BL,0C       ; pour D
05B7 MUL  BL          ;
05B9 MOV  BP,024A     ;
05BC ADD  BP,AX       ;
05BE MOV  [06A4],BP   ;
05C2 MOV  AX,BP       ; noté
05C4 ADD  AL,[BP+01]  ;
05C7 ADD  AL,02       ;
05C9 INC  AL          ;
05CB MOV  [06A6],AX   ;
05CE MOV  BP,[06A2]   ; calcul de
05D2 XOR  AH,AH       ; DX
05D4 MOV  AL,[BP+00]  ;
05D7 MOV  [06AA],AL   ;
05DA MOV  DL,[026E]   ;
05DE MUL  DL          ;
05E0 ADD  AL,AL       ;
05E2 INC  AL          ;
05E4 MOV  DL,AL       ;
05E6 XOR  DH,DH       ;
05E8 MOV  [06A8],DX   ; DX noté
05EC MOV  BP,[06A0]   ;
05F0 MOV  BH,[BP+00]  ;
05F3 DEC  BH          ; BH fixé
05F5 XOR  AH,AH       ;
05F7 MOV  AL,[06AC]   ;
05FA MOV  BL,16       ;
05FC MUL  BL          ;
05FE ADD  AX,000F     ;
0601 MOV  AH,BH       ; AX : abs. S
0603 PUSH AX          ;
0604 MOV  BL,AL       ;
0606 MOV  AX,DX       ;
0608 SHR  AX,1        ;
060A SUB  BL,AL       ; BL calculé
060C MOV  AH,07       ; vidéo
060E MOV  SI,0440     ;
0611 CALL 0230        ; calcul DS
0614 CALL 0394        ; effacer d
0617 POP  BX          ;
0618 CALL SI          ; adresse RAM
061A MOV  AX,07BA     ; 'raccord'
061D ES:              ; du plot re-
061E MOV  [BX],AX     ; découvert
0620 ADD  BX,00A0     ; (écritures
0624 MOV  AX,07D0     ; RAM vidéo)
0627 ES:              ;
0628 MOV  [BX],AX     ;
062A PUSH CS          ;
062B POP  DS          ; DS := CS
062C MOV  BP,[06A2]   ;
0630 MOV  BY [BP+00],00; effacement
0634 MOV  BP,[06A0]   ; S réactua-
0638 INC  BY [BP+00]  ;   -lisé
063B DEC  BY [BP+01]  ;
063E MOV  BP,[06A6]   ;
0642 MOV  AL,[06AA]   ;
0645 MOV  [BP+00],AL  ;
0648 MOV  BP,[06A4]   ;
064C DEC  BY [BP+00]  ;
064F INC  BY [BP+01]  ; D réactua-
0652 MOV  DX,[06A8]   ;   -lisé
0656 MOV  BP,[06A4]   ; on va re-
065A MOV  BH,[BP+00]  ; dessiner d
065D DEC  BH          ; BH calculé
065F XOR  AH,AH       ;
0661 MOV  AL,[06AD]   ;
0664 MOV  BL,16       ;
0666 MUL  BL          ;
0668 ADD  AX,000F     ;
066B MOV  AH,BH       ;
066D PUSH AX          ; AX = abs. D
066E MOV  BL,AL       ;
0670 MOV  AX,DX       ;
0672 SHR  AX,1        ;
0674 SUB  BL,AL       ; BL calculé
0676 MOV  AH,07       ; vidéo
0678 MOV  SI,0440     ;
067B CALL 0230        ; calcul DS
067E CALL 0358        ; dessiner d
0681 POP  BX          ;
0682 CALL SI          ; adresse RAM
0684 MOV  AX,07D0     ; 'raccord'
0687 ES:              ; du plot à
0688 MOV  [BX],AX     ; effacer
068A ADD  BX,00A0     ;
068E MOV  AX,07C4     ;
0691 ES:              ;
0692 MOV  [BX],AX     ;
0694 CALL 06B0        ; arrêt image
0697 POP  SI          ;
0698 POP  DS          ; registres
0699 POP  CX          ; restaurés
069A POP  AX          ;
069B RET              ;
```

BIBLIOGRAPHIE

Le lecteur soucieux d'approfondir le sujet après la lecture de ce livre peut se référer aux ouvrages indiqués ci-après. Nous précisons, pour chacun, son champ d'étude :

[1] CUTLAND N.J. — *Computability, an introduction to recursive function theory*. Cambridge University Press, 1980.

Introduction à la théorie des fonctions récursives à partir de la notion de machine à registre de Shepherdson et Sturgis considérée du seul point de vue théorique.

[2] MARGENSTERN M. — *Langage PASCAL et logique du premier ordre, tome 1*. Masson, 1989.
— *Langage PASCAL et logique du premier ordre, tome 2*. Masson, 1990.

Introduction à la programmation en *PASCAL* et à la logique. Fonctions récursives, machines de Turing et λ-calcul sont introduits et utilisés pour analyser le langage *PASCAL* mis à son tour à contribution pour modéliser ces modèles théoriques. Le second volume, tourné vers la théorie de la complexité des algorithmes introduit au λ-calcul typé.

[3] MINSKY M.L. — *Computation, Finite and Infinite Machines*. Prentice Hall, 1967.

Ouvrage classique, toujours d'actualité sur les machines théoriques : des automates finis à la machine de Turing en passant par les réseaux de neurones. Un regard, en passant, sur les fonctions récursives, moins étudiées, cependant que dans [1] ou [2].

[4] NOLIN L. — *Matériel et logiciel, tome 1, Cours de licence avec exercices*, Masson, 1988.
— *Matériel et logiciel, tome 2, Cours de maîtrise avec exercices*. Masson, 1988.

Ouvrage de référence sur le matériel et les relations qu'il entretient nécessairement avec les logiciels : l'implantation des opérations élémentaires et du traitement de base des instructions dans les circuits électroniques est examinée avec précision et clarté.

[5] STERN J. — *Fondements mathématiques pour l'informatique*. MacGraw Hill, 1990.

Introduction aux disciplines de la logique mathématique mises en oeuvre en informatique théorique. Théorie des langages, automates, fonctions récursives et machines de Turing.

[6] VIEILLEFOND C. — *Mise en oeuvre de l'iAPX 286*. Sybex, 1986.

Présentation du processeur 80286 sur le plan matériel et logiciel. Description détaillée du mode protégé et de son fonctionnement.

[7] Documentation INTEL — *80286 and 80287 Programmer's Reference Manual*, 1987.
— *80286 Operating Systems Writer's Guide*, 1986.
— *Microprocessors*, 1990.

[8] Documentation IBM — *Macro Assembler/2 Language Reference*, 1987

INDEX

Dans de nombreux cas, il convient de regarder également les pages qui suivent la page référencée.

Par ailleurs, la convention suivante a été prise en ce qui concerne les divers groupes de mnémoniques étudiés dans l'ouvrage :

majuscules standard : mnémoniques relatives au processeur 8086 ;

majuscules standard précédées d'un astérisque : mnémoniques des instructions Minsky ;

majuscules standard précédées du symbole ° : mnémoniques des instructions du mode protégé des processeurs 80x86, x ≥ 2 ;

majuscules italiques : directives de MASM.

MASSON Éditeurs
120, Bd Saint-Germain
75280 Paris Cedex 06
Dépôt légal : juillet 1991

Imprimé par Pierre Mardaga
12, rue Saint-Vincent, 4020 Liège
juin 1991

SYSTÈMES D'EXPLOITATION

- LES SYSTÈMES D'EXPLOITATION Structure et concepts fondamentaux C Lhermitte
  SYSTEMES D'EXPLOITATION Conception et fonctionnement D Barron
  PROGRAMMATION DE SYSTÈME SOUS CP/M-80 L E Hugues
  MULTICS Guide de l'usager J Berstel, J F Perrot
  UNIX System V Système et environnement A B Fontaine et Ph Hammes
  UNIX Programmation avancée M J Rochkind
  LE SYSTÈME D'EXPLOITATION PICK M Bull

MICROPROCESSEURS ET ARCHITECTURE DE L'ORDINATEUR

- TECHNOLOGIE DES ORDINATEURS pour les I U T et B T S Informatique Avec exercices P A Goupille
  LE MICROPROCESSEUR MC 68000 Contrôle et mise au point des composants associés J W Coffron
  PRINCIPES DE FONCTIONNEMENT DES ORDINATEURS R Dowsing et F Woodhams
  LE MICROPROCESSEUR 16 BITS 8086/8088 Matériel Logiciel Système d'exploitation A B Fontaine
  LES MICROPROCESSEURS 80286/80386 Nouvelles architectures PC A B Fontaine et F Barrand
- IMPLANTATION DES FONCTIONS USUELLES EN 68000 F Bret

RÉSEAUX

TÉLÉMATIQUE Téléinformatique et réseaux M Maiman
L'INTERFACE RS-232-C D Seyer
RÉSEAUX ET MICRO-ORDINATEURS Ph Jesty
ORDINATEURS. INTERFACES ET RÉSEAUX DE COMMUNICATION S Collin
LES RÉSEAUX LOCAUX Comparaison et choix J S Fritz, C F Kaldenbach et L M Progar
LE RÉSEAU SNA K C E Gee
CONNEXION DES MICROS AUX SYSTÈMES DE TÉLÉCOMMUNICATIONS J M Nilles
X 25 Protocoles pour les réseaux à commutation de paquets R J Deasington
RNIS. Concept et développement J Ronayne

INTELLIGENCE ARTIFICIELLE

INTELLIGENCE ARTIFICIELLE E Rich
LOGIQUES POUR L'INTELLIGENCE ARTIFICIELLE R Turner
LES SYSTÈMES INTELLIGENTS BASÉS SUR LA CONNAISSANCE W J Black
SYSTÈMES EXPERTS Concepts et exemples J L Alty et M J Coombs
PROGRAMMATION DES SYSTEMES EXPERTS EN PASCAL B Sawyer et D Foster
SYSTÈMES EXPERTS PROFESSIONNELS P Harmon et D King
TECHNOLOGIE DE L'INTELLIGENCE ARTIFICIELLE F Mizoguchi
APPORTS DE L'INTELLIGENCE ARTIFICIELLE AU GÉNIE LOGICIEL D Partridge
RÉFLEXIONS SUR L'INTELLIGENCE DES MACHINES 25 ans de recherches D Michie

INFORMATIQUE POUR L'ENTREPRISE

- MODÉLISATION DANS LA CONCEPTION DES SYSTÈMES D'INFORMATION, avec exercices commentés Acsiome
  INITIATION A L'INFORMATIQUE DE GESTION M Aumiaux
  LES FICHIERS INFORMATIQUES Conception et performances O Hanson
  COMPRENDRE LES BASES DE DONNÉES Théorie et pratique A Mesguich et B Normier
  SQL LANGAGE STRUCTURÉ D'INTERROGATION C J Hursch et J L Hursh
  L'AUDIT INFORMATIQUE Méthodes, règles, normes M Thorin
  APPLICATION SYSTEM Un système IBM de 4e génération J Rambaud
- INGÉNIERIE DES DONNÉES Bases de données, systèmes d'information, modèles et langages E Pichat, R Bodin

INFORMATIQUE INDUSTRIELLE ET APPLICATIONS SCIENTIFIQUES

MÉTHODES DE DÉVELOPPEMENT D'UN SYSTÈME À MICROPROCESSEURS A Amghar
ROBOTIQUE Contrôle, programmation, interaction avec l'environnement G Gini et M Gini
PASRO ET PASROC Le Pascal et le langage C appliqués à la robotique C Blume W Jacob et J Favaro
CONTRÔLE DES ROBOTS PAR MICRO-ORDINATEURS S Maekawa et E Ohno

- EXERCICES DE RECONNAISSANCE DE FORMES PAR ORDINATEUR Ph Fabre
  CALCUL DES FORMES PAR ORDINATEURS, J Woodwark
- INFOGRAHIE ET APPLICATIONS T. Liebling et H Rothlisberger
  GRAPHISME DANS LE PLAN ET DANS L'ESPACE AVEC TURBO-PASCAL 4 0 R Dony
  LE SYSTÈME GRAPHIQUE GKS R R A Hopgood, D A Duce, J R Gallop et D C Sutcliffe
  PROGRAMMATION EN INFOGRAPHIE Principes, exercices et programmes en C L Ammeraal
  FORMATION AU DIAGNOSTIC TECHNIQUE J Moustafiadès
  SPÉCIFICATION ET CONCEPTION DES SYSTÈMES Une méthodologie J P Calvez
- SPÉCIFICATION ET CONCEPTION DES SYSTÈMES Études de cas J. P Calvez

MANUELS INFORMATIQUES MASSON

MIM

ASSEMBLAGE

modélisation, programmation (80x86)

M. MARGENSTERN

Cet ouvrage est une initiation à l'assembleur des 80×86. Il en donne les rouages essentiels avec, toujours, les tenants et les aboutissants.

Les fondements théoriques des langages machines de *tous* les processeurs séquentiels, passés, présents et du proche avenir, y sont expliqués en termes simples et mathématiquement justifiés, dans le but d'apprendre au lecteur à se forger ses propres méthodes.

Dans sa seconde partie, l'ouvrage illustre les méthodes et les connaissances de la première partie : réalisation d'un mini-compilateur, simulation de fonctions-clés d'un système d'exploitation, les pièces maîtresses d'outils d'assemblage et de désassemblage ; des instruments de calcul multi-précision pouvant rivaliser avec les logiciels les plus réputés.

Les deux derniers chapitres introduisent au macro-assemblage d'une part et, d'autre part, aux cadres nouveaux de la protection, du privilège et de l'architecture à jeu d'instructions restreint.

Très utile aux professionnels de l'informatique, indispensable aux étudiants en informatique des universités et des grandes écoles, ce livre passionnera tous ceux qui veulent en savoir plus sur leur micro.

Maître de conférences à l'université Paris-Sud et membre du laboratoire d'informatique théorique et de programmation de l'Institut Blaise Pascal, ***Maurice Margenstern*** *est spécialiste de logique mathématique et dispose d'une solide expérience de la programmation sur gros systèmes et en micro-informatique.*

9 782225 825002

ISBN : 2-225-82500-9