Université Paris 13
Institut Galilée
Deug Mias 1ère année
2003-2004

Programmation Impérative
Polycopié de cours n° 2

Enseignants
A. Nazarenko et C. Recanati

Table des matières

3 Structure et fonctionnement d'un ordinateur

Nous allons examiner dans ce chapitre le mode de fonctionnement d'un ordinateur au niveau de l'unité centrale. Le mode de fonctionnement décrit ici s'applique aux ordinateurs possédant une architecture de type « Von Neumann » (la plupart des ordinateurs actuels). Commençons au préalable par présenter quelques unités courantes pour mesurer les capacités de traitement de l'information d'un ordinateur.

3.1 Quelques unités de mesures

a. Capacité mémoire

L'unité de base en informatique est l'unité binaire ou *bit* (binary unit). Toutefois, le bit est une unité d'information trop petite pour servir d'unité de mesure. Les quantités de mémoires dont dispose un ordinateur se mesure en fait en octets, un octet étant une séquence de 8 bits. On regroupe également les bits en mots, un mot étant constitué en général de quelques octets. On parle alors d'architecture 8, 16, 32, 64 ou 128 bits. Une architecture 8 bits est une architecture où le mot occupe 1 octet. Une architecture 128 bits est une architecture où un mot occupe 16 octets. Les PC actuels par exemple sont des machines 32 bits (i.e. un mot occupe 32 bits, soit 4 octets).

Pour décrire des quantités de mémoire plus importantes, on utilise :

Unité	Abréviation	Valeur
bit		
octet		8 bits
Kilo-octet	Ko	1024 octets
Mega-octet	Mo	1024 Ko
Giga-octet	Go	1024 Mo
Téra-octet	To	1024 Go

(notez : $1024 = 2^{10}$)

b. Fréquence d'horloge

La fréquence d'horloge mesure le nombre d'opérations élémentaires effectuées par le processeur pendant une seconde. Notez que le cycle d'exécution d'une instruction (cf. plus loin) peut nécessiter l'exécution de plusieurs opérations élémentaires et par conséquent de plusieurs battements d'horloge.

Cette fréquence se mesure en *Hertz* (Hz, MHz). Si un processeur possède une fréquence d'horloge de 200 MHz, cela signifie qu'il effectue 200.10^6 opérations élémentaires par seconde.

Aujourd'hui les PC ont des fréquences d'horloge entre 500 et 1000 MHz.

La fréquence d'horloge n'est pas une indication suffisante pour évaluer la rapidité d'un ordinateur : un ordinateur dont la fréquence d'horloge est plus élevée que celle d'un autre peut néanmoins être plus lent. En effet, la rapidité d'un ordinateur dépend aussi beaucoup de la vitesse à laquelle circule l'information entre le processeur et les autres composants de l'unité centrale. Si l'information circule lentement en dehors du processeur, il sera ralenti : beaucoup de cycles d'horloge seront consacrés à attendre des informations en provenance de l'extérieur.

La vitesse d'exécution dépend aussi beaucoup de la nature des opérations considérées : les opérations sur les réels sont en général bien plus longues à effectuer que les opérations sur les entiers. On mesure donc, en général, la vitesse d'un ordinateur à l'aide de deux nombres : le nombre d'opérations entières par seconde et le nombre d'opérations sur les réels par seconde. L'unité fréquemment utilisée est le *flops* (comme **fl**oating-point **o**perations **p**er **s**econd).

c. Débit

Le débit permet de mesurer la quantité de mémoire qu'un dispositif de stockage (disque dur, lecteur de CD-ROM) peut échanger par seconde avec l'extérieur.

L'unité est le *baud* (= bit/s). On utilise aussi les unités suivantes : octet/s, Mo/s, Go/s.

d. Temps d'accès

Le temps d'accès désigne le temps nécessaire pour accéder à l'information désirée sur un support de stockage mémoire. Ce temps ne prend pas en compte le transfert des données.

Quelques unités de mesure :

Unité	Abréviation	Valeur (en secondes)
milliseconde	ms	10^{-3}
microseconde	µs	10^{-6}
nanoseconde	ns	10^{-9}
picoseconde	ps	10^{-12}

Le temps d'accès moyen d'un disque dur est le temps moyen pour positionner la tête de lecture au début du bloc de mémoire auquel on veut accéder. Il est égal à 10ms.

Lorsque le temps d'accès aux mots mémoire gérés par la mémoire centrale est identique pour tous les mots, on parle de mémoire RAM (pour Random Access Memory) – la plus fréquemment rencontrée. Le temps nécessaire à l'écriture ou à la lecture d'un mot mémoire en mémoire RAM varie de 60 à 80 ns. Par extension, la mémoire RAM étant celle utilisée par la mémoire centrale, est souvent traduite en français par mémoire vive (par opposition à la mémoire ROM, pour Read Only Memory, qui désigne une mémoire externe uniquement accessible en lecture, comme celle d'un disque, et qui est qualifiée de mémoire morte).

3.2 Structure d'une machine Von Neumann

Dans une machine Von Neumann, on distingue traditionnellement deux parties :

- L'unité centrale qui assure le stockage et le traitement des données ainsi que le stockage des programmes et des résultats.
- L'unité d'échange qui a en charge des communications avec l'extérieur : entrée des données et des programmes ; sortie des résultats. Le clavier, le moniteur, la souris et l'ensemble des périphériques relèvent donc de cette notion abstraite d'unité d'échange.

Nous allons voir maintenant plus précisément les différents éléments constituant un ordinateur, à un niveau plus fin. Le schéma ci-dessous en décrit les différents éléments. Nous reviendrons plus loin sur ce schéma pour expliquer comment s'exécute une instruction.

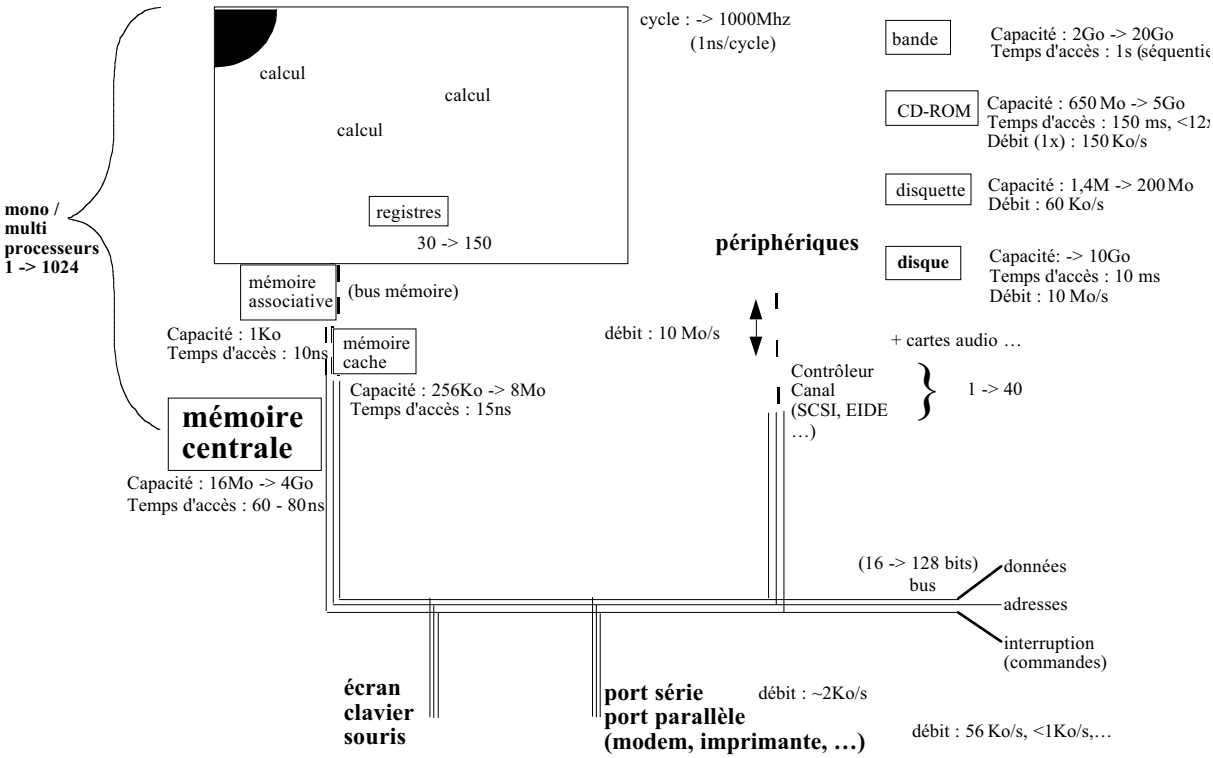


Schéma fonctionnel simplifié d'un ordinateur.

Nous nous intéressons plus particulièrement à l'unité centrale. Comme on le voit sur le schéma ci-dessus, elle se compose des éléments suivants :

- La **mémoire centrale** qui permet de stocker les données et les programmes.
- Le **processeur** (*Computing Processing Unit* ou CPU) qui interprète et exécute les programmes. Le processeur se compose de deux unités fonctionnellement séparées : **l'unité arithmétique et logique** (UAL) et **l'unité de commande** (UC). L'UAL est la zone du CPU qui effectue les opérations arithmétiques et logiques (les résultats intermédiaires sont stockés dans des mémoires appelés registres). L'UC dirige le fonctionnement de toutes les autres unités (UAL, Mémoire, Entrées/Sorties) en leur fournissant des signaux de cadence et de commande.

Un bus est un simple câble de n lignes qui permet de transmettre des données (dans les deux sens) entre le processeur et la mémoire. Un bus permet également de véhiculer des signaux entre l'Unité Centrale et les périphériques. Il existe trois types de bus : le bus de données qui sert à transporter les arguments d'une opération, et qui est constitué, pour les processeurs les plus récents, de 32 à 64 lignes parallèles ; le bus d'adresses qui permet d'identifier la case mémoire concernée par l'opération en cours (qui est également constitué de 32 à 64 lignes parallèles) et enfin le bus de commandes qui détermine l'opération à effectuer.

Les premiers ordinateurs étaient composés d'une unité centrale et de périphériques gérés directement par l'unité centrale. Des instructions spéciales permettaient de copier des données d'un périphérique vers un autre ou de/vers la mémoire centrale. L'unité centrale alternait donc des phases de calculs (additions, soustractions, etc. sur des registres) et des phases dites d'entrées/sorties (quand des données devaient être copiées ou lues ; par exemple, un

programme pour être exécuté doit d'abord être copié d'un disque vers la mémoire centrale). Or les accès aux périphériques sont très lents (comparés aux temps d'accès en mémoire centrale) et il faut cependant souvent lire ou écrire sur un disque ou une imprimante. Le processeur se trouve alors pendant ce temps sous utilisé.

Une première idée pour améliorer cette situation consiste à traiter à part les tâches de lecture et d'écriture sur les périphériques (disque dur, disquette, imprimante) en les confiant à un programme spécialisé. On associe alors à chaque périphérique de même type (i.e. utilisant la même méthode de gestion des données) un contrôleur. Le contrôleur sert d'intermédiaire entre le bus (relié à la mémoire centrale) et un ensemble de périphériques de même type dont il assure le contrôle.

Plus concrètement, imaginons que l'utilisateur demande à copier en mémoire centrale un fichier présent sur le disque D. Cette demande va correspondre à une instruction, exécutée par l'unité centrale, qui se réduira ici à envoyer une commande sur le bus relié au contrôleur associé au disque D. Cette commande spécifiera le nom du fichier sur le disque et l'emplacement où il doit être copié en mémoire centrale. Après cet envoi, l'unité centrale pourra continuer à exécuter d'autres instructions. Pendant ce temps, le contrôleur recevra l'instruction et l'exécutera, ce qui pourra nécessiter pour lui plusieurs pas de calculs, pendant lesquels l'unité centrale pourra continuer d'avancer. Quand le contrôleur a terminé, il peut en informer l'unité centrale au moyen d'un signal appeler *interruption*. Il s'agit d'un code envoyé sur une partie spéciale du bus et recopié dans un registre spéciale de l'unité centrale : le registre des interruptions. L'unité centrale est ainsi informée que la copie demandée a bien été effectuée et peut exécuter des instructions utilisant ces données en mémoire centrale.

La situation que nous venons de brosser peut encore être améliorée en évitant que la mémoire centrale soit la seule mémoire utilisée pour exécuter les instructions. On dispose pour cela de différents moyens :

- utiliser des mémoires *caches*. Une mémoire cache sert d'intermédaire entre la mémoire centrale et l'unité de calcul. D'acccès plus rapide que la mémoire centrale, elle est utilisée pour faire des copies de données anticipées entre les adresses en mémoire centrale et les registres.

- utiliser des mémoires *tampons*. Les tampons sont des mémoires auxiliaires dans les contrôleur de périphériques. Au lieu d'écrire par exemple, à chaque demande d'écriture d'un caractère, sur le périphérique concerné, on écrira dans un tampon du contrôleur. ce n'est qu'une fois le tampon rempli, qu'on transmettra la demande d'écriture sur le périphérique. La mémoire centrale est ainsi moins souvent sollicitée par les périphériques.

3.3 Le fonctionnement simplifié d'un ordinateur

L'exécution d'un programme se déroule selon le modèle suivant :

- Le programme et les données sont chargés en mémoire centrale (d'où le nom de machine à programme enregistré ou machine universelle),

- Les instructions du programme sont amenées séquentiellement à l'unité de commande (UC) qui les analyse et déclenche le traitement approprié en envoyant éventuellement des signaux à l'unité arithmétique et logique (UAL).

L'UC s'occupe donc de gérer l'exécution des instructions d'un programme. Elle comprend une mémoire d'accès très rapide dans laquelle sont stockés les résultats temporaires et les

informations de commande. Dans cette mémoire on trouve, entre autres, deux registres importants :

- le registre d'instructions (RI) qui contient l'instruction en cours d'exécution,
- le compteur ordinal (CO) qui contient l'adresse de la prochaine instruction à exécuter (et qu'il faut aller chercher en mémoire centrale).

Les registres de l'UC ne sont pas accessibles aux programmeurs. L'UC contient aussi un dispositif de décodage des instructions (décodeur) et un séquenceur de commandes qui active les circuits nécessaires à l'exécution de l'instruction en cours. Cette unité a besoin des signaux d'une horloge pour enchaîner les commandes. L'horloge est externe à l'unité.

L'UAL contient tous les circuits électroniques qui réalisent effectivement les opérations désirées. Ces opérations sont principalement l'addition, la soustraction, la multiplication, la division, la négation (inversion des bits) , les opérations logiques (ET, OU, et OU exclusif). Les opérandes nécessaires pour ces opérations se trouvent dans des registres contenus dans cette unité (l'un d'eux s'appelle l'accumulateur). Ces registres sont accessibles aux programmeurs.

La mémoire centrale contient deux types d'informations :

- les instructions de différents programmes,
- les données nécessaires à l'exécution des programmes.

Données et instructions sont stockées comme des suites de bit (0 ou 1) dans des mots-mémoires (généralement de 32 et 64 bits). Chaque mot-mémoire est repéré par une adresse unique. Les opérations possibles dans la mémoire centrale sont la lecture et l'écriture de mots-mémoire.

De façon simplifiée, voici un cycle d'exécution de l'UC :

1) Le registre CO (compteur ordinal) contient l'adresse du mot-mémoire représentant la prochaine instruction,
2) Le contenu de ce mot est transféré dans le registre RI (registre d'instruction),
3) Le contenu de RI est décodé afin de déterminer l'opération à exécuter,
4) L'opération est exécutée (le contenu d'un ou plusieurs registres est modifié),
5) CO est incrémenté (pour passer à l'instruction suivante),
6) Fin du cycle d'exécution et démarrage d'un nouveau cycle.

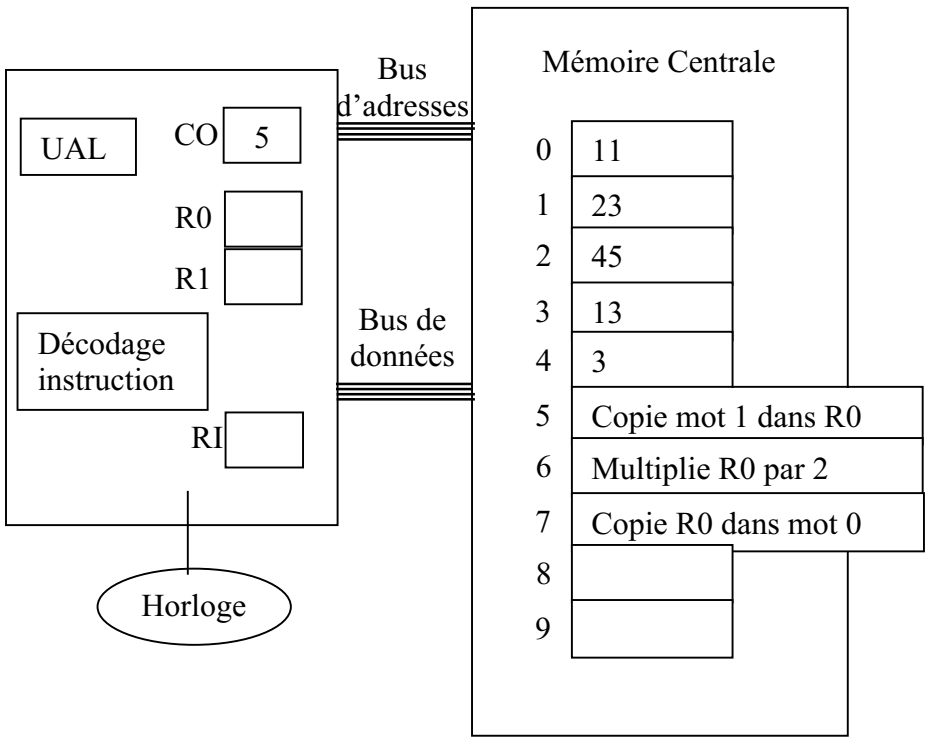


Schéma d'une unité centrale + mémoire centrale contenant un programme.

Nous détaillons ci-après, les différents cycles d'exécutions correspondant au programme contenu en mémoire dans l'exemple illustré ci-dessus :

1er cycle d'exécution :

1. La valeur 5 contenue dans le registre CO est envoyée sur le bus d'adresses, accompagnée d'un ordre de lecture,
2. Le contenu du mot mémoire d'adresse 5 (l'instruction "Copie mot 1 dans R0") est transféré dans le registre RI,
3. Le contenu de RI est décodé (par l'unité "Décodage instruction") afin de déterminer l'opération à exécuter,
4. L'opération est exécutée :
   a) la valeur 1 (adresse du mot mémoire) est envoyée sur le bus d'adresses, accompagnée d'un ordre de lecture,
   b) Le contenu du mot mémoire d'adresse 1 (valeur 23) est transféré dans le registre R0,
5. CO est incrémenté (la valeur de CO passe à 6)

2ème cycle d'exécution :

1. La valeur 6 contenue dans le registre C0 est envoyée sur le bus d'adresses, accompagnée d'un ordre de lecture,

2. Le contenu du mot mémoire d'adresse 6 (l'instruction "multiplie R0 par 2") est transféré dans le registre RI,

3. Le contenu de RI est décodé (par l'unité "Décodage instruction") afin de déterminer l'opération à exécuter,

4. L'opération est exécutée :

   l'UAL (Unité Arithmétique et Logique) effectue la multiplication de R0 par 2 et place le résultat (46) dans R0,

5. CO est incrémenté (la valeur de CO passe à 7)

3ème cycle d'exécution :

1. La valeur 7 contenue dans le registre CO est envoyée sur le bus d'adresses, accompagnée d'un ordre de lecture,

2. Le contenu du mot mémoire d'adresse 7 (l'instruction "copie R0 dans case n°0") est transféré dans le registre RI,

3. Le contenu de RI est décodé (par l'unité "Décodage instruction") afin de déterminer l'opération à exécuter,

4. L'opération est exécutée :

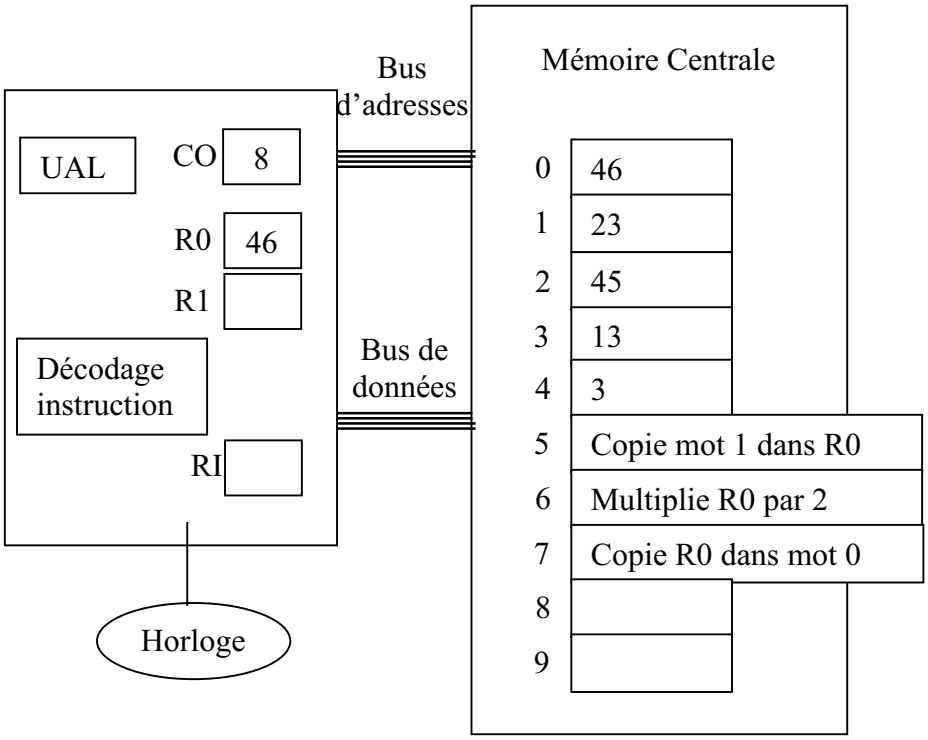
   a) La valeur 0 est envoyée sur le bus d'adresse, accompagnée d'un ordre d'écriture,

   b) La valeur 46 contenue dans le registre R0 est envoyée sur le bus de données et placée dans le mot mémoire d'adresse 0,

5. CO est incrémenté (la valeur de CO passe à 8)

6. Fin du 3ème cycle d'exécution

Une fois ces trois cycles exécutés, le contenu des différents registres et de la mémoire centrale est donnée sur la figure suivante:



Etat de l'Unité centrale et de la mémoire centrale après exécution des 3 instructions contenues en mémoire centrale.

3.4 Représentation des instructions en machine

Comme les entiers et les caractères, les instructions doivent être représentées sous forme de chaînes de bits.

3.4.1 Les instructions

Une instruction est composée de deux parties:

- le code opération : il détermine l'action que doit effectuer le processeur. Par exemple, une addition avec le contenu de l'accumulateur dans un microprocesseur Intel 8086 est codée par 10000001,
- le champ opérande : il indique l'argument impliqué dans l'opération. Il peut s'agir d'une donnée ou d'une adresse contenant des données.

La taille d'une instruction (i.e. le nombre de bits qu'elle occupe en mémoire) dépend de l'opération et de l'opérande. Elle est généralement de quelques octets.

L'ensemble des instructions disponibles est appelé un jeu d'instructions. On distingue plusieurs catégories d'instructions, selon le type d'action qu'elles déclenchent :

Les instructions d'affectation :

Ce sont les instructions qui permettent des transfert de données entre les registres et la mémoire. Il s'agit d'écriture (des registres vers la mémoire) ou de lecture (de la mémoire vers un registre). On les appelle affectations, car elles permettent d'affecter (i.e. d'attribuer) une certaine valeur à une case mémoire ou à un registre.

Les instructions arithmétiques et logiques :

Ce type d'instruction porte sur un registre donné. Elles permettent d'effectuer une opération entre le contenu du registre et une donnée, et place le résultat dans le registre concerné. Il y a en particulier une addition de donnée, une soustraction de donnée, l'incrémentation (addition de 1), la décrémentation (soustraction de 1) et des décalages de bits, vers la droite ou vers la gauche.

Les instructions de comparaison :

Elles permettent de comparer le contenu d'un registre à une donnée.

Les intructions de branchement (ou sauts) :

Ces instructions permettent d'enchaîner un nouveau cycle sur une instruction non consécutive en mémoire à l'instruction en cours. C'est en effet le registre CO qui repère l'instruction suivante à exécuter. Les instructions de branchement modifie ce registre et permettent donc de choisir la prochaine instruction à effectuer.

On distingue deux sortes de branchement :

- le branchements inconditionnel. Cette instruction modifie directement le registre CO.
- les branchements conditionnels : le saut à une instruction différente de l'instruction suivante n'est effectué que si une condition particulière est satisfaite. Par exemple, on peut tester si la valeur d'un registre donné est égale à zéro.

Sans ce type d'instructions, un programme devrait toujours effectuer les mêmes séquences d'instructions. Or il est fondamental de pouvoir écrire les programmes en fonction de certains paramètres, connus seulement à l'exécution. Cela permet de modéliser le programme en fonction des données et de traiter correctement les cas d'exceptions.

3.4.2 Le langage assembleur

Le langage assembleur est très proche du langage utilisé par la machine (ou langage machine) et dépend du processeur. Les instructions données en langage machine sont des suites de 0 et de 1 a priori illisibles pour un humain normalement constitué. Un premier pas serait de noter les suite de bits avec la notation hexadécimale (pour séparer plus facilement les octets), mais un tel programme reste cependant illisible.

Voici à quoi peuit ressembler un programme écrit (sous forme hexadécimale) en langage machine :

A1 01 10 03 01 12 A3 01 14 ...

C'est pourquoi les langages assembleurs ont été définis. Ils permettent essentiellement de noter les instructions par des noms explicites suivis de paramètres. Les différents registres sont également codés de manière symboliques. Dans l'exemple précédent, la séquence « A1 01 10 » a en réalité la signification « copier le contenu de la mémoire d'adresse 0110 dans le registre AX du processeur ». En langage assembleur, cette instruction sera notée :

MOV AX, [0110]

Cette notation est facile à mémoriser : MOV est l'abréviation de move (déplacer), AX désigne le registre de même nom, et la notation entre crochet indique qu'on fait référence à une adresse mémoire.

Toutes les instructions du jeu d'instructions du processeur ont ainsi une notation symbolique associée, fournie par le fabriquant du processeur. Ecrire en langage assembleur consiste donc à écrire des instructions machine dans une notation symbolique, plus facile à appréhender pour un être humain. Ces instructions seront d'abord stockées dans un fichier texte (constitué de caractères alpha-numériques) qu'on appellera le fichier source, et pourront être traduites automatiquement, grâce à un programme, l'*assembleur*, en langage machine (ou binaire).

Voici une interprétation du début de programme précédent en langage assembleur :

Instruction en langage machine	**Instruction en langage assembleur**	**Commentaires sur l'instruction**
A1 01 10	MOV Acc, [0110]	Copier le contenu de 0110 dans le registre Accumulateur
03 01 12	ADD Acc, [0112]	Ajouter le contenu de 0112 à l'accumulateur
A3 01 14	MOV [0110], Acc	Mettre le contenu de l'accumulateur à l'adresse 0114

3.4.3 Un langage assembleur très simplifié : le langage MAMIAS

Nous allons illustrer le codage des instructions par un exemple de langage très simple, le langage MAMIAS.

MAMIAS se caractérise par :

- les instructions et les données sont codées dans des **mots mémoire de 8 bits**,
- chaque mot mémoire est repéré par son **adresse**, à savoir un entier non signé n ($n \geq 0$) codé sur 5 bits ; le mot mémoire d'adresse n est noté par son adresse entre parenthèses, i.e. (n),
- il existe un registre (mot mémoire particulier) appelé l'**accumulateur** et désigné par *Acc*,
- une **instruction** est codée sur un mot mémoire (8 bits).; il s'agit d'un couple (code, argument), où le code (constitué des 3 premiers bits) désigne une opération, et l'argument (les 5 derniers bits) désigne un entier (qui représente, selon l'opération, soit une donnée, soit une adresse),
- une **donnée** est un entier signé codé sur un mot mémoire (8 bits).

Le langage MAMIAS permet de coder les instructions données à la machine, et nous avons comme nous l'avions déjà vu, exécution en boucle d'un cycle du type:

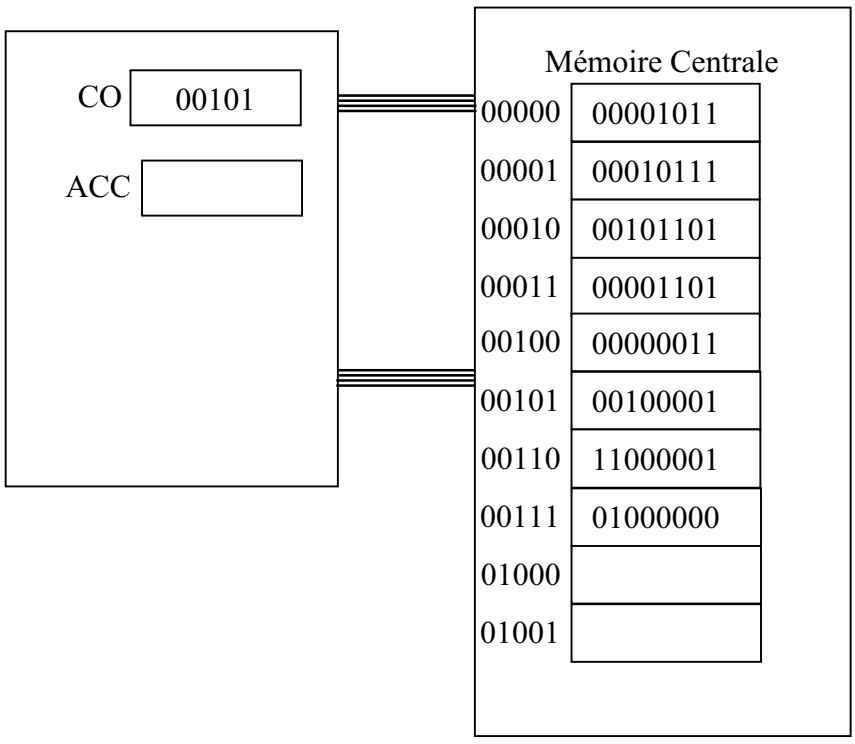
1. Le compteur ordinal CO contient l'adresse d'un mot-mémoire représentant la prochaine instruction MAMIAS
2. Le contenu de ce mot est transféré dans le registre RI (registre d'instruction),
3. Le contenu de RI (une instruction Mamias) est décodé afin de déterminer l'opération à exécuter
4. L'opération est exécutée (le contenu d'un ou plusieurs registres est modifié),
5. CO est incrémenté (pour passer à l'instruction suivante),

6. Fin du cycle d'exécution et démarrage d'un nouveau cycle.

Les opérations élémentaires sont :

Code	nom	Signification
000	INIT *x*	*Acc* ← *x* L'entier signé *x* codé sur 5 bits, est traduit en un entier signé codé sur 8 bits et rangé dans l'accumulateur
	CHARGE *n*	*Acc* ← (*n*) Le contenu du mot mémoire d'adresse n est recopié dans l'accumulateur
010	RANGE *n*	*(n)* ← *Acc* Le contenu de l'accumulateur est recopié à l'adresse n
011	ET *n*	*Acc* ← *Acc* **et** (*n*) Le contenu de l'accumulateur reçoit le et bit-à-bit entre son propre contenu et celui de la mémoire d'adresse n. L'opérateur de conjonction bit à bit est défini sur chaque bit par (*i*) 1 et 1 = 1, (*ii*) 0 et 1 = 0, (*iii*) 1 et 0 = 0, et (*iv*) 0 et 0 = 0
100	SAUTE *n*	si *Acc* = 0 alors aller à l'adresse *n.* Si le contenu de l'accumulateur est nul, le cycle d'interprétation du programme enchaîne sur l'instruction stockée à l'adresse n (i.e. l'entier n est mis dans le compteur ordinal et on entame un nouveau cycle)
101	ADD *n*	*(n)* ← *Acc* + (*n*) Ajoute au contenu de la mémoire d'adresse n celui de l'accumulateur et range le résultat à l'adresse n
110	DEC *x*	Décale le contenu de l'accumulateur de *x* positions vers la gauche si *x* > 0, et de *x* positions vers la droite si *x* < 0
111	STOP	Arrête l'exécution. Les 5 derniers bits n'ont aucune signification.

Voici l'exemple du paragraphe 3.3 codé dans le langage MAMIAS :



3.5 Le système d'exploitation

3.5.1 Introduction

Il n'y a pas de définition précise correspondant exactement à ce que recouvre l'expression ``système d'exploitation'' (SE en français, OS pour *Operating System* en anglais). On peut dire approximativement que *c'est la couche logicielle qui se situe entre le matériel et l'utilisateur de ce matériel*, c'est-à-dire un ensemble de programmes permettant à l'utilisateur d'exécuter ses programmes et contrôlant le matériel associé à l'ordinateur. Cet ensemble de programmes permet aussi la gestions des programmes et des données sous forme de fichiers.

Nous avons vu, dans la section précédente, comment l'unité centrale exécutait un programme qui lui était fourni en mémoire centrale : la suite d'instructions en langage machine (en code binaire) est exécutée instruction après instruction, par chargement de l'instruction courante dans le Registre d'Instruction, puis exécution de celle-ci et incrémentation du compteur ordinal. Reste à savoir comment amener le programme que l'utilisateur cherche à exécuter en mémoire centrale, à l'exécuter, puis, à la fin de l'exécution de ce programme, à demander à l'utilisateur le prochain programme à exécuter (en fait, on verra que l'on peut être plus efficace). Un des éléments principaux d'un système d'exploitation est justement un programme dont le rôle est d'amener en mémoire centrale et d'exécuter les programmes à la demande de l'utilisateur. Ces programmes, dont l'exécution est contrôlée par le système d'exploitation, sont appelés des *processus*.

Les systèmes d'exploitation couramment rencontrés sont les suivants :

- DOS et ses variantes MS-DOS (Microsoft) et PC-DOS (IBM). DOS est un acronyme pour *Disk Operating System*. Il s'agit d'un système d'exploitation dont l'interface alpha-numérique est un interprète de commandes: une commande est un programme que le système sait exécuter (*MKDIR rep* par exemple). Datant de 1978, DOS a été le premier système d'exploitation des PC.
- Unix et ses variantes, Linux, etc. C'est le premier vrai système multi-utilisateurs. Il a été conçu dans les années 70, en même temps que le langage C. Son noyau, écrit en C, a relativement peu évolué, et tous les systèmes actuels s'en inspirent. Tous les gros systèmes utilisent Unix.
- MacOS : c'est le système d'exploitation des ordinateurs MacIntosh. Premier vrai système graphique utilisant souris, icônes et fenêtres (notions proposées au départ par Rank Xerox), il a, avec la version X, été complètement réécrit en s'inspirant d'Unix et permet aujourd'hui aux utilisateurs de bénéficier à la fois de logiciels MacIntosh et d'un environnement de programmes Unix.
- Windows et ses variantes (Windows95, Windows98, Windows NT) : Il s'agit d'un système à interface graphique, qui a remplacé peu à peu le DOS sur les PC. Le noyau a beaucoup évolué entre les premières versions et la dernière, entièrement indépendante du DOS.

3.5.2 Caractéristiques des systèmes d'exploitation modernes

On peut caractériser les systèmes d'exploitation par la manière dont ils conçoivent la gestion des tâches et la gestion des utilisateurs. Les systèmes d'exploitation modernes sont des systèmes à la fois multi-programmes et multi-utilisateurs : ceci permet de prendre la mesure de la complexité de ces systèmes.

a. La multi-programmation

On dit d'une machine qu'elle est multi-programmée si plusieurs programmes peuvent être chargés en mémoire et partager l'unité centrale. L'objectif est d'éviter à celle-ci des temps morts, c'est-à-dire de rester inactive quand le programme en cours d'exécution est par exemple en attente de données. L'utilisation du processeur est alors optimisée.

Dans une machine mono-programmée, il y a alternance entre le travail du système d'exploitation et les phases d'exécution complète des programmes utilisateur. Ce type de fonctionnement est pleinement justifié dans deux cas de figure :

- Il n'y a qu'un seul utilisateur (premiers systèmes DOS). Ce cas ne se présente plus dans les nouvelles versions des systèmes d'exploitation (voir ci-dessous),
- Seul le temps d'exécution compte et pas le temps d'attente (cas de gros programmes). Cette situation se présente dans le mode de calcul « temps réel » de certains systèmes d'exploitation (les programmes sont alors exécutés sans interruption).

La multiprogrammation ne va pas sans poser de difficultés :

- Il faut partager l'espace mémoire entre un nombre important de programmes chargés en mémoire centrale.
- Il faut sauvegarder et restaurer le contexte de chaque programme (son état d'exécution) lorsque le processeur passe d'un programme à l'autre.
- Il faut contrôler l'intégrité des données manipulées par chaque programme.
- Il faut assurer le trafic ordonné et sans mélange des données entre la mémoire centrale et les différents périphériques.
- Il faut protéger les données et les programmes d'éventuelles erreurs.

Ces différentes tâches sont assurées par le système d'exploitation.

b. La multi-utilisation

Les systèmes à temps partagé ou multi-utilisateurs partagent les ressources de traitement entre différents utilisateurs situés sur différents terminaux et interagissant avec le système. En pratique le temps CPU est divisé en petites portions, attribuées à tour de rôle aux différents utilisateurs, qui ont de la sorte l'illusion d'avoir la machine pour eux seuls. Cette technique présente l'avantage d'utiliser le plus efficacement possible une machine, avec l'inconvénient d'alourdir le travail du système d'exploitation.

A l'inverse, un système est dit mono-utilisateur si, à un instant donné, un seul utilisateur peut se connecter sur l'ordinateur (ou, plus simplement, utiliser la machine).

Le système d'exploitation MS-DOS est ainsi un système mono-utilisateur, *a contrario* le système Unix est un système multi-utilisateur. Avec Unix, plusieurs personnes peuvent être connectées sur le même ordinateur en même temps, et demander au système d'exploitation d'exécuter différentes commandes. Bien entendu, il est impératif dans ce cas que le système d'exploitation soit capable de distinguer un programme demandé par un utilisateur d'un programme demandé par un autre.

Dans la pratique les systèmes actuels combinent des traitements *batch* (non interactifs) et des traitements en temps partagés que les utilisateurs peuvent interrompre ou suspendre à tout moment.

Dans les systèmes à temps partagés, le temps de réponse est un élément clef. Il faut trouver un équilibre entre la puissance de l'unité centrale et le nombre d'utilisateurs travaillant en parallèle.

3.5.3 Fonction du système d'exploitation

Du point de vue de l'utilisateur, le système d'exploitation remplit deux grandes fonctions : la gestion du parallélisme et la présentation d'une machine virtuelle conviviale.

- La gestion du parallélisme : du fait de la multiprogrammation et du temps partagé, on a l'impression que plusieurs programmes s'exécutent en même temps et on a l'illusion d'un fonctionnement parallèle du processeur. En fait, il ne s'agit que d'une apparence de parallélisme, le processeur ne traitant qu'une opération élémentaire à la fois. Donner cette apparence de parallélisme, c'est-à-dire gérer intégralement le partage des ressources et du processeurs entre ces différents programme est la première fonction des systèmes d'exploitation modernes.
- Une machine virtuelle conviviale : l'utilisateur n'a pas besoin de savoir comment marche le matériel. Le système d'exploitation sert d'interface avec le matériel et constitue de ce fait une machine virtuelle pour l'utilisateur. Celle-ci doit être la plus conviviale possible. Outre les systèmes à interface graphique, le système propose à l'utilisateur un *langage de commandes* qui doit être à la fois puissant et simple. Il permet à l'utilisateur d'interagir avec le système et cache les détails de la gestion du matériel.

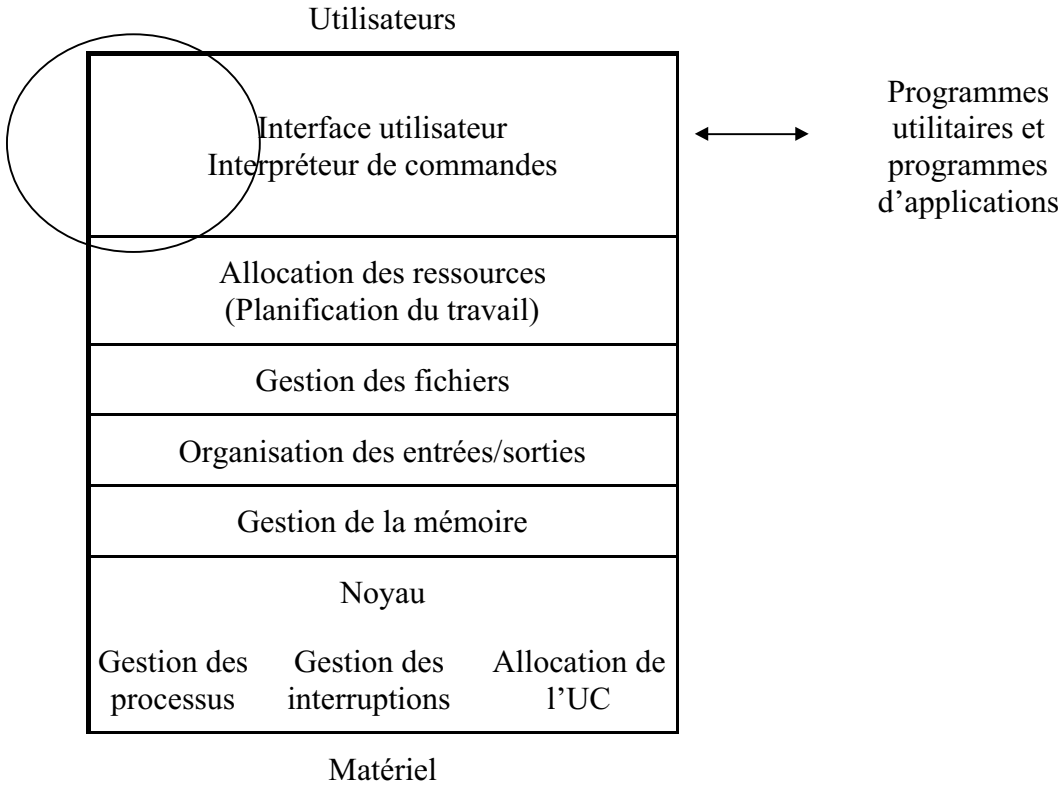
D'un point de vue plus technique, pour remplir les deux grandes fonctions précédentes, un système d'exploitation (c'est-à-dire les différents programmes qui composent un système d'exploitation) assument les tâches suivantes :

- Il comporte un *langage de contrôle ou de commandes* qui permet à l'utilisateur de modifier les données, faire exécuter des programmes, modifier l'environnement de travail, etc.
- Il spécifie comment s'effectue la *gestion des processus* (les programmes exécutés par l'ordinateur).
- Il assure la *gestion de la mémoire centrale*.
- Il spécifie une méthode de *gestion des fichiers*, et plus généralement des disques durs, du clavier, de l'écran ou de tout type de périphériques. Nous n'aborderons plus loin que la gestion des fichiers. La gestion des différents périphériques nécessite en effet une connaissance plus approfondie des composants physiques utilisés.
- Il permet de définir une *gestion des utilisateurs* - ce qui suppose de définir qui peut se connecter sur la machine, avec quel mot de passe et avec quels droits. Cela impose aussi la mise en place de mécanismes de sécurité. C'est l'administration des utilisateurs.
- Enfin, il définit et contrôle les mécanismes de *communications externes* et en particulier la gestion des connexions avec un réseau extérieur, et donc d'autres ordinateurs, la gestion des imprimantes, etc. C'est l'administration réseau.

Il existe différents moyens logiciels ou matériels pour mettre en place des procédures plus ou moins efficaces pour chacun des points précédents. Malheureusement, plus les outils correspondants sont sophistiqués, plus ces outils sont complexes à mettre en œuvre (la place occupée par les outils du système d'exploitation en mémoire centrale devient importante et

nécessite alors des ordinateurs puissants). Les systèmes développés réalisent donc toujours un compromis entre sophistication et taille.

La structure d'un système d'exploitation peut être organisée en une série de couches fonctionnelles séparant le matériel des utilisateurs, chaque couche de niveau supérieur faisant appel aux couches de niveaux inférieurs. Elle peut être représentée par le schéma suivant :



Structure d'un système d'exploitation

3.5.4 Fonctionnement simplifié du système d'exploitation

Le fonctionnement du cœur du système d'exploitation peut être décrit schématiquement comme suit :

- attente d'une commande faite par l'utilisateur demandant à ce qu'un programme soit exécuté : le lancement de l'exécution du programme se fera par exemple par un double clic de souris sur un icône d'une interface graphique (système Windows, environnement CDE sous système Unix), ou par la frappe au clavier d'un nom de commande sous une interface alpha-numérique (console Unix ou DOS par exemple),
- préparation d'une zone en mémoire centrale pouvant contenir le code du programme,
- chargement (i.e. copie) en mémoire centrale du code correspondant à ce programme. Le système place au mieux un programme en mémoire centrale (entre les adresses 35000 et 36000 par exemple). Mais l'adresse de début n'est pas connue au moment de la traduction du programme en code machine. Elle est en réalité susceptible de changer à chaque nouvelle exécution du programme et le code machine est écrit comme si cette adresse de début était l'adresse 0. Toutes les adresses du programmes

sont donc relatives et ce n'est que lors de l'exécution du programme qu'un calcul d'adresse réelle est effectué, avant l'exécution d'une instruction proprement dite.

- saut à l'adresse de la première instruction du programme,
- exécution du programme demandé. A la fin (instruction stop), on exécute un saut (i.e. retour) vers la prochaine instruction du programme du système d'exploitation,
- libération de la place occupée par le programme de l'utilisateur en mémoire centrale,
- retour à la première étape.

Un *processus* est le résultat du chargement en mémoire d'un programme, c'est un programme en cours d'exécution. Plusieurs processus peuvent être des exemplaires du même programme chargés à des adresses différentes de la mémoire centrale et donnant lieu à des exécutions différentes.

Les systèmes dits multi-programmés, même avec une seule unité centrale, autorisent plusieurs processus à effectuer des calculs *simultanément* (c'est-à-dire au même moment). Avec un seul processeur, il n'est évidemment pas question d'exécuter plusieurs instructions simultanément. Mais, dès lors qu'un processus est une suite d'instructions, on peut imaginer « entrelacer » les différentes suites d'instructions à exécuter. C'est justement ce que les systèmes actuels permettent : le système exécute pendant un certain temps un processus, puis *passe* d'un processus à un autre (il sauvegarde les informations qui permettront la reprise du processus arrêté, et charge les informations nécessaires à l'exécution du nouveau processus), etc., jusqu'à ce qu'il n'y ait plus de processus à exécuter. Cette opération s'appelle la *commutation de contextes*.

L'avantage principal d'une telle méthode est double : s'il y a plusieurs utilisateurs, le système fait croire à chacun d'eux que la machine est en train de travailler « pour lui », et quand un processus est interrompu parce qu'en attente de données, le système peut continuer néanmoins à travailler en exécutant un autre processus. On peut dès lors imaginer avoir l'exécution « en même temps » de plusieurs processus. Ce que l'on suggère pour l'exécution d'un ensemble de processus s'applique évidemment aussi à un ensemble de tâches (petits modules séparés d'un processus). Sur un micro-ordinateur en fonctionnement sous Windows NT ou sous Unix, il y a ainsi au moins une dizaine de tâches ou processus fonctionnant en permanence. La mise en œuvre d'une telle technique alourdit bien entendu considérablement la tâche du système d'exploitation, qui doit caractériser les processus selon leurs états, et intégrer de nouveaux modules. Mais cela permet d'optimiser l'utilisation de l'unité centrale.

Les processus alternent en permanence entre l'état « en exécution » et des états d'attente. Un seul processus est dans l'état « en exécution », tous les autres étant en attente d'exécution. Le fait d'interrompre l'exécution d'un processus pour en exécuter un autre s'appelle la *préemption*. Un processus alterne donc continuellement entre différents états, jusqu'à ce que son exécution soit terminée. Cette gestion de l'exécution fragmentée des processus est gérée elle-même par deux programmes : l'ordonnanceur (qui détermine dans quel ordre les processus seront exécutés) et le *scheduler* (qui détermine le temps de calcul alloué à chaque exécution). Le noyau du système d'exploitation exécute lui-même ces programmes, qui donne l'unité centrale à un programme externe, puis reprennent la main, etc.

Un processus doit être placé en mémoire centrale pour pouvoir être exécuté. La zone occupée par un processus peut être grande et doit être conservée pendant un temps plus ou moins long. Or la taille de la mémoire centrale est limitée. Il est dès lors indispensable de gérer au mieux l'allocation de la mémoire nécessaire. Comme toutes les zones n'ont pas à être nécessairement en mémoire centrale au même moment, le système d'exploitation peut décider d'en mettre

une partie sur un disque dur dans une zone spéciale (appelée *zone de swap*). Quand le système doit accéder à cette zone, il la charge alors en mémoire centrale. Ce système revient donc à avoir une mémoire centrale virtuellement plus grande qu'elle n'est en réalité. Bien entendu, en procédant ainsi, on perd un peu de temps pour recopier la zone de swap (sur disque) en mémoire centrale, mais globalement le gain est positif.

4 Utilisation de l'ordinateur : logiciels

Nous avons vu, dans les chapitres précédents, comment l'ordinateur traitait physiquement (ou presque) les instructions, telles qu'elles étaient données sous forme de suite de codes binaires en mémoire centrale. Sur les premiers ordinateurs, chaque programme binaire était implanté manuellement en mémoire centrale avant de pouvoir être exécuté. Le système d'exploitation permet aujourd'hui, d'effectuer cette tâche automatiquement : le programme à exécuter est copié à partir d'un disque (disque dur, disquette, CD-ROM) vers la mémoire centrale où il est exécuté.

Cette description permet de comprendre comment l'exécution des programmes est gérée par le système d'exploitation, mais n'éclaire pas sur la façon dont l'utilisateur pourra de son côté, exploiter sa machine. S'il n'est pas informaticien, l'utilisateur se contentera de lancer des programmes lui permettant d'éditer des textes ou de réaliser diverses tâches qui l'intéresse, car les programmes, et plus généralement les logiciels, répondent à des problèmes particuliers.

Pour l'informaticien, une question importante sera le développement d'un programme nouveau, ou d'un logiciel.

Trois notions sont donc essentielles pour l'exploitation de l'ordinateur :

- l'exécution des instructions par l'unité centrale,
- le système d'exploitation,
- les logiciels, objet de ce chapitre.

4.1 Notion de logiciels

L'informatique (*computer science*) est la *science du traitement* rationnel, notamment par machine automatique, *de l'information,* considérée comme le support des connaissances humaines et des communications dans les domaines technique, économique et social.

L'informatique a pour objet le traitement et la circulation de l'information. Traiter l'information consiste à passer d'informations, appelées *données*, à d'autres informations, appelées *résultats* (statistiques, gestion, traduction automatique, compilation, etc.).

Un *logiciel* est un ensemble de programmes permettant d'effectuer les traitements appropriés à une situation particulière, ou permettant de résoudre un problème donné.

La démarche qu'il faut adopter pour concevoir un logiciel peut se décomposer en plusieurs étapes :

- La première consiste à caractériser les différents traitements souhaités (les fonctionnalités du logiciel), ou le problème que l'on cherche à résoudre. Il faut définir tous les éléments : les données d'entrées, les résultats (les sorties), les différents élements qu'il faut représenter pour modéliser la situation ou résoudre le problème, leurs structures (*les types de données*), les opérations à effectuer ou les fonctionnalités à réaliser. On ne se préoccupe pas nécessairement de l'ordre dans lequel les opérations seront exécutées, ni de la manière dont les fonctionnalités souhaitées seront déclenchées. C'est une première étape d' *analyse*.
- La deuxième étape consiste, dans le cas d'une résolution de problème, à trouver une manière systématique de procéder pour aboutir à une solution. Cela conduit généralement à un *algorithme*. Si l'on souhaite simplement présenter à l'utilisateur un certain nombre

de fonctionnalités, il faudra alors réfléchir à un algorithme permettant de présenter à l'utilisateur une interface donnant accès aux différentes fonctionnalités du logiciel.

– La troisième étape, appelée implémentation, vise à écrire l'algorithme souhaité dans un *langage de programmation* particulier. Un langage de programmation suit des règles de syntaxe très strictes et ne présente pas les ambiguïtés des langues dites naturelles. On obtient alors un ou plusieurs modules de *programmes* (le *code source* du logiciel). Ces programmes sont écrits sous forme de textes (à l'aide d'un éditeur de texte ou de programmes) et stockés dans un ou plusieurs fichiers.

Pour réaliser l'impléméntation, il faudra choisir un langage de programmation, écrire le programme, puis tester le code obtenu. Le code s'ontiendra à partir d'un ou des programmes sources, en utilisant un compilateur. La compilation consiste à transformer le programme source en une suite d'instructions exécutables par le processeur. On passe d'un code source écrit dans plusieurs fichiers à un code binaire exécutable (on dit encore programme exécutable ou simplement exécutable) placé dans un fichier unique. Cette étape est effectuée par un *compilateur*. On peut également utiliser directement un *interpréteur*. Dans ce cas, l'étape de compilation n'existe pas car on donne directement le texte du programme source à interpréter.

Il existe différents types de langages de programmation, et plusieurs langages dans chaque catégorie. Chaque langage possède une syntaxe et des règles qui lui sont propres.

- *Langage machine* : les instructions, les données sont codées en binaire. Ces codes binaires sont directement exécutables par la machine. La signification des codes dépend du processeur. Un programme en langage machine pour un processeur P est *directement* exécutable par ce processeur. Mais un programme en langage machine pour un processeur P *ne pourra pas*, en général, être exécuté sur un autre processeur.
- *Langage d'assemblage* : les instructions sont spécifiées par un code mnémonique (SAUTE, STOP, DEC, etc.). Un langage d'assemblage n'est qu'une autre façon, plus facilement compréhensible, d'exprimer le code binaire. En général, à une instruction du langage d'assemblage correspond une instruction en langage machine, et la conversion est unique. Un programme, l'assembleur, permet de traduire un programme écrit en langage d'assemblage en langage binaire. Avec MAMIAS, on a spécifié à la fois le langage machine, et le langage d'assemblage.
- *Langages évolués* : les langages d'assemblage et les langages machine ont de gros inconvénients. Ils sont difficiles à suivre, peu compréhensibles, peu puissants (les opérations exécutables par la machine sont rudimentaires, des opérations plus complexes sont fréquentes et nécessitent d'écrire souvent les mêmes successions d'instructions), dépendants du processeur (donc l'exécution sur un autre processeur nécessite de presque tout réécrire). Un langage évolué permet de ne pas faire référence aux adresses en mémoire centrale, et autorise des instructions complexes (répétition d'opérations, découpage du code en morceaux, utilisation structures de données complexes comme des vecteurs ou tableaux, d'opérations complexes, etc.). Si l'on possède un texte de programme dans un langage évolué, on peut ensuite le compiler et l'exécuter sur différentes machines.

On peut classer les langages évolués en différents *paradigmes de programmation*.

– Langages impératifs : Pascal, C, Basic, FORTRAN, Cobol, Ada, etc. Ce sont les premiers langages de programmation. On les a qualifié *d'impératif* parce qu'ils formulent les algorithme en terme d'instructions devant être exécutées les unes après les autres, comme on exécute des ordres. Une instruction est ici un ordre donné à la machine.

L'accent est mis sur les algorithmes, le code étant répartis entre différentes procédures. Les structures de données ne sont pas considérées comme centrales mais ce sont elles qui détermineront l'efficacité des algorithmes.

Fortran est le premier langage algorithmique. Il est principalement utilisé pour le calcul scientifique. Cobol est le langage destiné aux applications de gestion : c'est le langage le plus utilisé dans le monde. Basic est un langage très rudimentaire initialement développé dans un but didactique. Aujourd'hui, les micro-ordinateurs ont des capacités suffisantes pour supporter des langages plus évolués. Pascal est essentiellement destiné à l'enseignement de la programmation : sa rigueur et sa simplicité lui valent un grand succès. Le langage C est un langage qui a été développé par les laboratoires Bell. Il est orienté vers la programmation système. C'est le langage de développement du système Unix.

- Langages fonctionnels : Lisp, Caml, Logo

  Ces langages reposent sur la notion de fonction. On formule les choses en termes de fonctions, et on peut même parfois définir des fonctions dont les arguments sont eux-mêmes des fonctions. List (*LISt Processing*) est un langage fonctionnel qui a été conçu pour la manipulation d'expressions symboliques : des listes. Les fonctions de base permettent de récupérer le premier élément d'une liste, et d'en manipuler la fin. Il a connu un grand succès en Intelligence Artificielle, où les données ne sont pas numériques.

- Langage de programmation logique : Prolog

  Prolog est un langage original un peu à part, inventé par un français : Colmerauer. Il reprend certains concepts de LISP, mais est avant tout basé sur la notion d'inférence logique. Il a surtout été utilisé en Intelligence Artificielle, notamment pour l'implémentation de systèmes experts.

- Langages orientés objets : Smalltalk, C++, Java

  La programmation objet met en avant les notions de modularité, d'encapsulation et de réutilisation. Les données et les fonctions sont encapsulées à l'intérieur des objets, et les objets sont organisés par classes d'objets et ensembles de classes d'objets. Dans cette approche, on ne s'attache pas à la fonctionnalité du système, mais on décrit le monde comme constitué d'objets typés, fournissant chacun des données et les méthodes permettant de les manipuler. Un programme consiste en un ensemble d'objets qui interagissent entre eux.

  SmallTalk est le premier langage orienté objet. Il a jeté les bases de la programmation objet. C++ est un successeur de C qui incorpore les concepts de la programmation objet tout en gardant des caractéristiques du langage C. C'est un langage très efficace, qui connait aujourd'hui un essor industriel. Java est inspiré de C et de C++. C'est le langage qui s'impose aujourd'hui pour le développement d'applications Internet car il fournit un nombre important de librairies, et assure la portabilité des programmes Java d'une machine à l'autre.

Tout programme écrit dans un langage évolué doit être traduit dans du langage machine pour pouvoir être exécuté. Il existe deux manières traditionnelles d'effectuer cette traduction :

- Utiliser un *compilateur* pour créer à partir du programme source un programme exécutable stocké dans un fichier. L'exécution est alors faite directement à partir des instructions binaires contenues dans ce fichier.
- Utiliser un *interpréteur* qui peut exécuter directement le programme source. L'interpréteur est un programme qui remplace chaque instruction du programme source en

une suite d'instructions équivalente compréhensible par le processeur. L'interprétation est effectuée instruction par instruction. Elle est plus lente que la compilation mais permet l'écriture de programmes complexes (comme des programmes qui se modifient eux-mêmes), utilisant des données a priori inconnues au moment de la compilation.

Une des originalités du langage Java est d'avoir imaginé un procédé intermédiaire entre la compilation et l'interprétation. Les programmes sources Java sont compilés dans un langage binaire appelé pseudo-code destiné à une machine abstraite imaginaire, appelée machine virtuelle Java. Sur chaque machine réelle particulière tourne alors un programme simulant la machine virtuelle java, c'est-à-dire en réalité un interpréteur de pseudo-code.

Un avantage annexe non négligeable est une plus grande portabilité des programmes sources qui nécessitent souvent des retouches particulières pour être compilés sur certaines machines.

4.2 Exemple de logiciel : le compilateur

Un compilateur est avant tout un programme : c'est lui qui permet de traduire automatiquement le texte source d'un programme écrit dans un langage évolué donné vers un code binaire en langage machine. Mais il a aussi d'autres fonctionnalités.

4.2.1 Fonctionnalités d'un compilateur

Les différentes fonctionnalités d'un compilateur sont les suivantes :

- Traduire un programme écrit sur un ou plusieurs fichiers dans un langage évolué en langage machine (création d'un fichier exécutable). Il s'agit de ce qu'on appelle la compilation du programme. C'est la fonctionnalité principale du compilateur.
- Traduire un ensemble de fonctions données dans un ou plusieurs fichiers dans un langage évolué en langage machine. Le résultat peut être non exécutable (car incomplet), et on parle alors de *fichier objet* ou *code objet*.
- Lier plusieurs fichiers objets pour obtenir un unique fichier objet ou un fichier exécutable. Cette étape de la compilation s'appelle l'*édition de liens*.
- Vérifier la syntaxe des programmes : l'ordinateur ne tolère aucun écart par rapport à la syntaxe, même si le sens du programme semble évident. S'il y a des fautes de syntaxe, le compilateur ne produira pas de code exécutable, mais signalera le type des erreurs détectées.
- Optimiser un code exécutable en l'adaptant au type de machine sur lequel le programme sera exécuté.

Outre ces fonctionnalités que l'on retrouve dans tout compilateur, on peut avoir d'autres fonctionnalités (ou d'autres logiciels dédiés à ces fonctionnalités) :

- Un *débogueur* permettant de tracer instruction par instruction l'exécution d'un programme, c'est-à-dire de montrer instruction après instruction comment la mémoire est modifiée (cf interpréteur MAMIAS). Un débogueur permet d'analyser des erreurs de programmation.
- Un éditeur spécialisé pour le langage de programmation choisi. Celui-ci va permettre de vérifier si ce qui est écrit est syntaxiquement correct, bien défini, au moment même de l'écriture du programme source. Les erreurs de syntaxes évidentes pourront ainsi être corrigées au moment même de l'écriture du programme, avant compilation.

Ces différentes fonctionnalités peuvent être accessibles au sein d'un même programme qui propose ainsi un « environnement de programmation », comme le fait par exemple VisualC++.

4.2.2 Composition d'un logiciel de compilation

Le logiciel fourni ne comprend pas que le programme du compilateur. Il est livré sous la forme d'un répertoire comportant lui-même plusieurs sous-répertoires et de nombreux fichiers qui seront utilisés par le programme du compilateur. On y trouve en particulier :

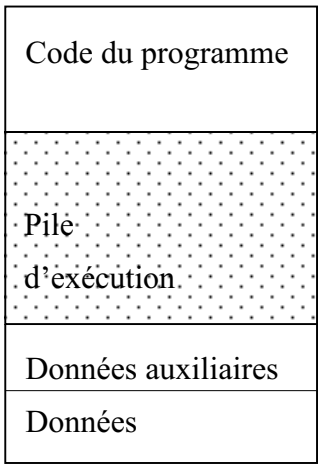
- L'*exécutable* (la commande) permettant la compilation (encore appelé le compilateur). Eventuellement d'autres exécutables, utilisés par la commande de compilation, mais pouvant également être lancés séparemment, comme une commande pour réaliser l'éditions de liens, ou obtenir un code écrit dans un langage d'assemblage.
- Des *fichiers d'en-tête* permettant au programmeur de faire appel à des fonctions prédéfinies dans le système d'exploitation ou décrites dans des fichiers du logiciel de compilation. Il s'agit en général de fonctions graphiques, mathématiques, etc.
- Des *librairies* contenant le code des fonctions définies par le logiciel de compilation, et dont les fonctions sont indiquées dans les fichiers d'en-tête.
- Des *fichiers de configuration*. Un fichier de configuration permet au compilateur de savoir où chercher les fichiers supplémentaires dont il a besoin, librairies ou fichiers d'en-tête. Un fichier de configuration peut aussi permettre au logiciel de conserver des informations sur les derniers fichiers compilés, sur la forme, la couleur, l'emplacement de la fenêtre de compilation.
- Des *exemples*. Il s'agit de petits programmes permettant à l'utilisateur de savoir comment utiliser le logiciel, comment utiliser les fonctions prédéfinies, etc.
- Des fichiers permettant de simplifier les opérations de compilation.
- Des *fichiers source* pour certaines opérations particulières.
- Des fichiers de *documentation*. Ils contiennent la documentation non présente dans l'aide en ligne.
- Des fichiers d' *aide*. Ces fichiers contiennent des textes accessibles directement à partir de l'exécutable.

4.2.3 Etapes d'une compilation

La compilation d'un programme source (écrit dans un langage évolué) vers un langage cible (le langage machine ou un langage d'assemblage par exemple) s'effectue en plusieurs étapes. Les deux principales sont les suivantes :

- Affecter à chaque variable du programme source une adresse en mémoire. En effet à chaque variable doit correspondre une zone mémoire (une cellule ou un ensemble consécutif de cellules) contenant la valeur (ou la suite des valeurs) que prendra la variable. Référencer une variable dans le source, c'est accéder à la cellule associée dans le code. Donc à chaque référence à cette variable dans le programme source, on substituera l'adresse en mémoire correspondant à cette variable.
- Remplacer une à une chaque instruction du programme source par le code correspondant dans le langage cible. En effet à chaque instruction du langage évolué correspond une succession stéréotypée d'instructions du langage cible.

Si le langage cible est le langage machine, le fichier exécutable est destiné à être placé dans une zone de la mémoire centrale pour y être directement exécuté. Cette zone sera généralement constituée d'une zone de code (les instructions proprement dites), d'une zone de données (mentionnées explicitement dans le programme-source) et d'une zone de données nécessaire à la traduction des instructions du langage source. Il existe aussi une zone de pile destinée à stocker des paramètres pour l'exécution de fonctions que l'on n'expliquera pas ici.



Indiquons au passage que lorsqu'une instruction du programme cherche à accéder à une zone qu'elle n'est pas censé utiliser (comme la pile d'exécution), l'exécution du programme est interrompu par un signal d'erreur. Ainsi, avec le langage C, on aura le message d'erreur : « memory fault - core dumped ». Ce qui signifie qu'il y a eu tentative de lire ou d'écrire dans une zone interdite, et que le contenu de la mémoire centrale concernant le programme a été « vidée » dans un fichier intitulé core. Ce fichier pourra être utilisé par un débogueur pour analyser l'état qui précédait l'interruption. En particulier, un débogueur symbolique permettra d'identifier directement la ligne du programme source qui a déclenché la faute d'accès mémoire.

4.2.4 Exemple: la compilation d'un petit langage évolué en MAMIAS

On souhaite pouvoir écrire de petits programmes, traduisibles en MAMIAS, mais écrits dans un langage plus simple, plus évolué. En particulier, on aimerait :

1. Avoir une syntaxe plus agréable, qui aide à mémoriser ce que fait l'instruction.
2. Avoir des instructions qui ne font pas référence au registre de l'accumulateur.
3. Disposer de règles d'assemblage d'instructions pour éviter de faire explicitement référence aux numéros (adresses) des mémoires dans les sauts.
4. S'abstraire des numéros des mémoires en y faisant référence par des noms : les variables.

On va montrer ici comment chacun des points précédents peut être réalisés, en définissant un nouveau langage, compilable en MAMIAS, dont la syntaxe se rapproche de celle des langages impératifs. Nous essayerons alors de montrer que le langage obtenu est bien compilable en MAMIAS en brossant les grandes lignes d'un algorithme général de compilation.

Reprenons un à un les points énumérés plus haut.

Point 1 : pour réaliser ce premier point, on peut utiliser la notation symbolique que nous avions mise en commentaires pour introduire la signification des instructions du langage MAMIAS. Il ne s'agit que d'un changement de notation, et les instructions restent les mêmes.

Instruction	**Signification**
INIT *x*	*Acc* ← *x*
CHARGE *n*	*Acc* ← (*n*)
RANGE *n*	*(n)* ← *Acc*
ET *n*	*Acc* ← *Acc* et (*n*)
SAUTE *n*	si *Acc* = 0 aller a *n*
ADD *n*	*(n)* ← *Acc* + (*n*)
DEC *x*	*Dec* (*Acc*, ←, *a*) ou *Dec* (*Acc*, →, *a*) selon le signe de *x*, avec *a*>0, égal à la valeur absolue de *x*.
STOP	*Stop*

La traduction du nouveau langage est très simple à réaliser puisqu'il ne s'agit au fond que d'une notation différente. On utilisera la colonne de gauche du tableau précédent pour effectuer la traduction, instruction par instruction. Seul le cas du décalage demande une attention particulière, car il nécessite un calcul (la valeur absolue de x) lorsque x est négatif.

Point 2: Se débarrasser de l'accumulateur. On aimerait avoir des instructions qui ne parlent que des mémoires, et ne font pas référence à l'accumulateur. Par exemple, au lieu d'avoir l'instruction INIT qui « initialise » la valeur contenue dans l'accumulateur, on souhaiterait pouvoir initialiser directement la valeur contenue dans une mémoire, et disposer, par exemple, d'une instruction notée (*n*) ← *x*, qui place dans la mémoire (n) la valeur *x* et permet donc d'initialiser la mémoire (n).

De même, au lieu d'avoir une addition qui fasse l'addition du contenu d'une mémoire (n) avec celui de l'accumulateur, on aimerait avoir une instruction qui permette de réaliser l'ajout d'une valeur *x* au contenu d'une mémoire (n) et place ensuite le résultat dans la mémoire (n). Cette instruction serait par exemple notée : (*n*) ← (*n*) + *x*.

Même chose pour le ET bit-à-bit et pour les instructions de décalages : on veut pouvoir faire le ET bit-à-bit du contenu d'une mémoire (n) avec celui d'une mémoire (p), et retrouver le résultat dans (n) (ou dans (p)) par exemple. Pour les décalages, on aimerait pouvoir décaler directement les bits d'une mémoire donnée, vers la gauche ou vers la droite.

Les tableaux suivants montrent que ces nouvelles instructions peuvent effectivement être traduites par une série d'instructions MAMIAS :

Nouvelle instruction	**Traduction MAMIAS**	**Signification**
(*n*) ← *x*	i INIT *x* i+1 RANGE *n*	*Acc* ← *x* (*n*) ← *Acc*

Nouvelle instruction	Traduction MAMIAS	Signification
$(n) \leftarrow (n) + x$	i INIT x i+1 ADD n	$Acc \leftarrow x$ $(n) \leftarrow (n) + Acc$
$(n) \leftarrow (n)$ et (p)	i CHARGE n i+1 ET p i+2 RANGE n	$Acc \leftarrow (n)$ $Acc \leftarrow Acc$ et (p) $(n) \leftarrow Acc$
Dec ((n) , $\leftarrow$, a)	i CHARGE n i+1 DEC a i+2 RANGE n	$Acc \leftarrow (n)$ Dec (Acc , $\leftarrow$, a) $(n) \leftarrow Acc$

On remarque au passage qu'une instruction du nouveau langage se trouve traduite par plusieurs instructions MAMIAS. (Pour simplifier le problème, nous supposerons pour la suite que la traduction d'une instruction du nouveau langage s'effectue sur une seule ligne).

On peut imaginer avoir accès à d'autres instructions dans ce petit langage, comme par exemple $(n) \leftarrow (p)$ ou $(n) \leftarrow (p) + (q)$, ou encore $(n) \leftarrow (p) + x$. Ces exemples pourront être traités en exercice.

Pour éviter de référencer les adresses de mémoires particulières (Point 3) et l'accumulateur (Point 2) dans l'instruction SAUTE, on aimerait disposer d'instructions de sauts plus générales, permettant de réaliser des assemblages d'instructions, en testant une condition. Ainsi par exemple, on pourrait disposer d'une instruction de saut permettant de tester si le contenu d'une mémoire est nul, afin d'effectuer ensuite un branchement, soit vers une instruction 1, soit vers une instruction 2. Cette nouvelle instruction, notée par exemple

si (n)=0
  <intruction 1>
sinon
  <instruction 2>
finsi

est bien traduisible en MAMIAS, comme le montre le tableau :

Nouvelle instruction	Traduction en MAMIAS	Signification
si (n) = 0	i CHARGE n	$Acc \leftarrow (n)$
<instruction 1>	i+1 SAUTE i+5	si Acc = 0 aller a i+5
sinon	i+2 <instruction 2>	. . .
<instruction 2>	i+3 CHARGE 0	$Acc \leftarrow 0$
finsi	i+4 SAUTE i+6	si Acc = 0 aller a i+6
	i+5 <instruction 1>	. . .

On pourrait de la même façon, compiler facilement des instructions de branchement conditionnel qui testerait si le contenu d'une mémoire est différent de zéro, ou s'il est positif.

Plus intéressant encore, on peut introduire une notion de boucle, avec des intructions complexes comme

pour (m) variant de 1 à j faire

<instruction>

finpour

ou encore

tantque (m) ≠ 0 faire

<instruction>

fintantque

La première instruction, l'instruction « pour (m) variant de 1 à j», exécute j fois de suite l'instruction qui suit, mais dans un contexte à chaque fois différent : la première fois, la mémoire (m) est initialisée à 1, et l'instruction est exécutée dans ce contexte, puis le contenu de (m) est augmenté de 1, l'instruction a nouveau exécutée, etc. La mémoire (m) prend donc successivement les valeurs 1, 2, etc., jusqu'à j, et pour chacune de ces valeurs, une exécution de l'instruction est effectuée. Il y a en tout exactement j exécutions de l'instruction, mais ces instructions ne s'exécutent pas toutes dans le même contexte, car elles sont entrelacées de nouvelles initialisations de la mémoire (m).

La seconde instruction, l'instruction « tantque (m) ≠0 » réalise en boucle l'instruction qui suit, en testant au préalable si la valeur de la mémoire (m) est non nulle. Tant que cette valeur est non nulle, on exécute une nouvelle fois l'instruction. Lorsque la valeur de (m) est nulle, on considère que l'instruction « tantque (m) ≠ 0 » est terminée, et on n'exécute pas l'instruction interne. Cette instruction « tantque (m) ≠0 » peut déclencher une infinité d'exécutions de l'instruction si la mémoire (m) n'est pas modifiée par cette instruction, et si elle est différente de zéro à l'origine. Si (m) vaut zéro au moment de l'exécution du « tantque (m) ≠0 », l'instruction ne sera jamais exécutée. Cette instruction est bien compilable en MAMIAS, comme le montre le tableau suivant :

Nouvelle instruction	**Traduction en MAMIAS**		**Signification**
tantque (m) ≠ 0	i	CHARGE m	*Acc* ← (m)
<instruction>	i+1	SAUTE i+5	si *Acc* = 0 aller a i+5
fintantque	i+2	<instruction>	. . .
	i+3	INIT 0	*Acc* ← 0
	i+4	SAUTE i	si *Acc* = 0 aller a i

On peut ensuite facilement étendre cette boucle « tantque (m) ≠ 0 » à l'exécution d'une série de P instructions :

Nouvelle instruction	Traduction en MAMIAS	Signification
tantque (m) ≠ 0	i CHARGE m	*Acc* ← (m)
<instruction 1>	i+1 SAUTE i+P+4	si *Acc* = 0 aller a i+P+4
. . .	i+2 <instruction 1>	. . .
<instruction P>	. . . . .	. . .
fintantque	i+P+1 <instruction P>	. . .
	i+P+2 INIT 0	*Acc* ← 0
	i+P+3 SAUTE i	si *Acc* = 0 aller a i

On voit que l'on peut étendre à loisir le nouveau langage avec des instructions plus complexes en autorisant de manière générale l'utilisation d'une série d'instructions aux emplacements initialement destinés à une simple instruction. C'est la démarche qu'ont suivie les langages de programmation impérative que nous avons mentionnés, comme Pascal, C, ou Fortran.

Quant au dernier point, l'introduction de noms pour remplacer les mémoires, il est très facile à réaliser et fondamental pour avoir un langage évolué digne de ce nom. On peut en effet utiliser des noms quelconques pour désigner les adresses mémoires. Ces noms, ou *variables* dans la terminologie des langages de programmation, seront nécessairement introduits dans un certain ordre dans le texte du programme. Pour attribuer automatiquement une adresse à une variable, on peut procéder de la façon suivante : puisque les variables apparaissent dans un certain ordre, on peut convenir que la première variable qui apparaît dans le texte du programme correspondra à la mémoire (31). La seconde, à la mémoire (30), et ainsi de suite, par ordre d'apparition des variables dans le texte du programme. On construit ainsi une table, appelée traditionnellement *table des symboles*, qui indique la correspondance univoque entre les noms (les variables) et les mémoires. Il suffira ensuite d'utiliser cette table pour traduire systématiquement toutes les instructions.

Par exemple, le programme suivant, écrit dans notre petit langage évolué :

```
x ← 0
i ← 1
j ← p
tantque j ≠ 0
   x ← x + i
   i ← x + i
   j ← j + -1
fintanque
```

peut être transformé automatiquement grâce à la table des symboles

variable	mémoire
x	(31)
i	(30)
j	(29)

en

```
(31) ← 0
(30) ← 1
(29) ← p
tantque (29) ≠ 0
   (31) ← (31) + (30)
   (30) ← (29) + (30)
   (29) ← (29) + -1
fintantque
```

Le nouveau texte obtenu peut être compilé ensuite à l'aide des tables de traduction précédentes.

Mais nous n'avons fait ici qu'esquisser une méthode possible de compilation qui utiliserait ces tables. La définir véritablement présente encore quelques difficultés. Une première chose à faire sera d'adapter ces tables pour rendre compte du fait que (et c'est le cas général) la traduction d'une instruction nécessite plusieurs lignes. Une autre difficulté est que les adresses des instructions données par nos tables de traduction sont, d'une part relatives (puisqu'elles sont indexées par i), et d'autre part incorrectes, car il fallait tenir compte du nombre d'instructions MAMIAS nécessaires pour traduire une <instruction> (ou une série d'instructions) dans les instructions complexes, avant de procéder à la traduction de l'instruction complexe elle-même. En effet, dans nos traductions, nous avons, pour simplifier le problème, fait comme si les instructions que l'on assemblaient étaient traduites par une seule ligne MAMIAS. Or le nombre de lignes nécessaires pour traduire une instruction figurant dans une instruction complexe n'est pas connu au moment où l'on veut effectuer la traduction de l'intruction complexe. Mais si l'on procède en commençant par traduire les instructions simples, puis les instructions complexes les plus internes, et ensuite les instructions complexes qui les englobent, etc., on sent bien qu'il doit être possible de préciser une méthode systématique (i.e. un algorithme) pour effectuer la compilation. A titre d'exercice, on pourra chercher à définir réellement une telle méthode, pour voir quelles sont les difficultés qui se présentent.