Chapitre 12

Codage en langage machine

Comment faire réagir un microprocesseur tel que le 8088 ? Comment programmer en langage machine sur un système tel que le PC ? Voici les questions auxquelles nous allons répondre dans ce chapitre.

12.1 Premières instructions du 8088

Commençons par étudier quelques instructions du 8088 de façon à programmer de façon concrète un premier programme.

12.1.1 Premières instructions de transfert du 8088

Nous avons vu la notion de registre. Voyons comment initialiser un registre, comment transférer d'un registre à un autre et comment initialiser un élément de mémoire vive.

12.1.1.1 Syntaxe générale d'un transfert en langage symbolique

Syntaxe.- Toute instruction de transfert sera représentée en **langage symbolique** (qui permet une meilleure compréhension à nous, pauvres humains) de la façon suivante :

```
MOV destination, source
```

où `MOV` est un mnémonique pour *MOVe* (*transférer* en anglais), `destination` ou `source` est le nom d'un registre et l'autre (`source` ou `destination` suivant le cas) dépend du contexte.

Remarque.- Attention à l'ordre, d'abord la destination, ensuite la source, ce qui n'est pas nécessairement l'ordre auquel on peut s'attendre. Une façon mnémothechnique de s'en souvenir est de penser à l'analogue en langage de haut niveau :

```
destination := source;
```

Que transférer ?.- Le 8088 est un microprocesseur 16 bits, c'est-à-dire que les transferts se font de façon naturelle sur seize bits, soit deux octets. En fait, puisque l'octet est la plus petite unité et que le bus des données est de huit bits, on peut également transférer un octet.

12.1.1.2 Initialisation d'un registre général

Langage symbolique.- L'instruction d'initialisation d'un registre s'écrit :

```
MOV reg, constante
```

où `reg` désigne le registre et `constante` est une constante, huit ou seize bits suivant la capacité du registre.

Sémantique.- La signification de cette instruction est l'initialisation du registre `reg` par cette constante.

Langage machine.- L'instruction en langage symbolique n'est là que pour nous aider à s'en souvenir. Il faut maitenant faire le lien avec le microprocesseur. Une telle instruction en langage machine exige deux octets ou trois octets, suivant que la constante occupe un ou deux octets :

le premier octet est appelé **code opération**, abrégé en **opcode**.

Opcodes des initialisations.- On pourrait choisir les codes opération au hasard. Pour une meilleure ergonomie (le 8088 comprend plusieurs centaines d'instructions), ceci n'est pas le cas. Le code d'opération d'une instruction d'initialisation est, en binaire :

```
1011 wreg
```

où :

- `w` (pour l'anglais *word*, soit mot, le nom pour une donnée de deux octets) est un bit valant 1 si la transfert s'effectue sur deux octets (soit un mot) et 0 s'il s'effectue sur un octet.
- `reg` est une suite de trois bits permettant de déterminer le registre de la façon spécifiée par la figure 12.1.

16 bits ($w = 1$)	8 bits ($w = 0$)
000 AX	000 AL
001 CX	001 CL
010 DX	010 DL
011 BX	011 BL
100 SP	100 AH
101 BP	101 CH
110 SI	110 DH
111 DI	111 BH

FIGURE 12.1 – Désignation des registres généraux du 8088

Remarquez que les registres de segment n'apparaissent pas dans ce tableau. Ils ne peuvent pas être initialisés directement.

Exercice corrigé.- *Traduire l'instruction* :

```
MOV AL, 10
```

en langage machine, la constante étant exprimé en décimal.

D'après ce que nous venons de dire, cette instruction se traduit par les deux octets (représentés en binaire) :

```
1011 0000 0000 1010
```

soit B0 0Ah en hexadécimal.

Remarque.- En général on utilisera l'hexadécimal en langage symbolique, c'est-à-dire qu'on aurait écrit :

```
MOV AL, 0Ah
```

dès le départ.

Vocabulaire.- Il est traditionnel de parler d'**adressage immédiat** au lieu d'initialisation des registres.

12.1.1.3 Chargement d'un registre de segment depuis un registre général ou une case mémoire

Langage symbolique.- L'instruction de transfert s'écrit toujours :

```
MOV destination, source
```

ce qui ne nous apprend pas grand chose.

Langage machine.- On a deux octets dans le cas de registres et deux ou quatre octets dans le cas d'une case mémoire :

```
| 1000 1110 | mod 0 reg r/m |           |           |
```

– Les deux bits de `reg` spécifient le registre de segment suivant le tableau de la figure 12.2. Dans le cas qui nous intéresse, on ne peut pas avoir $reg = 01$, c'est-à-dire que l'on ne peut pas charger le registre de segment de code.

reg	Segment
00	ES
01	CS
10	SS
11	DS

FIGURE 12.2 – Désignation des registres de segment du 8088

– les deux bits de `mod` et les trois bits de `r/m` spécifient le mode d'adressage. Nous reviendrons plus tard sur la signification générale. Contentons-nous ici du cas où `mod = 11`, ce qui signifie que la source est un registre général, `r/m` spécifiant alors le registre général suivant le tableau de la figure 12.1.

Exercice corrigé.- *Traduire l'instruction* :

```
MOV DS, AX
```

en langage machine.

D'après ce que nous venons de dire, cette instruction se traduit par les deux octets représentés en binaire par :

```
| 1000 1110 | 11 0 11 000 |
```

soit 8E D8h.

12.1.1.4 Chargement d'une donnée dans une case mémoire ou un registre

Introduction.- Voyons comment initialiser un élément de mémoire, disons celui dont l'adresse est `11Eh`, avec la valeur `25h`, par exemple.

On peut penser à une instruction du type :

```
mov [11E],25
```

où, par convention, une constante s'écrit telle quelle (en hexadécimal) et une adresse mémoire entre crochets.

Cependant le microprocesseur n'acceptera pas cette instruction et réagira par une erreur.

Pourquoi ?

L'instruction n'est pas suffisamment précise : s'agit-il de l'initialisation de l'octet d'adresse `11Eh` ou du mot d'adresse `11Eh`. Le résultat n'est pas le même.

Il faut donc un moyen de préciser cette instruction.

Syntaxe en langage symbolique.- On a les instructions :

```
mov word ptr [mem],val
mov byte ptr [mem],val
```

pour initialiser l'emplacement mémoire d'adresse `mem` avec la valeur `val`. Dans le premier cas il s'agit d'un emplacement d'un mot, dans le second d'un emplacement d'un octet.

Exemples.- On écrit :

```
mov word ptr [11E],25
mov byte ptr [122],30
```

Langage machine.- L'instruction en langage machine comporte trois à six octets :

```
| 1100 011w | mod 000 r/m | adresse bas | adresse haut | donnée | donnée haut |
```

où :

- comme ci-dessus, `w` est un bit valant 1 si le transfert s'effectue sur deux octets (soit un mot) et 0 s'il s'effectue sur un octet ;
- comme nous l'avons déjà dit, les deux bits de `mod` et les trois bits de `r/m` spécifient le mode d'adressage. Nous reviendrons plus tard sur la signification générale. Contentons-nous ici du cas où `mod = 00` et `r/m = 110`, indiquant qu'il s'agit d'un emplacement de mémoire vive spécifié par son décalage constituant les deux octets suivants (d'abord l'octet de poids fort puis celui de poids faible).
- La donnée est spécifiée par le dernier octet (si $w = 0$) ou les deux derniers octets (lorsque $w = 1$), d'abord l'octet de poids faible puis celui de poids fort.

Exercice corrigé.- *Traduire l'instruction* :

```
move byte ptr [0400], 05
```

en langage machine, la constante étant exprimé en décimal.

D'après ce que nous venons de dire, cette instruction se traduit par les cinq octets (représentés en binaire) :

```
| 1100 0110 | 00 000 110 | 0000 0000 | 0000 0100 | 0000 0101 |
```

soit C6 06 00 04 05h.

12.1.2 Instruction de retour

Notion.- Lorsqu'on fait appel, en langage machine, à un sous-programme, l'adresse du programme à laquelle on se trouve (c'est-à-dire les contenus des registres `CS` et `IP`, même si ce dernier registre n'est pas visible) est sauvegardée de façon à pouvoir revenir à cette instruction une fois le sous-programme exécuté. Le sous-programme doit lui-même indiquer qu'il a terminé son travail de

façon à retourner à cette adresse. L'instruction `STOP` ne conviendrait pas : elle indiquerait que le programme en son entier est terminé.

Langage symbolique.- Cette instruction s'écrit tout simplement :

```
RET
```

pour l'anglais *RETurn*.

Langage machine.- Cette instruction est codée par un octet `1100 0011`, soit C3h.

12.2 Programmer en langage machine avec debug

Nous allons voir comment entrer directement un programme en code machine (le seul qui est compréhensible par le microprocesseur) en utilisant debug.

12.2.1 Le programme

Le problème.- Lorsqu'on utilise un système d'exploitation, la mémoire vive n'est pas entièrement disponible pour l'utilisateur. Une partie de la mémoire vive est occupée par des variables du système d'exploitation, une autre par la mémoire graphique,... C'est le système d'exploitation qui décide de l'**organisation de la mémoire** (*memory map* en anglais).

N'entrons pas dans les détails de l'organisation de la mémoire sous MS-DOS. Cherchons, à l'aide de debug, une zone de mémoire vive non occupée et plaçons, grâce à un programme en langage machine, une valeur dans cette zone.

On s'aperçoit que la zone commençant à 1000 :400 ne contient que des 'KS' :

```
C:>debug
-D 1000:400
1000:0400 4B 53 4B 53 4B 53 4B 53-4B 53 4B 53 4B 53 4B 53 KSKSKSKSKSKSKSKS
1000:0410 4B 53 4B 53 4B 53 4B 53-4B 53 4B 53 4B 53 4B 53 KSKSKSKSKSKSKSKS
1000:0420 4B 53 4B 53 4B 53 4B 53-4B 53 4B 53 4B 53 4B 53 KSKSKSKSKSKSKSKS
1000:0430 4B 53 4B 53 4B 53 4B 53-4B 53 4B 53 4B 53 4B 53 KSKSKSKSKSKSKSKS
1000:0440 4B 53 4B 53 4B 53 4B 53-4B 53 4B 53 4B 53 4B 53 KSKSKSKSKSKSKSKS
1000:0450 4B 53 4B 53 4B 53 4B 53-4B 53 4B 53 4B 53 4B 53 KSKSKSKSKSKSKSKS
1000:0460 4B 53 4B 53 4B 53 4B 53-4B 53 4B 53 4B 53 4B 53 KSKSKSKSKSKSKSKS
1000:0470 4B 53 4B 53 4B 53 4B 53-4B 53 4B 53 4B 53 4B 53 KSKSKSKSKSKSKSKS
-q
```

Écrivons un programme en langage machine qui place 5 à l'emplacement mémoire 1000 :400.

Le programme en langage symbolique.- On doit initialiser le registre de segment DS à 1000h et placer 5 au décalage 400h, sans oublier de revenir du « sous-programme ». Puisqu'on ne peut pas initialiser directement le registre DS, on initialise le registre AX, puis on tranfère la valeur dans DS, ce qui donne :

```
mov ax,1000
mov ds,ax
mov byte ptr [0400],05
ret
```

Traduction du programme en langage machine.- La première instruction est une initialisation de registre :

```
1011 1 000   0000 0000   0001 0000
```

soit B8 00 10h.

Les autres instructions ont été traitées en exercice corrigé. La traduction de ce programme en code machine est donc :

```
B80010
8ED8
C606000405
C3
```

Nous sommes passé à la ligne pour plus de lisibilité mais, bien entendu, le programme est une suite de 0 et de 1 ou, de façon plus lisible pour les êtres humains, une suite de nombres hexadécimaux.

12.2.2 Un aparté : examen du contenu des registres

12.2.2.1 Examen de tous les registres

Introduction.- Comme nous l'avons dit, `debug` permet d'examiner et de changer le contenu de la mémoire, d'entrer un programme (en langage d'assemblage) et de faire exécuter un tel programme. Par mémoire, il faut entendre mémoire vive et mémoire de masse. Si on utilise un microprocesseur, le contenu des registres change sans cesse ; on ne peut donc pas obtenir d'information (même instantanée) sur le contenu de ceux-ci.

L'utilitaire `debug` permet cependant d'émuler la manipulation des registres.

Syntaxe.- Pour visualiser le contenu de tous les registres (de l'émulateur) avec `debug`, il suffit d'utiliser la commande 'R' (pour *Register*).

Exemple.- On a :

```
C:> DEBUG <retour>
-R <retour>
AX=0000  BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
CS=2377  DS=2377 ES=2377 SS=2377 IP=0100 NV UP DI PL NZ NA PO NC
2377:0100 D6       DB  D6
```

Commentaires.- 1^o) L'utilitaire `debug` répond à la commande 'R' par trois lignes : la première ligne indique le contenu des registres généraux, des registres de pointeur et des registres d'index ; la seconde ligne indique le contenu des registres de segment, la valeur du pointeur d'instruction et la valeur de huit des indicateurs ; la troisième ligne montre l'instruction pointée par `CS:IP`.

2^o) La valeur d'un registre (sauf pour le registre des indicateurs) est indiquée en hexadécimal, avec quatre chiffres de façon systématique (même si la valeur commence par des zéros).

3^o) Bien entendu, comme nous l'avons déjà dit, il ne s'agit pas des vraies valeurs des registres mais de ceux de l'émulateur en ce qui concerne la premièrre ligne. À chaque appel de `debug`, les registres de la première ligne sont remis à zéro et tous les indicateurs sont positionnés à zéro. Les contenus des registres de segment varient d'une session à l'autre mais on pourra remarquer qu'ils ont tous la même valeur. Les instructions exécutées par `debug` changeront la valeur des registres comme le ferait le microprocesseur.

4^o) Nous analyserons plus tard la signification de la troisième ligne.

Code des indicateurs.- Le code pour les indicateurs est le suivant :

Indicateur	**Code lorsque positionné** (= 1)	**Code lorsque non positionné** (= 0)
OF	OV (OVerflow)	NV (No oVerflow)
DF	DN (DowN)	UP (UP)
IF	EI (Enable Interrupt)	DI (Disable Interrupt)
SF	NG (NeGative)	PL (PLus)
ZF	ZR (ZeRo)	NZ (Not Zero)
AF	AC (Auxiliary Carry)	NA (No Auxiliary carry)
PF	PE (Parity Even)	PO (Parity Odd)
CF	CY (CarrY)	NC (No Carry)

12.2.2.2 Changement du contenu d'un registre

Syntaxe.- Pour changer le contenu d'un registre de l'émulateur (n'importe quel registre à 16 bits sauf le registre des indicateurs) en utilisant `debug`, il suffit de l'appeler avec la commande 'R' suivi de son nom. Sa valeur est alors indiquée (comme nous l'avons vu ci-dessus) et le prompteur est ' :'. Il suffit d'indiquer la valeur souhaitée (en hexadécimal, suivie d'un retour).

Exemple.- Si on veut que le registre `ax` contienne `FF01h`, on effectue les manipulations suivantes :

```
C:> DEBUG
-R AX
AX 0000
:FF01
-R AX
AX FF01
:
```

la dernière commande de `debug` permet de vérifier qu'on a bien changé la valeur du registre.

Remarques.- 1^o) Il est fortement recommandé de ne pas changer (pour l'instant) le contenu des registres de segment. En effet ces valeurs ont été données par le système d'exploitation là où il y a de la place disponible ; les changer pourrait avoir pour conséquence de modifier une zone utilisée par ailleurs.

2^o) Même si on met moins de quatre chiffres (hexadécimaux), `debug` affiche toujours avec quatre chiffres :

```
C:> DEBUG
-R CX
CX 0000
:2
-R CX
AX 0002
:
```

3^o) L'utilitaire `debug` indique une erreur si on essaie de placer un contenu à plus de cinq chiffres :

```
C:> DEBUG
-R CX
CX 0000
:12345
Erreur
```

ou si on essaie d'accéder à un registre à huit bits :

```
C:> DEBUG
-R AH
Erreur
```

4^o) Nous verrons plus tard comment changer le contenu du registre des indicateurs.

5^o) Rappelons qu'il s'agit d'une émulation. Le registre ne contient pas réellemnt la valeur indiquée.

12.2.3 Codage en langage machine avec debug

12.2.3.1 Introduction du programme en mémoire vive

On va entrer le programme en langage machine à un certain emplacement de la mémoire vive. Le segment de cet emplacement est déterminé par le registre CS. On va laisser le système d'exploitation et debug choisir la valeur de CS à notre place : lorsqu'on lance debug, des valeurs de CS, DS et SS, toutes égales, sont choisies par le système d'exploitation.

Pour le système d'exploitation MS-DOS, on entre le programme à partir du décalage 100h, les 256 premiers octets étant occupés par un *préfixe*.

Créons un répertoire, disons COM, pour y placer les divers programmes :

```
C:\>mkdir COM
C:\>cd COM
C:\COM\>
```

Entrons donc le programme comme nous l'avons vu ci-dessus :

```
C:\COM\>debug
-E CS:100 B8 00 10
-E CS:103 8E D8
-E CS:105 C6 06 00 04 05
-E CS:10A C3
-
```

Nous avons choisi d'entrer le nombre d'octets correspondant à une instruction à chaque fois mais nous pouvons en entrer le nombre que nous voulons. Si on se trompe (et cela risque d'arriver souvent), il suffit de recommencer pour la ligne en question, il suffit même de le faire pour le seul octet erronné.

12.2.3.2 Sauvegarde du programme sur disque

Pour sauvegarder un programme avec debug, il faut nommer un fichier qui le contiendra et écrire le programme dans ce fichier.

Nommer un fichier.- On donne un nom au fichier de sauvegarde. On se sert pour cela de la commande N (pour *Name*) dont la syntaxe est :

```
-N <nom du fichier>
```

où le nom du fichier vérifie les spécifications de MS-DOS. Ce fichier sera placé dans le répertoire depuis lequel on a lancé debug.

Écriture.- Pour sauvegarder un programme, il suffit d'indiquer le nom du fichier (à l'aide de la commande N), d'indiquer le nombre d'octets à sauvegarder à l'aide de BX:CX puis d'écrire ce programme sur le disque avec la commande W (pour *Write*).

Dans le cas ci-dessus on aura :

```
C:\COM\>debug
-E CS:100 B8 00 10
-E CS:103 8E D8
-E CS:105 C6 06 00 04 05
-E CS:10A C3
```

```
-R BX
BX 0000
:
-R CX
CX 0000
:0B
-N EX1.COM
-W
Ecriture de 0000B octets
-Q
```

Remarque.- On remarquera l'extension classique d'un fichier nommé avec la commande N, à savoir ".com".

12.2.3.3 Exécution du programme

Principe.- Puisque ex1.com est un fichier exécutable, il suffit de l'appeler sur la ligne de commande pour l'exécuter :

```
C:\COM\> ex1.com
C:\COM\>
```

Évidemment, cela ne donne pas grand chose. Il n'y a pas d'erreur, c'est déjà ça.

Vérification de la bonne exécution.- Pour vérifier que le programme a bien exécuté ce que l'on voulait, utilisons à nouveau debug pour explorer la mémoire vive :

```
C:\COM\>debug
-D 1000:400
1000:0400 05 53 4B 53 4B 53 4B 53-4B 53 4B 53 4B 53 4B 53 .SKSKSKSKSKSKSKS
1000:0410 4B 53 4B 53 4B 53 4B 53-4B 53 4B 53 4B 53 4B 53 KSKSKSKSKSKSKSKS
1000:0420 4B 53 4B 53 4B 53 4B 53-4B 53 4B 53 4B 53 4B 53 KSKSKSKSKSKSKSKS
1000:0430 4B 53 4B 53 4B 53 4B 53-4B 53 4B 53 4B 53 4B 53 KSKSKSKSKSKSKSKS
1000:0440 4B 53 4B 53 4B 53 4B 53-4B 53 4B 53 4B 53 4B 53 KSKSKSKSKSKSKSKS
1000:0450 4B 53 4B 53 4B 53 4B 53-4B 53 4B 53 4B 53 4B 53 KSKSKSKSKSKSKSKS
1000:0460 4B 53 4B 53 4B 53 4B 53-4B 53 4B 53 4B 53 4B 53 KSKSKSKSKSKSKSKS
1000:0470 4B 53 4B 53 4B 53 4B 53-4B 53 4B 53 4B 53 4B 53 KSKSKSKSKSKSKSKS
-q
```

L'emplacement mémoire 1000 :400 a bien changé de valeur.

Remarque.- L'instruction RET qui termine le programme se contente de mettre le registre à IP, c'est-à-dire qu'on se retrouve à l'instruction CS :0000. Nous avons déjà dit qu'avec le système d'exploitation MS-DOS, lorsque l'interpréteur de commandes (le programme qui tourne par défaut) rencontre un exécutable .com, caractérisé par son suffixe, il détermine un emplacement mémoire vide suffisamment grand, renvoie son adresse de début dans le segment CS, place un préfixe de 256 octects dans le segment puis, à partir du décalage 100h, le code machine contenu dans le fichier. Lorsqu'il rencontre l'instruction RET, la main est donc passée au début du préfixe, où doit se trouver la façon de revenir proprement au système d'exploitation, c'est-à-dire à l'interpréteur de commandes.

12.2.3.4 Récupération d'un programme avec debug

Introduction.- Pour lire un programme (sauvegardé antérieurement), il suffit d'indiquer le nom du fichier (toujours à l'aide de la commande N), puis de demander de charger (*to Load* en anglais) ce programme grâce à la commande L.

Syntaxe du chargement.- Si le fichier a été nommé, il suffit d'utiliser :

```
L [adresse]
```

pour charger le programme à une adresse déterminée. Si aucune adresse n'est indiquée, le programme est chargé à `CS:100`.

Exemple.- Dans le cas ci-dessus on aura :

```
C:\COM\>debug
-N EX1.COM
-L
```

Il y a un petit problème à ce moment-là puisque nous ne savons pas comment visualiser le programme (action que nous allons voir dans la sous-section suivante).

Autre façon.- Une autre façon de faire (il s'agit d'un raccourci) consiste à indiquer le nom du programme à l'appel de `debug` :

```
C:\COM\>debug EX1.COM
```

12.2.4 Visualisation d'un programme

La commande `U` (pour *Unassemble*, c'est-à-dire *désassembler* en anglais) permet à la fois de visualiser le code machine d'un programme et l'équivalent de ce code en langage symbolique de `debug`.

Syntaxe.- On peut utiliser l'un des formats suivants :

```
U <adresse de départ> <adresse de fin>
U <adresse de départ> <L nombre d'octets>
```

où le nombre d'octets est exprimé en hexadécimal. Si on n'indique rien, 32 octets sont affichés en commençant à l'adresse `CS:IP`.

Exemple.- Dans le cas de l'exemple ci-dessus on a :

```
C:\COM\>debug EX1.COM
-U 100 10B
24C3:0100 B80010      MOV AX,1000
24C3:0103 8ED8        MOV DS,AX
24C3:0105 C606000405 MOV BYTE PTR [0400],05
24C3:010A C3          RET
-
```

12.3 Assembler avec debug

Nous avons vu l'intérêt d'écrire un programme en langage symbolique avant de l'écrire en langage machine. La traduction d'un programme en langage symbolique en langage machine s'appelle l'**assemblage**. Nous avons vu comment assembler à la main. Cela devient très vite pénible. On s'est très vite aperçcu qu'on peut en confier la tâche à l'ordinateur lui-même. L'utilitaire `debug` permet cet assemblage, tout au moins dans certains cas.

Introduction du programme en langage symbolique.- Pour écrire un programme en langage symbolique avec `debug`, il suffit d'utiliser la commande 'A' (pour *Assemble*) de `debug` : on fait suivre 'A' de l'adresse de départ (en hexadécimal) ou, mieux, de rien (l'adresse constituée du segment de code choisi par `debug` et du décalage `100h` est prise par défaut) ; on répond à chaque adresse par une instruction en langage symbolique (celui compris par `debug`, évidemment).

Exemple.- Reprenons notre exemple :

```
C:\COM\>debug
-a
249c:0100 mov ax,1000
249c:0103 mov ds,ax
249c:0105 mov byte ptr [0400],05
249c:010A ret
249c:010B
-
```

Remarque.- Si vous écrivez une instruction (syntaxiquement) incorrecte que `debug` ne peut pas assembler, il indique une erreur.

12.4 Historique

12.4.1 Les langages symboliques

Pour VON NEUMANN, le langage de programmation le plus parfait et le plus universel est le langage machine. Malheureusement le cerveau humain – à part peut-être celui de VON NEUMANN – perd rapidement pied dans cette suite de 0 et de 1. De plus, si l'on veut qu'un jour un non-informaticien puisse commander le moindre travail à la machine, il faut bien qu'il puisse le faire sans pour antant devenir un spécialiste de la structure de l'ordinateur.

Le premier rédacteur d'un langage de programmation plus convivial est Alan TURING, qui veut faciliter l'usage du Manschester MARK 1. Ce premier langage contient cinquante instructions, qui sont automatiquement transcrites en langage machine par l'ordinateur lui-même. La même idée est reprise, par Grace HOPPER, sur l'UNIVAC 1, premier ordinateur civil, qui dispose d'un *short code* ; elle développe ce que la firme appelle la « programmation automatique », un programme interne qui transforme les instructions de l'utilisateur en instructions machine codées en binaire. À partir de ce moment, il est admis qu'un langage de programmation doit servir à écrire des programmes d'une façon qui permette des économies du point de vue du temps d'utilisation de la machine.

Nous avons déjà vu le langage symbolique utilisé par Maurice WILKES.

Sources.- Les tout premiers langages de programmation sont traités dans [KP-77] :

> *Cet article passe en revue l'évolution des langages de programmation de « haut-niveau » durant la première décade d'activité de la programmation des calculateurs.*

Nous discutons des contributions de Zuse en 1945 (le « Plankalkül »), Goldstine et von Neumann en 1946 (« Flow Diagrams »), Curry en 1948 (« Composition »), Mauchly et al. *en 1949 (« Short Code »), Burks en 1950 (« Intermediate PL »), Rutishauser en 1951 (« Klammerausdrücke »), Böhm en 1951 (« Formules »), Glennie en 1952 (« Autocode »), Hopper* et al. *en 1953 (« A-2 »), Laning et Zierler en 1953 (« Algebraic Interpreter »), Backus* et al. *en 1954–1957 (« Fortran »), Brooker en 1954 (« Mark I Autocode »), Kamynin et Lîubimskii en 1954 (« ШП-2 »), Ershov en 1955 (« ШП »), Grems et Porter en 1955 (« BACAIC »), Elsworth* et al. *en 1955 (« Kompiler 2 »), Blum en 1956 (« ADES »), Perlis* et al. *en 1956 (« IT »), Katz* et al. *en 1956–1958 (« MATH-MATIC »), Bauer et Samelson en 1956–1958 (U.S. Patent 3 047 228). Les caractéristiques principales de chaque contribution sont illustrées et discutées. Pour pouvoir les comparer, un algorithme particulier fixé a été codé (autant que faire se peut) dans chacun de ces langages. Cette recherche est fondée sur des sources non publiées et les auteurs espèrent qu'ils ont été capables de compiler une image aussi complète que possible des premiers développements dans ce thème.*

12.4.2 CP/M et MS-DOS

CP/M.- Comme nous l'avons déjà dit, les premiers micro-ordinateurs n'étaient pas munis de système d'exploitation. Puis des systèmes d'exploitation propriétaires apparurent : un par type d'ordinateur. Lorsque Gary KILDALL introduit CP/M, celui-ci devient très rapidement le système d'exploitation par excellence des micro-ordinateurs munis d'un microprocesseur 8 bits.

Gary Arlen KILDALL (1942–1994) est né et grandit à Seattle, dans l'état de Washington, où sa famille tenait une école de marins. Son père, Joseph KILDALL, d'origine norvégienne, était capitaine et sa mère, Emma, était à demi-suédoise. Gary va à l'université de Washington (UW), en espérant devenir enseignant de mathématiques, mais il est de plus en plus intéressé par la technologie des ordinateurs. Une fois diplômé, il remplit ses obligations militaires en enseignant à la *Naval Postgraduate School* (NPS) de Monterey, en Californie. Étant à une heure de voiture de la Silicon Valley, KILDALL entend parler du premier microprocesseur commerciallement disponible, le 4004 d'*Intel*. Il en achète un et commence à écrire des programmes expérimentaux pour celui-ci. Pour en apprendre plus sur les microprocesseurs, il travaille chez *Intel* comme consultant à sa libération.

Il retourne brièvement à UW où il termine son doctorate en informatique en 1972 et revient enseigner à NPS. Il publie un article qui introduit l'analyse des chemins de données utilisé de nos jours pour l'optimisation des compilateurs, tout en continuant à effectuer des expériences avec les micro-ordinateurs et la technologie émergeant des disquettes. En 1973, il dévelope le premier langage de haut niveau pour les microprocesseurs, appelé PL/M (*Programming Language for Microprocessors*). Il crée CP/M la même année pour permettre au 8080 de contrôler un lecteur de disquettes. Il effectue une démonstration de CP/M pour *Intel*, qui est peu intéressé mais choisit de diffuser PL/M.

KILDALL et sa femme Dorothy fondent une société, appelée d'abord *Intergalactic Digital Research* (plus tard renommée en *Digital Research, Inc*), pour vendre CP/M en faisant de la publicité dans les revues. *Digital Research* vend une licence de CP/M pour l'*IMSAI 8080*, une clone populaire de l'*Altair 8800*. Puis plusieurs fabricants achètent une licence CP/M, ce qui fait qu'il devient un standard de fait, qui doit s'adapter à un nombre croissant de matériels différents. En réponse, KILDALL invente le concept de BIOS, un ensemble de programmes simples stocké dans la partie matérielle du micro-ordinateur, ce qui permet à CP/M de s'exécuter sur différents systèmes sans modifications.

MS-DOS.- Sur la suggestion de Bill GATES, IBM contacte *Digital Research* en 1980 pour négocier

l'achat de la version à venir de CP/M, appelée CP/M-86, pour l'IBM PC. Gary confie la négotiation à sa femme, Dorothy, comme il a l'habitude de le faire, alors que lui et son collègue Tom Rolander utilisent l'avion privé de Gary pour livrer des logiciels au fabricant Bill Godbout. Avant d'expliquer le but de leur visite, les représentants d'IBM insistent pour que Dorothy signe un agrément de non-diffusion. Sur l'avis de l'avocat de DRI, Gerry Davis, Dorothy refuse de le signer sans l'approbation de Gary. Gary revient dans l'après-midi et essaie de revenir sur la discussion avec IBM, mais les représentants tiennent bon.

IBM s'adresse alors à Microsoft, qui est déjà d'accord pour fournir un interpréteur BASIC et plusieurs autres programmes pour le PC. Le représentant d'IBM, Sams, confie donc à Bill Gates la tâche de trouver un système d'exploitation utilisable. Quelques semaines plus tard, il propose d'utiliser un clone de CP/M, le 86-DOS de *Seattle Computer Products* (SCP). Paul Allen négocie une licence avec SCP, adapte 86-DOS au matériel de l'IBM PC et le présente comme PC-DOS.

On connaît le succès que connaîtra ce système d'exploitaion jusqu'en 1995, date à laquelle il sera remplacé par *Windows*. Disons un mot, pour finir, de ce qui arriva à *Digital Research*. Kildall obtient une copie de PC-DOS, l'examine et en conclut qu'il repose sur CP/M. Il demande à Gerry Davis de quels moyens légaux il dispose pour faire valoir ses droits, mais celui-ci lui explique qu'il n'est pas clair si la loi sur la propriété intellectuelle s'applique aux logiciels. Aussi Kildall se contente-t-il seulement de menacer IBM d'une action légale, qui lui répond en proposant d'offrir CP/M-86 en option pour le PC. Kildall accepte, pensant que le nouveau système d'exploitation d'IBM n'aura pas de succès.

12.4.3 Debug

En 1980, Tim Paterson, né en 1956, commence à travailler sur un système d'exploitation 16 bits (pour le 8086) pour la carte *S-100 Bus* qu'il a conçue pour SCP (*Seattle Computer Products*) l'année précédente. Pour faciliter la programmation de QDOS (plus tard appelé 86-DOS), Tim crée un débogueur tenant sur une puce ROM ; le code de cette version sur ROM est placée dans le domaine public. Plus tard, Tim en adapte le code afin qu'il s'exécute en tant que programme .COM sous QDOS. Il lui ajoute alors la capacité de désassembler du code machine 8086. Entretemps, *Microsoft* a acheté le droit de vendre le QDOS de Tim à IBM pour leur project 'secret' de PC. Tim est alors embauché par *Microsoft*. Lorsqu'il met au point DOS 1.00 en 1981, son utilitaire DEBUG.COM y est inclus. Peu y a été ajouté depuis, l'exception majeure étant la commande d'assemblage à partir de DOS 2.0 [Sed-04].

12.5 Bibliographie

[KP-77] Knuth, Donald E., Trabb Pardo, Luis, *The Early Development of Programming Languages*, in Belzer, J., Holzman, A. G., and Kent, A. (eds), **Encyclopedia of Computer Science and Technology**, vol. 6, Dekker, New-York, 1977, pp. 419–493. Reprinted in [Met-80], pp. 197–273. Traduction française dans Donald E. Knuth , **Éléments pour une histoire de l'informatique**, articles choisis et traduits par Patrick Cégielski, Lecture Notes 190, CSLI Publications, Stanford et Société Mathématique de France, 2011, xvi + 371 p.

[Met-80] Metropolis, N. and Howlett, J. and Rota, Gian-Carlo, eds, **A History of Computing in the Twentieth Century**, Academic Press, 1980.

[Sed-04] Sedory, Daniel B., **A Guide to Debug**, 2004. Téléchargeable sur :
`http://thestarman.pcministry.com/asm/debug/debug.htm`