Chapitre 11

Les chaînes de caractères

La plupart des ordinateurs modernes ne sont plus orientés calculs mais orientés mots. Il est donc important de savoir comment traiter les *chaînes de caractères*.

On peut toujours coder les mots et simuler les opérations sur ceux-ci. Mais, devant l'importance de la manipulation des mots par ordinateurs de nos jours, les concepteurs des microprocesseurs s'adaptent et créent des primitives pour celles-ci.

11.1 Définition des chaînes de caractères

Nous avons déjà vu ce qu'est, mathématiquement, un *mot*: il s'agit d'une suite finie sur un *alphabet*, ensemble fini non vide quelconque. En informatique on préfère parler de *chaînes de caractères* (*string* en anglais) pour insister sur le fait qu'on peut, en particulier, compter l'espace comme un caractère.

11.1.1 Représentation des caractères

Une chaîne de caractères étant une suite de caractères, il faut d'abord s'interroger sur la façon de repésenter les caractères.

Le microprocesseur ne connaît pas la notion de caractères. Il ne connaît que les **codes** des caractères, qui peuvent être considérés comme des entiers.

Combien faut-il d'entiers pour représenter les caractères ? Il y a vingt-six lettres en français, mais sans distinguer les minuscules des majuscules, et sans tenir compte des signes diacritiques : 'é' n'est pas la même chose que 'e' ou 'è'. DE plus on a besoin des chiffres, des signes de ponctuation et de quelques signes spéciaux.

Historiquement sont d'abord apparus des codes dont chaque caractère tenait sur sept bits, ce qui permet 128 caractères différents. Il existe deux codes répandus : le code EBDIC d'IBM et le code ASCII, ce dernier datant de 1967. Est ensuite apparu un code ASCII étendu sur 8 bits (un octet), ce qui permet 256 caractères. En fait on avait besoin de plusieurs codes ASCII étendus, suivant le pays dans lequel on se trouvait. ON utilise de nos jours, mais bien après la conception du microprocesseur `i8086`, un code sur deux octets, dit **unicode**.

Pour le `i8086`, un caractère est codé sur un octet. Qu'importe le code utilisé. C'est à l'utilisateur (dans la pratique au concepteur du système d'exploitation) de déterminer celui-ci.

11.1.2 Représentation des chaînes de caractères

Définition.- *Une* **chaîne de caractères** *est une suite d'octets d'adresses consécutives.*

Une chaîne de caractères est entièrement déterminée par son *adresse* et sa *longueur*.

Adresse d'une chaîne de caractères.- Soit la suite d'octets $w = o_1 o_2 ... o_n$. Si l'adresse, en mémoire centrale, du premier octet o_1 est p alors celle du second octet o_2 est $p + 1$, celle du troisième octet o_3 est $p + 2$, ... , celle du n-ième octet o_n est $p + n - 1$. Il suffit donc de connaître l'adresse du premier octet, p, qui est appelée l'**adresse de la chaîne de caractères**.

On remarquera que les octets sont numérotés naturellement de gauche à droite, contrairement aux bits d'un octet ou d'un mot, qui sont eux numérotés de droite à gauche.

Longueur d'une chaîne de caractères.- La détermination d'une chaîne de caractères en mémoire centrale se fait par son adresse mais aussi par sa **longueur** n, ce qui permet de déterminer la fin de celle-ci.

Exemple.- La chaîne de caractères “Bonjour” de longueur 7 à l'adresse 1024 sera représentée de la façon suivante en mémoire :

```
1024 'B'
1025 'o'
1026 'n'
1027 'j'
1028 'o'
1029 'u'
1030 'r'
```

11.1.3 Jeu de caractères utilisé

Les caractères sont des octets, ils sont donc limités à 256. Mais quels sont ces caractères ? Le microprocesseur ne se prononce pas : ils sont repérés par des numéros de 0 à 255, il appartiendra ensuite au concepteur du système d'exploitation de les interpréter, en particulier grâce à la *ROM caractère*.

Dans le cas de l'IBM-PC, il s'agit de l'alphabet ASCII étendu de deux façons : les caractères 0 à 31 sont remplacés par des caractères (dits **semi-graphiques**) propres à IBM et il y a le choix d'un ASCII national pour les caractères de 128 à 255.

11.2 Adressage indirect par registre et tableaux

Vous avez vu la notion de *tableau* dans votre cours d'initiation à la programmation et les services qu'elle rend au programmeur. Cette notion, très utile dans les langages évolués, n'existe pas en tant que telle dans le langage machine. Voyons ici la notion d'*adressage indirect*, plus particulièrement l'*adressage indirect par registre*, et comment celui-ci permet de mettre en place facilement les tableaux d'octets, autrement dit les chaînes de caractères.

11.2.1 Notion

Notion d'adressage indirect.- Jusqu'à maintenant nous avons manipulé des constantes (*adressage immédiat*), des valeurs se trouvant dans un registre (*adressage registre*) et des valeurs se trouvant à un emplacement donné de la mémoire vive (*adressage direct*).

Une autre façon de faire est de désigner un emplacement qui contient non pas la valeur désirée mais l'adresse (en mémoire vive) contenant la valeur désirée. On parle alors d'**adressage indirect**.

Syntaxe de l'adressage indirect par registre.- Comme nous l'avons déjà vu, pour désigner une valeur constante on écrit cette valeur (en hexadécimal), pour une valeur se trouvant dans un registre on écrit le nom du registre, pour une valeur se trouvant à un emplacement de la mémoire centrale on indique l'adresse (une constante hexadécimale) de cet emplacement entre crochets.

Pour une valeur se trouvant à l'adresse (de mémoire vive) désignée par le contenu d'un registre, on indique le nom du registre entre crochets, par exemple

`[bx]`. On parle alors d'**adressage indirect par registre**.

Registres concernés.- On ne peut pas utiliser n'importe quel registre pour l'adressage indirect par registre. Les seuls registres utilisables pour l'adressage indirect par registre pour le `i8086` sont les registres `BX`, `BP`, `DI` et `SI`.

Il ne s'agit pas d'une restriction logique mais d'une question d'économie de câblage du microprocesseur : on gagne sur le nombre de transistors en ne donnant pas toutes les fonctionnalités à chaque registre mais en les spécialisant quelque peu.

Indication de la taille de l'adressage indirect.- L'adressage indirect peut se faire pour des octets mais aussi pour des mots (deux octets) ou des mots doubles (quatre octets).
Lorsque cela n'est pas clair on utilise les **préfixes** suivants dans le cas du langage symbolique (il s'agit de codes d'opération différents pour le langage machine) :

- `byte ptr` pour un octet ;
- `word ptr` pour un mot ;
- `dword ptr` pour un mot double.

Par exemple :

```
mov al,[bx]
```

est visiblement uns instruction de copie d'un octet mais :

```
mov [bx],10
```

est ambigu. On doit donc préciser, par exemple :

```
mov byte ptr [bx],10
```

Exemple.- Avec le mot "Bonjour" écrit à l'adresse indiquée ci-dessus, la portion de programme suivant :

```
mov bx,1024
mov cl,[bx]
cmp byte ptr [bx],0
```

placera 'B' dans le registre `CL` et le pointeur `ZF` ne sera pas nul puisque `[bx]` ne contient pas le caractère nul lors de la dernière ligne.

Adresse physique.- Lors de l'adressage indirect on indique l'adresse logique et, en fait, que le décalage de celle-ci. Comme d'habitude, si ceci n'est pas suffisant, l'adresse logique complète, et donc l'adresse physique, est obtenue en utilisant l'adresse de segment indiquée dans le registre `DS` pour les registres `BX` et `SI`, dans le registre `ES` pour `DI`.

11.2.2 Un exemple

Écrivons un programme qui commence par placer "Bonjour" à partir de l'adresse `1024h`, suivi de la valeur sentinelle '0', puis qui compte le nombre de caractères de ce mot, c'est-à-dire le nombre de caractères depuis le début jusqu'à '0' non inclus, et qui place la longueur ainsi obtenue dans le registre `DX` :

```
C:>debug
```

```
-a
249C:0100 mov byte ptr [1024],42
249C:0105 mov byte ptr [1025],6F
249C:010A mov byte ptr [1026],6E
249C:010F mov byte ptr [1027],6A
249C:0114 mov byte ptr [1028],6F
249C:0119 mov byte ptr [1029],75
249C:011E mov byte ptr [102A],72
249C:0123 mov byte ptr [102B],0
249C:0128 mov dx,0
249C:012B mov bx,1024
249C:012E mov al,[bx]
249C:0130 cmp al,0
249C:0132 je 138
249C:0134 inc bx
249C:0135 inc dx
249C:0136 jmp 12E
249C:0138 int 3
249C:0139
-g
AX=0000 BX=102B CX=0000 DX=0007 SP=FFEE BP=0000 SI=0000 DI=0000
DS=249C ES=249C SS=249C CS=249C IP=0113 NV UP EI PL ZR NA PE NC
249C:0138 CC INT 3
-q
```

Ce programme nous donne bien la longueur 7.

11.3 Primitives de manipulation des mots

11.3.1 Copie

L'une des manipulations les plus courantes concernant les chaînes de caractères est la copie d'une telle chaîne d'un endroit à un autre de la mémoire vive. On peut évidemment réaliser une telle copie avec les instructions que nous connaissons déjà. On peut également utiliser les deux nouvelles instructions `movs` et `rep` pour cela.

Exercice.- Écrire un programme permettant de copier la chaîne de caractères d'adresse contenue dans `ax` et de longueur contenue dans `cx` à l'adresse contenue dans `bx`.

11.3.1.1 Copie d'un élément

On sait copier un élément (octet ou mot) d'un emplacement à un autre. Il existe cependant une instruction spécialisée pour ce faire, qui est surtout utile en conjonction avec une instruction de répétition spécialisée que nous verrons ensuite.

Syntaxe.- L'instruction `movs` (pour *MOVe a String*) se décline sous les deux formes :

```
movsb
```

ou :

```
movsw
```

pour copier un octet ou un mot.

Paramètres.- Bien entendu il manque des informations dans l'instructions ci-dessus, celles-ci étant apportées par les registres `cx`, `ds`, `si`, `es` et `di`.

Le couple de registres `ds:si` (avec `si` pour *Source Index*) doit contenir l'adresse de l'élément à copier. Le couple de registres `es:di` (avec `di` pour *Destination Index*) doit contenir l'adresse à laquelle il faut copier l'élément.

Exemple.- Écrivons un programme `debug` qui place 'A' à l'emplacement mémoire `1024h` et qui copie cette valeur à l'emplacement `1025h` :

```
C:\>debug
-d1024
15DC:1020              00 00 00 00-00 00 00 00 00 00 00 00       ............
15DC:1030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15DC:1040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15DC:1050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15DC:1060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15DC:1070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15DC:1080  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15DC:1090  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15DC:10A0  00 00 00 00                                       ....
-a
15DC:0100 mov byte ptr [1024],41
15DC:0105 mov ax,1024
15DC:0108 mov si,ax
15DC:010A mov ax,1025
15DC:010D mov di,ax
15DC:010F movsb
15DC:0110 int3
15DC:0111
-g

AX=1025  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=1025  DI=1026
DS=15DC  ES=15DC  SS=15DC  CS=15DC  IP=0110   NV UP EI PL NZ NA PO NC
15DC:0110 CC            INT     3
-d1024
15D9:1020              41 41 00 00-00 00 00 00 00 00 00 00       AA..........
15D9:1030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15D9:1040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15D9:1050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15D9:1060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15D9:1070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15D9:1080  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15D9:1090  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15D9:10A0  00 00 00 00                                       ....
-q
```

Nous vérifions que ces valeurs 'A' n'apparaissaient pas à ces emplacements avant et qu'elles apparaissent bien après, à l'aide de la commande `D`.

11.3.1.2 Copie répétée

Syntaxe.- L'instruction `rep` (pour *REPete*) permet de répéter un certain nombre de fois l'instruction qui suit, ce nombre de fois étant la valeur du registre `cx` (pour l'anglais *Count eXtended register*).

Elle peut être employée en conjonction avec l'une des deux instructions définies précédemment :

```
rep movsb
```

ou :

```
rep movsw
```

Façon de copier.- Imaginons que nous voulons copier le contenu de la mémoire vive `100h-200h` vers `400h-500h`. Il semble naturel de copier l'octet d'adresse

100h à l'adresse 400h, puis celui de 101h à 401h et ainsi de suite. Cependant, dans certaines applications, nous aurons à copier le contenu de 100h à 500h, celui de 101h à 4FFh, et ainsi de suite.

On utilise pour cela l'indicateur DF (pour *Direction Flag*) pour indiquer le sens de la copie : s'il vaut 0, on copie de façon naturelle dans l'ordre croissant ; s'il vaut 1, on copie dans l'ordre décroissant.

Il existe deux instructions :

```
CLD
```

et :

```
STD
```

pour spécifier ce sens. L'instruction CLD (pour *CLear Direction flag*) met l'indicateur de direction à zéro. L'instruction STD (pour *SeT Direction flag*) met l'indicateur de direction à 1.

Exemple.- Écrivons un programme, en langage symbolique de debug, qui permet d'écrire '**bonjour**' à l'adresse 1024h, puis de copier le contenu de cette chaîne de caractères à l'adresse 102Bh :

```
C:\>debug
-a
15BC:0100 mov byte ptr [1024],42
15BC:0105 mov byte ptr [1025],6F
15BC:010A mov byte ptr [1026],6E
15BC:010F mov byte ptr [1027],6A
15BC:0114 mov byte ptr [1028],6F
15BC:0119 mov byte ptr [1029],75
15BC:011E mov byte ptr [102A],72
15BC:0123 mov ax,1024
15BC:0126 mov si, ax
15BC:0128 mov ax, 102B
15BC:012B mov di, ax
15BC:012D mov cx, 6
15BC:0130 cld
15BC:0131 rep movsb
15BC:0133 int3
15BC:0134
-g

AX=102B  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=102A  DI=1031
DS=15BC  ES=15BC  SS=15BC  CS=15BC  IP=0133   NV UP EI PL NZ NA PO NC
15BC:0133 CC            INT     3
-d 1024
15BC:1020              42 6F 6E 6A-6F 75 72 42 6F 6E 6A 6F       BonjourBonjo
15BC:1030  75 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   u...............
15BC:1040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BC:1050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BC:1060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BC:1070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BC:1080  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BC:1090  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BC:10A0  00 00 00 00                                       ....
-q
```

On vérifie avec la commande D que la copie a bien été effectuée (sans le 'r' final puisque la longueur donnée est 6 de façon erronnée au lieu de 7).

On s'est servi implicitement du fait, qu'avec debug, les registres ds et es ont la même valeur.

11.3.2 Remplissage

On a souvent à remplir un partie de la mémoire avec le même octet, par exemple avec des '0' ou des espaces (dans le cas de l'effacement de l'écran, par

exemple). Les concepteurs du `i8086` ont donc implémenté une telle primitive, sous le nom de **stos** pour *STOre a String*.

11.3.2.1 Stockage d'un élément

Syntaxe.- Les deux instructions :

```
stosb
```

et :

```
stosw
```

permettent de stocker, respectivement, l'octet contenu dans le registre `al` ou le mot contenu dans le registre `ax`, à l'adresse `es:di`.

Exemple.- Écrivons un programme, en langage symbolique `debug`, permettant de placer 'A' à l'emplacement mémoire de décalage `1024h` (du segment choisi par MS-DOS pour `debug`) :

```
C:\>debug
-d 1024
15BA:1020              00 00 00 00-00 00 00 00 00 00 00 00       ............
15BA:1030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1080  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1090  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:10A0  00 00 00 00                                       ....
-a
15BA:0100 mov ax, 1024
15BA:0103 mov di, ax
15BA:0105 mov al, 41
15BA:0107 stosb
15BA:0108 int3
15BA:0109
-g

AX=1041  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=1025
DS=15BA  ES=15BA  SS=15BA  CS=15BA  IP=0108   NV UP EI PL NZ NA PO NC
15BA:0108 CC            INT     3
-d 1024
15BA:1020              41 00 00 00-00 00 00 00 00 00 00 00       A...........
15BA:1030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1080  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1090  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:10A0  00 00 00 00                                       ....
-q
```

11.3.2.2 Remplissage d'une zone

Syntaxe.- Les deux instructions :

```
rep stosb
```

et :

```
rep stosw
```

permettent de remplir la zone d'adresse `es:di` et de longueur celle contenue dans le registre `cs` avec l'octet se trouvant dans le registre `al` ou le mot dans le registre `ax`.

L'indicateur de direction indique si on doit décrémenter ou incrémenter le registre `di`.

Exemple.- Écrivons un programme, en langage symbolique `debug`, permettant de remplir ll'emplacement mémoire de décalage `1024h` (du segment choisi par MS-DOS pour `debug`) et de longueur `56h` avec des 'A' :

```
C:\>debug
-d 1024
15BD:1020              00 00 00 00-00 00 00 00 00 00 00 00       ............
15BD:1030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BD:1040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BD:1050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BD:1060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BD:1070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BD:1080  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BD:1090  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BD:10A0  00 00 00 00                                       ....
-a
15BD:0100 mov ax, 1024
15BD:0103 mov di, ax
15BD:0105 mov al, 41
15BD:0107 mov cx, 56
15BD:010A cld
15BD:010B rep stosb
15BD:010D int3
15BD:010E
-g

AX=1041  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=107A
DS=15BD  ES=15BD  SS=15BD  CS=15BD  IP=010D   NV UP EI PL NZ NA PO NC
15BD:010D CC            INT     3
-d 1024
15BD:1020              41 41 41 41-41 41 41 41 41 41 41 41       AAAAAAAAAAAA
15BD:1030  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41   AAAAAAAAAAAAAAAA
15BD:1040  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41   AAAAAAAAAAAAAAAA
15BD:1050  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41   AAAAAAAAAAAAAAAA
15BD:1060  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41   AAAAAAAAAAAAAAAA
15BD:1070  41 41 41 41 41 41 41 41-41 41 00 00 00 00 00 00   AAAAAAAAAA......
15BD:1080  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BD:1090  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BD:10A0  00 00 00 00                                       ....
-q
```

11.3.3 Chargement

Introduction.- Nous avons vu comment copier le contenu de l'accumulateur vers un emplacement mémoire, et même un groupe d'emplacements mémoire. On peut vouloir le contraire : copier d'un emplacement mémoire vers l'accumulateur.

On peut évidemment faire ceci avec l'instruction `mov`. Une instruction nouvelle a été introduite par les concepteurs du `i8086` qui permet en plus d'incrémenter (ou décrémenter) la valeur du registre `si`. Il s'agit de **lods** pour l'anglais *LOaD a String*.

Syntaxe.- On utilise les deux formes :

```
lodsb
```

et :

```
lodsw
```

pour charger, respectivement, l'octet ou le mot d'adresse `ds:si` dans le registre `al` ou `ax`, suivant le cas. De plus le registre `si` est incrémenté ou décrémenté suivant la valeur de l'indicateur de direction.

Contexte.- On n'utilise pas ces instructions avec `rep`, puisqu'on surchargerait l'accumulateur. On les utilise, en liaison avec `stos` pour une copie avec changement durant le transfert. Dans ce cas le nombre de répétitions est contrôlé par l'instruction :

`loop` *adresse*

où *adresse* indique l'adresse à laquelle renvoie la boucle et où le nombre de répétitions est indiqué par le contenu du registre `cx`.

Exemple.- Dans le cas du code ASCII, les minuscules 'a', ... , 'z' ont les mêmes numéros que les majuscules 'A', ... , 'Z' correspondantes avec `20h` de plus.

Plaçons une chaîne de caractères écrite exclusivement en majuscule, formée que de 'A' pour simplifier, à l'adresse de décalage `1024h` et de longueur `56h` puis transférons-la à l'adresse de décalage `1080h` en transformant les majuscules en minuscules :

```
C:\>debug
-d 1024
15BA:1020              00 00 00 00-00 00 00 00 00 00 00 00       ............
15BA:1030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1080  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1090  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:10A0  00 00 00 00                                       ....
-a
15BA:0100 mov ax, 1024
15BA:0103 mov di, ax
15BA:0105 mov al, 41
15BA:0107 mov cx, 56
15BA:010A cld
15BA:010B rep stosb
15BA:010D mov ax, 1024
15BA:0110 mov si, 1024
15BA:0113 mov ax, 1080
15BA:0116 mov di, ax
15BA:0118 mov cx,56
15BA:011B cld
15BA:011C lodsb
15BA:011D add al, 20
15BA:011F stosb
15BA:0120 loop 011C
15BA:0122 int3
15BA:0123
-g

AX=1061  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=107A  DI=10D6
DS=15BA  ES=15BA  SS=15BA  CS=15BA  IP=0122   NV UP EI PL NZ NA PO NC
15BA:0122 CC            INT     3
-d 1024
15BA:1020              41 41 41 41-41 41 41 41 41 41 41 41       AAAAAAAAAAAA
15BA:1030  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41   AAAAAAAAAAAAAAAA
15BA:1040  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41   AAAAAAAAAAAAAAAA
15BA:1050  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41   AAAAAAAAAAAAAAAA
15BA:1060  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41   AAAAAAAAAAAAAAAA
15BA:1070  41 41 41 41 41 41 41 41-41 41 00 00 00 00 00 00   AAAAAAAAAA......
15BA:1080  61 61 61 61 61 61 61 61-61 61 61 61 61 61 61 61   aaaaaaaaaaaaaaaa
15BA:1090  61 61 61 61 61 61 61 61-61 61 61 61 61 61 61 61   aaaaaaaaaaaaaaaa
15BA:10A0  61 61 61 61                                       aaaa
-q
```

On remarquera que `debug` accepte l'instruction :

```
mov si, 1024
```

là où nous voulions écrire :

```
mov si, ax
```

11.3.4 Comparaison

Introduction.- Nous avons souvent à *comparer* deux chaînes de caractères, c'est-à-dire à déterminer si elles sont égales ou si la première vient avant la seconde dans l'ordre lexicographique. Là encore, les concepteurs du *i8086* ont implémenté cette opération comme primitive avec l'instruction **CMPS** pour l'anglais *CoMPare Strings*.

Syntaxe.- On utilise les deux formes :

```
cmpsb
```

et :

```
cmpsw
```

pour la comparaison.

Sémantique.- Cette instruction soustrait, respectivement, l'octet ou le mot se trouvant à l'emplacement `es:di` à l'octet ou au mot se trouvant à l'emplacement `cs:si` et positionne les indicateurs en conséquence. Le contenu de la mémoire n'est pas affecté par cette instruction. Les registres `si` et `di` sont incrémentés ou décrémentés suivant la valeur de l'indicateur de direction.

Répétition conditionnelle.- Pour déterminer si deux chaînes de caractères sont égales, on va les comparer caractère par caractère. L'instruction de répétition `rep` n'est pas très adaptée à ce cas car on va décrire les deux chaînes en entier alors qu'on peut s'arrêter dès qu'on trouve deux emplacements qui divergent.

L'instruction `repz`*, ou* `repe`*, utilisée en conjonction avec l'une des cinq primitives de manipulation des chaînes de caractères, permet la répétition tant que* `cx` *est non nul et que l'indicateur* `ZF` *est égal à 1.*

Ainsi

```
repz cmpsb
```

s'arrêtera pour l'une des deux raisons suivantes. Soit on a parcouru toute la chaîne de caractères et les deux chaînes sont égales. Soit l'indicateur `ZF` a été positionné à zéro, indiquant ainsi que les deux chînes de caractères diffèrent en un certain emplacement.

Exemple.- Écrivons un programme `debug` qui place la chaîne de caractères "`Bon`" au décalage `1020h`, la chaîne de caractères "`BOn`" au décalage `1030h` puis qui compare les deux chaînes de caractères placées aux décalages `1020h` et `1030h` sur trois caractères, plaçant 0 dans le registre `al` si ces chaînes différente et 1 sinon :

```
C:\>debug
-d 1020
15BA:1020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

```
15BA:1080  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1090  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-a
15BA:0100 mov byte ptr [1020],42
15BA:0105 mov byte ptr [1021],6F
15BA:010A mov byte ptr [1022],6E
15BA:010F mov byte ptr [1030],42
15BA:0114 mov byte ptr [1031],4F
15BA:0119 mov byte ptr [1032],6E
15BA:011E mov ax,1020
15BA:0121 mov si,ax
15BA:0123 mov ax,1030
15BA:0126 mov di,ax
15BA:0128 mov cx,3
15BA:012B cld
15BA:012C repz cmpsb
15BA:012E jnz 0134
15BA:0130 mov al,1
15BA:0132 jmp 0136
15BA:0134 mov al,0
15BA:0136 int3
15BA:0137
-g

AX=1000  BX=0000  CX=0001  DX=0000  SP=FFEE  BP=0000  SI=1022  DI=1032
DS=15BA  ES=15BA  SS=15BA  CS=15BA  IP=0136   NV UP EI PL NZ NA PO NC
15BA:0136 CC            INT     3
--d 1020
15BA:1020  42 6F 6E 00 00 00 00 00-00 00 00 00 00 00 00 00   Bon.............
15BA:1030  42 4F 6E 00 00 00 00 00-00 00 00 00 00 00 00 00   BOn.............
15BA:1040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1080  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BA:1090  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-q
```

Le contenu du registre `ax` indique bien que les deux châines de caractères sont différentes.

Remarque.- Le langage symbolique de `debug` est trop proche du langage machine, d'où une difficulté lors du codage du programme : il faut tâtonner pour mettre les bonnes adresses aux instructions `jnz` et `jmp` des lignes `012Eh` et `0132h`. Ce problème sera réoslu par l'utilisation d'un langage d'assemblage.

11.3.5 Recherche

Introduction.- On a également souvent à rechercher si une chaîne de caractères contient tel ou tel caractère. Ceci donne lieu à la dernière primitive implémentée sur le `i8086` pour les chaînes de caractères, à savoir **scas** pour l'anglais *SCAn a String*.

Syntaxe.- On utilise les deux formes :

```
scasb
```

et :

```
scasw
```

pour la comparaison.

Sémantique.- Cette instruction soustrait, respectivement, l'octet ou le mot se trouvant à l'emplacement `es:di` à l'octet ou au mot se trouvant dans le registre `AL` ou `AX` et positionne les indicateurs en conséquence. Le contenu de la

mémoire n'est pas affecté par cette instruction. Le registre `di` est incrémenté ou décrémenté suivant la valeur de l'indicateur de direction.

Autre répétition conditionnelle.- Pour déterminer si le motif apparaît dans la chaîne de caractères, on va le comparer à chaque caractère de celle-ci. L'instruction de répétition `rep` n'est pas très adaptée à ce cas. On pourrait utiliser `repz` mais on a intérêt à utiliser une instruction nouvelle.

L'instruction `repnz`*, ou* `repne`*, utilisée en conjonction avec l'une des cinq primitives de manipulation des chaînes de caractères, permet la répétition tant que* `cx` *est non nul et que l'indicateur* `ZF` *est égal à 0.*

Ainsi

```
repnz scasb
```

s'arrêtera pour l'une des deux raisons suivantes. Soit on a parcouru toute la chaîne de caractères et le motif n'est pas apparu. Soit l'indicateur `ZF` a été positionné à un, indiquant ainsi que le motif est apparu dans la chaîne de caractères à un certain emplacement.

Exemple.- Écrivons un programme `debug` qui place la chaîne de caractères "`Bon`" au décalage `1020h` puis qui recherche si le caractère '`o`' apparaît dans celle-ci. Le registre `bx` prendra la valeur 2 ou 1 suivant qu'il apparaît ou non :

```
C:\>debug
-a
15BD:0100 mov byte ptr [1020],42
15BD:0105 mov byte ptr [1021],6F
15BD:010A mov byte ptr [1022],6E
15BD:010F mov ax,1020
15BD:0112 mov di,ax
15BD:0114 mov cx,3
15BD:0117 mov al,6F
15BD:0119 cld
15BD:011A repnz scasb
15BD:011C jnz 0123
15BD:011E mov bx,2
15BD:0121 jmp 0126
15BD:0123 mov bx,1
15BD:0126 int3
15BD:0127
-g

AX=106F  BX=0002  CX=0001  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=1022
DS=15BD  ES=15BD  SS=15BD  CS=15BD  IP=0126   NV UP EI NG NZ AC PO NC
15BD:0126 CC            INT     3
-d 1020
15BD:1020  42 6F 6E 00 00 00 00 00-00 00 00 00 00 00 00 00   Bon.............
15BD:1030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BD:1040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BD:1050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BD:1060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BD:1070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BD:1080  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
15BD:1090  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-q
```

11.3.6 Choix de la direction

Introduction.- Dans tous les exemples ci-dessus, nous avons choisi la direction croissante. Nous pouvons changer `CLD` par `STD`, et changer les valeurs de `SI` et `DI` correspondantes, nos programmes effectueront la même chose. Considérons un exemple pour lequel le choix de la direction ne peut pas être arbitraire.

Exemple.- Considérons le problème suivant : déplacer la chaîne de caractère de l'emplacement `100h-120h` à l'emplacement `102h-122h`. À cause de l'empiètement, on ne peut pas effectuer le déplacement dans l'ordre croissant : les octets `100h` et `101h` pourraient être copiés dans les octets `102h` et `103h` ; mais au moment de déplacer l'octet `102h`, la valeur originelle de celui-ci aurait disparue.

On doit donc nécessairement copier dans l'ordre inverse : d'abord l'octet de l'emplacement `120h` en `122h`.

Exercice.- Écrire un programme complet correspondant au problème ci-dessus.