Chương 4

GIỚI THIỆU HỢP NGỮ CHO IBM PC.

Tổng quan.

Chương này bao gồm các bước cần thiết để tạo ra, hợp dịch và thực hiện một chương trình Hợp ngữ. Đến cuối chương, bạn đã có thể viết được các chương trình đơn giản nhưng khá thú vị, thực hiện một số công việc hữu ích và chạy chúng trên máy tính.

Cũng như với bất cứ ngôn ngữ nào khác, bước đầu tiên ta phải học cú pháp. Điều này đối với Hợp ngữ tương đối đơn giản. Tiếp theo, chúng tôi sẽ chỉ ra cách khai báo biến và giới thiệu các lệnh số học và dịch chuyển số liệu cơ bản. Cuối cùng chúng tôi sẽ trình bày cách tổ chức chương trình. Bạn sẽ thấy các chương trình Hợp ngữ bao gồm mã lệnh, số liệu và ngăn xếp giống như chương trình mã máy.

Bởi vì các lệnh của Hợp ngữ quá cơ bản nên thực hiện các thao tác vào/ra trong Hợp ngữ khó khăn hơn nhiều so với các ngôn ngữ bậc cao. Chúng tôi sử dụng các hàm của DOS cho các thao tác vào/ra vì chúng khá dễ dùng và đủ nhanh cho hầu hết các yêu cầu ứng dụng.

Một chương trình Hợp ngữ trước khi có thể thực hiện phải được chuyển sang dạng mã máy. Mục 4.10 sẽ giải thích các bước. Nó minh hoạ vài kỹ thuật lập trình chuẩn bằng Hợp ngữ và như là mẫu cho các bài tập.

4.1 Cú pháp của Hợp ngữ.

Các chương trình Hợp ngữ được dịch ra các chỉ thị máy bằng một chương trình biên dịch vì thế chúng phải được viết ra sao cho phù hợp với các khuôn mẫu của trình biên dịch đó. Trong cuốn sách này chúng tôi sẽ sử dụng trình biên dịch MICROSOFT MACRO ASSEMBLER (MASM). Mã lệnh Hợp ngữ nói chung

không phân biệt chữ hoa hay thường nhưng chúng tôi sử dụng chữ hoa để phân biệt mã lệnh với phần còn lại của chương trình.

Các dòng lệnh.

Các chương trình bao gồm các dòng lệnh, mỗi dòng lệnh trên một dòng. Một dòng lệnh là một lệnh mà trình biên dịch sẽ dịch ra mã máy hay là một hướng dẫn biên dịch để chỉ dẫn cho trình biên dịch thực hiện một vài nhiệm vụ đặc biệt nào đó, chẳng hạn dành chỗ cho một biến nhớ hay khai báo một chương trình con. Mỗi lệnh hay hướng dẫn biên dịch thường có 4 trường:

```
Tên      Toán tử   Toán hạng              Lời bình
```

Các trường phải được phân cách nhau bằng ít nhất một ký tự trống hay TAB. Cũng không bắt buộc phải xắp xếp các trường theo cột nhưng chúng nhất định phải xuất hiện theo đúng thứ tự nêu trên.
Ví dụ một lệnh:

```
START:  MOV  CX,5          ;khởi tạo bộ đếm.
```

Trong ví dụ này, trường tên là nhãn START, toán tử là MOV, toán hạng là CX và 5, lời bình là ' khởi tạo bộ đếm'.

Ví dụ về hướng dẫn biên dịch:

```
MAIN PROC
```

MAIN là tên và toán hạng là PROC. Dẫn hướng biên dịch này khai báo một chương trình con có tên PROC.

4.1.1 Trường tên.

Trường tên được sử dụng làm nhãn lệnh, các tên thủ tục và các tên biến. Chương trình biên dịch sẽ chuyển các tên thành các địa chỉ bộ nhớ.
Các tên có thể có chiều dài từ 1 đến 31 ký tự, có thể chứa các chữ cái, chữ số và các ký tự đặc biệt (? . @ _ $ %). Không được phép chèn dấu trống vào giữa một tên. Nếu sử dụng dấu chấm (.) thì nó phải đứng đầu tiên. Các tên không được bắt đầu bằng một chữ số. Chương trình biên dịch không phân biệt chữ hoa và chữ thường trong tên.

Ví dụ các tên hợp lệ:

```
COUNTER1
```

```
@character
SUM_OF_DIGITS
$1000
DONE?
.TEST
```

Ví dụ các tên không hợp lệ:

```
TWO  WORD      chứa một khoảng trống
2abc           bắt đầu bằng một chữ số
A45.28         dấu chấm không phải là ký tự đầu tiên
YOU&ME         chứa một ký tự không hợp lệ
```

4.1.2 Trường toán tử.

Trong một lệnh, trường toán tử chứa mã lệnh dạng tượng trưng. Chương trình biên dịch sẽ chuyển mã lệnh dạng tượng trưng sang mã lệnh của ngôn ngữ máy. Tượng trưng của mã lệnh thường biểu thị chức năng của các thao tác. Ví dụ như: MOV, ADD, SUB.

Trong một hướng dẫn biên dịch, trường toán tử chứa **toán tử giả** (pseudo-op). Các toán tử giả sẽ không được dịch ra mã máy mà đơn giản chúng chỉ báo cho trình biên dịch làm một việc gì đó. Chẳng hạn toán tử giả PROC được dùng để tạo ra một thủ tục.

4.1.3 Trường toán hạng.

Đối với một chỉ thị, trường toán hạng xác định dữ liệu sẽ được các thao tác tác động lên. Một chỉ thị có thể không có, có 1 hoặc 2 toán hạng. Ví dụ:

```
NOP              không toán hạng, không làm gì cả.
INC  AX          một toán hạng, cộng 1 vào nội dung AX.
ADD  WORD1,2     hai toán hạng, cộng 2 vào từ nhớ WORD1.
```

Trong một chỉ thị hai toán hạng, toán hạng đầu tiên gọi là toán hạng đích. Nó có thể là một thanh ghi hoặc một ô nhớ, là nơi chứa kết quả (lưu ý một số chỉ thị không lưu giữ kết quả). Toán hạng thứ hai là toán hạng nguồn. Các chỉ thị thường không làm thay đổi toán hạng nguồn.

Đối với một hướng dẫn biên dịch, trường toán hạng thường chứa thêm thông tin về việc dẫn hướng.

4.1.4 Trường lời giải thích.

Người lập trình thường sử dụng trường lời giải thích của một dòng lệnh để giải thích dòng lệnh đó làm cái gì. Mở đầu trường này là một dấu chấm phẩy (;) và trình biên dịch bỏ qua mọi cái được đánh vào sau dấu chấm phẩy này. Lời giải thích có thể tuỳ ý (có hoặc không) nhưng vì Hợp ngữ là ngôn ngữ bậc thấp cho nên ta hầu như không thể hiểu được một chương trình viết bằng Hợp ngữ khi không có lời bình. Trên thực tế điền các lời giải thích vào hầu hết các dòng lệnh là một phương pháp học lập trình tốt. Nghệ thuật chú giải sẽ được phát triển cùng với quá trình thực hành.

Không nên viết những điều đã quá rõ ràng như:

```
MOV     CX,0            ;chuyển 0 vào CX.
```

Thay vào đó, ta nên sử dụng các lời giải thích để đặt các chỉ thị vào trong ngữ cảnh của chương trình:

```
MOV     CX,0            ;CX đếm số vòng lặp,khởi tạo 0
```

Cũng có thể tạo nên cả một dòng ghi chú và dùng chúng để tạo ra các dòng trống trong chương trình:

```
;
;khởi tạo các thanh ghi.
;
MOV     AX,0
MOV     BX,0
```

4.2 Dữ liệu chương trình.

Bộ vi xử lý chỉ thao tác với dữ liệu nhị phân, vì thế trình biên dịch phải chuyển đổi tất cả các dạng dữ liệu khác nhau thành các số nhị phân. Tuy nhiên trong một chương trình Hợp ngữ chúng ta có thể biểu diễn dữ liệu dưới dạng các số nhị phân, thập phân, số hex và thậm chí cả các ký tự.

Các số.

Một số nhị phân được viết như là một chuỗi các bit kết thúc bằng chữ cái 'B' hay 'b'.Ví dụ 1011b.

Một số thập phân là chuỗi các chữ số thập phân kết thúc bằng chữ cái 'D' hay 'd' (hoặc không có).

Một số hex phải bắt đầu bằng một chữ số thập phân và kết thúc bằng chữ cái 'H' hay 'h'. Ví dụ 0ABCH (với cách này, trình biên dịch có thể biết được ký hiệu 'ABCH' biểu diễn một biến có tên 'ABCH' hay số hex ABC).

Tất cả các số kể trên có thể có dấu tuỳ ý.

Sau đây là các ví dụ các số hợp lệ và không hợp lệ trong MASM:

Số	Kiểu
1100	thập phân
1100b	nhị phân
12345	thập phân
-3568D	thập phân
1,234	không hợp lệ, chứa ký tự không là chữ số
1B4DH	số hex
1B4D	số hex không hợp lệ, không kết thúc là 'h'
FFFFH	số hex không hợp lệ, không bắt đầu bằng một chữ số thập phân
0FFFFh	số hex

Các ký tự.

Các ký tự và chuỗi các ký tự phải được bao trong dấu nháy đơn hay nháy kép. Ví dụ "A", 'hello'. Các ký tự được trình biên dịch dịch ra mã ASCII của chúng. Vì vậy chương trình không phân biệt giữa 'A' và 41h (là mã ASCII của 'A').

Bảng 4.1 Các toán tử giả định nghĩa số liệu.

Toán tử giả	Biểu diễn
DB	Định nghĩa byte
DW	Định nghĩa word

DD	Định nghĩa từ kép
DQ	Định nghĩa 4 word (4 từ liên tiếp)
DT	Định nghĩa 10 byte (10 byte liên tiếp)

4.3 Các biến.

Trong Hợp ngữ các biến có vai trò giống như trong các ngôn ngữ bậc cao. Mỗi biến có một kiểu dữ liệu và được chương trình gán cho một địa chỉ bộ nhớ. Các toán tử giả định nghĩa số liệu và ý nghĩa của chúng được liệt kê trong bảng 4.1. Mỗi toán tử giả có thể được dùng để thiết lập một hay nhiều dữ liệu của kiểu đã được đưa ra.

Trong phần này chúng ta sử dụng DB và DW để định nghĩa các biến kiểu byte và các biến kiểu word. Các toán tử giả khác được dùng trong chương 18 có liên quan đến các thao tác với số có độ chính xác kép và số không nguyên.

4.3.1 Các biến kiểu byte.

Dẫn hướng định nghĩa một biến kiểu byte của trình biên dịch có dạng sau đây:

```
Tên     DB      giá_trị_khởi_tạo
```

Trong đó toán tử giả DB được hiểu là "định nghĩa byte"

Ví dụ:

```
ALPHA   DB      4
```

Với dẫn hướng này, Hợp ngữ sẽ gán tên ALPHA cho một byte nhớ và khởi tạo nó giá trị 4. Một dấu chấm hỏi (?) đặt ở vị trí của giá trị khởi tạo sẽ tạo nên một byte không được khởi tạo.Ví dụ:

```
BYTE    DB      ?
```

Giới hạn thập phân của các giá trị khởi tạo nằm trong khoảng từ -128 đến 127 với kiểu có dấu và từ 0 đến 255 với kiểu không dấu. Các khoảng này vừa đúng giá trị của một byte.

4.3.2 Các biến kiểu word.

Dẫn hướng định nghĩa một biến kiểu word của trình biên dịch có dạng sau đây:

```
Tên     DW      giá_trị_khởi_tạo
```

Toán tử giả DW có nghĩa là "định nghĩa word". Ví dụ:

```
WRD     DW      -2
```

Giống như với biến kiểu byte một dấu chấm hỏi ở vị trí giá trị khởi tạo có nghĩa là word không được khởi tạo giá trị đầu. Giới hạn thập phân của giá trị khởi tạo được xác định từ -32768 đến 32767 đối với kiểu có dấu và từ 0 đến 65535 đối với kiểu không dấu.

4.3.3 Các mảng.

Trong ngôn ngữ hợp ngữ, mảng chỉ là một chuỗi các byte nhớ hay từ nhớ. Ví dụ để định nghĩa mảng 3 byte có tên B_ARRAY với các giá trị khởi tạo là 10h, 20h, 30h chúng ta có thể viết:

```
B_ARRAY     DB   10h,20h,30h
```

Tên B_ARRAY được gán cho byte đầu tiên, B_ARRAY+1 cho byte thứ hai và B_ARRAY+2 cho byte thứ ba. Nếu như trình biên dịch gán địa chỉ offset 0200h cho B_ARRAY thì bộ nhớ sẽ như sau:

Phần tử	Địa chỉ	Nội dung
B_ARRAY	0200h	10h
B_ARRAY+1	0201h	20h
B_ARRAY+2	0202h	30h

Mảng các word có thể được định nghĩa một cách tương tự,.Ví dụ:

```
W_ARRAY         DW      1000,356,248,13
```

sẽ tạo nên một mảng có 4 phần tử với các giá trị khởi tạo là 1000,356,248,13. Từ đầu tiên được gán với tên W_ARRAY, từ tiếp theo gán với W_ARRAY+2, rồi đến W_ARRAY+4 v.v. Nếu mảng bắt đầu tại 0300h thì bộ nhớ sẽ như sau:

Phần tử	Địa chỉ	Nội dung
W_ARRAY	0300h	1000d
W_ARRAY+2	0302h	356d
W_ARRAY+4	0304h	248d
W_ARRAY+6	0306h	13d

Byte thấp và byte cao trong một word.

Đôi khi chúng ta muốn tham chiếu đến byte thấp và byte cao của biến word. Giả sử ta định nghĩa:

```
WORD1   DB      1234H
```

byte thấp của WORD1 sẽ chứa 34h còn byte cao chứa 12h. Byte thấp có địa chỉ ký hiệu là WORD1, còn byte cao có địa chỉ ký hiệu là WORD1+1.

Các chuỗi ký tự.

Một chuỗi ký tự có thể được khởi tạo bằng mảng các mã ASCII. Ví dụ:

```
LETTER  DB      'ABC'
```

tương đương với:

```
LETTER  DB      41h,42h,43h
```

Trong một chuỗi, trình biên dịch phân biệt các chữ hoa và chữ thường. Vì vậy chuỗi 'abc' được dịch ra 3 byte với các giá trị 61h,62h và 63h.

Cũng có thể kết hợp các ký tự và các số trong một định nghĩa. Ví dụ:

```
MSG     DB      'HELLO',0Ah,0Dh,'S'
```

tương đương với:

```
MSG     DB  48h,45h,4Ch,4Ch,4Fh,0Ah,0Dh,24h
```

4.4 Các hằng có tên.

Để tạo ra các mã lệnh Hợp ngữ dễ hiểu, người ta thường dùng các tên tượng trưng để biểu diễn các hằng số.

EQU (EQUates: *coi bằng như, coi như*).

Để gán tên cho hằng, chúng ta có thể sử dụng toán tử giả EQU. Cú pháp:

```
Tên     EQU     hằng_số
```

Ví dụ:

```
LF      EQU     0Ah
```

sẽ gán tên LF cho 0Ah, là mã ASCII của ký tự xuống dòng. Tên LF bây giờ có thể được dùng để thay cho 0Ah tại bất cứ đâu trong chương trình. Trình biên dịch sẽ dịch các lệnh:

```
MOV     DL,0AH
```

và:

```
MOV     DL,LF
```

ra cùng một chỉ thị máy.

Phần tử bên phải EQU cũng có thể là một chuỗi. Ví dụ:

```
PROMPT  EQU     'TYPE YOUR NAME"
```

Sau đó thay vì:

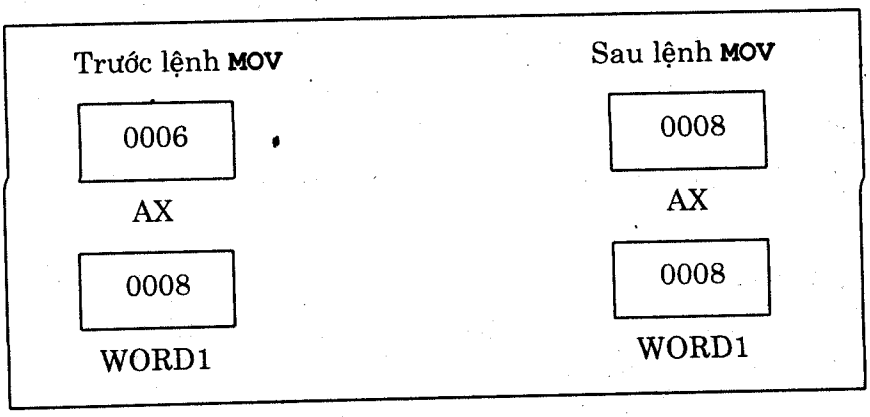
```
MSG     DB      "TYPE YOUR NAME"
```

Ta có thể viết:

```
MSG     DB      PROMPT
```

Chú ý: **Bộ nhớ không dành chỗ cho các hằng có tên** (khi biên dịch, nơi nào chứa tên hằng thì ở đó sẽ được thay đổi bởi giá trị của hằng).

Hình 4.1 Mov AX,WORD1

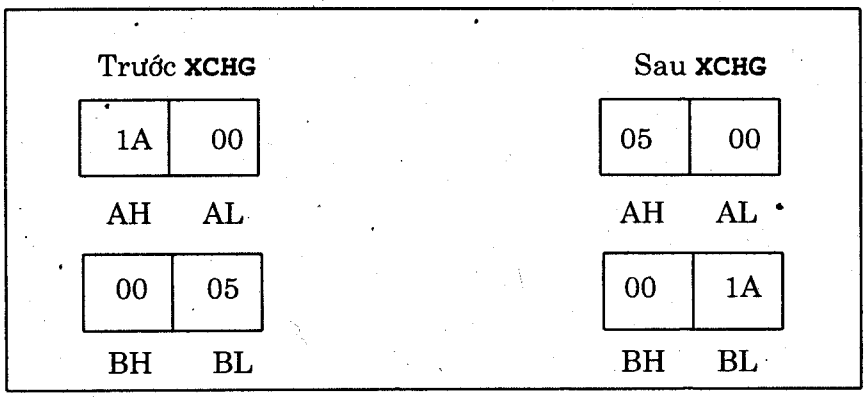


4.5 Vài lệnh cơ bản.

Hệ lệnh của bộ vi xử lý 8088 có đến hơn một trăm lệnh, trong đó có các lệnh được thiết kế dành riêng cho các bộ vi xử lý cao cấp (xem chương 20). Trong phần này chúng ta sẽ xem xét 6 lệnh tiện dụng nhất cho việc chuyển dữ liệu và thực hiện các phép tính số học. Các lệnh mà chúng tôi đưa ra ở đây có thể dùng được cho cả các toán hạng byte và word.

Trong phần sau đây, WORD1 và WORD2 là các biến kiểu word, BYTE1 và BYTE2 là các biến kiểu byte. Như đã nêu trong chương 3, AH là byte cao của thanh ghi AX; BL là byte thấp của BX.

Hình 4.2 XCHG AH,BL



4.5.1 MOV và XCHG.

Lệnh MOV được sử dụng để chuyển dữ liệu giữa các thanh ghi, giữa một thanh ghi và một ô nhớ hoặc chuyển trực tiếp một số vào một thanh ghi hay ô nhớ. Cú pháp:

```
MOV     đích,nguồn
```

Sau đây là một vài ví dụ:

```
MOV     AX,WORD1
```

Lệnh này đọc là "chuyển WORD1 vào AX". Nội dung của thanh ghi AX được thay bằng nội dung của ô nhớ WORD1. Nội dung của WORD1 không bị thay đổi. Nói một cách khác, một bản sao của WORD1 được gửi vào AX (Hình 4.1).

```
MOV     AX,BX
```

AX lấy giá trị chứa trong BX, còn BX không bị thay đổi.

```
MOV     AH,'A'
```

Lệnh này chuyển số 41h (mã ASCII của 'A') vào thanh ghi AH. Giá trị trước đó của thanh ghi AH bị viết đè lên (thay bằng giá trị mới).

Lệnh XCHG (hóán chuyển) được dùng để hoán chuyển nội dung của hai thanh ghi, thanh ghi và một ô nhớ. Cú pháp là:

```
XCHG    đích,nguồn
```

Ví dụ:

XCHG AH,BL

Lệnh này sẽ hoán chuyển nội dung của hai thanh ghi AH và BL, như vậy AH sẽ chứa nội dung trước đây của BL còn BL lại chứa nội dung trước đây của AH (Hình 4.2). Một ví dụ khác:

XCHG AX,WORD1

Lệnh sẽ hoán chuyển nội dung của thanh ghi AX và ô nhớ WORD1.

Bảng 4.2

Các khả năng kết hợp cho phép của các toán hạng trong lệnh MOV và XCHG

Chỉ thị MOV

Toán hạng đích / Toán hạng nguồn	Thanh ghi công dụng chung	Thanh ghi đoạn	Ô nhớ	Hằng số
Thanh ghi công dụng chung	YES	YES	YES	NO
Thanh ghi đoạn	YES	NO	YES	NO
Ô nhớ	YES	YES	NO	NO
Hằng số	YES	NO	YES	NO

Chỉ thị **XCHG**

Toán hạng đích / Toán hạng nguồn	Thanh ghi công dụng chung	Ô nhớ
Thanh ghi công dụng chung	YES	YES
Ô nhớ	YES	NO

Các hạn chế của MOV và XCHG.

Vì lý do kỹ thuật, có một vài hạn chế khi sử dụng lệnh MOV và XCHG. Bảng 4.2 chỉ ra các khả năng kết hợp cho phép. Cần chú ý rằng các chỉ thị MOV và XCHG không hợp lệ tròng trường hợp hai toán hạng cùng là các ô nhớ ví dụ :

```
MOV     WORD1,WORD2     ;không hợp lệ
```

Nhưng chúng ta có thể giải quyết vấn đề này bằng cách sử dụng các thanh ghi, chẳng hạn:

```
MOV     AX,WORD2
MOV     WORD1,AX
```

4.5.2 Các chỉ thị ADD, SUB, INC và DEC.

Các chỉ thị ADD và SUB được sử dụng để cộng hoặc trừ nội dung của hai thanh ghi, một thanh ghi và một ô nhớ hoặc cộng (trừ) một số vào (từ) một thanh ghi hay một ô nhớ. Cú pháp:

```
ADD     đích,nguồn
SUB     đích,nguồn
```

Ví dụ:

```
ADD     WORD1,AX
```

Chỉ thị này "cộng AX vào WORD1", sẽ cộng nội dung của thanh ghi AX với nội dung của ô nhớ WORD1 và chứa tổng trong WORD1. AX không bị thay đổi (Hình 4.3).

```
SUB     AX,DX
```

Trong ví dụ này "trừ DX từ AX", giá trị của AX trừ đi giá trị của DX, kết quả được chứa trong AX, thanh ghi DX không bị thay đổi (Hình 4.4).

```
ADD     BL,5
```

Chỉ thị này cộng số 5 vào nội dung của thanh ghi BL.

Cũng giống như trường hợp MOV và XCHG, có một vài hạn chế khi kết hợp các toán hạng của ADD và SUB. Các trường hợp cho phép được tổng kết trong bảng 4.3. Phép cộng hay trừ trực tiếp giữa các ô nhớ là không hợp lệ. Ví dụ:

```
ADD     BYTE1,BYTE2     ;không hợp lệ
```

Có một giải pháp là chuyển BYTE2 vào một thanh ghi trước khi cộng:

```
MOV     AL,BYTE2        ;AL lấy giá trị BYTE2
```

```
ADD     BYTE1,AL          ;cộng vào BYTE1
```

INC (INCrement) được dùng để cộng 1 vào nội dung của một thanh ghi hay ô nhớ , DEC (DECrement) trừ 1 từ nội dung của một thanh ghi hay ô nhớ. Cú pháp:

```
INC     đích
DEC     đích
```

Ví dụ:

```
INC     WORD1
```

cộng 1 vào nội dung của WORD1 (Hình 4.5).

```
DEC     BYTE1
```

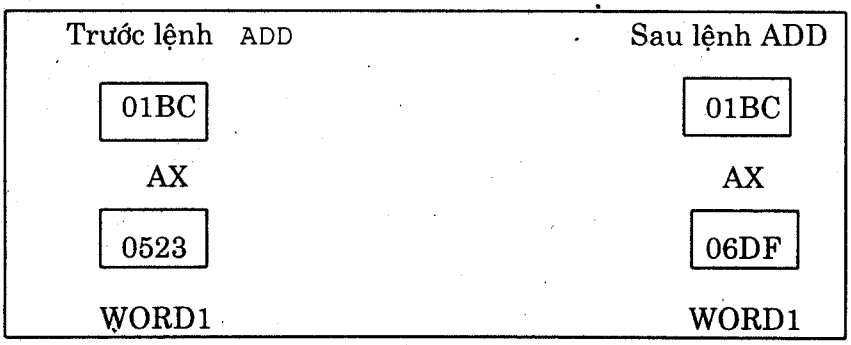
trừ 1 từ biến BYTE1 (Hình 4.6).

Bảng 4.3.

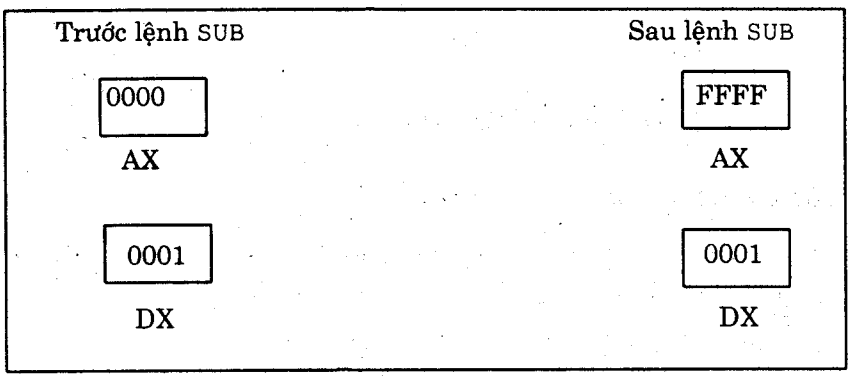
Các kết hợp cho phép của các toán hạng trong phép cộng và phép trừ.

	Toán hạng đích	
Toán hạng nguồn	Thanh ghi công dụng chung	Ô nhớ
Thanh ghi công dụng chung	yes	yes
Ô nhớ	yes	no
Hằng số	yes	yes

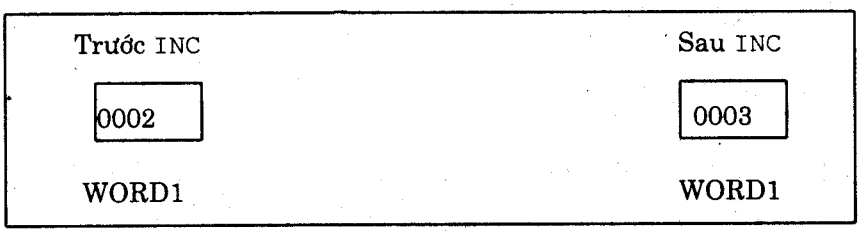
Hình 4.3. ADD `WORD1,AX`



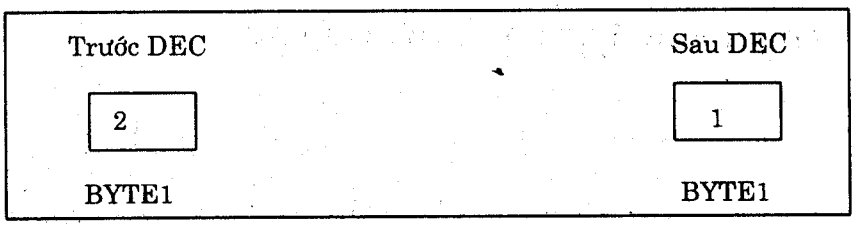
Hình 4.4. SUB AX,DX



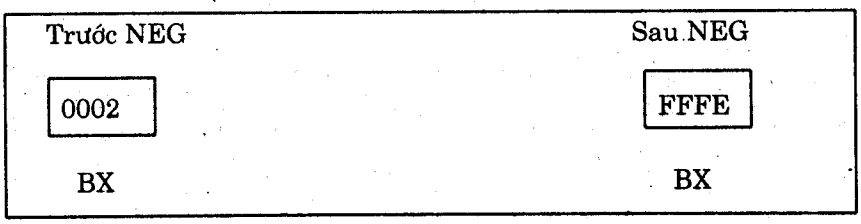
Hình 4.5. INC WORD1



Hình 4.6. DEC BYTE1



Hình 4.7. NEG BX



4.5.3 NEG

NEG dùng để phủ định nội dung của toán hạng đích. DEC sẽ thay thế nội dung bởi phần bù 2 của nó. Cú pháp:

```
NEG đích
```

Toán hạng đích có thể là một thanh ghi hay một ô nhớ. Ví dụ:

```
NEG BX
```

Sẽ phủ định nội dung của thanh ghi BX (Hình 4.7).

Kiểu quy ước của các toán hạng.

Các toán hạng của các lệnh hai toán hạng đã nêu phải có cùng kiểu, tức là cùng là byte hoặc word. Vì thế một lệnh như sau:

```
MOV AX,BYTE1          ;không hợp lệ
```

là không được phép. Tuy nhiên trình biên dịch chấp nhận cả hai lệnh sau đây:

```
MOV AH,'A'
```

và

```
MOV AX,'A'
```

Trong trường hợp đầu, trình biên dịch xét thấy do toán hạng đích AH là một byte nên toán hạng nguồn cũng phải là một byte và nó chuyển 41h vào AH. Đến trường hợp sau, vì toán hạng nguồn là một từ, nó giả thiết toán hạng đích cũng như vậy và chuyển 0041h vào AX.

4.6 Dịch từ ngôn ngữ bậc cao sang Hợp ngữ .

Để các bạn hiểu rõ hơn những lệnh đã nêu ở trên, chúng tôi sẽ dịch vài dòng lệnh gán của ngôn ngữ bậc cao sang Hợp ngữ . Chúng tôi chỉ sử dụng các lệnh MOV, ADD, SUB, INC, DEC và NEG mặc dù trong vài trường hợp nếu ta dùng các lệnh nêu sau này sẽ thực hiện tốt hơn. Trong các ví dụ, A và B là các biến word:

Dòng lệnh	**Dịch**
B=A	MOV AX,A;chuyển A vào AX MOV B,AX;rồi sang B

Như đã chỉ ra, chuyển trực tiếp giữa các ô nhớ là không hợp lệ, do vậy chúng ta phải chuyển nội dung của A sang một thang ghi trước khi chuyển sang B.

```
A=5-A       MOV AX,5        ;đưa 5 vào AX
            SUB AX,A        ;AX chứa 5-A
            MOV A,AX        ;đưa nó vào A
```

Ví dụ này minh hoạ một phương pháp dịch các lệnh gán: thực hiện các thao tác số học trong một thanh ghi (chẳng hạn AX), sau đó chuyển kết quả vào biến đích. Trong ví dụ này có một cách khác ngắn hơn:

```
NEG A          ;A=-A
ADD A,5        ;A=5-A
```

Ví dụ tiếp theo sẽ chỉ ra phương pháp nhân với một hằng số.

```
A=B-2*A    MOV AX,B       ;AX chứa B
           SUB AX,A       ;AX chứa B-A
           SUB AX,A       ;AX chứa B-2*A
           MOV A,AX       ;lưu kết quả vào A
```

4.7 Cấu trúc chương trình.

Như chương 3 đã nêu, các chương trình bằng ngôn ngữ máy gồm có mã, dữ liệu và ngăn xếp. Mỗi phần chiếm một đoạn bộ nhớ. Chương trình bằng Hợp ngữ cũng có tổ chức như vậy. Trong trường hợp này, mã dữ liệu và ngăn xếp được cấu trúc như các đoạn chương trình. Mỗi đoạn chương trình sẽ được dịch thành một đoạn bộ nhớ bởi trình biên dịch.

Chúng ta sử dụng các định nghĩa đoạn đơn giản hoá mà đã được dùng cho MASM 5.0. Ta sẽ thảo luận kỹ hơn trong chương 14 cùng với các định nghĩa đoạn toàn phần.

4.7.1 Các chế độ bộ nhớ.

Kích thước của đoạn mã và dữ liệu trong một chương trình có thể được xác định bằng cách chỉ ra chế độ bộ nhớ nhờ sử dụng dẫn hướng biên dịch .MODEL Cú pháp:

```
.MODEL  kiểu_bộ_nhớ
```

Các chế độ bộ nhớ thường sử dụng nhất là: SMALL, MEDIUM, COMPACT và LARGE. Chúng được trình bày ở hình 4.4. Trừ khi có rất nhiều mã lệnh hay số liệu, kiểu thích hợp nhất là SMALL. Dẫn hướng biên dịch .MODEL phải được đưa vào trước bất kỳ một định nghĩa đoạn nào.

Bảng 4.4. Các kiểu bộ nhớ.

Kiểu	Miêu tả
SMALL	Mã lệnh trong một đoạn . Dữ liệu trong một đoạn.

MEDIUM	Mã lệnh chiếm nhiều hơn một đoạn. Dữ liệu trong một đoạn.
COMPACT	Mã lệnh trong một đoạn. Dữ liệu chiếm nhiều hơn một đoạn.
LARGE	Mã lệnh chiếm nhiều hơn một đoạn. Dữ liệu chiếm nhiều hơn một đoạn. Không có mảng nào lớn hơn 64 Kbyte.
HUGE	Mã lệnh chiếm nhiều hơn một đoạn. Dữ liệu chiếm nhiều hơn một đoạn. Các mảng có thể lớn hơn 64 Kbyte.

4.7.2 Đoạn dữ liệu (Data segment).

Đoạn dữ liệu của một chương trình chứa tất cả các định nghĩa biến. Cũng như vậy các định nghĩa hằng cũng thường được tạo ra ở đây nhưng chúng cũng có thể được đưa vào một chỗ khác trong chương trình bởi vì không có ô nhớ nào liên quan đến nó. Để khai báo một đoạn dữ liệu chúng ta sử dụng dẫn hướng biên dịch **.DATA**, theo sau là các khai báo biến hay hằng. Ví dụ:

```
.DATA
WORD1   DW    2
WORD2   DW    5
MSG     DB    'THIS IS A MESSAGE'
MASK    EQU   10010010B
```

4.7.3 Đoạn ngăn xếp (Stack segment).

Mục đích của khai báo đoạn ngăn xếp là tạo ra một khối bộ nhớ (vùng ngăn xếp) để chứa ngăn xếp. Vùng ngăn xếp có thể đủ lớn để chứa ngăn xếp với kích thước lớn nhất của nó. Cú pháp khai báo như sau:

```
.STACK kích_thước
```

Trong đó kích thước là một số tuỳ ý, xác định kích thước của vùng ngăn xếp tính theo byte. Ví dụ:

```
.STACK 100h
```

sẽ tạo ra 100h byte cho vùng ngăn xếp (kích thước hợp lý cho hầu hết các chương trình ứng dụng). Nếu kích_thước bị bỏ qua, 1 Kbyte sẽ được thiết lập cho vùng ngăn xếp.

4.7.4 Đoạn mã (Code segment).

Đoạn mã chứa các lệnh của chương trình. Cú pháp khai báo là:

```
.CODE   tên
```

Trong đó tên là một tện đoạn tuỳ ý (không cần thiết phải có tên trong một chương trình dùng kiểu bộ nhớ .SMALL bởi vì như vậy trình biên dịch sẽ phát sinh một lỗi).

Bên trong đoạn mã, các lệnh được tổ chức như các thủ tục. Một định nghĩa thủ tục đơn giản nhất là:

```
Tên_thủ_tục             PROC
;thân của thủ tục
Tên_thủ_tục             ENDP
```

ở đây Tên_thủ_tục là tên của thủ tục; PROC và ENDP là các toán tử giả đánh dấu bắt đầu và kết thúc thủ tục.

Sau đây là một ví dụ định nghĩa đoạn mã:

```
.CODE
MAIN    PROC
;các lệnh của chương trình chính
MAIN    ENDP
;các thủ tục khác
```

4.7.5 Tổng hợp lại.

Giờ đây sau khi đã nghiên cứu tất cả các đoạn chương trình, chúng ta có thể chỉ ra khuôn mẫu chung cho một chương trình dùng kiểu bô nhớ .SMALL. Chỉ với vài thay đổi nhỏ, mẫu này có thể dùng cho tất cả các chương trình ứng dụng:

```
.MODEL  SMALL
.STACK  100H
.DATA
;các định nghĩa số liệu ở đây
```

```
.CODE
MAIN    PROC
;các lệnh ở đây
MAIN    ENDP
;các thủ tục khác ở đây
END     MAIN
```

Dòng cuối cùng của chương trình phải là dẫn hướng biên dịch END, theo sau là tên của chương trình chính.

4.8 Các lệnh vào ra.

Trong chương 1 bạn đã biết rằng CPU liên lạc với các thiết bị ngoại vi thông qua các thanh ghi vào/ra hay còn được gọi là các cổng vào/ra. Có hai lệnh có thể truy nhập trực tiếp các cổng đó là IN và OUT. Các lệnh này được sử dụng khi yêu cầu tốc độ cao, ví dụ như trong các chương trình trò chơi. Tuy nhiên hầu hết các chương trình ứng dụng không dùng các lệnh IN và OUT bởi vì thứ nhất là các địa chỉ cổng thay đổi giữa các loại máy tính và sau nữa lập trình với các chương trình phục vụ được cung cấp bởi các nhà sản xuất dễ hơn nhiều.

Có hai loại chương trình phục vụ vào/ra: Các chương trình của DOS và BIOS (Basic Input Output System). Các chương trình BIOS được chứa trong ROM và tác động trực tiếp tới các cổng vào/ra. Trong chương 12, chúng ta sẽ dùng chúng để thực hiện các thao tác cơ bản với màn hình như di chuyển con trỏ hay cuốn màn hình. Các chương trình của DOS có thể thực hiện các công việc phức tạp hơn, ví dụ như in một chuỗi ký tự. Thực ra chúng sử dụng các chương trình của BIOS để thực hiện các thao tác vào/ra trực tiếp.

Lệnh INT.

Lệnh INT được dùng để gọi các chương trình ngắt của DOS và BIOS. Nó có dạng sau:

```
INT số_hiệu_ngắt
```

ở đây, số_hiệu_ngắt là một con số xác định một chương trình. Ví dụ: INT 16 ,sẽ gọi các phục vụ bàn phím của BIOS. Chương 15 sẽ trình bày về lệnh INT một cách chi tiết hơn. Sau đây chúng ta sẽ sử dụng một chương trình đặc biệt của DOS, phục vụ ngắt 21h.

4.8.1 Ngắt 21h

Ngắt 21h được dùng để gọi rất nhiều hàm của DOS (xem phụ lục C). Mỗi hàm được gọi bằng cách đặt số hàm vào trong thanh ghi AH và gọi INT 21h. Chúng ta hãy xem xét các hàm sau đây:

Số hiệu hàm	Chương trình
1	Vào một phím
2	Đưa một ký tự ra màn hình
9	Đưa ra một chuỗi ký tự

Các hàm của ngắt 21h nhận dữ liệu trong các thanh ghi nào đó và trả về kết quả trong các thanh ghi khác. Các thanh ghi này sẽ được liệt kê khi chúng tôi mô tả mỗi hàm.

Hàm 1:
Vào một phím.

Vào: AH=1
Ra: AL= Mã ASCII nếu một phím ký tự được ấn.
= 0 Nếu một phím điều khiển hay chức năng được nhấn

Để gọi phục vụ này, bạn hãy thực hiện các lệnh sau:

```
MOV     AH,1            ;hàm vào một phím
INT     21h             ;mã ASCII trong AL
```

Bộ vi xử lý sẽ đợi người sử dụng ấn một phím nếu cần thiết. Nếu một phím ký tự được ấn, AL sẽ nhận mã ASCII và ký tự được hiện lên trên màn hình. Nếu một phím khác được ấn, chẳng hạn phím mũi tên, F1-F10..., thì AL sẽ chứa 0. Trong các lệnh tiếp theo INT 21h có thể kiểm tra AL và thực hiện tác vụ thích hợp.

Bởi vì hàm 1 của ngắt 21 không đưa ra thông báo để người sử dụng vào một phím nên bạn sẽ không biết được là máy tính đang đợi nhập số liệu hay đang làm các công việc khác. Hàm tiếp theo có thể được dùng để đưa ra các thông báo nhập số liệu:

Hàm 2:

Hiển thị một ký tự hay thi hành một chức năng điều khiển.

Vào:	AH=2
	DL= mã ASCII của ký tự hiển thị hay ký tự điều khiển.
Ra:	AL= mã ASCII của ký tự hiển thị hay ký tự điều khiển.

Để dùng hàm này hiển thị một ký tự, ta đặt mã ASCII của nó trong DL. Ví dụ các lệnh sau đây sẽ làm xuất hiện dấu chấm hỏi trên màn hình:

```
MOV   AH,2
MOV   DL,'?'
INT   21h
```

Sau khi ký tự được hiển thị, con trỏ màn hình dịch sang vị trí tiếp theo của dòng (nếu ở cuối dòng, con trỏ màn hình sẽ dịnh chuyển sang đầu đòng tiếp theo).

Hàm 2 cũng có thể được dùng để thực hiện một chức năng điều khiển. Nếu như DL chứa mã ASCII của ký tự điều khiển, hàm này sẽ thi hành chức năng điều khiển đó. Các ký tự điều khiển quan trọng được chỉ ra sau đây:

Mã ASCII(Hex)	Ký hiệu	Chức năng
7	BEL	phát tiếng bíp (beep).
8	BS	lùi lại một vị trí.
9	HT	tab.
A	LF	xuống dòng.
D	CR	xuống dòng và về đầu dòng.

Khi thực hiện, AL nhận mã ASCII của ký tự điều khiển.

4.9 Chương trình đầu tiên.

Chương trình đầu tiên của chúng ta sẽ đọc một ký tự từ bàn phím và hiển thị nó ở đầu dòng tiếp theo.

Chúng ta bắt đầu bằng việc hiển thị một dấu chấm hỏi:

```
MOV   AH,2        ; hàm hiển thị ký tự
MOV   DL,'?'      ; ký tự là "?"
```

```
INT   21h          ; hiển thị ký tự
```

Lệnh thứ hai chuyển 3Fh (mã ASCII của '?') vào thanh ghi DL.

Tiếp theo ta hãy đọc một ký tự:

```
MOV   AH,1         ; hàm đọc một ký tự
INT   21h          ; ký tự trong AL
```

Bây giờ chúng cần hiển thị ký tự ở dòng tiếp theo. Trước khi thực hiện điều này, ký tự phải được cất vào một thanh ghi khác (chúng tôi sẽ giải thích điều này trong chốc lát).

```
MOV   BL,AL        ; cất ký tự trong BL
```

Để dịch chuyển con trỏ màn hình đến vị trí đầu dòng tiếp theo chúng ta phải thực hiện các tác vụ xuống dòng và về đầu dòng. Chúng ta có thể thực hiện các hàm này bằng cách đưa mã ASCII của chúng vào DL và gọi ngắt 21h.

```
MOV   AH,2         ; hàm hiển thị ký tự
MOV   DL,0Dh       ; về đầu dòng
INT   21h          ; thực hiện về đầu dòng
MOV   DL,0Ah       ; xuống dòng
INT   21h          ; thực hiện xuống dòng
```

Lý do mà chúng ta phải đưa ký tự từ AL vào BL là hàm 2 của ngắt 21h làm thay đổi AL.

Cuối cùng chúng ta đã sẵn sàng hiển thị ký tự:

```
MOV     DL,BL    ;lấy ký tự
INT     21h      ;và hiển thị nó
```

Sau đây là chương trình đầy đủ:

Chương trình PGM4_1.ASM

```
TITLE   PGM4_1:SAMPLE PROGRAM
.MODEL  SMALL
.STACK  100H
.CODE
MAIN    PROC
```

```
        MOV     AH,2            ; hàm hiển thị ký tự
        MOV     DL,'?'          ; ký tự là "?"
        INT     21h             ; hiển thị ký tự
;vào một ký tự
        MOV     AH,1            ; hàm đọc một ký tự
        INT     21h             ; ký tự trong AL
        MOV     BL,AL           ; cất ký tự trong BL
;xuống dòng mới:
        MOV     AH,2            ; hàm hiển thị ký tự
        MOV     DL,0Dh          ; về đầu dòng
        INT     21h             ; thực hiện về đầu dòng
        MOV     DL,0Ah          ; xuống dòng
        INT     21h             ; thực hiện xuống dòng
;hiển thị ký tự:
        MOV     DL,BL           ;lấy ký tự
        INT     21h             ;và hiển thị nó
;trở về DOS
        MOV     AH,4CH          ;hàm thoát về DOS
        INT     21H             ;thoát về DOS
MAIN    ENDP
        END     MAIN
```

Do không dùng các biến nên ta bỏ qua đoạn dữ liệu.

Kết thúc một chương trình.

Hai dòng cuối cùng của chương trình MAIN cần có đôi lời giải thích. Khi một chương trình kết thúc, nó phải trả điều khiển về cho DOS. Chúng ta có thể thực hiện điều này bằng cách gọi hàm 4Ch của ngắt 21h.

4.10 Tạo lập và chạy một chương trình.

Bây giờ chúng ta đã sẵn sàng để xem xét các bước tạo lập và chạy một chương trình. Chương trình nêu ở trên như là một ví dụ để khảo sát. Có 4 bước cụ thể (hình 4.8):

1. Dùng một chương trình soạn thảo văn bản tạo ra một file chương trình nguồn (source program file).
2. Dùng một chương trình biên dịch tạo ra file đối tượng ngôn ngữ máy (object file).

2. Dùng một chương trình biên dịch tạo ra file đối tượng ngôn ngữ máy (object file).
3. Dùng chương trình LINK (sẽ trình bày chi tiết sau) liên kết một hay nhiều file đối tượng tạo ra các file chương trình (run file).
4. Cho chạy file chương trình.

Trong chỉ dẫn này, các file cần thiết (chương trình biên dịch và chương trình liên kết) có trong ổ đĩa C, các đĩa của người lập trình trong ổ đĩa A. Chúng ta đặt ổ đĩa A là ổ đĩa mặc định để các file tạo ra sẽ được chứa trong đĩa của người lập trình.

Bước 1. Tạo lập file chương trình nguồn.

Chúng ta đã sử dụng một chương trình soạn thảo văn bản tạo chương trình ở trên với tên là PGM4_1.ASM. Phần mở rộng .ASM được quy ước dùng để định nghĩa một file nguồn của Hợp ngữ .

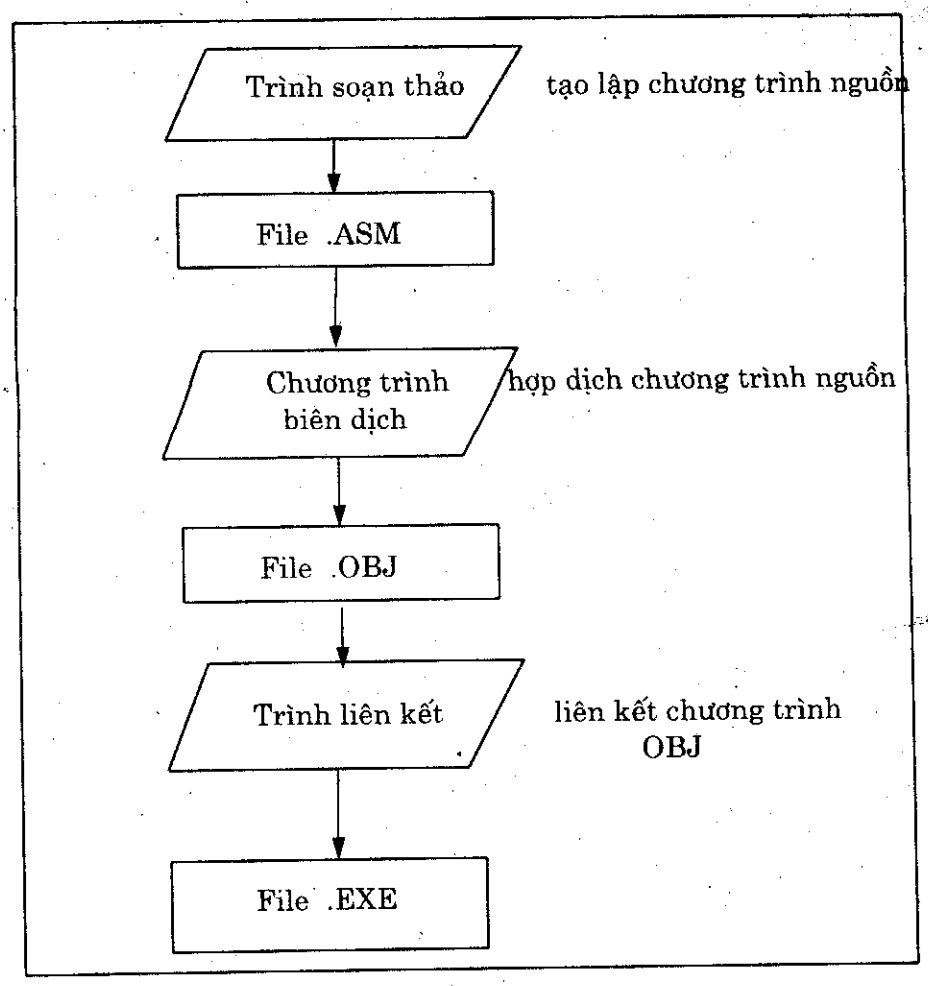
Bước 2. Hợp dịch chương trình.

Chúng ta sử dụng MASM (Microsoft Macro Assembler) để dịch file nguồn PGM4_1.ASM sang file đối tượng ngôn ngữ máy PGM4_1.OBJ. Lệnh đơn giản nhất (câu trả lời của người sử dụng xuất hiện trong vùng đệm) là:

```
A>C:MASM PGM4_1;
Microsoft  (R) Macro Assembler Version 5.10
Copyright  (C)  Microsoft  Corp  1981,  1988.  All  right
reserved.
      50060  +  418673  Byte symbol space free
          0  Warning Errors
          0  Severe Errors
```

Sau khi in ra các thông tin bản quyền, MASM kiểm tra các lỗi cú pháp của file nguồn. Nếu tìm thấy một lỗi nào đó nó sẽ hiển thị số dòng của mỗi lỗi và một hướng dẫn ngắn gọn. Trong trường hợp này do không có lỗi nào, MASM sẽ dịch mã Hợp ngữ thành file đối tượng ngôn ngữ máy với tên PGM4_1.OBJ.

Hình 4.8 Các bước lập trình.



Dấu chấm phẩy theo sau câu lệnh có nghĩa là chúng ta không muốn phát sinh thêm các file khác nữa. Ta hãy bỏ nó đi và xem cái gì xảy ra:

```
A>C:MASM PGM4_1
Microsoft  (R) Macro Assembler Version 5.10
Copyright  (C)  Microsoft  Corp  1981,  1988.  All  right
reserved.
Object  filename [  PGM4_1.OBJ ]  :
Source  listing  [ NUL.LST ]  :     PGM4_1
Cross_reference [ NUL.CRF ]:    PGM4_1
       50060  +  418673   Byte symbol space free
           0  Warning Errors
           0  Severe Errors
```

Lần này MASM in ra các tên file có thể được tạo nên và đợi chúng ta đưa vào tên mới. Các tên mặc định được viết trong dấu ngặc vuông. Muốn chấp nhận một tên chỉ cần nhấn phím Return. Tên mặc định NUL có nghĩa là sẽ không tạo tên file đó nếu ta không đưa vào một tên, do vậy ở đây ta trả lời với tên PGM4_1.

File nguồn listing.

File nguồn listing (file .LST) là một file văn bản các dòng được đánh số hiển thị mã Hợp ngữ bên cạnh mã máy tương ứng đồng thời đưa ra các thông tin khác về chương trình. Điều này đặc biệt có ích cho các mục đích gỡ rối bởi vì các lỗi của MASM đưa ra kèm theo số dòng.

File tham khảo ngang (Cross_reference file).

File tham khảo ngang (file .CRF) là một bảng liệt kê các tên xuất hiện trong chương trình và số thứ tự các dòng mà nó có mặt. Nó cần thiết khi xác định các biến và các nhãn trong một chương trình lớn.

Ví dụ về các file .LST và .CRF được chỉ ra trong phụ lục D cùng với các phần chọn khác của MASM.

Bước 3. Hợp dịch chương trình.

File .OBJ được tạo lập ở bước 2 là một file ngôn ngữ máy nhưng nó không thể thực hiện được bởi lẽ khuôn mẫu của nó chưa thích hợp với một File chương trình:

1. Nó không biết được nơi mà chương trình sẽ được nạp vào trong bộ nhớ để thi hành, và địa chỉ mã máy có thể chưa được điền vào.
2. Một vài tên dùng trong chương trình có thể chưa được định nghĩa. Ví dụ: trong một chương trình lớn có thể cần phải tạo ra vài file, và một thủ tục trong một file này có thể tham trỏ tới một tên được định nghĩa trong file khác.

Chương trình LINK sẽ gồm một hay nhiều file đối tượng, điền vào mọi địa chỉ còn thiếu và kết hợp các file đối tượng thành một file khả thi duy nhất (file .EXE). File này có thể được nạp vào trong bộ nhớ và chạy luôn.

Để liên kết chương trình bạn hãy đánh vào:

```
A>C:LINK  PGM4_1;
```

Giống như trên, nếu không có dấu chấm phẩy, chương trình liên kết sẽ đưa ra thông báo để bạn vào các tên của các file phát sinh (xem phụ lục D).

Bước 4. Chạy một chương trình.

Để chạy một chương trình bạn chỉ cần đánh vào tên File chương trình có hoặc không có phần mở rộng .EXE.

```
A> PGM4_1
?A
A
```

Chương trình in ra dấu '?' và đợi chúng ra đưa vào một ký tự. Ta đánh vào A và chương trình lặp lại nó ở dòng tiếp theo.

4.11 Hiển thị một chuỗi.

Trong chương trình đầu tiên, ta dùng các hàm 1 và 2 của ngắt 21h để đọc và hiển thị một ký tự. Sau đây là một hàm khác của ngắt 21h được dùng để hiển thị một chuỗi các ký tự:

Ngắt 21h, Hàm 9:
Hiển thị một chuỗi.

Vào: DX=địa chỉ tương đối (offset) của chuỗi.
Chuỗi phải kết thúc bằng ký tự '$ '.

Ký tự '$ ' đánh dấu kết thúc chuỗi và không được hiển thị. Nếu như chuỗi chứa mã ASCII của ký tự điều khiển thì các chức năng điều khiển sẽ được thi hành.

Để làm ví dụ cho hàm này, chúng ta sẽ viết một chương trình in chuỗi "HELLO!" ra màn hình. Lời chào này được định nghĩa trong đoạn dữ liệu :

```
MSG     DB      "HELLO!$"
```

Lệnh LEA.

Hàm 9 của ngắt 21h yêu cầu địa chỉ tương đối của chuỗi ký tự chứa trong DX. Để thực hiện điều này chúng ta sẽ dùng một lệnh mới:

```
LEA     đích,nguồn
```

Trong đó đích là một thanh ghi công dụng chung, nguồn là một ô nhớ. LEA có nghĩa là "Load Effective Address" (nạp địa chỉ thực). Nó sẽ lấy ra và chép địa chỉ tương đối của nguồn sang đích. Ví dụ:

```
LEA     DX,MSG
```

sẽ nhập địa chỉ tương đối của biến MSG vào DX.

Bởi vì chương trình thứ hai của chúng ta có chứa đoạn dữ liệu nên nó sẽ được bắt đầu với lệnh khởi tạo DS. Phần tiếp sau đây sẽ giải thích tại sao các lệnh đó là cần thiết.

Đoạn mở đầu chương trình.

Khi một chương trình được nạp vào bộ nhớ, DOS sẽ dành cho nó 256 byte đoạn mở đầu chương trình (PSP_Program Segment Prefix). PSP chứa các thông tin về chương trình vì thế chương trình có thể truy nhập vùng này. DOS sẽ đưa địa chỉ của PSP vào cả trong DS lẫn ES trước khi thi hành chương trình. Kết quả là DS không chứa địa chỉ của đoạn dữ liệu. Để chạy đúng, một chương trình có chứa đoạn dữ liệu sẽ được bắt đầu với hai lệnh sau:

```
MOV             AX,@DATA
MOV             DS,AX
```

@DATA là tên của đoạn dữ liệu được định nghĩa bởi .DATA. Chương trình biên dịch sẽ dịch @DATA thành địa chỉ. Ta phải dùng hai lệnh vì là một số (địa chỉ dữ liệu) không thể chuyển trực tiếp vào một thanh ghi đoạn. Với thanh ghi DS đã được khởi tạo, chúng ta có thể in ra lời chào "HELLO!" bằng cách đưa địa chỉ của nó vào thanh ghi DX rồi gọi ngắt 21h:

```
LEA     DX,MSG          ; lấy thông báo
MOV     AH,9            ; hàm hiển thị chuỗi
INT     21h             ; hiển thị chuỗi
```

Sau đây là chương trình đầy đủ:

Chương trình PGM4_2.ASM

Sau đây là chương trình đầy đủ:

Chương trình PGM4_2.ASM

```
TITLE     PGM4_2:   Chương trình in chuỗi ký tự.
.MODEL          SMALL
.STACK          100H
.DATA
MSG     DB      "HELLO!$"
.CODE
MAIN    PROC
;khởi tạo DS
        MOV     AX,@DATA
        MOV     DS,AX
;hiển thị thông báo
        LEA     DX,MSG          ; lấy thông báo
        MOV     AH,9            ; hàm hiển thị chuỗi
        INT     21h             ; hiển thị chuỗi
;trở về DOS
        MOV     AH,4CH
        INT     21H
MAIN    ENDP
        END     MAIN
```

Và đây là kết quả khi chạy chương trình:

```
A> PGM4_2
HELLO!
```

4.12 Một chương trình đổi chữ thường thành chữ hoa.

Bây giờ chúng ta sẽ tổng kết các kiến thức trình bày trong chương này vào một chương trình duy nhất. Chương trình này sẽ bắt đầu bằng việc nhắc người sử dụng đưa vào một chữ thường, trên dòng tiếp theo nó sẽ đưa ra một thông báo khác với chữ đã được đổi sang dạng in hoa. Ví dụ:

```
ENTER  A  LOWER CASE  LETTER : a
IN  UPPER  CASE  IT  IS  :  A
```

Chúng ta dùng các tên CR và LF để định nghĩa các hằng số 0DH và 0AH.

```
MSG1    DB      'ENTER  A  LOWER CASE  LETTER : $ '
MSG2    DB      CR,LF,' IN  UPPER  CASE IT IS :'
CHAR    DB      ?,' $ '
```

Khi định nghĩa MSG2 và CHAR chúng ta sử dụng một mánh khoé hữu hiệu: Bởi vì chương trình phải hiển thị thông báo thứ hai và một chữ cái (sau khi đã đổi thành chữ hoa) trên dòng tiếp theo, MSG2 bắt đầu với mã ASCII của ký tự xuống dòng và trở về đầu dòng. Khi MSG2 được hiển thị bằng hàm 9 của ngắt 21h các chức năng điều khiển này sẽ được thi hành và thông báo được hiển thị ở dòng tiếp theo. Do MSG2 không kết thúc bằng ký tự '$', ngắt 21h sẽ hiển thị cả ký tự chứa trong CHAR.

Chương trình của chúng ta bắt đầu với việc hiển thị thông báo thứ nhất và đọc ký tự:

```
LEA     DX,MSG1         ; lấy thông báo đầu tiên
MOV     AH,9            ; hàm hiển thị chuỗi
INT     21h             ; hiển thị thông báo đầu tiên
MOV     AH,1            ; hàm đọc một ký tự
INT     21h             ; đọc một chữ thường vào AL
```

Sau khi đã đọc vào chữ thường, chương trình phải đổi nó sang dạng chữ hoa. Trong bảng mã ASCII, các chữ thường bắt đầu tại 61h và các chữ hoa bắt đầu tại 41h, vì vậy để đổi kiểu chữ chỉ cần lấy nội dung AL trừ đi 20h:

```
SUB     AL,20H          ; đổi thành chữ hoa
MOV     CHAR,AL         ; và lưu trữ nó
```

Bây giờ chương trình sẽ hiển thị thông báo thứ hai và chữ ở dạng in hoa:

```
LEA     DX,MSG2         ; lấy thông báo thứ hai
MOV     AH,9            ; hàm hiển thị chuỗi
INT     21h             ; hiển thị thông báo thứ hai
                ; và chữ hoa
```

Sau đây là chương trình đầy đủ:

Chương trình PGM4_3.ASM

```
TITLE   PGM4_3:SAMPLE PROGRAM
```

```
.MODEL  SMALL
.STACK  100H
.DATA
CR      EQU     0DH
LF      EQU     0AH
MSG1    DB      'ENTER  A  LOWER CASE  LETTER : $ '
MSG2    DB      CR,LF,' IN  UPPER  CASE IT IS :'
CHAR    DB      ?,' $ '
.CODE
MAIN    PROC
;khởi tạo DS
        MOV     AX,@DATA
        MOV     DS,AX
;in dòng nhắc người sử dụng
        LEA     DX,MSG1   ;lấy thông báo đầu tiên
        MOV     AH,9      ;hàm hiển thị chuỗi
        INT     21h       ;hiển thị thông báo đầu tiên
;vào một ký tự và đổi thành chữ hoa
        MOV     AH,1      ;hàm đọc một ký tự
        INT     21h       ;đọc một chữ thường vào AL
        SUB     AL,20H    ;đổi thành chữ hoa
        MOV     CHAR,AL   ;và lưu trữ nó
;hiển thị trên dòng tiếp theo
        LEA     DX,MSG2         ;lấy thông báo thứ hai
        MOV     AH,9            ;hàm hiển thị chuỗi
        INT     21h             ;hiển thị thông báo thứ
                                ;hai và chữ hoa
;trở về DOS
        MOV     AH,4CH
        INT     21H
MAIN    ENDP
        END     MAIN
```

TỔNG KẾT:

- Các chương trình bằng Hợp ngữ được tạo nên từ các dòng lệnh. Mỗi dòng lệnh có thể là một lệnh sẽ được máy tính thi hành hay một dẫn hướng cho chương trình biên dịch.
- Các dòng lệnh bao gồm các trường tên, toán tử, (các) toán hạng và trường lời bình.
- Một tên có chứa đến 31 ký tự. Các ký tự có thể là các chữ cái, chữ số hay các ký hiệu đặc biệt nào đó.
- Các số có thể được viết ở dạng nhị phân, thập phân hay hex.
- Các ký tự và chuỗi ký tự phải được bao bọc bởi dấu ngoặc đơn hay ngoặc kép.
- Các dẫn hướng DB và DW được dùng để định nghĩa các biến byte và các biến word. EQU sử dụng khi muốn gán tên cho hằng.
- Nói chung một chương trình chứa một đoạn mã, một đoạn dữ liệu và một đoạn ngăn xếp.
- MOV và XCHG được dùng để chuyển số liệu. Có một vài hạn chế khi sử dụng các lệnh này, ví dụ chúng không thể thao tác trực tiếp giữa các ô nhớ.
- ADD, SUB, INC, DEC và NEG là các lệnh số học cơ bản.
- Có hai cách xuất và nhập dữ liệu đối với IBM PC : liên lạc trực tiếp với các thiết bị ngoại vi thông qua cổng và sử dụng các phục vụ ngắt của DOS và BIOS.
- Phương pháp trực tiếp khó lập trình và phụ thuộc vào các vi mạch phần cứng nhất định.
- Xuất và nhập các ký tự hay các chuỗi có thể thực hiện nhờ phục vụ ngắt 21h của DOS.
- Hàm 1 của ngắt 21h đọc một ký tự từ bàn phím vào thanh ghi AL.
- Hàm 2 của ngắt 21h hiển thị ký tự có mã ASCII chứa trong thanh ghi DL. Nếu DL chứa mã ASCII của một ký tự điều khiển thì chức năng điều khiển sẽ được thi hành.
- Hàm 9 của ngắt 21h hiển thị chuỗi có địa chỉ tương đối trong DX. Chuỗi phải được kết thúc bằng ký tự '$ '.

Thuật ngữ tiếng Anh

Array	Một chuỗi các byte hay từ nhớ.
assembler directive	Dẫn hướng biên dịch: Dẫn hướng cho chương trình biên dịch thực hiện một vài nhiệm vụ đặc biệt.
CODE segment	Đoạn mã. Vùng chương trình chứa các lệnh.
.CRF file	File tạo nên bởi trình biên dịch trong đó liệt kê các tên xuất hiện trong chương trình và số của các dòng mà chúng xuất hiện.
DATA segment	Đoạn dữ liệu. Vùng chương trình chứa các biến.
destinasion operand	Toán hạng đích. Toán hạng đầu tiên của lệnh. Nó sẽ chứa kết quả trả về.
.EXE file	Giống như File chương trình.
instruction	Lệnh hay chỉ thị. Một dòng lệnh được chương trình biên dịch dịch ra mã máy.
.LST file	File có các dòng được đánh số tạo bởi trình biên dịch trong đó chứa mã Hợp ngữ, mã máy và các thông tin khác về chương trình.
memory MODEL	Kiểu bộ nhớ. Tổ chức của một chương trình cho biết tổng số các mã và dữ liệu.
object file	File đối tượng. File ngôn ngữ máy được chương trình biên dịch tạo ra từ file chương trình nguồn.
PSP (program segment prefix)	Đoạn mở đầu chương trình :Vùng 256 byte đứng trước chương trình trong bộ nhớ chứa các thông tin về chương trình.
pseudo_op	Toán tử giả. Dẫn hướng biên dịch.
run file	File chương trình. File ngôn ngữ máy khả thi tạo nên bởi chương trình LINK.
source operand	Toán hạng nguồn. Toán hạng thứ hai, thường không thay đổi khi thực hiện lệnh.
source program file	Tệp chương trình nguồn. Tệp văn bản chương trình tạo ra bằng một chương trình soạn thảo văn bản.

STACK segment	Đoạn ngăn xếp. Phần của chương trình chứa ngăn xếp hiện hành.
variable	Biến. Tên tượng trưng cho một ô nhớ chứa dữ liệu.

Các lệnh mới

ADD	INT	NEG
DEC	LEA	SUB
INC	MOV	XCHG

Các toán tử giả mới

.CODE	.MODEL	EQU
.DATA	.STACK	

Bài tập:

1. Tên nào trong các tên sau đây là hợp lệ trong Hợp ngữ cho IBM PC ?
   a. TWO_WORD
   b. ?1
   c. Two_word
   d. .@?
   e. $145
   f. LET'S_GO
   g. T=.

2. Số nào trong các số sau đây là hợp lệ. Nếu chúng hợp lệ, hãy chỉ rõ chúng là số nhị phân, thập phân hay số hex.
   a. 246
   b. 246h
   c. 1001
   d. 1,101
   e. 2A3h
   f. FFF Eh
   g. 0ah
   h. bh
   i. 1110b

3. Hãy nêu toán tử giả định nghĩa dữ liệu cho các biến và hằng sau đây nếu như chúng hợp lệ:
   a. Một biến kiểu word được khởi tạo với giá trị 52.
   b. Một biến kiểu word WORD1, không được khởi tạo.
   c. Một biến kiểu byte B được khởi tạo với giá trị 52h.
   d. Một biến kiểu byte C1 không được khởi tạo.
   e. Một biến kiểu word WORD2 được khởi tạo với giá trị 65536.
   f. Một mảng kiểu word được khởi tạo với 5 giá trị đầu nguyên (từ 1 đến 5).
   g. Một hằng BEL bằng 07h.
   h. Một hằng MSG bằng ' THIS IS A MESSAGE $'.

4. Giả thiết rằng các số liệu sau đây được nạp vào bộ nhớ bắt đầu tại vị trí offset 0000h:

A	DB	7
B	DW	1ABCh
C	DB	'HELLO'

a. Hãy cho biết các địa chỉ offset của các biến A, B và C.

b. Hãy cho biết nội dung của byte tại offset 0002h dưới dạng số hex.

c. Hãy cho biết nội dung của byte tại offset 0004h dưới dạng số hex.

d. Hãy cho biết địa chỉ offset của ký tự 'O' trong 'HELLO'.

5. Hãy cho biết mỗi lệnh dưới đây là hợp lệ hay không hợp lệ, trong đó W1 và W2 là các biến WORD; B1, B2 là các biến BYTE.

```
a.      MOV     DS,AX
b.      MOV     DS,100h
c.      MOV     DS,ES
d.      MOV     W1,DS
e.      XCHG    W1,W2
f.      SUB     5,B1
g.      ADD     B1,B2
h.      ADD     AL,256
i.      MOV     W1,B1
```

6. Chỉ dùng các lệnh MOV, ADD, SUB, INC, DEC và NEG hãy dịch các dòng lệnh gán của ngôn ngữ bậc cao sau đây sang Hợp ngữ với A, B, C là các biến kiểu word.

```
a.      A=B-A
b.      A=-(A+1)
c.      C=A+B
d.      B=3*B+7
e.      B=B-A-1
```

7. Hãy viết các lệnh (không phải các chương trình đầy đủ) thực hiện các công việc sau đây:

a. Đọc một ký tự và hiển thị nó ở vị trí tiếp theo trên cùng một dòng.

b. Đọc một chữ hoa (bỏ qua việc kiểm tra lỗi) và hiển thị nó ở vị trí tiếp theo trên cùng một dòng dưới dạng chữ thường.

8. Viết một chương trình thực hiện các công việc sau đây:

a. Hiển thị dấu hỏi chấm (?).

b.Đọc hai chữ số thập phân có tổng nhỏ hơn 10.

c. Hiển thị các số đó với tổng của chúng với dòng thông báo tương ứng.

Ví dụ:

?27

Tổng của 2 và 7 là 9.

9. Hãy viết một chương trình thực hiện các công việc sau đây:

a. Đưa ra thông báo cho người sử dụng.

b. Đọc 3 chữ cái đầu của họ, tên đệm, tên của một người.

c. Hiển thị chúng từ trên xuống trên lề trái.

Ví dụ:

Bạn hãy vào 3 chữ cái đầu: NTB

N

T

B

10. Viết một chương trình đọc một chữ số hex trong khoảng (A-F) rồi hiển thị nó trên dòng tiếp theo ở dạng nhị phân.

Ví dụ:

Bạn vào một chữ số hex: C

Dạng thập phân của nó là: 12

11. Viết một chương trình hiển thị một bảng 10x10 điền đầy dấu sao.

Gợi ý: Khai báo một chuỗi xác định hộp trong đoạn dữ liệu rồi hiển thị nó bằng hàm 9 của ngắt 21h.

12. Viết một chương trình:

a. Hiển thị dấu '?'.

b. Đọc 3 chữ cái.

c. Hiển thị chúng trong một bảng 11x11 được điền đầy các dấu sao.

d. Phát tiếng kêu bip của máy tính.

Chương 5

TRẠNG THÁI CỦA BỘ XỬ LÝ VÀ THANH GHI CỜ

Tổng quan

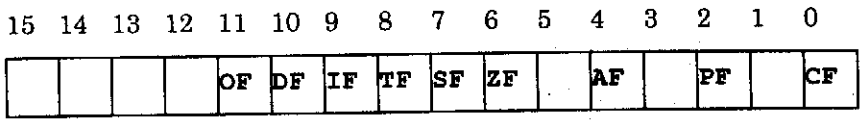
Một điểm khác biệt quan trọng giữa máy tính và các loại máy khác đó là máy tính có khả năng quyết định. Các mạch trong máy tính có khả năng thực hiện những quyết định đơn giản dựa trên trạng thái hiện tại của bộ xử lý. Đối với bộ vi xử lý 8086, trạng thái của bộ vi xử lý được thể hiện trong 9 bit riêng biệt gọi là các cờ. Mọi quyết định của bộ vi xử lý đều dựa trên giá trị của các cờ này.

Các cờ được đặt trong thanh ghi cờ và chúng được phân thành 2 loại cờ trạng thái và cờ điều khiển. Cờ trạng thái phản ánh kết quả của các phép tính. Trong chương này chúng ta sẽ thấy chúng bị ảnh hưởng ra sao bởi các chỉ thị máy. Trong chương 6 chúng ta sẽ nghiên cứu việc sử dụng chúng để xây dựng các chương trình có nhiều lệnh rẽ nhánh và vòng lặp. Cờ điều khiển được sử dụng để cho phép hoặc không cho phép một thao tác nào đó của bộ vi xử lý, chúng sẽ được mô tả ở các chương cuối.

Trong phần 5.1 chúng tôi sẽ giới thiệu chương trình DEBUG của DOS, chúng ta sẽ thấy cách sử dụng DEBUG để thực hiện từng lệnh trong chương trình của người sử dụng, hiển thị các thanh ghi, các cờ và các ô nhớ.

5.1 Thanh ghi cờ

Hình 5.1 cho thấy cấu trúc của thanh ghi cờ. Các cờ trạng thái nằm ở các bit 0, 2, 4, 6, 7 và 11 còn các cờ điều khiển nằm ở các bit 8, 9 và 10. các bit khác không có ý nghĩa. Chú ý rằng không cần nhớ cờ nào nằm ở bit nào. Bảng 5.1 trình bày tên các cờ và ký hiệu của chúng. Trong chương này chúng ta sẽ tập trung vào các cờ trạng thái.



Hình 5.1 Thanh ghi cờ

Các cờ trạng thái

Như đã nói trên, bộ xử lý sử dụng cờ trạng thái để phản ánh kết quả của một phép tính, chẳng hạn khi lệnh SUB AX,AX được thực hiện cờ ZF sẽ được thiết lập 1 nhờ vậy nó chỉ ra rằng kết quả bằng 0 đã được tạo ra. Bây giờ chúng ta hãy xem các cờ trạng thái.

Cờ nhớ (Carry Flag- CF)

Cờ CF được thiết lập 1 khi có nhớ từ bit msb trong phép cộng hay có vay vào bit msb trong phép trừ. Ngược lại nó bằng 0. Cờ CF cũng bị ảnh hưởng bởi các lệnh quay và dịch (xem chương 7).

Cờ chẵn lẻ (Parity Flag- PF)

Cờ PF được thiết lập 1 nếu byte thấp của kết quả có số chẵn các bit 1(parity chẵn). Nó bằng 0 nếu byte thấp có số lẻ bít 1(parity lẻ). Ví dụ kết quả của một phép cộng các word là FFFEh, như vậy byte thấp có 7 bit 1 do đó PF= 0

Cờ nhớ phụ (Auxiliary Flag- AF)

Cờ AF được thiết lập 1 nếu có nhớ từ bit 3 trong phép cộng hoặc có vay vào bit 3 trong phép trừ. Cờ AF được sử dụng trong các thao tác với số thập phân mã hoá nhị phân (số BCD).

Cờ Zero (Zero Flag- ZF)

Cờ ZF được thiết lập 1 khi kết quả bằng 0 và ngược lại.

Cờ dấu (Sign Flag- SF)

Cờ SF được thiết lập 1 khi bit msb của kết quả bằng 1 có nghĩa là kết quả là âm nếu bạn làm việc với số có dấu. Ngược lại SF=0 nếu bit msb của kết quả bằng 0.

Cờ tràn (Overflow Flag OF)

Cờ OF được thiết lập 1 khi xảy ra tràn ngược lại nó bằng 0, khái niệm tràn sẽ được giải thích sau đây.

Cờ trạng thái		
Bit	Tên gọi	Ký hiệu
0	cờ nhớ	CF
2	Cờ chẵn lẻ	PF
4	Cờ nhớ phụ	AF
6	Cờ ZERO	ZF
7	Cờ dấu	SF
11	Cờ tràn	OF
Cờ điều khiển		
Bit	Tên gọi	Ký hiệu
8	Cờ bẫy	TF
9	Cờ ngắt	IF
10	Cờ định hướng	DF

Bảng 5.1 Tên cờ và các ký hiệu.

5.2 Hiện tượng tràn

Hiện tượng tràn gắn liền với một sự thật là phạm vi của các số biểu diễn trong máy tính có giới hạn.

Chương 2 đã chỉ ra rằng phạm vi của các số thập phân có dấu có thể biểu diễn bằng một word 16 bit là từ -32768 đến 32767, với một byte 8 bit thì phạm vi là từ -128 đến 127.

Đối với các số không dấu thì phạm vi là từ 0 đến 65535 cho một word và từ 0 đến 255 cho một byte. Nếu kết quả của một phép tính nằm ngoài phạm vi này thì hiện tượng tràn sẽ xảy ra và kết quả nhận được bị cắt bớt sẽ không phải là kết quả đúng.

Các ví dụ về hiện tượng tràn

Sự tràn có dấu và không có dấu là các hiện tượng độc lập nhau. Khi chúng ta thực hiện một thao tác số học như cộng hay trừ, có 4 khả năng xảy ra là: (1) không tràn, (2) chỉ tràn có dấu, (3) chỉ tràn không dấu, (4) tràn có dấu và không dấu đồng thời.

Đây là một ví dụ về hiện tượng tràn không dấu nhưng không tràn có dấu:

Giả sử AX chứa FFFFh, BX chứa 0001h và lệnh ADD AX,BX được thực hiện. Kết quả dạng nhị phân như sau:

```
   1111 1111 1111 1111
+  0000 0000 0000 0001
-------------------------
 1 0000 0000 0000 0000
```

Nếu chúng ta làm việc với các số không dấu, kết quả đúng phải là 100000h= 65536, nhưng kết quả này nằm ngoài phạm vi biểu diễn của một word nên kết quả còn lại trong thanh ghi AX là 0h, đây là một kết quả sai, như vậy hiện tượng tràn không dấu đã xảy ra. Nhưng kết quả nhận được lại đúng với các số có dấu, FFFFh = -1 khi hiểu là số có dấu, trong khi đó 0001h = 1 vậy tổng của chúng bằng 0, rõ ràng hiện tượng tràn có dấu đã không xảy ra.

Bây giờ chúng ta hãy xem một ví dụ về hiện tượng tràn có dấu nhưng lại không tràn không dấu. Giả sử AX và BX cùng chứa 7FFFh, hãy thực hiện phép cộng ADD AX,BX

Kết quả dạng nhị phân như sau

```
   0111 1111 1111 1111
   0111 1111 1111 1111
-------------------------
   1111 1111 1111 1110     =FFFEh
```

Trong cả 2 dạng có dấu và không có dấu 7FFFh đều bằng 32767, bởi vậy trong cả 2 phép cộng không dấu và có dấu đều cho kết quả là 32767+ 32767 = 65534. Giá trị này nằm ngoài phạm vi của số có dấu, kết quả nhận được dạng có dấu là FFFEh= -2, như vậy xảy ra hiện tượng tràn có dấu. Tuy nhiên FFFEh lại bằng 65534 ở dạng không dấu do vậy không có hiện tượng tràn không dấu.

Có 2 vấn đề được đặt ra đó là làm sao CPU chỉ ra có hiện tượng tràn và làm sao nó biết được có hiện tượng tràn xảy ra ?

CPU đã chỉ ra có hiện tượng tràn như thế nào ?

Bộ xử lý lập cờ OF =1 để báo có hiện tượng tràn có dấu và CF = 1 để báo có hiện tượng tràn không dấu. Thông báo này dùng cho chương trình để tiến hành những hành động thích hợp, nếu như không có hành động nào được thực hiện ngay lập tức thì kết quả của lệnh sau có thể xoá cờ báo tràn.

Khi xác định có hiện tượng tràn bộ xử lý không coi kết quả như là số không dấu cũng như số có dấu, thay vào đó nó sử dụng cả 2 cách hiểu có dấu và không có dấu cho mỗi thao tác đồng thời thiết lập các cờ CF hay OF báo tràn có dấu hay không có dấu một cách tương ứng.

Trách nhiệm thuộc về người lập trình, người có quy ước về kết quả. Nếu anh ta đang làm việc với số có dấu thì chỉ có cờ OF đáng quan tâm trong khi cờ CF có thể bỏ qua, ngược lại khi làm việc với số không dấu thì cờ quan trọng là CF chứ không phải là OF.

Làm cách nào CPU biết được có hiện tượng tràn xảy ra ?

Có rất nhiều lệnh có thể gây ra hiện tượng tràn, nhưng để đơn giản ở đây chúng ta chỉ nói về phép cộng và phép trừ.

Hiện tượng tràn không dấu

Khi thực hiện phép cộng, hiện tượng tràn xảy ra khi có nhớ từ bit msb. Có nghĩa là kết quả đúng của phép tính lớn hơn số không dấu lớn nhất có thể biểu diễn, đó là FFFFh cho một word và FFh cho một byte. Khi thực hiện phép trừ hiện tượng tràn không dấu xảy ra khi có sự vay vào bit msb, có nghĩa là kết quả đúng nhỏ hơn 0.

Hiện tượng tràn có dấu

Trong phép cộng các số cùng dấu, hiện tượng tràn có dấu xảy ra khi tổng nhận được có dấu khác với dấu của 2 số hạng. Điều này đã xảy ra trong ví dụ trên khi chúng ta cộng 2 số dương 7FFFh nhưng lại nhận được kết quả là một số âm FFFEh.

Phép trừ các số khác dấu cũng giống như phép cộng các số cùng dấu, ví dụ A- (-B) = A+ B và -A-(+B) = -A + -B. Hiện tượng tràn xảy ra khi kết quả có dấu khác với dấu chúng ta chờ đợi (xem ví dụ 5.3 trong phần sau).

Trong phép cộng 2 số khác dấu hiện tượng tràn là không thể xảy ra vì A + (-B) chính bằng A - B và bởi vì A và B là 2 số đủ nhỏ để chứa trong toán hạng

đích thì hiệu của chúng tất nhiên không thể vượt ra ngoài phạm vi của nó được. Cũng vì lý do như vậy mà phép trừ 2 số cùng dấu cũng không thể gây ra hiện tượng tràn.

Thực ra bộ xử lý sử dụng phương pháp sau để thiết lập cờ OF:

Nếu việc nhớ vào bit msb và việc nhớ ra từ nó không đồng thời, có nghĩa là có nhớ vào msb nhưng không có nhớ ra từ nó, hay có nhớ ra từ nó mà không có nhớ vào thì hiện tượng tràn có dấu xuất hiện và OF được lập 1 (xem ví dụ 5.2 trong phần tiếp theo).

5.3 Sự ảnh hưởng của các lệnh đến các cờ

Nhìn chung mỗi khi bộ xử lý thực hiện một lệnh, các cờ được thay đổi để phản ánh kết quả. Tuy nhiên có một số lệnh không ảnh hưởng tới cờ, chỉ ảnh hưởng tới một số trong chúng hay có thể làm cho chúng không xác định. Vì lệnh nhảy sẽ nghiên cứu trong chương 6 phụ thuộc vào việc lập cờ, chúng ta cần biết mỗi lệnh ảnh hưởng tới cờ như thế nào. Chúng ta hãy trở lại với 7 lệnh cơ bản đã học trong chương 4, chúng ảnh hưởng tới cờ như sau:

Chỉ thị	Các cờ bị ảnh hưởng
MOV/XCHG	Không cờ nào
ADD/SUB	Tất cả các cờ
INC/DEC	Tất cả các cờ trừ cờ CF
NEG	Tất cả các cờ (CF = 1 trừ khi kết quả là 0, OF = 1 nếu toán hạng word là 8000h hay toán hạng byte là 80h

Để các bạn làm quen với việc các lệnh có ảnh hưởng như thế nào đến các cờ, chúng tôi sẽ nêu ra ở đây một vài ví dụ. Trong mỗi ví dụ chúng tôi sẽ trình bày một lệnh, nội dung của các toán hạng và dự đoán kết quả cũng như việc thiết lập các cờ CF, PF, ZF và OF (chúng ta bỏ qua AF vì nó chỉ được sử dụng cho các phép tính số học với số BCD).

Ví dụ 5.1: Thực hiện phép cộng AX,BX, trong đó AX và BX đều chứa FFFFh

Trả lời :

```
   FFFFh
+  FFFFh
-------
  1FFFEh
```

Kết quả thu được chứa trong AX là FFFEh = 1111 1111 1111 1110.

SF = 1 vì msb = 1.

PF = 0 vì có 7 bit 1 (lẻ các bit 1) trong byte thấp của kết quả.

ZF = 0 vì kết quả thu được khác 0

CF = 1 vì có nhớ từ bit msb trong phép cộng

OF = 0 vì dấu của tổng nhận được giống như dấu của các số hạng tham gia phép cộng (còn khi thực hiện phép cộng dưới dạng nhị phân bạn sẽ thấy có nhớ vào bit msb đồng thời cũng có nhớ từ msb).

Ví dụ 5.2 : Thực hiện phép cộng AL,BL trong đó AL và BL cùng chứa 80h

Trả lời:

```
  80
  80
-----
 100
```

Kết quả nhận được trong AL bằng 0

SF = 0 vì msb = 0.

PF = 1 vì tất cả các bit của tổng bằng 0.

ZF = 1 vì kết quả thu được bằng 0

CF = 1 vì có nhớ từ bit msb trong phép cộng

OF = 1 vì các số hạng tham gia phép cộng đều là các số âm nhưng kết quả nhận được là một số dương (còn khi thực hiện phép cộng dưới dạng nhị phân bạn sẽ thấy không có nhớ vào bit msb nhưng lại có nhớ từ bit msb).

Ví dụ 5.3 : Thực hiện phép trừ AX,BX trong đó AX chứa 8000h còn BX chứa 0001h

Trả lời :

```
  8000
  0001
------
  7FFF
```

Kết quả nhận được trong AX là 7FFFh = 0111 1111 1111 1111

SF = 0 vì msb = 0.

PF = 1 vì có 8 bit 1 (chẵn các bit 1) trong byte thấp của kết quả.

ZF = 0 vì kết quả thu được khác 0

CF = 0 vì số không dấu nhỏ hơn bị trừ từ số lớn hơn.

Còn về cờ OF, xét về phương diện các số có dấu, chúng ta đã trừ một số dương từ một số âm, cũng giống như cộng 2 số âm. Tuy nhiên kết quả nhận được lại là số dương (sai dấu) do đó OF = 1.

Ví dụ 5.4: Tăng AL lên 1 khi AL chứa FFh.

Trả lời:

```
  FF
   1
-----
 100
```

Kết quả nhận được trong AL là 00h, SF = 0, PF = 1, ZF = 1. Mặc dù có nhớ CF không bị ảnh hưởng bởi lệnh INC. Có nghĩa là nếu trước đó CF = 0 thì cuối cùng CF vẫn bằng 0.

OF = 0 vì 2 số hạng có dấu khác nhau (có nhớ vào msb đồng thời cũng có nhớ từ msb).

Ví dụ 5.5 : Chuyển -5 vào AX

Trả lời :

Kết quả nhận được trong AX là -5 = FFFBh, không có cờ nào bị ảnh hưởng bởi lệnh MOV.

Ví dụ 5.6 : Nghịch đảo AX, trong đó AX chứa 8000h

Trả lời :

8000h	1000 0000 0000 0000
Số bù 1 của 8000h	0111 1111 1111 1111
	+1
Số bù 2 của 8000h	1000 0000 0000 0000

Kết quả nhận được trong AX là 1000 0000 0000 0000 = 8000h

SF = 1, PF = 1, ZF = 0.

CF = 1 vì trong lệnh NEG, CF luôn bằng 1 trừ khi kết quả bằng 0.

OF = 1 vì kết quả là 8000h, khi lấy nghịch đảo của một số kết quả nhận được phải có dấu ngược lại nhưng ở đây số bù 2 của 8000h lại chính bằng nó tức là dấu không thay đổi.

Trong chương tới chúng tôi sẽ giới thiệu một chương trình cho phép chúng ta thấy được sự thiết lập cờ thực sự.

5.4 Chương trình DEBUG

Chương trình DEBUG tạo ra một môi trường trong đó chương trình có thể được kiểm tra. Người sử dụng có thể từng bước duyệt qua chương trình đồng thời hiển thị và thay đổi nội dung các thanh ghi cũng như ô nhớ. Cũng có thể đưa vào trực tiếp các lệnh của Hợp ngữ mà sau đó chương trình DEBUG sẽ đổi nó sang dạng mã máy và lưu trong bộ nhớ. Những hướng dẫn chi tiết về DEBUG và CODEVIEW (một chương trình gỡ rối phức tạp hơn) được trình bày trong phụ lục E.

Bây giờ chúng ta hãy sử dụng DEBUG để mô tả sự ảnh hưởng của các lệnh tới cờ, trước hết chúng ta tạo ra chương trình sau.

Chương trình PGM5_1.ASM

```
TITLE   PGM5_1:CHECK FLAGS
;Chương trình được sử dụng trong DEBUG để kiểm tra việc lập
cờ
.MODEL   SMALL
.STACK   100H
.CODE
MAIN    PROC
        MOV    AX,4000H          ;AX = 4000h
        ADD    AX,AX             ;AX = 8000h
```

```
        SUB     AX,0FFFFH           ;AX = 8001h
        NEG     AX                  ;AX = 7FFFh
        INC     AX                  ;AX = 8000h
        MOV     AH,4CH
        INT     21H                 ;Trở về DOS
MAIN    ENDP
        END     MAIN
```

Chúng ta hãy hợp dịch và liên kết chương trình để tạo ra file PGM5_1. EXE nằm trong đĩa ở ổ A. Trong đoạn dưới đây những gì người sử dụng đánh vào được in đậm.

Chương trình DEBUG chứa trong đĩa DOS ở ổ C, Để vào DEBUG và mô tả chương trình của chúng ta, hãy đánh vào:

C>DEBUG A:PGM5_1.EXE

DEBUG sẽ đáp lại bằng dấu nhắc của nó: "-", và đợi lệnh đưa vào. Trước hết chúng ta có thể xem các thanh ghi bằng cách đánh vào lệnh "R".

```
-R
AX=0000 BX=0000 CX=001F DX=0000 SP=000A BP=0000 SI=0000
DI=0000 DS=0ED5 ES=0ED5 SS=0EE5 CS=0EE6 IP=0000 NV UP DI PL NZ
NA PO NC
0EE6:0000 B80040        MOV     AX,4000
```

Trên màn hình cho thấy nội dung của các thanh ghi ở dạng hex, tại dòng thứ 3 chúng ta thấy:

```
0EE6:0000 B80040                MOV     AX,4000
```

`0EE6:0000` Là địa chỉ của lệnh tiếp theo sẽ được thực hiện cho dưới dạng `segment:offset`.

`B80040` là mã máy của lệnh đó. Đoạn 0EE6 là nơi DOS quyết định nạp chương trình vào, nếu bạn thử làm ví dụ này có thể bạn sẽ nhận được địa chỉ khác. 8 cặp chữ xuất hiện bên phải của dòng thứ 2 là trạng thái hiện tại của các cờ. Các cờ xuất hiện theo trình tự: OF, DF, IF, SF, ZF, AF, PF và CF. Bảng 5.2 trình bày các ký hiệu DEBUG sử dụng cho các cờ. Các bạn có thể thấy các cờ

đã bị DEBUG xoá, ý nghĩa của các cờ điều khiển sẽ được giải thích ở chương 11 và chương 15.

Để từng bước thực hiện chương trình, chúng ta sử dụng lệnh "T" (Trace), trước khi làm điều đó hãy hiển thị lại nội dung các thanh ghi một lần nữa:

```
-R
AX=0000  BX=0000  CX=001F  DX=0000  SP=000A  BP=0000  SI=0000
DI=0000 DS=0ED5 ES=0ED5 SS=0EE5 CS=0EE6 IP=0000 NV UP DI PL NZ
NA PO NC
0EE6:0000 B80040        MOV     AX,4000
```

Lệnh đầu tiên là MOV AX ,4000h

```
-T
AX=4000  BX=0000  CX=001F  DX=0000  SP=000A  BP=0000  SI=0000
DI=0000 DS=0ED5 ES=0ED5 SS=0EE5 CS=0EE6 IP=0003 NV UP DI PL NZ
NA PO NC
0EE6:0003 03C0  ADD     AX,AX
```

Việc thực hiện lệnh MOV AX, 4000h đưa 4000h vào thanh ghi AX, các cờ vẫn không thay đổi vì lệnh MOV không làm ảnh hưởng tới cờ. Bây giờ chúng ta hãy thực hiện lệnh ADD AX, AX:

```
-T
AX=8000  BX=0000  CX=001F  DX=0000  SP=000A  BP=0000  SI=0000
DI=0000 DS=0ED5 ES=0ED5 SS=0EE5 CS=0EE6 IP=0005 OV UP DI NG NZ
NA PE NC
0EE6:0005 2DFFFF        SUB     AX,FFFF
```

Bây giờ AX chứa 4000h + 4000h = 8000h. SF trở thành 1(NG) để chỉ ra kết quả là số âm. Dấu hiệu tràn được chỉ ra bởi OF = 1 (OV) vì chúng ta cộng 2 số dương và nhận được một số âm. PF đặt bằng 1 (PE) vì byte thấp của kết quả không có bit 1 nào.

Bây giờ chúng ta thực hiện lệnh SUB AX,0FFFFh:

```
-T

AX=8001 BX=0000 CX=001F DX=0000 SP=000A BP=0000 SI=0000
DI=0000 DS=0ED5 ES=0ED5 SS=0EE5 CS=0EE6 IP=0008 NV UP DI NG NZ
AC PO CY

0EE6:0008 F7D8      NEG   AX
```

AX chứa 8000h- FFFFh = 8001h. OF đổi trở lại thành 0 (NV), vì chúng ta trừ 2 số cùng dấu không thể xảy ra hiện tượng tràn có dấu. Tuy nhiên CF = 1 chỉ ra rằng đã có hiện tượng tràn không dấu vì chúng ta đã trừ một số không dấu lớn hơn từ một số nhỏ hơn, khiến cho có sự vay vào bit msb. PF = 0 vì byte thấp của AX có một bit 1 duy nhất.

Bây giờ chúng ta tiếp tục với lệnh NEG AX:

```
-T

AX=7FFF BX=0000 CX=001F DX=0000 SP=000A BP=0000 SI=0000
DI=0000 DS=0ED5 ES=0ED5 SS=0EE5 CS=0EE6 IP=000A NV UP DI PL NZ
AC PE CY

0EE6:000A 40        INC   AX
```

AX chứa số bù 2 của 8001h = 7FFFh. Đối với lệnh NEG CF luôn bằng 1 trừ khi kết quả bằng 0 nhưng trường hợp này lại không xảy ra ở đây. Cờ OF = 0 vì kết quả không phải là 8000h.

Cuối cùng chúng ta thực hiện lệnh INC AX:

```
-T

AX=8000 BX=0000 CX=001F DX=0000 SP=000A BP=0000 SI=0000
DI=0000 DS=0ED5 ES=0ED5 SS=0EE5 CS=0EE6 IP=000B OV UP DI NG NZ
AC PE CY

0EE6:000B B44C      MOV   AH,4C
```

Cờ OF thay đổi trở lại thành 1 (OV) vì chúng ta cộng 2 số dương (7FFFh và 1) nhưng lại nhận được kết quả là một số âm. Mặc dù không có nhớ từ bit msb nhưng CF vẫn giữ nguyên bằng 1 vì lệnh INC không ảnh hưởng tới cờ này.

Để kết thúc việc thực hiện chương trình chúng đánh vào "G" (Go):

```
-G

Program terminated normally
```

Và để thoát khỏi DEBUG chúng ta đánh Q (Quit)

```
-Q
C>
```

Bảng 5.2 Các ký hiệu cờ của DEBUG

Cờ trạng thái	Ký hiệu thiết lập (1)	Ký hiệu xoá(0)
CF	CY (có nhớ)	NC (không nhớ)
PF	PE (chẵn)	PO (lẻ)
AF	AC (có nhớ phụ)	NA (không nhớ phụ)
ZF	ZR (zero)	NZ(không zero)
SF	NG (âm)	PL(dương)
OF	OV (tràn)	NV(không tràn)
Cờ điều khiển		
DF	DN (giảm)	UP(tăng)
IF	EI (cho phép ngắt)	DI (cấm ngắt)

TỔNG KẾT

Trong chương này chúng ta đã nghiên cứu những vấn đề sau

- Thanh ghi cờ là một thanh ghi của bộ vi xử lý 8086. 6 bit của nó gọi là cờ trạng thái và 3 bit khác gọi là cờ điều khiển.
- Các cờ trạng thái phản ánh kết quả của một thao tác, chúng là các cờ: cờ nhớ(CF), cờ chẵn lẻ(PF), cờ nhớ phụ(AF), cờ Zero(ZF), cờ dấu(SF) và cờ tràn(OF).
- Cờ CF được thiết lập 1 khi một thao tác cộng hay trừ gây ra nhớ hoặc vay ở bit msb, ngược lại nó bằng 0.
- Cờ PF được thiết lập nếu có một số chẵn các bit 1 trong byte thấp của kết quả, ngược lại nó bằng 0.
- Cờ AF được thiết lập 1 nếu có nhớ hoặc vay ở bit 3 trong kết quả, ngược lại chúng bằng 0.
- Cờ ZF được thiết lập 1 nếu kết quả bằng 0, ngược lại nó bằng 0.
- Cờ SF được thiết lập nếu bit msb trong kết quả bằng 1, ngược lại nó bằng 0.
- Cờ OF được thiết lập 1 nếu kết quả đúng của phép tính quá lớn để có thể chứa trong toán hạng đích, ngược lại nó bằng 0.
- Hiện tượng tràn xảy ra khi kết quả đúng của phép tính nằm ngoài phạm vi biểu diễn của máy tính. Hiện tượng tràn đó gọi là tràn có dấu nếu kết quả được xem xét như là số có dấu tương tự như vậy nó được gọi là hiện tượng tràn không dấu nếu kết quả được xem xét như là số không dấu.
- Bộ xử lý sử dụng các cờ CF và OF để báo tràn: CF = 1 nghĩa là đã xảy ra hiện tượng tràn không dấu, OF = 1 có nghĩa là xuất hiện hiện tượng tràn có dấu.
- Bộ xử lý thiết lập cờ CF nếu có nhớ từ bit msb trong phép cộng hoặc có vay vào bit msb trong phép trừ, trong trường hợp sau điều đó có nghĩa là chúng ta đã lấy một số không dấu trừ đi số lớn hơn nó
- Bộ xử lý thiết lập cờ OF nếu có nhớ vào bit msb nhưng không có nhớ từ đó hoặc ngược lại.
- Có một phương pháp khác để nhận ra hiện tượng tràn có dấu trong các phép cộng hay trừ. Trong phép cộng các số cùng dấu, hiện tượng tràn xảy ra nếu kết quả nhận được có dấu khác với dấu của các số hạng. Phép trừ các số

khác dấu cũng giống như phép cộng các số cùng dấu và hiện tượng tràn có dấu xảy ra khi kết quả nhận được có dấu khác với dấu của số bị trừ.

- Trong phép cộng các số khác dấu hay trừ các số cùng dấu hiện tượng tràn có dấu không thể xảy ra.
- Nói chung các lệnh đều có ảnh hưởng đến cờ, nhưng có một số lệnh không ảnh hưởng tới cờ hoặc chỉ ảnh hưởng tới một số cờ.
- Trạng thái của các cờ được cho thấy trong chương trình DEBUG.
- DEBUG có thể dùng để duyệt qua một chương trình, một số lệnh của nó là: "R" để hiển thị nội dung của các thanh ghi, "T" để thực hiện một lệnh của chương trình và "G" để thực hiện chương trình.

Các thuật ngữ tiếng Anh

Control Flags	Cờ điều khiển- Các cờ dùng để cho phép hoặc không cho phép một thao tác nào đó của CPU
Flags	Các cờ- Các bit của thanh ghi cờ biểu diễn trạng thái của CPU
FLAGS register	Thanh ghi cờ- Thanh ghi trong CPU mà các bit của nó chính là các cờ
Status Flags	Cờ trạng thái- Các cờ phản ánh kết quả của một lệnh được CPU thực hiện

Bài tập

1. Hãy cho biết nội dung mới của toán hạng đích và trạng thái mới của các cờ CF, SF, ZF, PF Và OF trong các lệnh sau (giả sử ban đầu các cờ đều bị xoá về 0)

   a. `ADD AX, AX` Trong đó AX chứa 7FFFh và BX chứa 0001h
   b. `SUB AL, BL` Trong đó AL chứa 01h và BL chứa FFh
   c. `DEC AL` Trong đó AL chứa 00h
   d. `NEG AL` Trong đó AL chứa 7Fh
   e. `XCHG AX, BX` Trong đó AX chứa 1ABCh và BX chứa 712Ah
   f. `ADD AL, BL` Trong đó AL chứa 80h và BL chứa FFh
   g. `SUB AX, BX` Trong đó AX chứa 0000h và BX chứa 8000h
   h. `NEG AX` Trong đó AX chứa 0001h

2. a. Giả sử AX và BX đều chứa các số dương, lệnh ADD AX, BX được thực hiện, hãy chứng minh rằng hiện tượng có nhớ vào bit msb xảy ra nhưng không có nhớ ra từ nó khi và chỉ khi có hiện tượng tràn có dấu.

   b. Giả sử AX và BX đều chứa các số âm, lệnh ADD AX, BX được thực hiện, hãy chứng minh rằng hiện tượng có nhớ từ msb nhưng không có nhớ vào nó xảy ra khi và chỉ khi có hiện tượng tràn có dấu.

3. Giả sử lệnh ADD AX, BX được thực hiện. Trong các phần sau đây số hạng thứ nhất được chứa trong AX, số hạng thứ 2 chứa trong BX, hãy cho biết kết quả trong thanh ghi AX và có hiện tượng tràn (có dấu và không có dấu) xảy ra không ?

   a.

```
   512Ch
+  4185h
```

   b.

```
   FE12h
+  1ACBh
```

   c.

```
   E1E4h
+  DAB3h
```

   d.

7132h
+ 7000h

e.

6389h
+ 1176h

4. Giả sử phép trừ SUB AX,BX được thực hiện. Trong các phần sau đây số hạng thứ nhất được chứa trong AX, số hạng thứ 2 chứa trong BX, hãy cho biết kết quả trong thanh ghi AX và có hiện tượng tràn (có dấu và không có dấu) xảy ra không ?

a.

2143h
- 1986h

b.

81FEh
- 1986h

c.

19BCh
- 81FEh

d.

0002h
- FE0Fh

e.

8BCDh
- 71ABh

Chương 6

CÁC LỆNH ĐIỀU KHIỂN RẼ NHÁNH.

Tổng quan.

Để đảm bảo thực hiện được các công việc có ích, các chương trình bằng Hợp ngữ phải có các phương pháp chọn lựa và lặp lại các đoạn mã lệnh. Trong chương này, chúng tôi sẽ chỉ ra các phương pháp đó được thực hiện như thế nào nhờ các lệnh nhảy và lặp.

Các lệnh nhảy và lặp chuyển điều khiển cho các phần khác trong chương trình. Sự chuyển giao này có thể không có điều kiện, có thể phụ thuộc vào các tổ hợp riêng rẽ các cờ trạng thái.

Sau khi làm quen với các cấu trúc nhảy, chúng ta sẽ sử dụng chúng để thực hiện các cấu trúc chọn lựa và lặp của ngôn ngữ bậc cao. Các ứng dụng này sẽ làm cho việc đổi một thuật toán với các lệnh giả sang các lệnh Hợp ngữ dễ dàng hơn.

6.1 Một ví dụ về lệnh nhảy.

Để bạn có khái niệm về cách thức làm việc của cấu trúc nhảy, chúng tôi sẽ viết một chương trình hiển thị toàn bộ tập hợp các ký tự của IBM.

Chương trình PGM6_1.ASM

```
1:  TITLE   PGM6_1: Hiển thị các ký tự IBM.
2:  .MODEL  SMALL
3:  .STACK  100H
4:  .CODE
5:  MAIN  PROC
6:        MOV   AH,2          ; hàm hiển thị ký tự
```

```
 7:          MOV   CX,256        ; số ký tự được hiển thị
 8:          MOV   DL,0  ; DL chứa mã ASCII ký tự NULL
 9:  PRINT_LOOP:
10:          INT   21h           ; hiển thị một ký tự
11:          INC   DL            ; tăng mã ASCII
12:          DEC   CX            ; giảm bộ đếm
13:          JNZ   PRINT_LOOP    ; lặp lại nếu CX khác 0
14:;trở về DOS
15:          MOV   AH,4Ch
16:          INT   21H
17: MAIN     ENDP
18:          END   MAIN
```

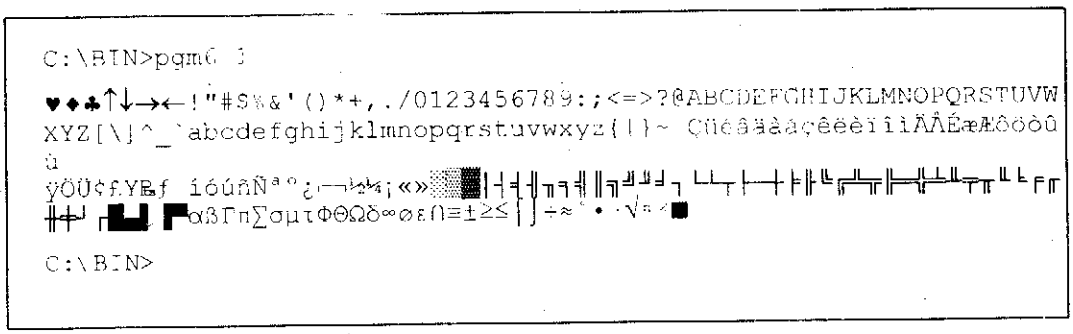
Tập hợp ký tự IBM bao gồm 256 ký tự. Các ký tự có mã từ 32 đến 127 là các ký tự ASCII chuẩn in được đã giới thiệu trong chương 2. IBM cũng cung cấp một hệ thống các ký tự đồ hoạ có mã từ 0 đến 32 và từ 128 đến 255.

Để hiển thị các ký tự chúng ta dùng một vòng lặp (từ dòng 9 đến dòng 13). Trước khi vào vòng lặp AH được khởi tạo giá trị 2 (hàm hiển thị một ký tự) và DL được đặt bằng 0 là mã ASCII ký tự đầu tiên. CX là bộ đếm vòng lặp, nó được đặt bằng 256 trước khi vào vòng lặp và giảm 1 sau khi mỗi ký tự được hiển thị.

Lệnh JNZ (Jump if Not Zero) điều khiển vòng lặp. Nếu kết quả của lệnh kế trước (DEC CX) khác 0, lệnh JNZ sẽ chuyển điều khiển đến lệnh có nhãn PRINT_LOOP. Cuối cùng khi DX bằng 0 chương trình tiếp tục thực hiện các lệnh trở về DOS. Hình 6.1 là kết quả khi chạy chương trình. Tất nhiên với các mã ASCII của các ký tự điều khiển như lùi một ký tự, về đầu dòng v.v. các chức năng điều khiển sẽ được thực hiện thay vì hiển thị chúng.

Lưu ý: PRINT_LOOP là nhãn dòng lệnh lần đầu tiên chúng ta sử dụng trong một chương trình. Các nhãn cần thiết khi một lệnh trỏ đến lệnh khác giống như trong trường hợp trên. Nhãn kết thúc bằng dấu hai chấm và để dễ nhận ra, chúng được đặt riêng một dòng. Các nhãn tham trỏ tới lệnh ngay sau chúng.

Hình 6.1. Kết quả chương trình PRG6_1.



6.2 Các lệnh nhảy có điều kiện.

JNZ là một ví dụ của lệnh **nhảy có điều kiện**. Cú pháp:

```
JNZ          nhãn_đích
```

Nếu như điều kiện của lệnh nhảy được thoả mãn, lệnh có nhãn nhãn_đích sẽ được thi hành. Lệnh này có thể ở trước hoặc sau lệnh nhảy. Nếu điều kiện không thoả mãn, lệnh ngay sau lệnh nhảy được thi hành. Với lệnh JNZ điều kiện là kết quả của lệnh trước nó khác 0.

Phạm vi của lệnh nhảy có điều kiện.

Cấu trúc mã máy của lệnh nhảy có điều kiện đòi hỏi nhãn_đích phải đứng trước lệnh nhảy không quá 126 byte hoặc đứng sau không quá 127 byte (chúng tôi sẽ chỉ ra cách để vượt qua giới hạn này sau).

CPU thực hiện một lệnh nhảy như thế nào?.

Để thực hiện một lệnh nhảy, CPU nhìn vào thanh ghi cờ. Ta đã biết rằng thanh ghi phản ánh kết quả công việc gần nhất mà bộ vi xử lý thực hiện. Nếu điều kiện của lệnh nhảy (được phát biểu như một tổ hợp các sự lập cờ trạng thái) thoả mãn, CPU điều chỉnh IP trỏ đến nhãn đích và như thế lệnh ở sau nhãn này sẽ được thi hành tiếp theo. Nếu điều kiện nhảy không thoả mãn, IP không bị sửa đổi. Điều này có nghĩa là lệnh trên dòng tiếp theo sẽ được thi hành.

Trong chương trình trên, CPU thi hành lệnh JNZ PRINT_LOOP phụ thuộc vào ZF. Nếu ZF=0 điều khiển sẽ được chuyển tới nhãn PRINT_LOOP; nếu ZF=1 chương trình tiếp tục thực hiệm lệnh MOV AH,4Ch.

Bảng 6.1 chỉ ra các lệnh nhảy có điều kiện. Có ba loại đó là:

(1) Các lệnh nhảy có dấu được dùng khi kết quả trả về là các số có dấu.

(2) Các lệnh nhảy không dấu dùng với các số không dấu.

(3) Các lệnh nhảy điều kiện đơn: điều kiện phụ thuộc vào một cờ riêng biệt.

Lưu ý: Các lệnh nhảy tự nó không ảnh hưởng tới các cờ.

Cột đầu tiên trong bảng 6.1 là mã lệnh nhảy. Rất nhiều lệnh nhảy có hai mã lệnh. Ví dụ: JG và JNLE. Cả hai mã lệnh này có cùng một mã máy. Khi thực hiện chúng cho kết quả hoàn toàn giống nhau.

Bảng 6.1. Các lệnh nhảy có điều kiện.

Các lệnh nhảy có dấu.

Ký hiệu	Chức năng	Điều kiện nhảy
JG/JNLE	nhảy nếu lớn hơn nhảy nếu không nhỏ hơn hay bằng	ZF=0 và SF=OF
JGE/JNL	nhảy nếu lớn hơn hay bằng nhảy nếu không nhỏ hơn	SF=OF
JL/JNGE	nhảy nếu nhỏ hơn nhảy nếu không lớn hơn hay bằng	SF<>OF
JLE/JNG	nhảy nếu nhỏ hơn hay bằng nhảy nếu không lớn hơn	ZF=1 hay SF=OF

Các lệnh nhảy không dấu.

Ký hiệu	Chức năng	Điều kiện nhảy
JA/JNBE	nhảy nếu lớn hơn nhảy nếu không nhỏ hơn hay bằng	CF=0 và ZF=0
JAE/JNB	nhảy nếu lớn hơn hay bằng nhảy nếu không nhỏ hơn	CF=0
JB/JNAE	nhảy nếu nhỏ hơn nhảy nếu không lớn hơn hay bằng	CF=1

JBE/JNA	nhảy nếu nhỏ hơn hay bằng	CF=1 hay ZF=1
	nhảy nếu không lớn hơn	

Các lệnh nhảy điều kiện đơn.

Ký hiệu	Chức năng	Điều kiện nhảy
JE/JZ	nhảy nếu bằng	ZF=1
	nhảy nếu bằng 0	
JNE/JNZ	nhảy nếu không bằng	ZF=0
	nhảy nếu khác 0	
JC	nhảy nếu có nhớ	CF=1
JNC	nhảy nếu không nhớ	CF=0
JO	nhảy nếu nếu tràn	OF=1
JNO	nhảy nếu không tràn	OF=0
JS	nhảy nếu dấu âm	SF=1
JNS	nhảy nếu dấu dương	SF=0
JP/JPE	nhảy nếu cờ chẵn	PF=1
JNP/JPO	nhảy nếu cờ lẻ	PF=0

Lệnh CMP.

Các điều kiện nhảy thường được cung cấp bởi lệnh CMP (compare). Nó có dạng sau:

```
CMP    đích,nguồn
```

Lệnh này so sánh toán tử đích với toán tử nguồn bằng cách lấy toán tử đích trừ đi toán tử nguồn. Kết quả không được lưu lại nhưng các cờ bị ảnh hưởng. Các toán hạng của lệnh CMP không thể cùng là các ô nhớ. Toán hạng đích không được phép là hằng số. Chú ý: CMP giống hệt như SUB ngoại trừ việc toán hạng đích không bị thay đổi.

Ví dụ. Giả thiết chương trình chứa hai dòng lệnh sau đây:

```
CPM    AX,BX
JG     BELOW
```

Ở đây AX=7FFFh, BX=0001. Kết quả so sánh AX và BX là 7FFFh-0001h=7FFEh.

Bảng 6.1 chỉ ra rằng điều kiện nhảy cho lệnh JG đã được thoả mãn bởi vì ZF=SF=OF=0, và do đó điều khiển được chuyển đến nhãn BELOW.

Dịch các lệnh có điều kiện.

Trong ví dụ vừa nêu, khi xem xét các cờ sau lệnh CMP ta thấy rằng điều khiển sẽ được chuyển đến nhãn BELOW. Đó cũng là cách CPU thực hiện một lệnh nhảy có điều kiện. Người lập trình không cần thiết là phải suy nghĩ nhiều về các cờ, bạn có thể chỉ dùng tên của lệnh nhảy để quyết định việc chuyển điều khiển đến nhãn đích. Các lệnh sau đây:

```
CMP     AX,BX
JG      BELOW
```

Nếu như AX lớn hơn BX (coi là số có dấu) thì JG (jump if greater than) sẽ chuyển đến BELOW.

Đặc biệt cả khi nghĩ rằng lệnh CMP được thiết kế chỉ dùng cho các lệnh nhảy có điều kiện, chúng vẫn có thể đi kèm với các lệnh khác như trong chương trình PGM6_1. Một ví dụ khác:

```
DEC     AX
JL      THERE
```

trường hợp này nếu nội dung của AX (coi là số có dấu) nhỏ hơn 0, điều khiển sẽ được chuyển đến THERE.

So sánh các lệnh nhảy có dấu và không dấu.

Mỗi lệnh nhảy có dấu đều tương ứng với một lệnh nhảy không dấu. Ví dụ lệnh nhảy có dấu JG và lệnh nhảy không dấu JA. Dùng lệnh nhảy có dấu hay không dấu tuỳ thuộc vào kiểu số được đưa ra. Trên thực tế, như bảng 6.1 chỉ ra, các lệnh này thao tác với các cờ khác nhau: các lệnh nhảy có dấu sử dụng các cờ ZF,SF và OF trong khi các lệnh nhảy không dấu lại dùng các cờ ZF và CF. Sử dụng không đúng loại có thể đưa đến kết quả sai.

Ví dụ. giả thiết rằng chúng ta làm việc với các số không dấu. Nếu như AX=7FFFh và BX=8000h, khi ta thực hiện:

```
CMP     AX,BX
JA      BELOW
```

thậm chí 7FFFh > 8000h trong dạng có dấu, chương trình vẫn không nhảy đến nhãn BELOW. Nguyên nhân ở đây là 7FFFh < 8000h ở dạng không dấu và ở đây chúng ta lại dùng lệnh nhảy không dấu JA.

Làm việc với các ký tự.

Khi làm việc với tập hợp ký tự ASCII chuẩn cả các lệnh nhảy có điều kiện và không điều kiện đều có thể được sử dụng bởi lẽ bit dấu của byte chứa mã ký tự luôn là 0. Dù sao thì các lệnh nhảy không dấu phải được sử dụng khi so sánh các ký tự ASCII mở rộng (mã từ 80h đến FFh).

Ví dụ 6.1.

Giả sử AX và BX chứa các số có dấu. Hãy viết các lệnh để đưa số lớn nhất vào CX.

Trả lời:

```
            MOV   CX,AX        ; đưa AX vào CX
            CMP   BX,CX        ; BX lớn hơn?
            JLE   NEXT         ; không, tiếp tục
            MOV   CX,BX        ; đúng, đưa BX vào CX
NEXT:
```

6.3. Lệnh JMP.

Lệnh JMP (jump) dẫn đến việc chuyển điều khiển không điều kiện (nhảy không điều kiện). Cú pháp:

```
JMP   đích
```

Ở đây đích phải là một nhãn trong cùng một đoạn với JMP (xem phụ lục F với chỉ dẫn chi tiết hơn).

JMP có thể được dùng để khắc phục khoảng giới hạn của lệnh nhảy có điều kiện. Ví dụ: giả sử chúng ta muốn thực hiện vòng lặp:

```
TOP:
; thân vòng lặp
          DEC   CX          ; giảm bộ đếm
          JNZ   TOP         ; lặp nếu CX >0
          MOV   AX,BX
```

nhưng thân vòng lặp lại chứa quá nhiều lệnh đến mức nhãn TOP nằm ngoài khoảng giới hạn của lệnh JNZ (nhiều hơn 126 byte trước JNZ TOP). Chúng ta có thể sửa lại:

```
TOP:
; thân vòng lặp
          DEC   CX          ; giảm bộ đếm
```

```
            JNZ     BOTTOM          ; lặp nếu CX >0
            JMP     EXIT
BOTTOM:     JMP     TOP
EXIT:       MOV     AX,BX
```

6.4. Các cấu trúc ngôn ngữ bậc cao.

Chúng tôi đã có lần nói rằng cấu trúc nhảy có thể được dùng để thực hiện các công việc rẽ nhánh và lặp. Tuy nhiên do các lệnh nhảy quá sơ khai nên rất khó mã hoá một thuật toán (có hoặc không có các dòng hướng dẫn) nhất là đối với những người mới lập trình.

Bởi vì đa số các bạn đã có chút ít kinh nghiệm về các cấu trúc của ngôn ngữ bậc cao như cấu trúc chọn lựa IF_THEN_ELSE hay các vòng lặp WHILE, chúng tôi sẽ nêu ra cách giả lập các cấu trúc này trong ngôn ngữ Hợp ngữ. Trong trường hợp đầu tiên chúng tôi sẽ đưa ra cấu trúc các toán tử giả của ngôn ngữ bậc cao.

6.4.1 Các cấu trúc rẽ nhánh.

Trong các ngôn ngữ bậc cao, các cấu trúc rẽ nhánh của một chương trình để chọn các đường dẫn khác nhau và phụ thuộc vào các điều kiện. Phần này chúng ta sẽ xem xét 3 cấu trúc.

IF_THEN

Cấu trúc IF_THEN có thể được khai báo dưới dạng toán tử giả như sau:

```
IF điều_kiện
    THEN
          nhánh_đúng
    END_IF
```

Xem hình 6.2.

Điều_kiện là một biểu thức có thể đúng hoặc sai. Nếu nó đúng, nhánh_đúng sẽ được thực hiện. Ngược lại, cấu trúc không thực hiện lệnh nào, chương trình tiếp tục với các lệnh theo sau.

Ví dụ 6.2. Thay số trong AX bằng giá trị tuyệt đối của nó.

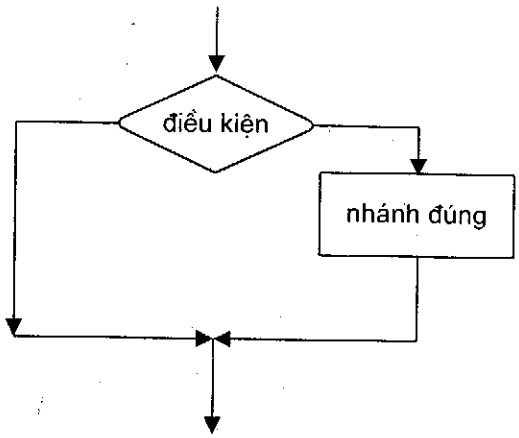
Trả lời : Một thuật toán với mã lệnh giả:

```
IF  AX>0
THEN
     thay AX bằng -AX
END_IF
```

Nó có thể được mã hoá như sau:

```
; if  AX<0
    CMP    AX,0          ; AX<0 ?
    JNL    END_IF        ; không, thoát ra.
; then
    NEG    AX            ; đúng, đổi dấu
END IF:
```

Hình 6.2. IF THEN



Điều kiện AX < 0 được kiểm tra bởi lệnh CMP AX,0. Nếu AX không nhỏ hơn 0, ta không phải làm gì cả, JNL được dùng để nhảy qua lệnh NEG AX. nếu điều kiện AX < 0 thoả mãn, chương trình tiếp tục thực hiện lệnh NEG AX.

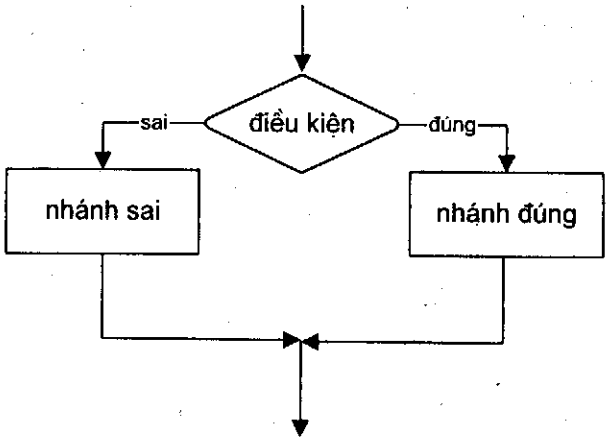
IF_THEN_ELSE.

```
IF  điều_kiện
THEN
    nhánh_đúng
ELSE
    nhánh_sai
END_IF
```

Xem hình 6.3

Trong cấu trúc này nếu điều_kiện là đúng, nhóm lệnh nhánh_đúng sẽ được thi hành. Còn nếu điều_kiện sai, nhóm lệnh nhánh_sai sẽ được thi hành.

Hình 6.3. IF_ THEN_ELSE



Ví dụ 6.3. Giả sử AL và BL chứa các ký tự ASCII mở rộng. Hãy hiển thị ký tự đứng trước trong bảng mã.

Trả lời :

```
IF  AL<=BL
    THEN
        hiển thị ký tự trong AL
ELSE
        hiển thị ký tự trong BL
END_IF
```

Ta có thể mã hoá nó như sau:

```
        MOV    AH,2          ;chuẩn bị hiển thị
;if  AL<=BL
        CMP    AL,BL         ; AL<=BL  ?
        JNBE   ELSE_         ;không, hiển thị ký tự trong BL
;then                        ;AL<=BL
        MOV    DL,AL         ;chuyển ký tự vào DL.để hiển thị
        JMP    DISPLAY       ;tới DISPLAY
ELSE_:                       ; BL < AL
        MOV    DL,BL         ;chuyển ký tự vào DL để hiển thị
DISPLAY:
        INT    21h           ; hiển thị nó.
;END_IF
```

Chú ý: Ta dùng nhãn ELSE_ vì ELSE là từ dành riêng.

Điều kiện AL<=BL được kiểm tra bởi lệnh CMP AL,BL. Nếu nó sai, chương trình sẽ nhảy qua nhánh_đúng tới ELSE_. Chúng ta sử dụng lệnh nhảy không dấu JNBE bởi lẽ chúng ta đang so sánh các kí tự mở rộng.

Nếu AL<=BL thoả mãn, nhánh_đúng được thực hiện. Lưu ý rằng chỉ thị JMP DISPLAY là cần thiết để nhảy qua nhánh_sai. Điều này khác trong với ngôn ngữ bậc cao: nhánh_sai được tự động nhảy qua nếu nhánh-đúng được thực hiện.

CASE.

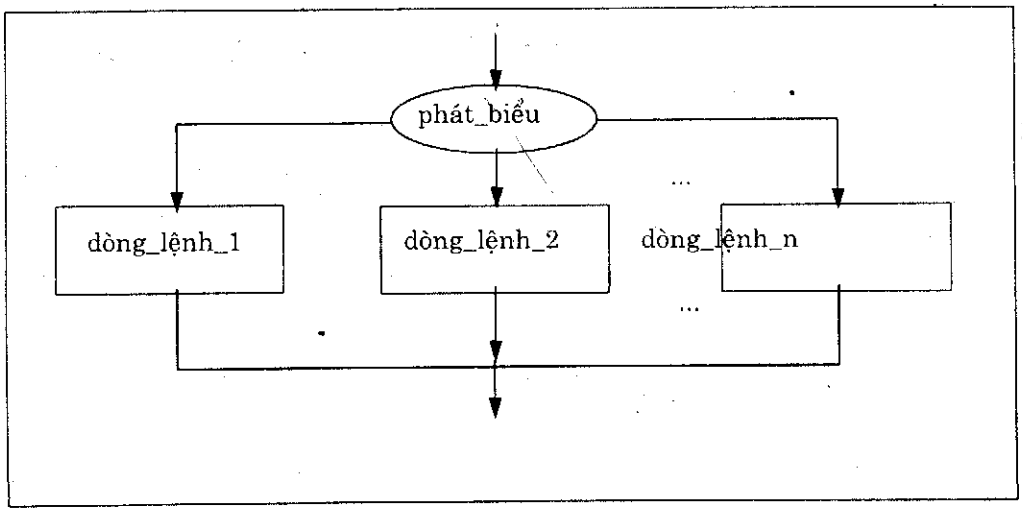
CASE là một cấu trúc đa nhánh, nó kiểm tra các thanh ghi, các biến hay các biểu thức với các giá trị riêng rẽ trong miền giá trị. Dạng tổng quát của nó là:

```
CASE  phát_biểu
    giá_trị_1: dòng_lệnh_1
    giá_trị_2: dòng_lệnh_2
    .
    .
    giá_trị_n: dòng_lệnh_n
END_CASE
```

Xem hình 6.4

Trong cấu trúc này phát_biểu được kiểm tra, nếu giá trị của nó bằng với giá_trị_i thì dòng_lệnh _i sẽ được thi hành. Ta giả thiết tập hợp giá_trị_1... giá_trị_n tách biệt nhau

Hình 6.4. CASE



Ví dụ 6.4. Nếu AX chứa một số âm, hãy nhập -1 vào BX, nếu AX chứa 0, cho BX bằng 0, nếu AX dương đổi BX thành 1.

Lời giải:

```
CASE AX
    <0: gán BX bằng -1
    =0: gán BX bằng 0
    >0: gán BX bằng 1
END_CASE
```

Ta có thể mã hoá như sau:

```
;case AX
        CMP     AX,0            ; kiểm tra AX
        JL      NEGATIVE        ; AX<0
        JE      ZERO            ; AX=0
        JG      POSITIVE        ; AX>0
NEGATIVE:
        MOV     BX,-1           ; nhập -1 vào BX
        JMP     END_CASE        ; rồi thoát
ZERO:
        MOV     BX,0            ; nhập 0 vào BX
        JMP     END_CASE        ; rồi thoát
POSITIVE:
        MOV     BX,1            ; nhập 1 vào BX
END_CASE:
```

Các nhánh với điều kiện kép.

Đôi khi điều kiện nhánh của IF hay CASE có dạng:

```
điều_kiện_1  AND  điều_kiện_2
```

hay

```
điều_kiện_1  OR   điều_kiện_2
```

Ở đây điều_kiện_1 và điều_kiện_2 có thể đúng hoặc sai. Đầu tiên chúng ta hãy xem xét điều kiện AND (AND condition), sau đó đến điều kiện OR (OR condition).

Các điều kiện AND.

Điều kiện AND chỉ đúng khi cả hai điều kiện: điều_kiện_1 và điều_kiện_2 cùng đúng. Ngược lại nếu một trong chúng sai, điều kiện AND cũng sẽ sai.

Ví dụ 6.6. Đọc một ký tự. Nếu là chữ hoa thì hiển thị nó.

Lời giải:

```
Đọc một ký tự ( vào DL)
IF  ( 'A' <= ký_tự ) và ( ký_tự <= 'Z' )
THEN
    hiển thị ký tự
END_IF
```

Để mã hoá, đầu tiên chúng ta kiểm tra xem ký tự trong AL có đứng sau 'A' trong bảng mã hay không, nếu sai ta có kết thúc. Nếu đúng, trước khi hiển thị ký tự ta vẫn còn phải kiểm tra ký tự có đứng trước 'Z' hay không. Sau đây là mã lệnh:

```
;đọc một ký tự
        MOV   AH,1          ; chuẩn bị đọc
        INT   21h           ; ký tự vào AL
;if  ( 'A' <= ký_tự ) và ( ký_tự <= 'Z' )
        CMP   AL,'A'        ; ký_tự >='A' ?
        JNGE  END_IF        ; không, thoát ra
        CMP   AL,'Z'        ; kýtự <='Z' ?
        JNLE  END_IF        ; không, thoát ra
; then hiển thị ký tự
        MOV   DL,AL         ; lấy ký tự
        MOV   AH,2          ; chuẩn bị hiển thị
        INT   21h           ; hiển thị ký tự
END_IF:
```

Các điều kiện OR.

Điều_kiện_1 OR điều_kiện_2 là đúng khi điều_kiện_1 hoặc điều_kiện_2 đúng. Nó chỉ sai khi cả hai điều kiện thành phần cùng sai.

Ví dụ 6.7. Đọc một ký tự. Nếu là 'y' hay 'Y' thì hiển thị nó. Nếu ngược lại, kết thúc chương trình.

Lời giải:

```
Đọc một ký tự (vào AL).
IF  (ký_tự = 'y') hoặc (ký_tự ='Y')
THEN
    hiển thi ký tự
ELSE
    kết thúc chương trình.
END_IF
```

Để mã hoá, đầu tiên chúng ta kiểm tra ký_tự = 'y' ?. Nếu thoả mãn, điều kiện OR đúng và chúng ta có thể thực hiện dòng lệnh THEN. Ngược lại vẫn cơ hội để điều kiện OR đúng, đó là khi ký_tự bằng 'Y', và dòng lệnh THEN được thi hành. Nếu điều này vẫn sai, điều kiện OR là sai và chúng ta sẽ thực hiện dòng lệnh ELSE. Sau đây là mã lệnh:

```
;đọc một ký tự
        MOV   AH,1          ; chuẩn bị đọc
        INT   21h           ; ký tự trong AL
; if  ( ký_tự = 'y') hoặc ( ký_tự ='Y' )
        CMP   AL,'y'        ; ký_tự = 'y' ?
        JE    THEN          ; đúng, chuyển đến hiển thị ký tự
        CMP   AL,'Y'        ; ký_tự = 'Y'
        JE    THEN          ; đúng, chuyển đến hiển thị ký tự
        JMP   ELSE_         ; sai, kết thúc
THEN:
        MOV   AH,2          ; chuẩn bị hiển thị
        MOV   DL,AL         ; lấy ký tự
        INT   21h           ; hiển thị nó
        JMP   END_IF        ; và thoát ra
ELSE_:
        MOV   AH,4Ch        ;
```

```
        INT    21h              ; trở về DOS
END_IF:
```

6.4.2 Các cấu trúc lặp.

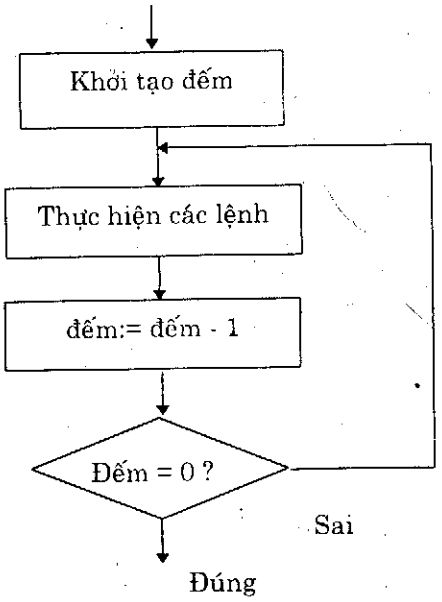
Một vòng lặp là một chuỗi các lệnh được lặp lại. Số lần lặp có thể đã xác định trước hoặc phụ thuộc vào các điều kiện.

Vòng lặp FOR.

Đây là một cấu trúc lặp mà số lần lặp lại các dòng lệnh đã biết trước (vòng lặp điều khiển bằng biến đếm). Dạng mã lệnh giả:

```
FOR  số_lần_lặp  DO
    các dòng lệnh
END_ FOR
```

Xem hình 6.5



Hình 6.5: Vòng FOR

Ta có thể sử dụng lệnh LOOP để thực hiện vòng lặp FOR. Lệnh này có dạng:

```
LOOP  nhãn_đích
```

Bộ đếm vòng lặp là thanh ghi CX được khởi tạo bằng số_lần_lặp. Mỗi lần thực hiện lệnh LOOP , thanh ghi CX tự động giảm đi 1 và nếu CX khác 0 thì điều khiển được chuyển tới nhãn đích. Nếu CX = 0, lệnh tiếp sau lệnh LOOP sẽ được thi hành. Nhãn đích phải ở trước lệnh lặp không quá 126 byte.

Vòng lặp FOR có thể được thực hiện nhờ lệnh LOOP như sau:

```
    ;khởi tạo CX bằng số_lần_lặp
TOP:
    ; thân vòng lặp
    LOOP  TOP
```

Ví dụ 6.8. Viết một vòng lặp điều khiển bằng biến đếm hiển thị một dòng 80 dấu sao.

Lời giải:

```
FOR  80 times DO
    hiển thị '*'
END_FOR
```

Mã lệnh là:

```
    MOV   CX,80        ; số các dấu sao được hiển thị
    MOV   AH,2         ; hàm hiển thị ký tự
    MOV   DL,'*'       ; ký tự hiển thị
TOP:
    INT   21h          ; hiển thị một dấu sao
    LOOP  TOP          ; lặp lại 80 lần
```

Bạn hãy lưu ý rằng vòng lặp FOR thực hiện bởi lệnh LOOP sẽ được thực hiện ít nhất một lần. Thực ra nếu CX bằng 0 khi vào vòng lặp, lệnh LOOP giảm CX thành 0FFFFh và lệnh LOOP sẽ được thực hiện 0FFFFh = 65535 lần nữa. Để khắc phục điều này, lệnh JCXZ (jump if CX is zero) được đặt trước vòng lặp. Cú pháp của nó là:

```
JCXZ  nhãn_đích
```

Nếu CX bằng 0, điều khiển sẽ được chuyển đến nhãn đích. Như vậy vòng lặp sẽ bị bỏ qua nếu CX bằng 0:

```
        JCXZ   SKIP
TOP:
        ;thân vòng lặp
        LOOP   TOP
SKIP:
```

Vòng lặp WHILE.

Đây là vòng lặp phụ thuộc vào một điều kiện. Dạng mã lệnh giả:

```
WHILE   điều_kiện  DO
    các dòng lệnh
END_WHILE
```

Xem hình 6.6

Điều_kiện được kiểm tra ở đầu vòng lặp. Nếu nó đúng thì các dòng lệnh sẽ được thi hành. Ngược lại nếu điều_kiện sai, chương trình tiếp tục thực hiện lệnh ở sau vòng lặp. Rất có thể ngay khi khởi đầu điều_kiện đã không thoả mãn. Trong trường hợp này thân vòng lặp sẽ không được thực hiện lần nào. Vòng lặp tiếp tục được thực hiện khi điều kiện còn đúng.

Ví dụ 6.9. Viết các lệnh để đếm số ký tự trong một dòng.

Lời giải:

```
khởi tạo bộ đếm bằng 0,
đọc một ký tự
WHILE  ký tự <> ký tự về đầu dòng  DO
    đếm = đếm + 1
    đọc một ký tự
END_WHILE
```

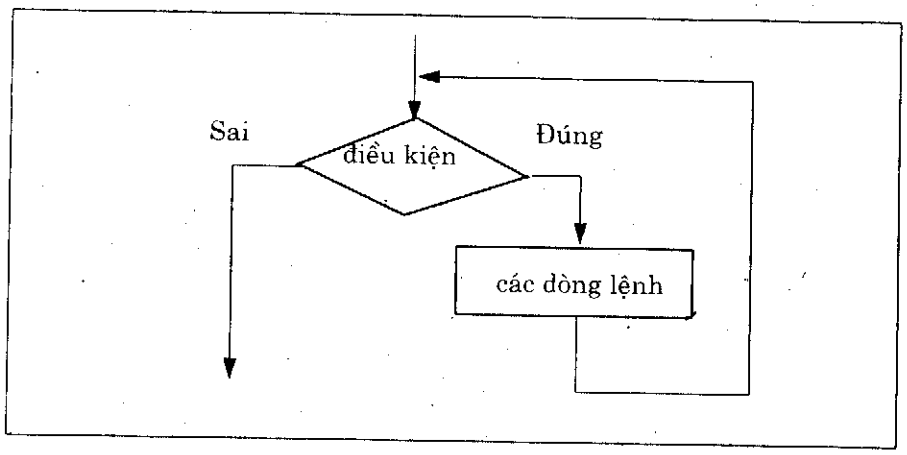
Các lệnh là:

```
        MOV    DX,0              ; DX đếm số ký tự
        MOV    AH,1              ; chuẩn bị đọc
```

```
        INT     21h                 ; ký tự trong AL
WHILE_:
        CMP     AL,0Dh              ; CR ?
        JE      END_WHILE           ; đúng, thoát ra
        INC     DX                  ; không phải CR, tăng bộ đếm
        INT     21h                 ; đọc một ký tự
        JMP     WHILE_              ; lặp lại
END_WHILE:
```

Lưu ý là do vòng lặp WHILE kiểm tra điều kiện kết thúc ở đầu vòng lặp nên bạn cần chắc chắn rằng bất cứ biến nào liên quan đến điều kiện vòng lặp đều phải được khởi tạo trước khi vào vòng lặp. Vì vậy bạn phải đọc một ký tự trước khi vào vòng lặp rồi lại phải đọc ký tự khác ở cuối nó. Ta dùng nhãn WHILE_ vì WHILE là từ dành riêng.

Hình 6.6. Vòng lặp WHILE



Vòng lặp REPEAT

Có một vòng lặp có điều kiện khác đó là vòng lặp REPEAT. Dạng mã lệnh giả của nó là:

```
REPEAT
    các dòng lệnh
UNTIL   điều_kiện
```

Xem hình 6.7.

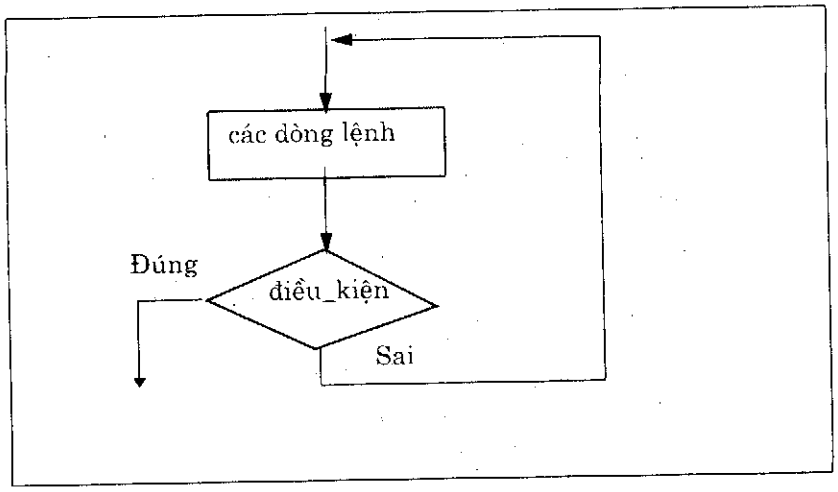
Trong một vòng lặp REPEAT ...UNTIL, các dòng lệnh được thi hành sau đó mới kiểm tra điều kiện. Nếu điều_kiện đúng, vòng lặp kết thúc, nếu nó sai điều khiển rẽ nhánh đến đầu vòng lặp.

Ví dụ 6.10. Viết các lệnh đọc vào các ký tự, kết thúc khi gặp một ký tự trắng.

Lời giải:

```
REPEAT
    đọc một ký tự
UNTIL  ký tự trắng
```

Hình 6.7 Vòng lặp REPEAT



Các lệnh là:

```
        MOV    AH,1          ; chuẩn bị đọc
REPEAT:
        INT    21h           ; ký tự trong AL
;until
        CMP    AL,' '        ; ký tự trắng ?
        JNE    REPEAT        ; không, đọc tiếp
```

So sánh WHILE và REPEAT.

Trong nhiều trường hợp khi cần một vòng lặp có điều kiện, sử dụng vòng lặp WHILE hay REPEAT là tuỳ ý thích mỗi người. Ưu điểm của vòng lặp WHILE là vòng lặp có thể được bỏ qua khi điều kiện kết thúc khởi tạo với giá trị lôgic sai, trong khi đó các lệnh trong vòng lặp REPEAT luôn được thực hiện ít nhất một lần. Tuy nhiên các lệnh cho một vòng lặp REPEAT có vẻ ngắn hơn đôi chút bởi lẽ nó chỉ có một lệnh nhảy có điều kiện ở cuối trong khi vòng lặp WHILE có những hai: một lệnh nhảy có điều kiện ở đầu và lệnh JMP ở cuối.

6.5. Lập trình với các cấu trúc bậc cao.

Để chỉ ra một chương trình có thể được phát triển từ các toán tử giả bậc cao thành các lệnh Hợp ngữ (Assembly) như thế nào, chúng ta hãy giải quyết vấn đề sau đây:

Yêu cầu:

Thông báo cho người sử dụng nhập vào một dòng văn bản. Hiển thị trên dòng tiếp theo chữ hoa đầu tiên và sau cùng tính theo thứ tự mã ASCII của chuỗi vừa nhập. Nếu không có chữ hoa nào được nhập vào thì hiển thị thông báo ' Không có chữ hoa ! '. Một ví dụ khi thực hiện chương trình:

```
Ban hãy vào một dòng văn bản:
DONG CHAY THU CHUONG TRINH!
Chữ hoa đầu tiên = A  Chữ hoa sau cùng = U
```

Để giải quyết vấn đề này chúng ta sẽ sử dụng phương pháp thiết kế chương trình top_down (từ trên xuống) mà bạn đã có thể gặp trong lập trình ngôn ngữ bậc cao. Trong phương pháp này, vấn đề nguyên thuỷ được chia thành một chuỗi các vấn đề con mà thực hiện mỗi trong chúng đơn giản hơn nhiều so với vấn đề ban đầu. Mỗi vấn đề con lại có thể được chia nhỏ hơn nữa ... Cứ tiếp tục như thế cho đến khi mỗi vấn đề con có thể được mã hoá trực tiếp.Việc sử dụng các chương trình con có thể phát triển phương pháp này

Sự phân chia đầu tiên:

1. Hiển thị thông báo ban đầu.

2. Đọc và xử lý dòng văn bản.

3. Hiển thị kết quả.

Bước 1. Hiển thị thông báo ban đầu.

Bước này có thể mã hoá ngay tức khắc:

```
MOV   AH,9          ; chức năng hiển thị chuỗi
LEA   DX,PROMPT     ; lấy thông báo ban đầu
INT   21h           ; hiển thị nó
```

Thông báo có thể chứa trong đoạn dữ liệu như sau:

```
PROMPT  DB 'Ban hãy vào một dòng văn bản',0Dh,0Ah ,'$'
```

Ta đưa vào cả ký tự xuống dòng và về đầu dòng để chuyển con trỏ xuống đầu dòng tiếp theo và như vậy người sử dụng có thể đánh vào toàn bộ một dòng văn bản.

Bước 2. Đọc và xử lý dòng văn bản.

Bước này thực hiện hầu hết các công việc của chương trình. Nó thực hiện nhập từ bàn phím, trả về các chữ cái thoả mãn (nó cũng đưa ra thông báo nếu không có chữ hoa nào được đọc vào). Sau đây là các bước thực hiện:

```
Đọc một ký tự
WHILE  không phải ký tự xuống dòng  DO
    IF  ký tự là chữ hoa (*)
    THEN
          IF  đứng trước chữ hoa đầu
          THEN
                chữ hoa đầu = ký tự
          END_IF
                IF  đứng sau chữ hoa cuối
          THEN
                chữ hoa cuối = ký tự
          END_IF
    EDN_IF
    đọc một ký tự
END_WHILE
```

Dòng (*) thực chất là một điều kiện AND:

```
IF  ( 'A' <= ký tự)  AND ( ký tự <='Z' )
```

Bước 2 có thể được mã hoá như sau:

```
        MOV    AH,1                ; hàm đọc ký tự
        INT    21h                 ; ký tự trong AL
WHILE_:
; while  không phải ký tự xuống dòng  do
        CMP    AL,0Dh              ; CR ?
        JE     END_WHILE           ; đúng, thoát ra
;if ký tự là chữ hoa
        CMP    AL,'A'              ; ký tự >= 'A' ?
        JNGE   END_IF              ; không phải chữ hoa
        CMP    AL,'Z'              ; ký tự <= 'Z' ?
        JNLE   END_IF              ; không phải chữ hoa
;then
;if ký tự đứng trước chữ hoa đầu
        CMP    AL,FIRST            ; ký tự < FIRST ?
        JNL    CHECK_LAST          ; không, kiểm tra tiếp
;then   chữ hoa đầu = ký tự
        MOV    FIRST,AL            ; FIRST = ký tự
;end_if
CHECK_LAST:
;if ký tự đứng sau chữ hoa cuối
        CMP    AL,LAST             ; ký tự > LAST ?
        JNG    END_IF              ; không, cho qua
;then   chữ hoa cuối = ký tự
        MOV    LAST,AL             ; LAST = ký tự
;end_if
END_IF:
;đọc một ký tự
        INT    21h                 ; ký tự trong AL
        JMP    WHILE_              ; lặp lại
END_WHILE:
```

Các biến FIRST và LAST phải được khởi tạo trước khi vào vòng lặp WHILE. Chúng có thể được định nghĩa trong đoạn dữ liệu như sau:

```
FIRST       DB     '['
LAST        DB     '@'
```

Các giá trị khởi tạo '[' và '@' được chọn bởi vì '[' đứng sau 'Z' và '@' đứng trước 'A'. Theo cách này, chữ hoa đầu tiên được đưa vào sẽ làm thay đổi giá trị của cả hai biến.

Với bước 2 đã được mã hoá, chúng ta có thể tiếp tục đến bước cuối cùng.

Bước 3. Hiển thị kết quả:

```
IF  không có chữ hoa
THEN
    hiển thị  'Không có chữ hoa ! '
ELSE
    hiển thị chữ hoa đầu và cuối.
END_IF
```

Bước này có thể hiển thị một trong hai thông báo: NOCAP_MSG nếu không có chữ hoa nào được đánh vào hay CAP_MSG nếu ngược lại. Chúng ta có thể khai báo chúng trong đoạn dữ liệu như sau:

```
NOCAP_MSG DB          ' Không có chữ hoa ! $'
CAP_MSG       DB      ' Chữ hoa đầu tiên = '
FIRST         DB      '['
              DB      ' Chữ hoa sau cùng = '
LAST          DB      '@ $ '
```

Khi CAP_MSG được hiển thị, nó sẽ đưa ra thông báo ' Chữ hoa đầu tiên = ', sau đó là giá trị của FIRST rồi đến 'Chữ hoa cuối cùng = 'và giá trị của LAST. Chúng ta đã sử dụng kỹ thuật này trong chương trình cuối cùng của chương 4.

Chương trình của chúng ta sẽ kiểm tra biến FIRST biết có chữ hoa nào được đọc vào hay không. Nếu FIRST chứa '[' là giá trị khởi đầu của nó thì có nghĩa là không đọc vào chữ hoa nào cả.

Bước 3 có thể được mã hoá như sau:

```
        MOV   AH,9          ; hàm hiển thị chuỗi
;if  không có chữ hoa
        CMP   FIRST ,'['    ; FIRST = '[' ?
        JNE   CAPS          ; không, hiển thị kết quả
;then
        LEA   DX,NOCAP_MSG
        JMP   DISPLAY
CAPS:
```

```
        LEA    DX,CAP_MSG
DISPLAY:
        INT    21h             ; hiển thị thông báo
;end if
```

Sau đây là chương trình đầy đủ:

Chương trình PGM6_2.ASM

```
    TITLE    PGM6_2: Chữ hoa đầu và cuối.
    .MODEL   SMALL
    .STACK   100H
    .DATA
    PROMPT DB' Bạn hãy vào một dòng văn bản',0Db,0Ah,'$'
    NOCAP_MSG DB    0DH,0AH,' Không có chữ hoa ! $'
    CAP_MSG  DB     ' Chữ hoa đầu tiên = '
    FIRST    DB     '?'
             DB     ' Chữ hoa sau cùng = '
    LAST     DB     '@ $ '
    .CODE
    MAIN     PROC
; khởi tạo DS
        MOV    AX,@DATA
        MOV    DS,AX
; hiển thị thông báo ban đầu
        MOV    AH,9            ; chức năng hiển thị chuỗi
        LEA    DX,PROMPT       ; lấy thông báo ban đầu
        INT    21h             ; hiển thị nó
; đọc và xử lý một dòng văn bản
        MOV    AH,1            ; hàm đọc ký tự
        INT    21h             ; ký tự trong AL
WHILE :
; while không phải ký tự xuống dòng do
        CMP    AL,0Dh          ; CR ?
        JE     END_WHILE       ; đúng, thoát ra
;if ký tự là chữ hoa
        CMP    AL,'A'          ; ký tự >= 'A' ?
        JNGE   END_IF          ; không phải chữ hoa
        CMP    AL,'Z'          ; ký tự <= 'Z' ?
        JNLE   END_IF          ; không phải chữ hoa
```

```
;then
;if ký tự đứng trước chữ hoa đầu
        CMP    AL,FIRST      ; ký tự < FIRST ?
        JNL    CHECK_LAST    ; không, kiểm tra tiếp
;then  chữ hoa đầu = ký tự
        MOV    FIRST,AL      ; FIRST = ký tự
;end_if
CHECK_LAST:
;if ký tự đứng sau chữ hoa cuối
        CMP    AL,LAST       ; ký tự > LAST ?
        JNG    END_IF        ; không, cho qua
;then  chữ hoa cuối = ký tự
        MOV    LAST,AL       ; LAST = ký tự
;end_if
END_IF:
;đọc một ký tự
        INT    21h           ; ký tự trong AL
        JMP    WHILE_        ; lặp lại
END_WHILE:
; hiển thị kết quả
        MOV    AH,9          ; hàm hiển thị chuỗi
;if  không có chữ hoa
        CMP    FIRST ,'['    ; FIRST = '[' ?
        JNE    CAPS          ; không, hiển thị kết quả
;then
        LEA    DX,NOCAP_MSG
        JMP    DISPLAY
CAPS:
        LEA    DX,CAP_MSG
DISPLAY:
        INT    21h           ; hiển thị thông báo
;end_if
;trở về DOS
        MOV    AH,4CH
        INT    21h
MAIN    ENDP
        END    MAIN
```

TỔNG KẾT.

- Có hai loại lệnh nhảy: lệnh nhảy có điều kiện và lệnh nhảy không điều kiện. Lệnh nhảy có điều kiện được chia thành các lệnh nhảy có dấu, không dấu và các lệnh nhảy điều kiện đơn.
- Các lệnh nhảy có điều kiện được thực hiện dựa vào việc thiết lập các cờ trạng thái. Lệnh CMP (compare) chỉ dùng để thiết lập cờ trước các lệnh nhảy.
- Nhãn đích của lệnh nhảy có điều kiện phải đứng trước không quá 126 byte hoặc đứng sau không hơn 127 byte kể từ lệnh đó. Một lệnh JMP thường được dùng để nhảy qua giới hạn này.
- Trong một cấu trúc chọn lựa IF-THEN, nếu điều kiện kiểm tra là đúng, các dòng lệnh nhánh_đúng sẽ được thực hiện. Trong trường hợp ngược lại, dòng lệnh theo sau cấu trúc được thực hiện.
- Trong một cấu trúc chọn lựa IF-THEN_ELSE, nếu điều kiện kiểm tra là đúng các dòng lệnh nhánh_đúng sẽ được thực hiện. Trong trường hợp ngược lại các dòng lệnh nhánh_sai sẽ được thực hiện. Cần phải có lệnh JMP ở cuối nhánh_đúng để nhảy qua các dòmg lệnh của nhánh_sai.
- Trong một cấu trúc CASE, việc rẽ nhánh được điều khiển bởi một biểu thức. Các nhánh tương ứng với các giá trị có thể của biểu thức.
- Vòng lặp FOR được thực hiện với số lần lặp biết trước. Nó có thể được tạo lập bởi lệnh LOOP. Trước khi vào vòng lặp, CX phải được khởi tạo bằng số lần lặp lại của vòng lặp.
- Trong một vòng lặp WHILE, điều kiện lặp được kiểm tra ở đầu vòng lặp. Các dòng lệnh của vòng lặp được lặp lại khi điều kiện đúng. Nếu điều kiện khởi đầu là sai, các dòng lệnh này sẽ không được thực hiện lần nào.
- Trong một vòng lặp REPEAT, điều kiện lặp được kiểm tra ở cuối vòng lặp. Các dòng lệnh của vòng lặp được lặp lại đến khi điều kiện đúng. Do điều kiện kiểm tra ở cuối vòng lặp nên các dòng lệnh của vòng lặp này sẽ được thực hiện ít nhất một lần.

Thuật ngữ tiếng Anh

AND condition Phép **và** lôgic hai điều kiện.

condition jump instruction Lệnh nhảy có điều kiện. Một lệnh nhảy có điều kiện được thực hiện dựa vào việc thiết lập các cờ trạng thái.

loop Vòng lặp. Một chuỗi các lệnh được lặp lại.

OR condition Phép **hoặc** lôgic hai điều kiện.

signed jump Nhảy có dấu. Lệnh nhảy có điều kiện dùng với các số có dấu.

single_flag jump Nhảy điều kiện đơn. Lệnh nhảy có điều kiện thao tác dựa trên một cờ riêng biệt.

top_down program design Thiết kế chương trình từ trên xuống. Khai triển chương trình bằng cách phân nhỏ một vấn đề lớn thành rất nhiều các vấn đề đơn giản hơn.

uncondition jump Nhảy không điều kiện. Sự chuyển điều khiển không điều kiện.

unsigned jump Nhảy không dấu. Lệnh nhảy có điều kiện dùng với các số không dấu.

Các lệnh mới:

CMP	JCXZ	JLE/ING
JA/JNBE	JE/JZ	JMP
JAE/JNB	JG/JNLE	JNC
JB/JNAE	JGE/JNL	JNE/JNZ
JC	JL/JNLE	LOOP
JBE/JNA		

Bài tập:

1. Viết các lệnh Hợp ngữ cho mỗi cấu trúc chọn lựa sau đây:

a.
```
IF  AX <0
        THEN
        BX = -1
    END_IF
```

b.
```
IF  AL <0
    THEN
          AH = 0FFh
     ELSE
          AH = 0
    END_IF
```

c. Giả thiết DL chứa mã ASCII của một ký tự:
```
IF  ( DL >= 'A' )  AND  ( DL <= 'Z' )
THEN
      Hiển thị DL
END_IF
```

d.
```
IF   AX < BX
THEN
      IF  BX < CX
          THEN
                AX = 0
          ELSE
                BX = 0
      END_IF
 END_IF
```

e.
```
IF   ( AX < BX )  OR  (BX <CX )
 THEN
      DX = 0
 ELSE
      DX = 1
END_IF
```

f.
```
IF   AX < BX
 THEN
```

```
        AX = 0
ELSE
        IF  BX < CX
                THEN
                        BX = 0
                ELSE
                        CX = 0
        END_IF
END_IF
```

2. Dùng cấu trúc CASE để mã hoá các công việc sau đây:

Đọc một ký tự.

Nếu là 'A', chuyển con trỏ về đầu dòng.

Nếu là 'B', chuyển con trỏ xuống dòng.

Nếu là một ký tự khác, trở về DOS.

3. Viết các lệnh thực hiện các công việc sau đây:

a. AX = 1 + 4 + 7 + ... + 148.

b. AX = 100 + 95 + ... + 5.

4. Dùng các lệnh LOOP thực hiện các công việc sau đây:

a. Nhập tổng của 50 phần tử đầu tiên của dãy số học: 1, 5, 9, 13 ... vào DX.

b. Đọc một ký tự và hiển thị nó 80 lần trên dòng tiếp theo.

c. Đọc một mật khẩu 5 ký tự và viết đè lên nó bằng cách trở về đầu dòng rồi hiển thị 5 chữ X. Bạn không cần phải lưu giữ lại các ký tự này.

5. Thuật toán sau đây có thể sử dụng để chia hai số dương bằng cách lặp lại phép trừ:

```
khởi động thương số bằng 0
WHILE  số bị chia >= số chia  DO
        tăng thương số
        trừ bớt số chia từ số bị chia
END WHILE
```

Hãy viết các lệnh thực hiện chia AX cho BX rồi cất thương trong CX.

6. Thuật toán sau đây có thể sử dụng để nhân hai số dương M và N bằng cách lặp lại phép cộng:

```
khởi động tích số bằng 0
REPEAT
      cộng M vào tích số
      giảm N
UNTIL  N = 0
```

Hãy viết các lệnh thực hiện nhân AX với BX rồi cất tích trong CX. Bạn có thể bỏ qua trường hợp tràn.

7. Ta có thể tạo lập một vòng lặp điều khiển bởi biến đếm mà nó còn tiếp tục thi hành khi điều kiện được thoả mãn. Các lệnh:

```
LOOPE     nhãn          ; lặp khi bằng
```

và

```
LOOPZ     nhãn          ; lặp khi bằng ZERO
```

sẽ làm giảm CX. Sau đó nếu CX <> 0 và ZF = 1, điều khiển sẽ chuyển đến lệnh sau nhãn đích. Nếu như CX =0 hoặc ZF = 0, lệnh ở sau vòng lặp được thực hiện. Tương tự như vậy, các lệnh:

```
LOOPNE    nhãn          ; lặp khi khônh bằng
```

và

```
LOOPNZ    nhãn          ; lặp khi bằng khác ZERO
```

sẽ làm giảm CX. Sau đó nếu CX <> 0 và ZF = 0, điều khiển sẽ chuyển đến lệnh sau nhãn đích. Nếu như CX =0 hoặc ZF = 1, lệnh ở sau vòng lặp được thực hiện.

a. Hãy sử dụng vòng lặp LOOPE viết các lệnh đọc các ký tự đến khi một ký tự khác ký tự trắng được đánh vào hoặc đã nhập đủ 80 ký tự.

b. Dùng vòng lặp LOOPNE viết các lệnh đọc các ký tự đến khi một ký tự về đầu dòng được đánh vào hoặc đã nhập đủ 80 ký tự.

Chương 7

CÁC LỆNH LOGIC, DỊCH VÀ QUAY

Tổng quan

Trong chương này chúng ta sẽ nghiên cứu về các chỉ thị dùng để thay đổi đến từng bit trong byte hay word. Khả năng thao tác với các bit thường không có ở trong các ngôn ngữ bậc cao (trừ ngôn ngữ C), và đó cũng là một lý do quan trọng để lập trình bằng Hợp ngữ .

Trong phần 7.1 chúng tôi sẽ giới thiệu các lệnh lôgic AND, OR, XOR và NOT. Các lệnh này có thể sử dụng để xoá, thiết lập và kiểm tra từng bit trong các thanh ghi hay các biến. Chúng ta sẽ sử dụng các lệnh này để thực hiện một số công việc đã quen thuộc như đổi chữ thường thành chữ hoa hay còn mới như xác định xem một thanh ghi chứa số chẵn hay lẻ.

Phần 7.2 sẽ trình bày về các lệnh dịch, các bit có thể được dịch phải hoặc trái trong thanh ghi hay trong một ô nhớ. Khi một bit bị dịch ra khỏi thanh ghi nó sẽ được chứa trong cờ CF. Vì dịch trái cũng có nghĩa là nhân đôi số và dịch phải có nghĩa là chia đôi nó, lệnh này cho phép chúng ta thực hiện phép nhân và chia cho một luỹ thừa của 2. Trong chương 9 chúng ta sẽ sử dụng các lệnh MUL và DIV để thực hiện các phép nhân và chia một cách tổng quát hơn, tuy nhiên các lệnh này chậm hơn nhiều so với các lệnh dịch.

Trong phần 7.3 chúng tôi sẽ giới thiệu các lệnh quay. Các lệnh này làm việc giống như các lệnh dịch ngoại trừ một điều là khi một bit bị ra khỏi một phía của toán hạng nó sẽ được đưa trở về phía kia của toán hạng đó. Các lệnh này có thể được sử dụng trong các trường hợp chúng ta muốn kiểm tra hoặc thay đổi các bit hay nhóm các bit.

Trong phần 7.4 chúng ta sẽ sử dụng các lệnh lôgic, dịch và lệnh quay để thực hiện các thao tác vào ra với số nhị phân và số hex. Khả năng đọc và viết các số cho phép chúng ta giải quyết rất nhiều vấn đề khác nhau.

7.1 Các lệnh lôgic

Như đã nói ở trên khả năng thao tác với từng bit riêng biệt là một trong những ưu điểm của hợp ngữ. Chúng ta có thể thay đổi từng bit trong máy tính bằng các lệnh lôgic. Các giá trị nhị phân 0 và 1 được xem như là các giá trị lôgic TRUE hoặc FALSE một cách tương ứng. Bảng 7.1 là bảng sự thật của các toán tử lôgic AND, OR, XOR(hoặc phủ định- exclusive OR) và NOT.

Khi lệnh lôgic được áp dụng với các toán hạng 8 và 16 bit kết quả nhận được bằng cách áp dụng chúng với từng bit một.

Ví dụ 7.1 Thực hiện các thao tác lôgic sau đây:

1. 10101010 AND 11110000
2. 10101010 OR 11110000
3. 10101010 XOR 11110000
4. NOT 10101010

Trả lời :

1.

```
    10101010
AND 11110000
------------
  = 10100000
```

2.

```
    10101010
 OR 11110000
------------
  = 11111010
```

3.

```
    10101010
XOR 11110000
------------
  = 01011010
```

4.

```
    10101010
------------
NOT 01010101
```

a	b	a AND b	a OR b	a XOR b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

a	NOT a
0	1
1	0

Bảng 7.1 Bảng sự thật của các lệnh lôgic AND, OR, XOR và NOT
(0 = TRUE, 1 = FALSE)

7.1.1 Các lệnh AND, OR và XOR

Các lệnh AND, OR và XOR thực hiện các thao tác lôgic như tên gọi của chúng. Cú pháp như sau:

```
AND     Đích, nguồn
OR      Đích, nguồn
XOR     Đích, nguồn
```

Kết quả của thao tác được chứa trong toán hạng đích, nó phải là một thanh ghi hay một ô nhớ. Toán hạng nguồn có thể là hằng số, thanh ghi hay ô nhớ. Tuy nhiên các thao tác giữa 2 ô nhớ là không hợp lệ.

Ảnh hưởng tới cờ:

```
SF, ZF, PF phản ánh kết quả của lệnh
AF không xác định
CF, OF = 0
```

Khi sử dụng các lệnh AND, OR và XOR chúng ta có thể thay đổi một cách có chọn lọc các bit của toán hạng đích. Để làm điều đó chúng ta tạo lên một mẫu bit được gọi là mặt nạ (MASK). Các bit của mặt nạ được chọn để sao cho các bit tương ứng của toán hạng đích được thay đổi đúng như chúng ta mong muốn khi lệnh lôgic được thực hiện.

Để chọn các bit mặt nạ chúng ta sử dụng các tính chất sau đây của các lệnh AND, OR và XOR. Từ hình 7.1 chúng ta thấy, nếu b biểu diễn một bit (0 hoặc 1) thì:

```
b  AND  1 = b, b  OR   0 = b,   b  XOR  0 = b
b  AND  1 = b, b  OR   1 = 1,   b  XOR  1 = ~b
                                          (bù của b)
```

Từ đó chúng ta rút ra kết luận sau:

1. Lệnh AND có thể sử dụng để xoá các bit nhất định của toán hạng đích trong khi giữ nguyên những bit còn lại. Bit 0 của mặt nạ xoá bit tương ứng, còn bit 1 của mặt nạ giữ nguyên bit tương ứng của toán hạng đích.
2. Lệnh OR có thể được dùng để thiết lập các bit xác định của toán hạng đích trong khi vẫn giữ nguyên những bit còn lại. Bit 1 của mặt nạ sẽ thiết lập bit tương ứng trong khi bit 0 của nó sẽ giữ nguyên bit tương ứng của toán hạng đích.
3. Lệnh XOR có thể dùng để đảo các bit xác định của toán hạng đích trong khi vẫn giữ nguyên những bit còn lại. Bit 1 của mặt nạ làm đảo bit tương ứng còn bit 0 giữ nguyên bit tương ứng của toán hạng đích.

Ví dụ 7.2 Xoá bit dấu của AL trong khi giữ nguyên các bit còn lại.

Trả lời:

Sử dụng lệnh AND với 0111111b = 7Fh làm mặt nạ : `AND  AL, 7Fh`

Ví dụ 7.3 Thiết lập các bit trọng lượng cao nhất và thấp nhất của AL trong khi giữ nguyên các bit còn lại

Trả lời :

Sử dụng lệnh OR với 10000001b = 81h làm mặt nạ : `OR  AL, 81h`

Ví dụ 7.4 Thay đổi bit dấu của DX

Trả lời :

Sử dụng lệnh XOR với mặt nạ 8000h chúng ta có: `XOR  DX, 8000h`

Chú ý !: Để tránh nhầm lẫn khi viết, tốt nhất các bạn nên biểu diễn mặt nạ dưới dạng số hex thay vì dưới dạng số nhị phân đặc biệt là khi sử dụng mặt nạ dài 16 bit.

Các lệnh lôgic đặc biệt có ích khi thực hiện các công việc dưới đây:

Đổi mã ASCII của một số thành số tương ứng.

Chúng ta đã thấy rằng khi một chương trình đọc một ký tự từ bàn phím, AL sẽ chứa mã ASCII của ký tự đó. Điều đó cũng đúng với các ký tự biểu diễn số. Chẳng hạn khi phím "5" được ấn, AL sẽ chứa 35h thay vì 5. Để nhận được 5 trong thanh ghi AL chúng ta có thể làm như sau:

```
SUB    AL,30h
```

Một phương pháp khác là sử dụng lệnh AND để xoá nửa byte cao của AL:

```
AND    AL,0Fh
```

Vì mã ASCII của các chữ số từ "0" đến "9" là từ 30h đến 39h, phương pháp này có thể dùng để đổi mã ASCII của bất cứ chữ số nào thành giá trị thập phân tương ứng.

Bằng cách sử dụng lệnh AND thay cho lệnh SUB chúng ta nhấn mạnh rằng chúng đang thay đổi những mẫu bit của AL, nhờ đó chương trình trở nên dễ đọc hơn.

Việc đổi ngược lại một số thập phân ra mã ASCII của nó để giành cho các bạn như là một bài tập.

Đổi chữ thường thành chữ hoa

Mã ASCII của các chữ thường từ "a" đến "z" nằm từ 61h đến 7Ah, mã ASCII của các chữ hoa từ "A" đến"Z" nằm từ 41h đến 5Ah. Bởi vậy chẳng hạn nếu DL chứa mã ASCII của một chữ thường chúng ta có thể đổi ra chữ hoa bằng cách thực hiện lệnh:

```
SUB    DL,20h
```

Phương pháp này được sử dụng trong chương 4, tuy nhiên nếu chúng ta so sánh mã ASCII dạng nhị phân của các chữ thường và chữ hoa tương ứng sẽ thấy rằng:

Ký tự	Mã ASCII	Ký tự	MãASCII
a	01100001	A	01000001
b	01100010	B	01000010
.	.	.	.
.	.	.	.
.	.	.	.
z	01111010	Z	01011010

Rõ ràng là để đổi từ chữ thường thành chữ hoa chúng ta chỉ cần xoá bit 5, điều này có thể thực hiện bằng cách dùng lệnh AND với mặt nạ 11011111b hay 0DFh. Do đó nếu một chữ thường chứa trong DL thì ta có thể đổi nó thành chữ in như sau:

```
AND    DL,0DFh
```

Việc đổi ngược lại từ chữ hoa ra chữ thường xem như bài tập cho các bạn.

Xoá một thanh ghi

Chúng ta đã biết 2 cách để xoá một thanh ghi, chẳng hạn để xoá thanh ghi AX chúng ta có thể làm như sau:

```
MOV    AX,0
```

hay

```
SUB    AX,AX
```

Bằng cách sử dụng một tính chất là 1 XOR 1 = 0 và 0 XOR 0 = 0 chúng ta có một phương pháp thứ 3 như sau:

```
XOR    AX,AX
```

Mã máy cho lệnh đầu tiên là 3 byte so với 2 byte của mỗi lệnh sau, như vậy các lệnh sau hiệu quả hơn. Tuy nhiên các thao tác giữa ô nhớ với ô nhớ là không hợp lệ nên khi cần xoá một ô nhớ thì chúng ta bắt buộc phải sử dụng lệnh thứ nhất.

Kiểm tra xem một thanh ghi có bằng 0 hay không

Vì 1 OR 1 = 1 và 0 OR 0 = 0 có vẻ như chúng ta sẽ phí thời gian nếu thực hiện lệnh sau:

```
OR    CX,CX
```

Bởi vì chúng không làm thay đổi nội dung của CX. Tuy nhiên chúng lại tác động đến cờ ZF và SF do đó trong trường hợp đặc biệt khi CX chứa 0 thì ZF = 1 và vì vậy nó có thể được dùng thay cho lệnh :

```
CMP   CX,0
```

Để kiểm tra xem nội dung của một thanh ghi có bằng 0 hay không hoặc để kiểm tra dấu của số chứa trong thanh ghi.

7.1.2 Lệnh NOT

Lệnh NOT lấy số bù 1 của một toán hạng đích. Cú pháp như sau:

```
NOT   toán hạng đích
```

Lệnh này không làm ảnh hưởng tới các cờ.

Ví dụ 7.5 Đảo các bit trong thanh ghi AX

Trả lời :

Dùng lệnh NOT:

```
NOT   AX
```

7.1.3 Lệnh TEST

Lệnh TEST thực hiện thao tác và lôgic giữa toán hạng đích với nguồn nhưng không làm thay đổi toán hạng đích. Mục đích của lệnh TEST là thiết lập các cờ. Cú pháp :

```
TEST   toán hạng đích, toán hạng nguồn
```

Các cờ bị tác động:

SF, ZF, PF phản ánh kết quả
AF không xác định
CF, OF = 0

Kiểm tra các bit

Lệnh TEST có thể sử dụng để kiểm tra các bit riêng biệt trong một toán hạng. Mặt nạ có giá trị 1 ở các bit tương ứng với với các bit cần kiểm tra trong toán hạng đích và giá trị 0 ở các bit khác. Vì 1 AND b = b và 0 AND b = 0 kết quả của lệnh:

```
TEST    toán hạng đích,mặt nạ
```

Sẽ có giá trị 1 ở các bit đã chọn khi và chỉ khi toán hạng đích cũng có giá trị 1 ở các vị trí đó, còn lại các bit khác có giá trị 0. Nếu toán hạng đích có giá trị 0 ở tất cả các bit được kiểm tra thì kết quả sẽ bằng 0 và cờ ZF được thiết lập 1.

Ví dụ 7.6 Viết đoạn lệnh thực hiện lệnh nhảy đến nhãn BELOW nếu AL chứa số chẵn.

Trả lời :

Vì các số chẵn có bit 0 bằng 0 nên mặt nạ là 00000001b = 1. Chúng ta có:

```
TEST  AL,1        ;AL chứa số chẵn ?
JZ    BELOW       ;Đúng ! ,nhảy đến BELOW
```

7.2 Các lệnh dịch

Các lệnh dịch và quay dịch các bit trong toán hạng đích sang trái hoặc phải một hoặc một số vị trí. Đối với lệnh dịch các bit bị dịch ra khỏi toán hạng sẽ bị mất, còn đối với lệnh quay các bit dịch ra khỏi một phía của toán hạng đích sẽ được đưa trở lại phía kia. Các lệnh này có 2 dạng sau :

Khi muốn dịch hoặc quay 1 vị trí chúng ta có:

```
Mã lệnh     toán hạng đích,1
```

Khi muốn dịch hoặc quay N vị trí chúng ta có:

```
Mã lệnh     toán hạng đích, CL
```

Trong đó CL chứa N. Trong cả 2 trường hợp toán hạng đích là một thanh ghi 8 hoặc 16 bit hay là một ô nhớ. Chú ý rằng trong các bộ vi xử lý tiên tiến của INTEL, các lệnh quay hoặc dịch cho phép sử dụng các hằng số 8 bit. Như chúng ta sẽ thấy ngay sau đây các lệnh này có thể dùng để nhân hoặc chia cho luỹ thừa của 2, và chúng ta sẽ sử dụng chúng trong các chương trình vào ra với số nhị phân và số hex.

7.2.1 Các lệnh dịch trái

Lệnh SHL

Lệnh SHL (shift left) dịch các bit của toán hạng đích sang bên trái. Cú pháp

cho lệnh dịch một vị trí như sau:

```
SHL        toánhạngđích,1
```

Một giá trị 0 sẽ được đưa vào vị trí bên phải nhất của toán hạng đích, còn bit msb của nó sẽ được đưa vào cờ CF (xem hình 7.2)

Nếu số đếm vị trí dịch N khác 1 thì lệnh có dạng sau:

```
SHL   toánhạngđích,CL
```

Trong đó CL chứa N. Trong trường hợp này N phép dịch trái đơn được thực hiện. Giá trị của CL vẫn giữ nguyên không thay đổi khi lệnh thực hiện xong.

Các cờ bị tác động:

SF, PF, ZF phản ánh kết quả

AF không xác định

CF chứa bit cuối cùng được dịch ra khỏi toán hạng

OF bằng 1 nếu kết quả bị thay đổi dấu trong phép dịch cuối cùng.

Ví dụ 7.7 Giả sử DH chứa 8Ah và CL chứa 3, cho biết giá trị của DH và CF sau khi lệnh: SHL DH, CL được thực hiện

Trả lời :

Giá trị nhị phân trong DH là 10001010. Sau 3 lần dịch CF sẽ chứa giá trị 0 còn nội dung của DH có thể nhận được bằng cách xoá đi 3 bit bên trái nhất của DH và thêm 3 bit 0 vào bên 0 phải nó. Bởi vậy chúng ta nhận được kết quả :01010000b = 50h.

Thực hiện phép nhân bằng cách dịch trái

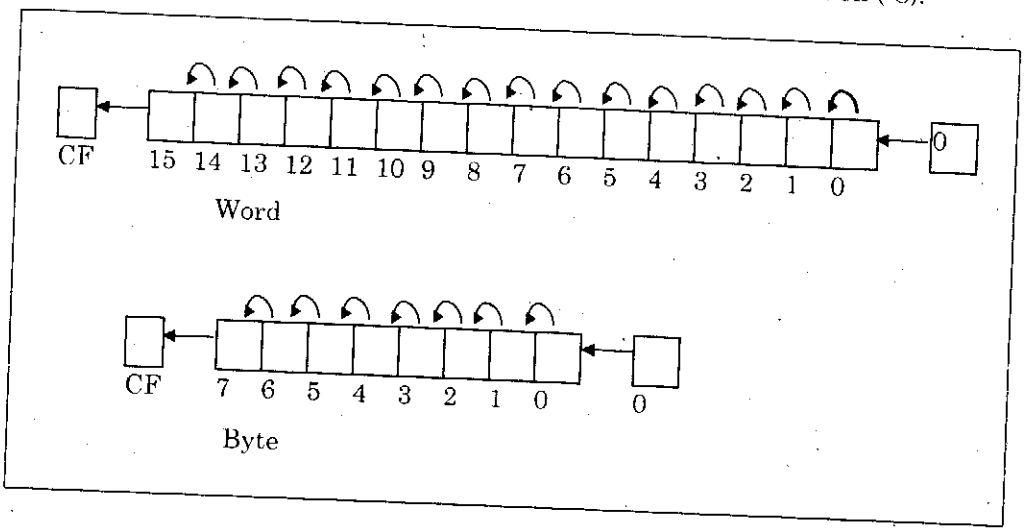
Giả sử có số thập phân 235, nếu mỗi chữ số của nó được dịch sang bên trái một vị trí và thêm vào bên phải chữ số 0 thì ta nhận được 2350 có nghĩa là bằng 235 nhân với 10.

Tương tự như vậy việc dịch trái một số nhị phân cũng như nhân nó với 2. Ví dụ AL chứa 5 = *00000101b. Sau khi dịch trái một bit chúng ta có : 00001010b* = 10d nghĩa là bằng 2 giá trị của nó. Tiếp tục dịch trái chúng ta nhận được 00010100b = 20d nghĩa là lại nhân đôi nó một lần nữa.

Lệnh SAL

Như vậy lệnh SHL có thể dùng để nhân một toán hạng với luỹ thừa của 2. Tuy nhiên để làm nổi bật bản chất số học của thao tác, lệnh SAL (shift arithmetic left) thường được sử dụng để thay cho việc nhân các số. Cả 2 lệnh đều tạo ra cùng một mã máy.

Các số âm cũng có thể được nhân 2 bằng cách dịch trái.Ví dụ nếu AX chứa FFFFh (-1) chúng ta dịch trái nó 3 bit thì sẽ nhận được AX = FFF8h (-8).



Hình 7.2 Lệnh SHL và SAL

Sự tràn

Khi chúng ta sử dụng lệnh dịch trái để làm phép nhân hiện tượng tràn có thể xảy ra. Với lệnh dịch trái một bit CF và OF tương ứng chỉ ra một cách chính xác sự tràn không dấu và có dấu. Tuy nhiên cờ tràn không còn đáng tin cậy trong lệnh dịch trái nhiều bit. Sở dĩ như vậy là vì lệnh dịch nhiều bit thực ra là nhiều lệnh dịch một bit, và cờ CF, OF chỉ phản ánh kết quả của sự dịch cuối cùng. Ví dụ khi BL chứa 80h, CL chứa 2 và chúng ta thực hiện lệnh SHL BL.CL thì CF = OF = 0 mặc dù cả 2 hiện tượng tràn có dấu và không dấu đều xảy ra.

Ví dụ 7.8 Viết một đoạn lệnh thực hiện phép nhân AX với 8, giả sử hiện tượng tràn không xảy ra.

Trả lời:

Để thực hiện phép nhân với 8 chúng ta cần thực hiện 3 lần dịch trái:

```
MOV   CL,3  ;Số lần dịch
SHL   AX,CL ;Nhân với 8
```

7.2.2 Các lệnh dịch phải

Lệnh SHR

Lệnh SHR (shift Right) dịch các bit của toán hạng đích sang bên phải. Cú pháp cho lệnh dịch một vị trí như sau:

```
SHR   toánhạngđích,1
```

Một giá trị 0 sẽ được đưa vào bit msb của toán hạng đích, còn bit bên phải nhất của nó (lsb) sẽ được đưa vào cờ CF (xem hình 7.3)

Nếu số đếm vị trí dịch N khác 1 thì lệnh có dạng sau:

```
SHL   toánhạngđích,CL.
```

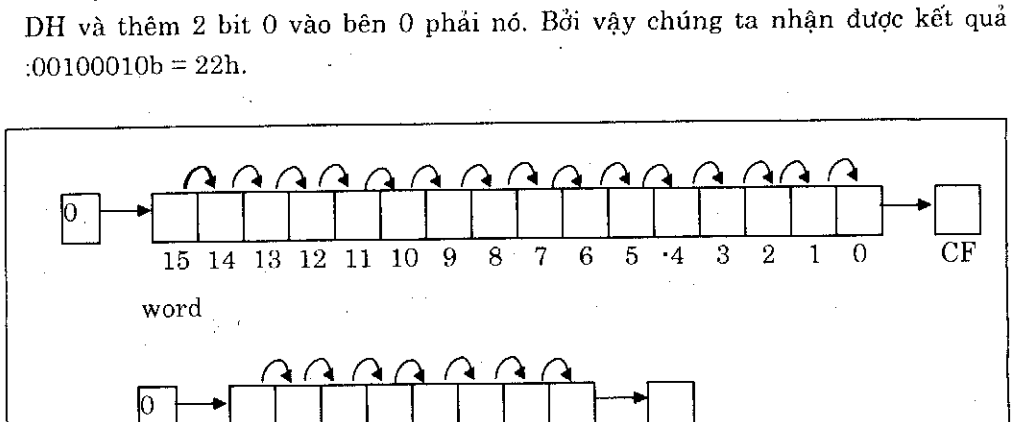
Trong đó CL chứa N. Trong trường hợp này N phép dịch phải đơn được thực hiện. Giá trị của CL vẫn giữ nguyên không thay đổi khi lệnh thực hiện xong.

Các cờ bị tác động cũng giống như trong trường hợp dịch trái

Ví dụ 7.9 Giả sử DH chứa 8Ah và CL chứa 2, cho biết giá trị của DH và CF sau khi lệnh: SHL DH, CL được thực hiện

Trả lời :

Giá trị nhị phân trong DH là 10001010. Sau 2 lần dịch CF sẽ chứa giá trị 1 còn nội dung của DH có thể nhận được bằng cách xoá đi 2 bit bên phải nhất của DH và thêm 2 bit 0 vào bên 0 phải nó. Bởi vậy chúng ta nhận được kết quả :00100010b = 22h.



Hình 7.3 Lệnh SHR

Lệnh SAR

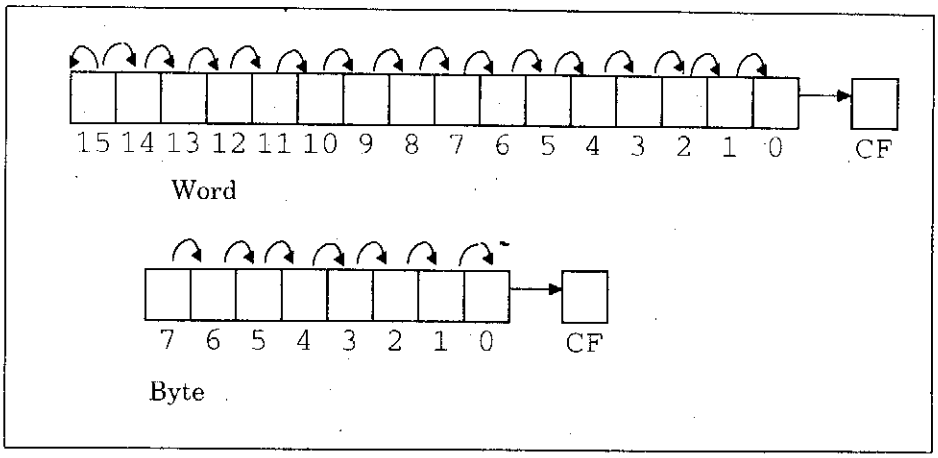
Lệnh SAR (shift arimethic right) làm việc gần giống lệnh SHR với một điểm khác biệt là bit msb của toán hạng đích giữ nguyên giá trị ban đầu của nó. (xem hình 7.4). Cú pháp như sau:

```
SAR   toán hạng đích,1
```

Và

```
SAR   Toán hạng đích,CL
```

Các cờ bị tác động giống như trong lệnh SHR



Hình 7.4 Lệnh SAR

Thực hiện phép chia bằng cách dịch phải

Vì việc dịch trái nhân đôi giá trị của toán hạng đích cho nên cũng có lý khi đoán rằng việc dịch phải sẽ chia đôi nó. Điều đó đúng cho các số chẵn, đối với các số lẻ, dịch phải sẽ chia đôi nó và làm tròn xuống số nguyên gần nhất. Ví dụ nếu BL chứa

00000101b = 5 thì sau khi dịch phải BL sẽ chứa 0000010 = 2.

Phép chia không dấu và có dấu

Trong phép chia bằng cách dịch phải chúng ta phải phân biệt 2 trường hợp đối với các số không dấu và có dấu. Nếu đang làm việc với các số không dấu chúng ta phải sử dụng lệnh SHR còn khi làm việc với số có dấu chúng ta phải sử dụng lệnh SAR vì lệnh này giữ nguyên dấu.

Ví dụ 7.10 Sử dụng các phép dịch phải để thực hiện phép chia số không dấu 65143 cho 4, thương số chứa trong AX.

Trả lời :

Để chia cho 4 chúng ta cần dịch phải 2 lần, do số bị chia là số không dấu chúng ta sẽ sử dụng lệnh SHR. Đoạn lệnh như sau:

```
MOV   AX,65143    ;AX chứa số bị chia
MOV   CL,2        ;CX chứa số lần dịch phải
SHR   AX,CL       ;Chia cho 4
```

Ví dụ 7.11 Giả sử AL chứa -15, hãy cho biết giá trị thập phân của AL sau khi thực hiện lệnh SAR AL,1.

Trả lời :

Lệnh SAR chia số cho 2 và làm tròn xuống. Chia -15 cho 2 chúng ta nhận được -7,5 sau khi làm tròn chúng ta nhận được -8. Dưới dạng số nhị phân chúng ta có -15 = 11110001b. Sau khi dịch phải chúng ta có 11111000b = -8.

Các phép nhân và chia tổng quát hơn

Chúng ta đã thấy rằng việc nhân và chia cho luỹ thừa của 2 có thể thực hiện bằng các lệnh dịch trái và phải. Để nhân với một số bất kỳ như 10d chúng ta có thể kết hợp các lệnh dịch và cộng (xem chương 8).

Trong chương 9 chúng ta sẽ học các lệnh MUL và IMUL, DIV và IDIV. Chúng không bị giới hạn trong việc nhân hay chia các luỹ thừa của 2, nhưng lại chậm hơn nhiều so với các lệnh dịch

7.3 Các lệnh quay

Lệnh quay trái

Lệnh ROL (rotate left) dịch các bit sang bên trái. Bit msb được dịch vào bit bên phải nhất, đồng thời được đưa vào cờ CF. Các bạn có thể tưởng tượng các bit của toán hạng đích tạo thành một vòng tròn với bit lsb theo sau bit msb, xem hình 7.5. Cú pháp :

```
ROL   toán hạng đích,1
```

Và

```
ROL   toán hạng đích,CL
```

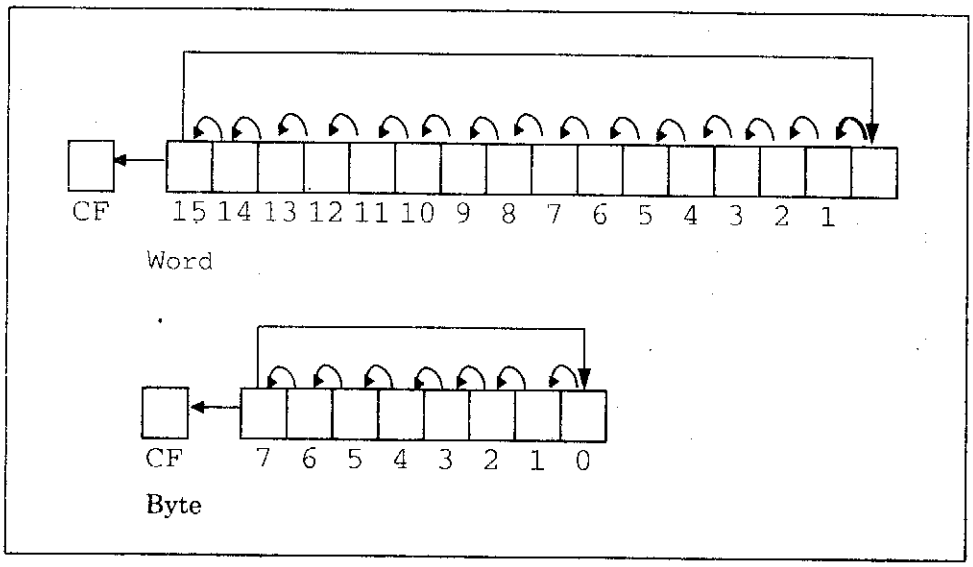
Lệnh quay phải

Lệnh ROR (rotate right) làm việc giống như lệnh ROL trừ một điểm khác biệt là các bit được dịch sang phải, bit bên phải nhất được dịch vào bit msb đồng thời cũng được đưa vào cờ CF. Cú pháp của lệnh:

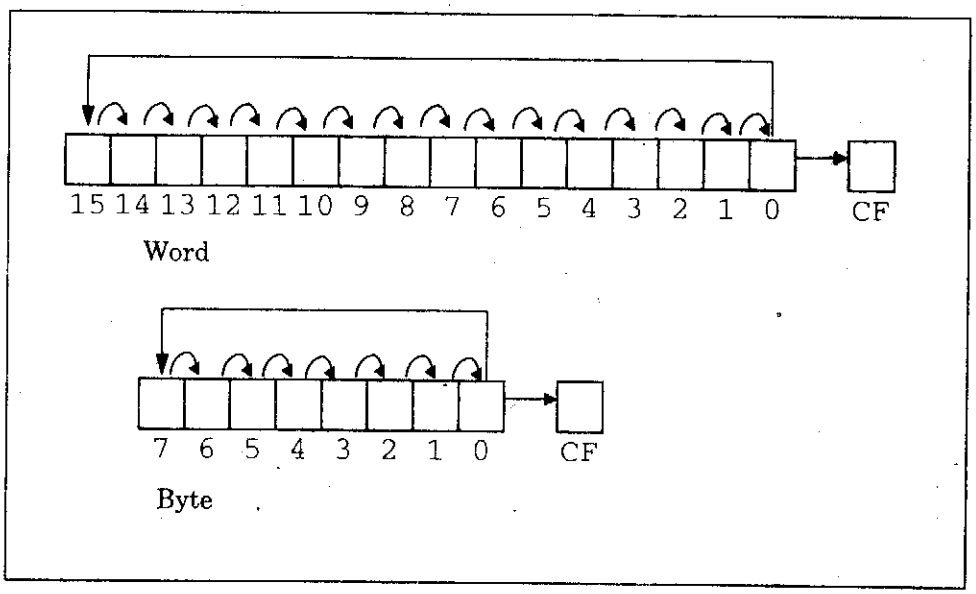
```
ROR   toán hạng đích,1
```

Và

```
ROR   toán hạng đích,CL
```



Hình 7.5 Lệnh ROL



Hình 7.6 Lệnh ROR

Trong các lệnh ROL,ROR cờ CF chứa bit bị dịch ra khỏi toán hạng. Ví dụ sau sẽ trình bày một phương pháp để xác định các bit trong một byte hay word mà không làm thay đổi nội dung của chúng.

Ví dụ 7.12 Sử dụng lệnh ROL để đếm số bit 1 trong thanh ghi BX mà không làm thay đổi nội dung của nó, chứa kết quả trong AX.

Lời giải:

```
    XOR   AX,AX        ;AX đếm số bit
    MOV   CX,16        ;Biến đếm vòng lặp
TOP:
    ROL   BX,1         ;CF chứa bit bị đưa ra
    JNC   NEXT         ;bit 0 ?
    INC   AX           ;không phải !, tăng số bit 1
NEXT:
    LOOP  TOP          ;lặp lại cho đến khi làm xong
```

Trong ví dụ trên đây, chúng ta đã sử dụng lệnh JNC (Jump if No Carry), lệnh này thực hiện việc nhảy nếu CF = 0. Trong phần 7.4 chúng ta sẽ sử dụng để đưa ra nội dung của một thanh ghi dưới dạng nhị phân.

Lệnh quay trái qua cờ nhớ

Lệnh RCL (Rotate Carry Left) dịch các bit của toán hạng đích sang trái. Bit msb được đặt vào cờ CF, giá trị cũ của CF được đưa vào bit phải nhất của toán hạng đích. Nói cách khác RCL làm việc giống như ROL ngoại trừ một điều là cờ CF cũng là một phần của vòng tròn tạo lên bởi các bit đang được quay (xem hình 7.7), cú pháp của lệnh:

```
RCL  toán hạng đích,1
```

Và

```
RCL  toán hạng đích,CL
```

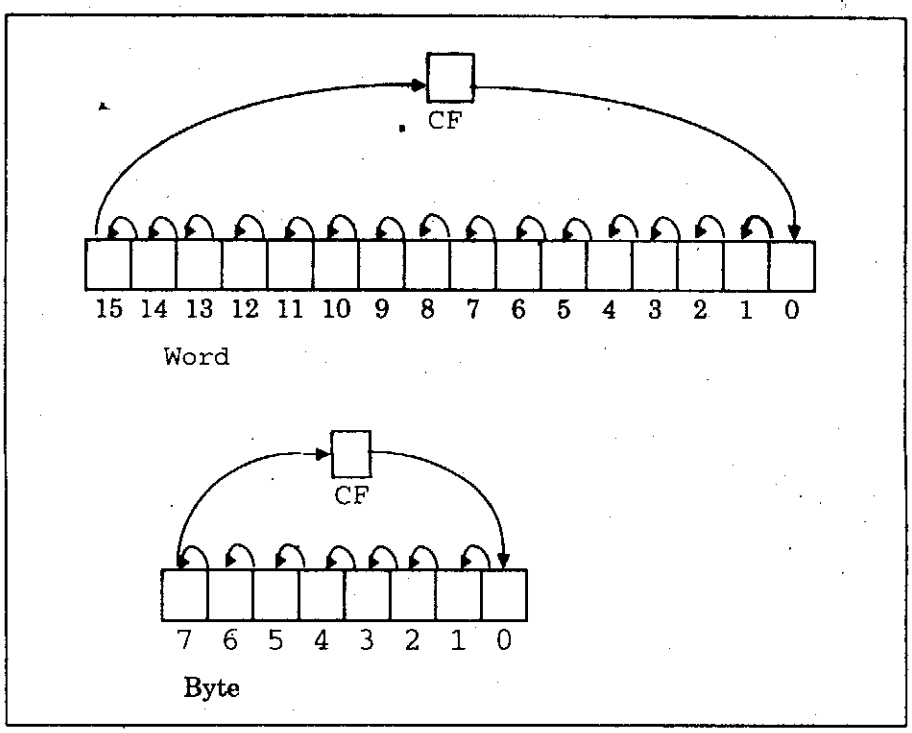
Lệnh quay phải qua cờ nhớ

Lệnh RCR (Rotate Carry Right) hoạt động giống như lệnh RCL nhưng các bit được quay sang phải (xem hình 7.8), cú pháp của lệnh như sau:

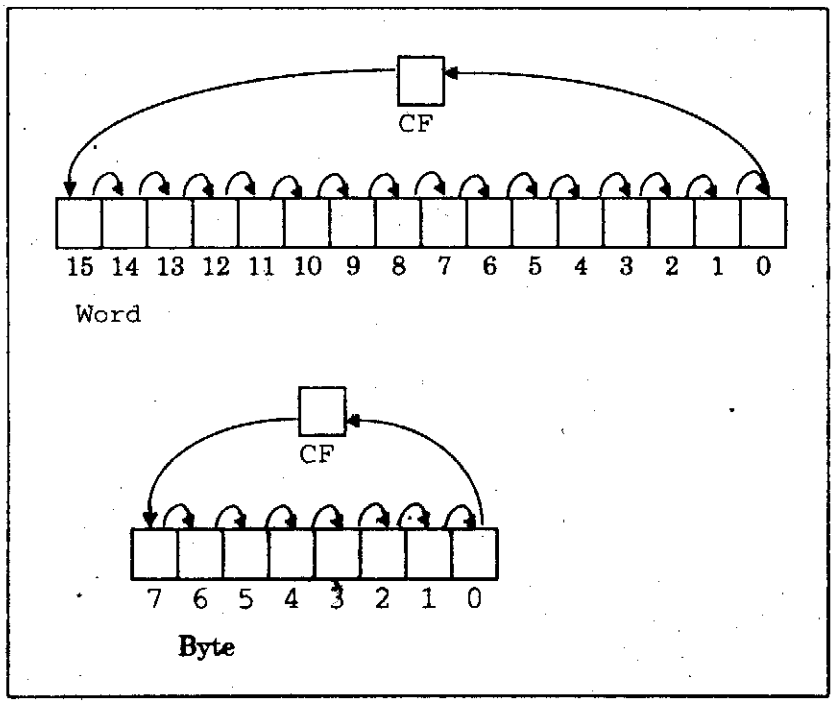
```
RCR  toán hạng đích,1
```

Và

```
RCR  toán hạng đích,CL
```



Hình 7.7 Lệnh RCL



Hình 7.8 Lệnh RCR

Ví dụ 7.13 Giả sử DH chứa 8Ah, CF = 1, và CL chứa 3. Cho biết nội dung của DH và CF khi thực hiện lệnh RCR DH,CL ?

Trả lời :

	CF	DH
Các giá trị đầu	1	10001010
Sau 1 lần dịch	0	11000101
Sau 2 lần dịch	1	01100010
Sau 3 lần dịch	0	10110001

Như vậy sau khi thực hiện lệnh DH chứa 10110001b = B1h

Tác động của các lệnh quay đến cờ

SF, PF, ZF phản ánh kết quả

AF không xác định

CF = bit cuối cùng bị dịch ra khỏi toán hạng

OF = 1 nếu kết quả đổi dấu trong lần quay cuối cùng.

Một ứng dụng: Đảo các mẫu bit

Chúng ta hãy xem xét vấn đề đảo các mẫu bit trong một byte hoặc một word để làm một ví dụ về ứng dụng của các lệnh quay và dịch. Chẳng hạn AL chứa 11011100 chúng ta muốn đổi lại thành 00111011. Một phương pháp đơn giản là sử dụng lệnh SHL để dịch các bit ra khỏi đầu bên trái của AL rồi dùng lệnh RCR để đưa nó vào đầu bên trái của một thanh ghi khác như BL chẳng hạn. Sau khi lặp lại công việc đó 8 lần BL sẽ chứa các mẫu bit đã đảo ngược của AL và có thể chép lại nội dung của nó vào AL. Đoạn lệnh như sau:

```
        MOV   CX,8          ;Số lần cần thực hiện
REVERSE:
        SHL   AL,1          ;Lấy bit vào CF
        RCR   BL,1          ;Quay để đưa nó vào BL
        LOOP  REVERSE       ;lặp cho đến khi DO xong
        MOV   AL,BL         ;AL chứa mẫu bit đảo ngược
```

7.4 Vào ra với các số nhị phân và số hex.

Một ứng dụng khá tiện lợi của các lệnh quay và dịch là thực hiện các thao tác vào/ra với các số nhị phân và số hex.

Nhập các số nhị phân

Để nhập các số nhị phân chúng ta giả thiết rằng chương trình đọc vào các số nhị phân từ bàn phím kết thúc bằng phím ENTER. Các số thực sự ở dạng chuỗi các chữ số 0 và 1. Khi mỗi ký tự được nhập chúng ta phải đổi chúng ra giá trị của từng bit rồi kết hợp các bit vào trong thanh ghi. Thuật toán sau thực hiện việc đọc một số nhị phân từ bàn phím rồi sau đó lưu nó vào thanh ghi BX.

Thuật toán nhập số nhị phân

```
Xoá BX /* BX sẽ giữ kết quả */
Nhập một ký tự /* '0' hoặc '1' */
WHILE ký tự <> CR DO
    Đổi ký tự ra giá trị nhị phân
    Dịch trái BX
    Chèn giá trị nhận được vào bit lsb của BX
    Nhập ký tự
END_WHILE
```

Biểu diễn với việc nhập 110

```
Xoá BX
    BX=0000 0000 0000 0000
Nhập vào ký tự '1', đổi nó thành 1
Dịch trái BX
    BX=0000 0000 0000 0000
Chèn gia trị nhận được vào bit lsb của BX
    BX=0000 0000 0000 0001
Nhập vào ký tự '1', đổi nó thành 1
Dịch trái BX
    BX=0000 0000 0000 0010
Chèn gia trị nhận được vào bit lsb của BX
    BX=0000 0000 0000 0011
    Nhập vào ký tự '0', đổi nó thành 0
Dịch trái BX
    BX=0000 0000 0000 0110
Chèn gia trị nhận được vào bit lsb của BX
    BX=0000 0000 0000 0110
BX chứa 110b
```

Thuật toán trên đây đã coi rằng các ký tự nhập vào chỉ là một trong 3 ký tự '1', '0' hay CR và có nhiều nhất 16 ký tự được nhập. Khi một chữ số mới được nhập vào, các bit có sẵn trong thanh ghi BX phải được dịch trái để giành chỗ cho nó. Do đó có thể dùng lệnh O R để chèn bit mới vào BX. Các lệnh hợp ngữ như sau:

```
        XOR     BX,BX       ;Xoá BX
        MOV     AH,1  ;Hàm con đọc ký tự từ bàn phím
        INT     21H         ;Đọc ký tự
WHILE_:
        CMP     AL,0DH      ;CR?
        JE      END_WHILE   ;Đúng !,kết thúc
        AND     AL,0FH ;Không!đổi ra giá trị nhị phân
        SHL     BX,1        ;Giành chỗ cho giá trị mới
        OR      BL,AL       ;Chèn giá trị mới vào BX
        INT     21H         ;Đọc tiếp ký tự
        JMP     WHILE_      ;Lặp lại
END_WHILE:
```

Việc xuất các giá trị nhị phân

Việc đưa ra nội dung của BX dưới dạng nhị phân cũng dùng các lệnh dịch. Dưới đây chúng tôi chỉ đưa ra thuật toán, còn phần chương trình bằng hợp ngữ giành cho các bạn làm bài tập.

Thuật toán đưa ra số nhị phân

```
FOR 16 lần, DO:
   Quay trái BX /* BX chứa giá trị đưa ra, bit msb đưa
                   vào CF */
   IF CF=1
      THEN
            Đưa ra '1'
      ELSE
            Đưa ra '0'
   END_IF
END_FOR
```

Nhập các số hex

Việc nhập các số hex bao gồm chữ số từ "0' đến "9" và các chữ cái từ "A" đến "F" kết thúc bằng ký tự CR. Để đơn giản chúng ta giả thiết :

- Chỉ sử dụng các chữ hoa
- Người sử dụng chỉ đưa vào tối đa 4 chữ số hex.

Quá trình đổi các ký tự thành trị nhị phân phức tạp hơn so với trường hợp nhập số nhị phân, ngoài ra BX phải được dịch 4 lần để giành chỗ cho một giá trị hex.

Thuật toán nhập số hex

```
Xoá BX /*BX sẽ chứa giá trị nhập vào*/
Nhập ký tự hex
WHILE     ký tự<>CR DO
   Đổi ký tự ra trị nhị phân
   Dịch trái BX 4 lần
   Chèn giá trị mới vào 4 bit thấp của BX
   Nhập ký tự
END_WHILE
```

Biểu diễn việc nhập 6ABh

```
Xoá BX
   BX=0000 0000 0000 0000
Nhập vào "6", đổi thành 0110
Dịch trái BX 4 lần
   BX=0000 0000 0000 0000
Chèn giá trị nhận được vào 4 bit thấp của BX
   BX=0000 0000 0000 0110
Nhập vào "A", đổi thành Ah=1010
Dịch trái BX 4 lần
   BX=0000 0000 0110 0000
Chèn giá trị nhận được vào 4 bit thấp của BX
   BX=0000 0000 0110 1010
Nhập vào "B", đổi thành Bh=1011
Dịch trái BX 4 lần
   BX=0000 0110 1010 0000
Chèn giá trị nhận được vào 4 bit thấp của BX
   BX=0000 0110 1010 1011
BX chứa 06ABh.
```

Đoạn chương trình

```
        XOR   BX,BX        ;Xoá BX
        MOV   CL,4         ;Bộ đếm 4 lần dịch
        MOV   AH,1         ;Hàm con nhập ký tự từ bàn phím
        INT   21H          ;Nhập một ký tự
WHILE_:
        CMP   AL,0DH       ; CR ?
        JE    END_WHILE    ;Đúng!,kết thúc
;Đổi ký tự ra giá trị nhị phân
        CMP   AL,39H       ;Đó là chữ số ?
        JG    LETTER       ;Không phải!, đó là một chữ cái
;Nhập một số
```

```
        AND   AL,0FH        ;đổi chữ số ra giá trị nhị phân
        JMP   SHIFT         ;Đem chèn vào BX
LETTER:                     ;Đổi chữ cái ra trị nhị phân
        SUB   AL,37H
SHIFT:
        SHL   BX,CL         ;Giành chỗ cho giá trị mới
;đưa giá trị vào BX
        OR    BL,AL         ;Chèn giá trị mới vào 4 bit thấp
                            ;của BX
        INT   21H           ;Nhập tiếp ký tự từ bàn phím
        JMP   WHILE_        ;Lặp lại cho đến khi phím ENTER
                            ;được ấn
END_WHILE:
```

Chú ý rằng chương trình không kiểm tra xem các ký tự nhập vào có hợp lệ hay không.

Đưa ra số hex

BX chứa số 16 bit bằng giá trị của số hex 4 chữ số. Để đưa ra nội dung của BX, chúng ta bắt đầu từ bên trái, lấy ra từng nhóm bit của mỗi chữ số rồi đổi nó thành chữ số hex tương ứng sau đó đưa ra.

Thuật toán đưa ra số hex

```
FOR 4 lần  DO
      Chuyển BH vào DL /*BX chứa giá trị đưa ra*/
      Dịch DL về bên phải 4 lần
      IF DL<10
            THEN
                  Đổi thành một trong các ký tự:
                              "0", ..."9"

            ELSE
                  Đổi thành một trong các chữ cái:
                              "A"..."F"
      END_IF
      Đưa ký tự ra
      Quay BX 4 lần về bên trái
      END_FOR
```

Biểu diễn việc đưa ra số 4CA9h trong BX

```
BX=4CA9h=0100 1100 1010 1001
```

```
Chuyển BH vào DL
    DL=0100 1100
Dịch phải DL 4 lần
    DL=0000 0100
Đổi thành ký tự và đưa ra
    DL=0011 0100=34h= '4'
Quay BX 4 lần về bên trái
    BX=1100 1010 1001 0100
Chuyển BH vào DL
    DL=1100 1010
Dịch phải DL 4 lần
    DL=0000 1100
Đổi thành ký tự và đưa ra
    DL=0100 0011=43h= 'C'
Quay BX 4 lần về bên trái
    BX=1010 1001 0100 1100
Chuyển BH vào DL
    DL=1010 1001
Dịch phải DL 4 lần
    DL=0000 1010
Đổi thành ký tự và đưa ra
    DL=0100 0010=42h= 'B'
Quay BX 4 lần về bên trái
    BX=1001 0100 1100 1010
Chuyển BH vào DL
    DL=1001 0100
Dịch phải DL 4 lần
    DL=0000 1001
Đổi thành ký tự và đưa ra
    DL=0011 1001=39h= '9'
Quay BX 4 lần về bên trái
    BX=0100 1100 1010 1001= Giá trị ban đầu.
```

Việc lập chương trình theo thuật toán này chúng tôi giành cho các bạn.

TỔNG KẾT

Trong chương này chúng ta đã học được:

- 5 lệnh lôgic là AND, OR, XOR, NOT và TEST
- *Lệnh OR được sử dụng để thiết lập các bit* riêng biệt của toán hạng đích hay để kiểm tra xem toán hạng đích có bằng 0 hay không.
- Lệnh XOR được dùng để lấy bù các bit riêng biệt của toán hạng đích hay để xoá nó về 0.
- Lệnh NOT dùng để lấy bù 1 của toán hạng đích.
- Lệnh TEST dùng để kiểm tra từng bit riêng biệt của toán hạng đích. Chẳng hạn nó có thể kiểm tra xem một số là chẵn hay lẻ.
- Các lệnh SHL và SAL dịch từng bit của toán hạng đích sang trái một vị trí. Bit msb được đưa vào cờ CF và một giá trị 0 được đưa vào bit lsb.
- Lệnh SHR dịch từng bit của toán hạng đích sang phải 1 vị trí, bit msb được đưa vào cờ CF và một giá trị 0 được đưa vào bit lsb.
- Lệnh SAR hoạt động tương tự như lệnh SHR nhưng bit msb được giữ nguyên giá trị.
- Các lệnh dịch có thể dùng để nhân hoặc chia cho 2. Các lệnh SHL và SAL nhân đôi giá trị trừ khi có hiện tượng tràn xảy ra. Các lệnh SHR và SAR chia đôi giá trị của toán hạng đích nếu nó là số chẵn. Trong trường hợp nó là số lẻ các lệnh này chia đôi giá trị của nó và làm tròn xuống số nguyên gần nhất. Lệnh SHR dùng cho số học không dấu và lệnh SAR dùng cho số học có dấu.
- Lệnh ROL dịch từng bit của toán hạng đích sang trái, bit msb được đưa vào vị trí của bit lsb. Đối với lệnh ROR từng bit dịch sang phải một vị trí, bit lsb được đưa vào vị trí của bit msb. Trong cả 2 trường hợp cờ CF chứa bit cuối cùng được dịch ra khỏi toán hạng đích.
- Các lệnh RCL và RCR hoạt động giống như các lệnh ROL và ROR nhưng bit bị quay ra khỏi toán hạng đích được đưa vào cờ CF còn nội dụng của cờ CF được đưa vào toán hạng đích
- Cũng có thể thực hiện các phép dịch và quay nhiều lần cùng một lúc, khi đó CL phải chứa số lần dịch hoặc quay cần thực hiện
- Các lệnh dịch và quay được sử dụng rất hữu hiệu để vào/ra với các số nhị phân và số hex.

Các thuật ngữ tiếng Anh

Clear	Xoá - đổi giá trị thành 0
Complement	Lấy bù - Đổi từ 0 về 1 và 1 về 0
Mask	Mặt nạ - Một mẫu bit dùng trong các thao tác lô gic để xoá, thiết lập hay kiểm tra các bit xác định trong 1 toán hạng.
SET	Thiết lập - Đổi giá trị của bit thành 1

Các lệnh mới

AND	**RCR**	**SAR**
NOT	**ROL**	**SHR**
OR	**ROR**	**TEST**
RCL	**SAL/SHL**	**XOR**

Bài tập

1. Thực hiện các phép tính lôgic sau đây

```
a. 10101111 AND 10001001
b. 10110001 OR  01001001
c. 01111100 XOR 11011010
d. NOT  01011110
```

2. Viết các lệnh lôgic để thực hiện các công việc sau đây.
   a. Xoá các bit ở vị trí chẵn của AX, giữ nguyên các bit khác.
   b. Thiết lập các bit lsb và msb của BL trong khi giữ nguyên các bit khác.
   c. Đảo bit msb của BL, giữ nguyên các bit khác.
   d. Thay nội dung của biến word1 bằng số bù 1 của nó.

3. Dùng lệnh TEST để :
   a. Thiết lập cờ ZF nếu nội dung của AX bằng 0.
   b. Xoá cờ ZF nếu nội dung của là một số lẻ.
   c. Thiết lập SF nếu DX chứa số âm.
   d. Thiết lập ZF nếu DX chứa số dương hoặc 0.
   e. Thiết lập PF nếu BL chứa một số chẵn các bit 1.

4. Giả sử,AL chứa 11001011b và CF=1, cho biết nội dung mới của AL khi mỗi lệnh sau được thực hiện. Điều kiện đầu đúng cho tất cả các phần của câu hỏi.
   a. SHL AL,1
   b. SHR AL,1
   c. ROL AL,CL trong đó CL chứa 2
   d. ROR AL,CL trong đó CL chứa 3
   e. SAR AL,CL trong đó CL chứa 2
   f. RCL AL,1
   g. RCR AL,CL trong đó CL chứa 3

5. Viết 1 hay nhiều lệnh để thực hiện các công việc sau, giả sử không có hiện tượng tràn xảy ra:
   a. Nhân đôi một biến kiểu byte có giá trị B5h
   b. Nhân nội dung của AL với 8.
   c. Chia 32142 cho 4 và lưu kết quả trong AX.
   d. Chia -2145 cho 16 và lưu kết quả trong BX.

6. Viết các lệnh thực hiện các công việc sau:
   a. Giả sử AL chứa giá trị nhỏ hơn 10, hãy đổi nó thành một chữ số thập phân.
   b. Giả sử DL chứa mã ASCII của một chữ hoa, hãy đổi nó thành chữ thường.

7. Viết các lệnh thực hiện các công việc sau:

a. Nhân nội dung của BL với 10d, giả sử không có hiện tượng tràn.

b. Giả sử AL chứa số âm, hãy chia nó cho 8 và lưu số dư vào AH. (Gợi ý :dùng lệnh ROR)

Các bài tập lập trình

8. Viết một chương trình thông báo cho người sử dụng vào một ký tự, in ra trên các dòng liên tiếp nhau mã ASCII của ký tự đó dưới dạng nhị phân, số các chữ số 1 trong mã ASCII dưới dạng nhị phân đó.

Ví dụ:
Đánh vào một ký tự : A
Mã ASCII của A dưới dạng nhị phân là 01000001
Số các bit 1 là 2

9. Viết chương trình thông báo cho người sử dụng đánh vào một ký tự và in ra mã ASCII của ký tự dưới dạng hex ở dòng tiếp theo. Lập lại cho đến khi người sử dụng đánh ENTER.

Ví dụ:
Đánh vào một ký tự: Z
Mã ASCII của Z dưới dạng hex là 5A
Đánh vào một ký tự:

10. Viết chương trình thông báo cho người sử dụng đánh vào một số hex nhỏ hơn hay bằng 4 chữ số. Đưa ra số dưới dạng nhị phân ở dòng kế tiếp. Khi người sử dụng đưa vào một ký tự không hợp lệ, thông báo để họ vào lại. Chương trình chỉ nhận các chữ in hoa.

Ví dụ:
Đánh vào một số hex(0..FFFF): 1a
Chữ số hex không hợp lệ, hãy vào lại :1ABC
Dưới dạng nhị phân nó bằng: 0001101010111100

Chương trình của bạn có thể bỏ qua các ký tự sau 4 ký tự đầu tiên.

11. Viết chương trình thông báo cho người sử dụng đánh vào một số nhị phân có nhỏ hơn hay bằng 16 chữ số. Đưa ra số dưới dạng hex ở dòng kế tiếp. Khi người sử dụng đưa vào một ký tự không hợp lệ, thông báo để họ vào lại.

Ví dụ:
Đánh vào một số nhị phân (nhiều nhất 16 chữ số):111000001
Dưới dạng hex nó bằng:E1

Chương trình của bạn có thể bỏ qua các ký tự sau 16 ký tự đầu tiên.

12. Viết một chương trình thông báo cho người sử dụng đưa vào 2 số nhị phân, mỗi số có 8 chữ số, in ra màn hình ở dòng tiếp theo tổng của chúng dưới dạng nhị phân. Mỗi khi người sử dụng đánh vào một ký tự không hợp lệ sẽ có thông báo yêu cầu

vào lại. Mỗi số được nhận sau khi người sử dụng sử dụng đánh ENTER.

Ví dụ:

Đánh vào một số nhị phân 8 chữ số:11001010

Đánh vào một số nhị phân 8 chữ số:10011100

Tổng của chúng dạng nhị phân bằng:101100110

13. Viết một chương trình thông báo cho người sử dụng đưa vào 2 số hex không dấu trong khoảng từ 0 đến FFFFh, in ra màn hình ở dòng tiếp theo tổng của chúng dưới dạng hex. Mỗi khi người sử dụng đánh vào một ký tự không hợp lệ sẽ có thông báo yêu cầu vào lại. Chương trình của bạn phải có khả năng kiểm soát hiện tượng tràn không dấu. Mỗi số được nhận sau khi người sử dụng sử dụng đánh ENTER.

Ví dụ:

Đánh vào một số hex, 0 - FFFF:21AB

Đánh vào một số hex, 0 - FFFF:FE03

Tổng của chúng là 11FAE

14. Viết một chương trình thông báo cho người sử dụng đánh vào một chuỗi các chữ số thập phân kết thúc bằng phím ENTER, và in ra màn hình ở dòng kế tiếp tổng của chúng ở dạng hex. Nếu người sử dụng đánh vào một ký tự không hợp lệ phải thông báo để họ vào lại.

Ví dụ:

Đánh vào một chuỗi các chữ số thập phân: 1299843

Tổng của các chữ số dưới dạng hex là 0024

Chương 8

NGĂN XẾP VÀ CÁC THỦ TỤC

Tổng quan.

Đoạn ngăn xếp của chương trình dùng để lưu trữ tạm thời các dữ liệu và địa chỉ. Trong chương này chúng tôi trình bày về các thao tác với ngăn xếp và cách thức sử dụng ngăn xếp để khai báo các thủ tục.

Trong mục 8.1, chúng tôi giới thiệu về các lệnh để cất vào và lấy ra dữ liệu từ ngăn xếp: PUSH và POP. Vì từ cuối cùng cất vào ngăn xếp sẽ được lấy ra đầu tiên nên ngăn xếp có thể dùng để đảo ngược thứ tự một dãy dữ liệu. Tính chất này sẽ được khai thác trong mục 8.2.

Các thủ tục là phần hết sức quan trọng trong lập trình ngôn ngữ bậc cao, điều này cũng đúng với ngôn ngữ Hợp ngữ. Mục 8.3 và 8.4 sẽ trình bày những điểm cơ bản của các thủ tục trong ngôn ngữ Hợp ngữ. Về mặt kỹ thuật, chúng ta có thể biết chính xác một thủ tục được gọi và trở về chương trình gọi nó như thế nào. Trong mục 8.5, chúng tôi sẽ đưa ra ví dụ một thủ tục thực hiện phép nhân nhị phân bằng phương pháp cộng và dịch bit. Qua ví dụ này bạn sẽ học thêm được chút ít về chương trình DEBUG.

8.1. Ngăn xếp.

Ngăn xếp (LIFO) là cấu trúc dữ liệu một chiều. Các phần tử được cất vào và lấy ra từ một đầu của cấu trúc, tức là nó được xử lý theo phương thức 'vào trước, ra sau' (LIFO: Last-In, First-Out). Phần tử được cất vào cuối cùng gọi là đỉnh của ngăn xếp. Ta có thể hình dung ngăn xếp như một chồng đĩa; chiếc đĩa cuối cùng được xếp vào nằm trên đỉnh và chỉ có nó mới có thể được lấy ra đầu tiên.

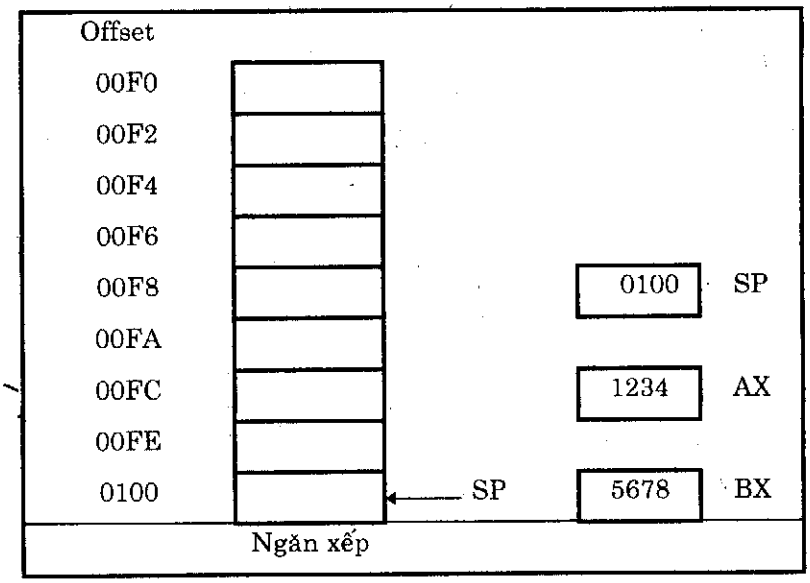
Mỗi chương trình phải dành ra một khối bộ nhớ để làm ngăn xếp. Chúng ta thực hiện điều này khi khai báo đoạn ngăn xếp. Ví dụ:

```
.STACK        100h
```

Khi chương trình được biên dịch và nạp vào bộ nhớ, SS sẽ chứa địa chỉ đoạn ngăn xếp. Trong khai báo ngăn xếp nêu trên, con trỏ ngăn xếp SP được khởi tạo bằng 100h. Điều này làm phát sinh một vị trí ngăn xếp rỗng. Khi ngăn xếp không rỗng, SP chứa địa chỉ offset của đỉnh ngăn xếp.

PUSH và PUSHF.

Hình 8.1.a Ngăn xếp rỗng:



Lệnh PUSH được dùng để thêm một từ mới vào trong ngăn xếp. Cú pháp:

```
PUSH    nguồn
```

Ở đây nguồn là một thanh ghi 16 bit hoặc một từ nhớ. Ví dụ:

```
PUSH    AX
```

Lệnh PUSH thực hiện các công việc sau đây:

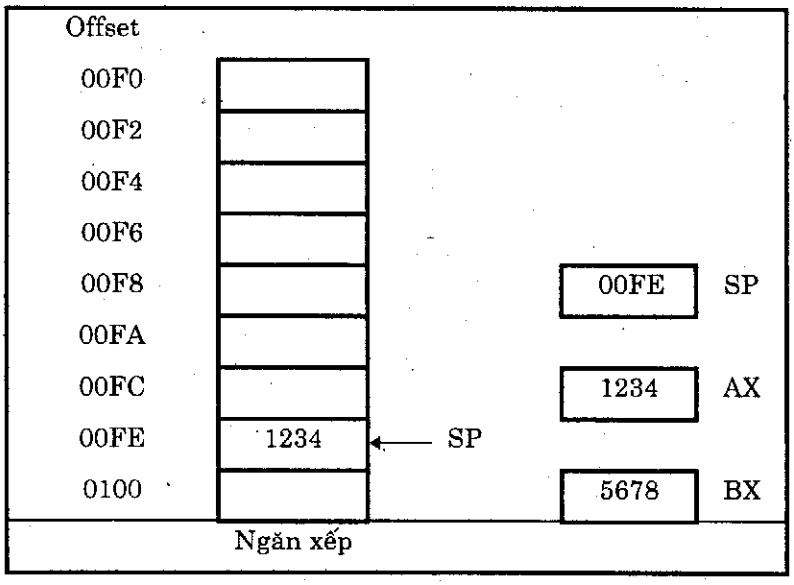
1. Giảm SP đi 2.

2. Một bản sao của nội dung của toán hạng nguồn được chuyển vào địa chỉ xác định bởi SS : SP. Toán hạng nguồn không bị thay đổi.

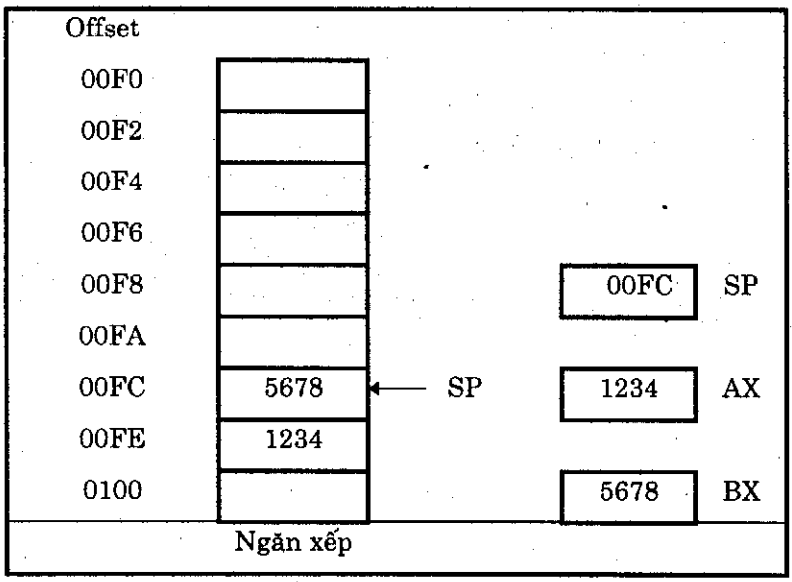
Lệnh PUSHF không có toán hạng cất nội dung của thanh ghi cờ vào ngăn xếp.

Ban đầu, SP chứa địa chỉ offset của ô nhớ theo sau đoạn ngăn xếp. Lệnh PUSH đầu tiên giảm SP đi 2 làm cho con trỏ chỉ đến từ cuối cùng của trong đoạn ngăn xếp. Bởi vì lệnh PUSH làm giảm SP nên ngăn xếp phát triển về phía đỉnh bộ nhớ. Hình 8.2 chỉ ra lệnh PUSH làm việc như thế nào.

Hình 8.1.b Sau lệnh PUSH AX



Hình 8.1.c Sau lệnh PUSH BX



POP và POPF.

Lệnh POP được dùng để lấy ra phần tử đỉnh ngăn xếp. Cú pháp:

```
POP   đích
```

trong đó toán tử đích là một thanh ghi 16 bit (trừ IP) hoặc là một từ nhớ. Ví dụ:

```
POP   BX
```

Lệnh POP thực hiện các công việc sau đây:

1. Nội dung của ô nhớ SS : SP (đỉnh ngăn xếp) được chuyển tới toán tử đích.

2. SP tăng lên 2.

Hình 8.2 chỉ ra một lệnh POP làm việc như thế nào.

Lệnh POPF đưa vào thanh ghi cờ nội dung của đỉnh ngăn xếp.

Các lệnh PUSH, PUSHF, POP và POPF đều không ảnh hưởng đến cờ.

Lưu ý rằng các lệnh PUSH và POP chỉ thao tác với các WORD, vậy nên nếu dùng với các byte như sau:

```
PUSH  DL            ;không hợp lệ
```

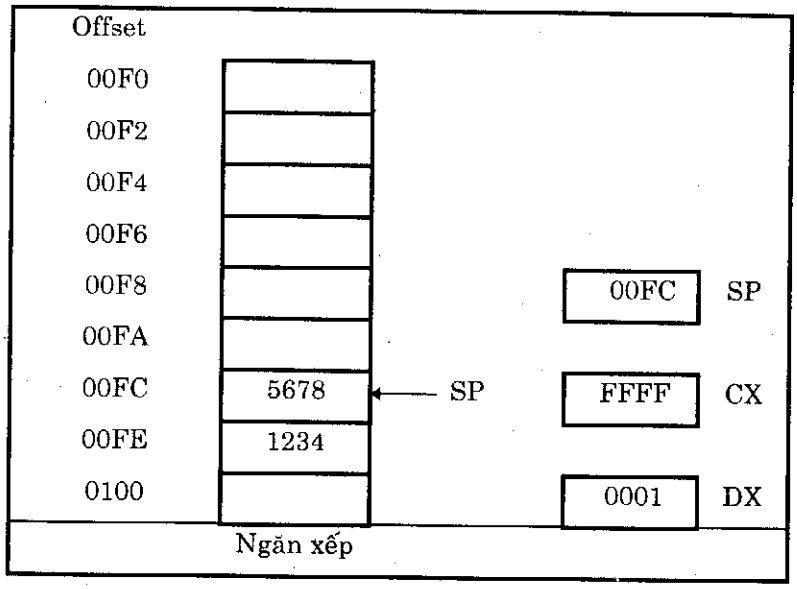
là không hợp lệ. Cũng như vậy với số liệu trực tiếp:

```
PUSH  2             ;không hợp lệ
```

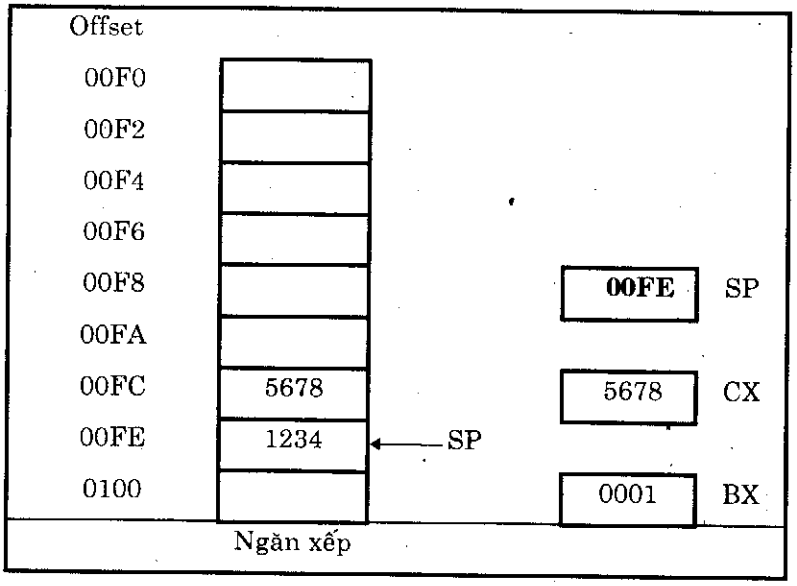
Chú ý: Cất một số liệu trực tiếp là được phép đối với các bộ vi xử lý 80186, 80486. Các bộ vi xử lý này sẽ được trình bày ở chương 20.

Ngoài chương trình của người sử dụng, hệ điều hành cũng sử dụng ngăn xếp cho các mục đích của riêng nó. Ví dụ để thực hiện hàm *INT 21h*, DOS ghi lại mọi thanh ghi mà nó dùng đến vào ngăn xếp và phục hồi chúng khi phục vụ ngắt được hoàn thành. Người sử dụng không cần quan tâm đến vấn đề này bởi *vì tất cả các giá trị mà DOS cất trong ngăn xếp sẽ được lấy ra hết trước khi trả* điều khiển cho chương trình của người sử dụng.

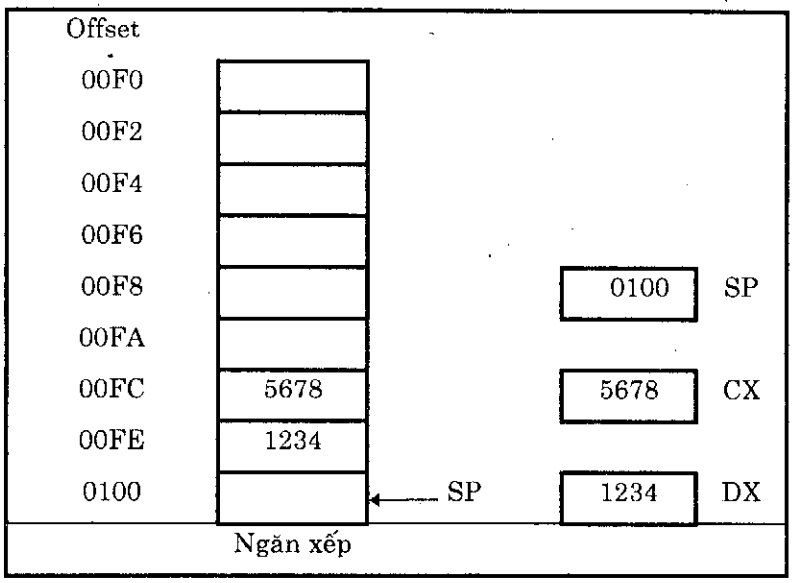
Hình 8.2A. Trước lệnh POP.



Hình 8.2B. Sau lệnh POP CX.



Hình 8.2C. Sau lệnh POP DX.



8.2. Một ứng dụng của ngăn xếp.

Bởi vì ngăn xếp hoạt động theo phương thức vào trước-ra sau nên thứ tự của các phần tử cất vào ngăn xếp sẽ bị đảo ngược khi lấy ra. Chương trình sau đây sẽ sử dụng tính chất này để đọc một chuỗi ký tự và hiển thị chúng theo thứ tự ngược lại trên dòng tiếp theo.

Thuật toán đảo ngược thứ tự.

```
Hiển thị dấu '?'
Khởi động bộ đếm bằng 0.
Đọc một ký tự.
WHILE  không phải ký tự về đầu dòng  DO
    Cất ký tự vào ngăn xếp.
    Tăng biến đếm.
    Đọc một ký tự.
END_WHILE
Chuyển con trỏ xuống dòng mới.
FOR  biến đếm  DO
    Lấy một ký tự từ ngăn xếp.
    Hiển thị nó.
```

```
END_FOR
```

Sau đây là chương trình:

Chương trình PGM8_1.ASM

```
1:      TITLE    PGM8_1:REVERSE INPUT
2:      .MODEL   SMALL
3:      .STACK   100H
4:      CODE
5:      MAIN     PROC
6:      ;hiển thị lời nhắc người sử dụng
7:               MOV    AH,2          ;chuẩn bị hiển thị
8:               MOV    DL,'?'        ;ký tự để hiển thị
9:               INT    21h           ;hiển thị'?'
10:     ;khởi động biến đếm ký tự
11:              XOR    CX,CX
12:     ;đọc một ký tự
13:              MOV    AH,1          ;chuẩn bị đọc
14:              INT    21h           ;đọc một ký tự
15:     ;while không phải ký tự xuống dòng do
16:     WHILE_:
17:              CMP    AL,0Dh        ;CR?
18:              JE     END_WHILE     ;đúng,thoát khỏi vòng lặp
19:     ;cất ký tự vào ngăn xếp và tăng biến đếm
20:              PUSH   AX            ;cất vào ngăn xếp
21:              INC    CX            ;đếm = đếm+1
22:     ;đọc một ký tự
23:              INT    21h           ;đọc một ký tự
24:              JMP  WHILE         ; trở về vòng lặp
25:     END_WHILE:
26:     ;xuốngdòng tiếp theo
27:              MOV    AH,2          ;hàm hiển thị ký tự
28:              MOV    DL,0Dh        ;CR
29:              INT    21h           ;thi hành
30:              MOV    DL,0Ah        ;LF
31:              INT    21h           ;thi hành
32:              JCXZ   EXIT          ;thoát nếu đếm=0
33:     ;for biến đếm do
34:     TOP:
35:     ;lấy một ký tự từ ngăn xếp
36:              POP    DX          ; lấy một ký tự từ ngăn xếp
```

```
37:     ;hiển thị nó
38:             INT   21h          ;hiển thị nó
39:             LOOP  TOP
40:     ;end_for
41:     EXIT:
42:             MOV   AH,4CH
43:             INT   21H
44:     MAIN    ENDP
45:             END   MAIN
```

Bởi lẽ số các ký tự nhập vào là không xác định trước, chương trình dùng CX để đếm nó. Sau đó CX điều khiển vòng lặp FOR hiển thị các lý tự theo thứ tự ngược lại.

Trong các dòng 16-24, chương trình dùng một vòng lặp WHILE để cất các ký từ ngăn xếp và đọc các ký tự mới cho đến khi ký tự về đầu dòng được đánh vào. Mặc dù các ký tự chỉ chứa trong AL, chúng ta vẫn phải cất cả AX vào ngăn xếp bởi vì toán tử trong lệnh PUSH phải là một WORD.

Khi chương trình thoát khỏi vòng lặp WHILE (dòng 25), tất cả các ký tự đều đã được cất vào ngăn xếp với byte thấp của đỉnh ngăn xếp chứa ký tự cất vào sau cùng, AL chứa mã ASCII của ký tự về đầu dòng.

Dòng 32 chương trình kiểm tra xem có ký tự nào được đọc vào hay không. Nếu không thì CX bằng 0 và chương trình nhảy trở về DOS. Nếu ít nhất một ký tự được đọc, chương trình đi vào vòng lặp FOR, trong đó các lệnh POP lặp lại sẽ đưa vào DX số liệu lấy từ ngăn xếp (do đó DL chứa mã của ký tự) và hiển thị các ký tự.

Một ví dụ khi chạy chương trình:

```
C>PRG8_1
?THIS  IS  A  TEST
TSET  A  SI  SIHT

C>PRG8_1
?   ( chỉ ký tự về đầu dòng được đánh vào)

C>
```

8.3. Các thuật ngữ của thủ tục.

Trong chương 6 chúng tôi đã đề cập đến ý tưởng lập trình từ trên xuống. Nội dung của ý tưởng này là phân tích vấn đề ban đầu thành chuỗi các vấn đề con dễ thực hiện hơn. Các ngôn ngữ bậc cao thường dùng các thủ tục để giải quyết các vấn đề con này và chúng ta cũng có thể làm tương tự đối với Hợp ngữ. Như vậy một chương trình Hợp ngữ có thể được tạo nên bằng cách kết hợp các thủ tục.

Trong số các thủ tục sẽ có một thủ tục chính, nó chứa điểm xuất phát của chương trình. Để thực hiện một nhiệm vụ, thủ tục chính gọi một trong số các thủ tục còn lại. Rất có thể các thủ tục này lại gọi các thủ tục khác hay một thủ tục gọi chính nó.

Khi một thủ tục gọi một thủ tục khác, điều khiển được chuyển đến thủ tục được gọi và các lệnh của nó được thi hành. Thủ tục được gọi luôn trả điều khiển về cho thủ tục gọi tại lệnh tiếp theo của dòng lệnh gọi (Hình 8.3). Trong các ngôn ngữ bậc cao cơ chế gọi và trở về là ẩn dối với người lập trình còn trong Hợp ngữ chúng ta có thể thấy rõ nó làm việc như thế nào (xem mục 8.4).

Khai báo thủ tục.

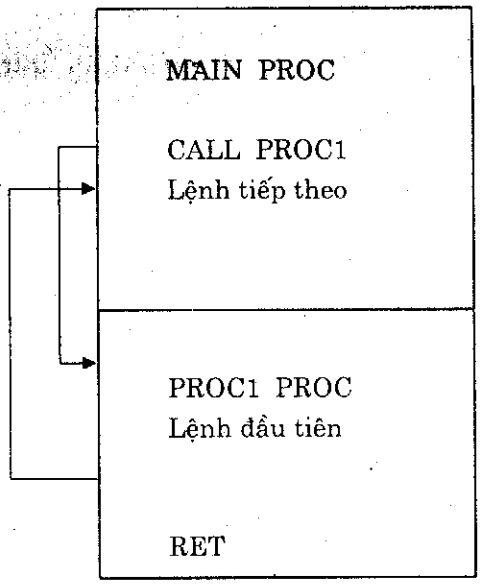
Cú pháp khai báo một thủ tục như sau:

```
name   PROC   type
;thân thủ tục
....
       RET
name   ENDP
```

Trong đó, name là tên của thủ tục định nghĩa bởi người sử dụng. Toán hạng tuỳ chọn type có thể là NEAR hay FAR (NEAR được ngầm định nếu bỏ qua type). NEAR có nghĩa là dòng lệnh gọi thủ tục ở cùng đoạn với thủ tục đó, ngược lại tuỳ chọn FAR có nghĩa dòng lệnh gọi ở trong một đoạn khác. Các phần sau đây ta giả thiết mọi thủ tục đều có kiểu NEAR. Các thủ tục FAR sẽ được trình bày trong chương 14.

RET

Để trả điều khiển về chương trình gọi ta sử dụng lệnh RET (return). Mọi thủ tục (ngoại trừ thủ tục chính) đều phải có một lệnh RET ở đâu đó. Thường thì nó là lệnh cuối cùng của thủ tục.



Liên lạc giữa các thủ tục

Một thủ tục phải có cách nào đó để nhận các giá trị và trả về kết quả cho thủ tục gọi nó. Khác với các thủ tục của ngôn ngữ bậc cao, các thủ tục của Hợp ngữ không có danh sách tham số, vì thế các lập trình viên phải nghĩ cách cho việc liên lạc giữa các thủ tục. Ví dụ nếu chỉ có vài số liệu vào. ra ta có thể chứa chúng trong các thanh ghi. Phương pháp chung cho việc liên lạc giữa các thủ tục sẽ được trình bày trong chương 14.

Chú giải các thủ tục.

Ngoài các yêu cầu về cú pháp, mỗi thủ tục cũng nên có những lời giải thích để bất kỳ ai đọc chương trình nguồn đều có thể hiểu được các thủ tục làm gì, nó lấy số liệu và trả về kết quả ở đâu. Trong cuốn sách này, thông thường chúng tôi chú giải các thủ tục như bằng khối lời bình như sau:

```
; ( miêu tả thủ tục làm gì)
; Vào:  ( nơi lấy thông tin từ chương trình gọi)
; Ra: ( nơi trả về thông tin cho chương trình goi)
; Sử dụng: ( Các chương trình mà thủ tục gọi)
```

8.4. CALL và RET.

Lệnh CALL được dùng để gọi một thủ tục. Có hai kiểu gọi thủ tục là gọi trực tiếp và gián tiếp. Cú pháp của lệnh gọi thủ tục trực tiếp:

```
CALL   name
```

Ở đây name là tên của thủ tục. Cú pháp của lệnh gọi thủ tục gián tiếp:

```
CALL   address_expression
```

với address_expression là một thanh ghi hay ô nhớ chứa địa chỉ của thủ tục.

Lệnh CALL thực hiện các công việc sau đây:

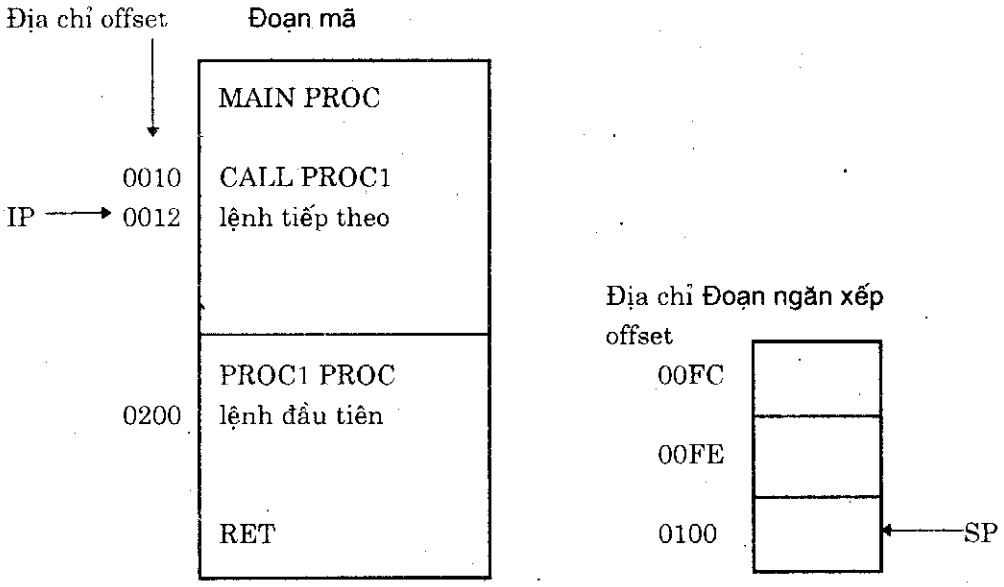
1. Địa chỉ trở về của chương trình gọi được cất vào ngăn xếp. Địa chỉ này là offset của lệnh ngay sau dòng lệnh CALL, dạng segment:offset của nó tại thời điểm thi hành lệnh CALL chứa trong CS : IP.
2. IP được gán bằng địa chỉ offset lệnh đầu tiên của thủ tục. Thao tác này chuyển điều khiển cho thủ tục. (Xem hình 8.4A và 8.4B)

Để trở về từ một thủ tục chúng ta dùng lệnh:

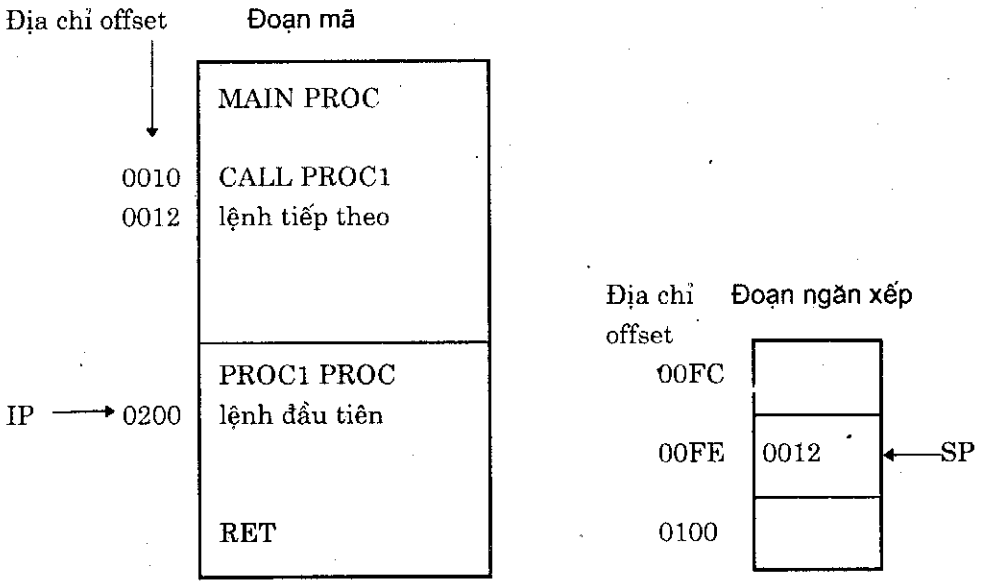
```
RET    pop_value
```

Tham số nguyên pop_value là tuỳ chọn. Trong một thủ tục NEAR, lệnh RET đưa giá trị ở đỉnh ngăn xếp vào IP. Nếu pop_velue xác định bằng N thì N sẽ được cộng vào SP. Điều này tương đương với việc lấy N byte khỏi ngăn xếp. `CS:IP` lúc này chứa địa chỉ trở về dạng `segment:offset` và điều khiển được trả lại cho chương trình gọi (Xem hình 8.5A và 8.5B).

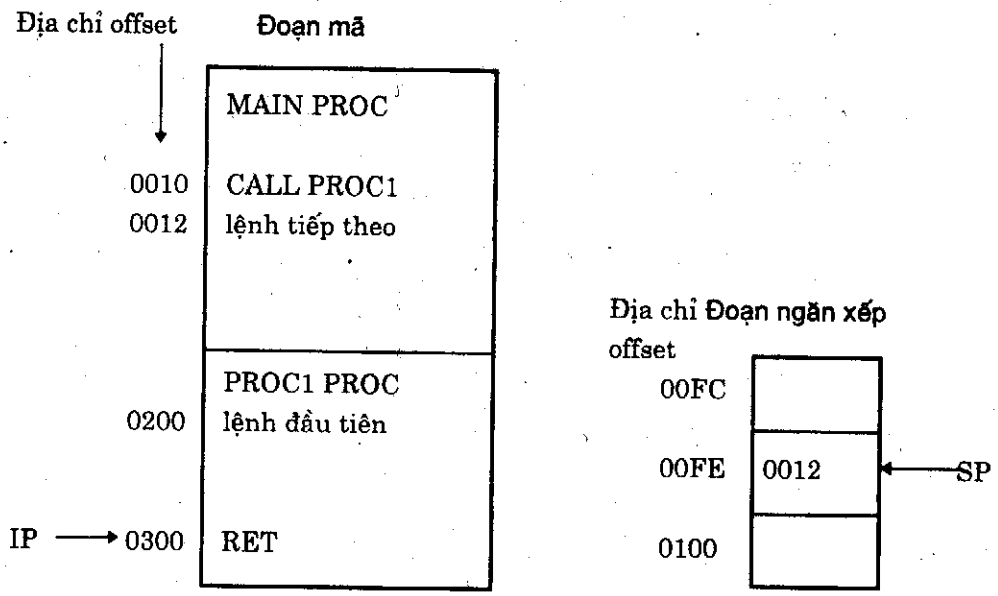
Hình 8.4A Trước lệnh CALL.



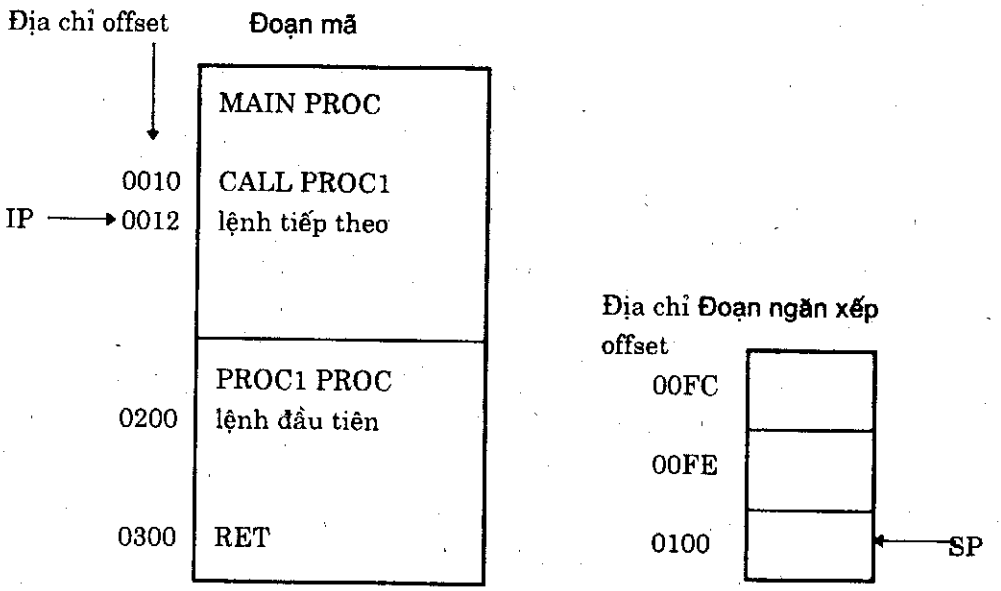
Hình 8.4. Sau lệnh CALL.



Hình 8.5A.Trước lệnh RET.



Hình 8.5B. Sau lệnh RET



8.5 Ví dụ về thủ tục

Để làm ví dụ chúng ta sẽ viết một thủ tục nhân 2 số nguyên dương A và B bằng cách cộng và dịch các bit, đây là một phương pháp để thực hiện phép nhân không dấu trong máy tính (trong chương 9 chúng tôi sẽ trình bày về lệnh nhân).

Thuật toán thực hiện phép nhân:

```
Tích = 0
REPEAT
    IF lsb của B bằng 1 (lsb là bit có trọng lượng
                         thấp nhất)
    THEN
       Tích = Tích+ A
    END_IF
    Dịch trái A
    Dịch phải B
UNTIL B = 0
```

Ví dụ nếu A =111b = 7 và B = 1101b = 13

```
Tích = 0
Vì bit lsb của B bằng 1 nên Tích = Tích+A = 111b
Dịch trái A:  A=1110b
Dịch phải B:  B=110b

Vì bit lsb của B bằng 0 nên:
Dịch trái A:  A=11100b
Dịch phải B:  B=11b
Vì bit lsb của B bằng 1 nên Tích=Tích+A=111b+111100b
                                        =100011b
Dịch trái A:  A=111000b
Dịch phải B:  B=1b

Vì bit lsb của B bằng 1 nên Tích=Tích+A=100011b+111000
                                        =1011011b
Dịch trái A:  A=1110000b
Dịch phải B:  B=0

Vì bit lsb của B bằng 0 nên :
```

```
Tích nhận được=1011011b = 91d
```

Chú ý chúng ta cũng có thể nhận được kết quả như vậy bằng cách thực hiện quá trình nhân thập phân thông thường trên các số nhị phân như sau:

```
        111b
*       1101b
-------------
         111
        000
       111
      111
-------------
     1011011b
```

Trong chương trình sau đây, thuật toán nhân được thực hiện bởi chương trình MULTIPLY. Chương trình chính không có số liệu vào, ra. Chúng ta sẽ dùng chương trình DEBUG để thực hiện các thao tác vào/ra.

Chương trình PGM8_1.ASM

```
1:      TITLE    PGM8_2: MULTIPLICATION  BY  ADD  AND  SHIFT
2:      .MODEL    SMALL
3:      .STACK    100H
4:      .CODE
5:      MAIN      PROC
6:      ;đưa A vào AX và B vào BX nhờ DEBUG
7:                CALL  MULTIPLY
8:      ;DX sẽ chứa kết quả
9:                MOV   AH,4Ch
10:               INT   21h
11:     MAIN      ENDP
12:     MULTIPLY  PROC
13:     ;nhân hai số A và B bằng phép cộng và dịch
14:     ;vào:AX=A; BX=B; Các số nằm trong khoảng(0-FFh)
15:     ;ra: DX=kết quả
16:               PUSH  AX
17:               PUSH  BX
18:               XOR DX,DX    ;tích=0
19:     REPEAT:
20:     ;if B lẻ
21:               TEST  BX,1         ;B lẻ?
```

```
22:                JZ END_IF     ;không, B chẵn
23:      ;then
24:                ADD DX,AX     ;tích=tích+A
25:      END_IF:
26:                SHL AX,1            ;dịch trái A
27:                SHR BX,1            ;dịch phải B
28:      ;until
29:                JNZ REPEAT
30:                POP BX
31:                POP AX
32:                RET
33:      MULTIPLY  ENDP
34:                END MAIN
```

Thủ tục MULTIPLY nhận các biến vào A và B thông qua các thanh ghi AX và BX. Người sử dụng nạp giá trị cho các thanh ghi này trong chương trình DEBUG. Kết quả của phép tính trả về thanh ghi DX. Để tránh tràn, A và B phải được giới hạn trong khoảng từ 0 đến FFh.

Mỗi thủ tục thường bắt đầu bằng việc cất tất cả các thanh ghi mà nó sử dụng trong ngăn xếp và khôi phục lại các thanh ghi này khi kết thúc. Thao tác này là cần thiết bởi lẽ chương trình có thể có các dữ liệu chứa trong các thanh ghi và các thủ tục có thể gây ra các hiệu ứng lề nếu không ghi lại chúng trước. Thậm chí là không cần thiết đối với chương trình này, chúng tôi vẫn cất các thanh ghi AX và BX (dòng 16 và 17) và khôi phục lại chúng (dòng 30 và 31) để làm ví dụ minh hoạ. Các thanh ghi được lấy ra khỏi ngăn xếp theo thứ tự ngược lại với khi chúng được cất vào.

Sau khi xoá DX, thanh ghi sẽ chứa tích, chương trình tiến vào vòng lặp REPEAT (dòng 19-29). Dòng 22 chương trình kiểm tra bit trọng lượng thấp nhất (lsb) của BX. Nếu BX có lsb bằng 1, AX sẽ được cộng vào tích chứa trong DX và BX dịch phải. Vòng lặp kết thúc khi BX = 0. Khi thoát ra, thủ tục sẽ chứa tích trong DX.

Sau khi biên dịch và liên kết chương trình, ta gọi nó trong DEBUG (trong dòng lệnh sau đây, người sử dụng trả lời bằng chữ đậm):

```
C> DEBUG   PGM8_2.EXE

-
```

DEBUG sẽ trả lời bằng dấu nhắc lệnh"-". Để liệt kê chương trình ta dùng lệnh U (Unassemble):

```
-U
177F:0000 E80400      CALL  0007
177F:0003 B44C  MOV   AH,4C
177F:0005 CD21  INT   21
177F:0007 50    PUSH  AX
177F:0008 53    PUSH  BX
177F:0009 33D2  XOR   DX,DX
177F:000B F7C30100    TEST  BX,0001
177F:000F 7402  JZ    0013
177F:0011 03D0  ADD   DX,AX
177F:0013 D1E0  SHL   AX,1
177F:0015 D1EB  SHR   BX,1
177F:0017 75F2  JNZ   000B
177F:0019 5B    POP   BX
177F:001A 58    POP   AX
177F:001B C3    RET
177F:001C E3D1  JCXZ FFEF
177F:001E E38B  JCXZ FFAB
```

Lệnh U của DEBUG sẽ dịch nội dung bộ nhớ thành các lệnh ngôn ngữ máy. Mỗi lệnh được hiển thị địa chỉ dạng segmet:offset, mã máy và mã lệnh Hợp ngữ. Tất cả các số biểu diễn dưới dạng hex. Kết quả thực hiện cho ta thấy chương trình chính nằm từ địa chỉ offset 0000 đến 0005; thủ tục MULTIPLY bắt đầu tại 0007 và kết thúc với lệnh RET tại 001B. Các lệnh sau đó không thuộc chương trình.

Trước khi vào số liệu, ta hãy xem các thanh ghi.

```
-R
AX=0000  BX=0000  CX=001C  DX=0000  SP=0100  BP=0000  SI=0000
DI=0000
DS=176F ES=176F SS=1781 CS=177F IP=0000 NV UP EI PL NZ NA PO NC
177F:0000 E80400 CALL 0007
```

Giá trị khởi tạo SP bằng 100h có nghĩa là chúng ta dành ra 100h byte cho ngăn xếp. Để xem ngăn xếp, chúng ta có thể liệt kê bộ nhớ bằng lệnh D.

```
-DSS:F0 FF
1781:00f0 00 00 00 00 00 00 00 6f 17-A4 13 07 00 6F 17 00 00
```

Lệnh DSS:F0 FF hiển thị các byte nhớ từ địa chỉ SS:F0 đến SS:FF. Đây chính là 16 byte cuối cùng của ngăn xếp. Nội dung mỗi byte được hiển thị dưới dạng số hex hai chữ số. Vì là ngăn xếp rỗng, các số được hiển thị ở đây không có ý nghĩa.

Trước khi thực hiện chương trình, chúng ta phải đưa các số A và B vào các thanh ghi AX và BX (ở đây ta sẽ cho A=7 và B=13=Dh). Để vào A ta dùng lệnh R:

```
-RAX
AX 0000:7
```

RAX có nghĩa là chúng ta muốn thay đổi nội dung của AX. DEBUG sẽ hiển thị giá trị hiện tại, theo sau là dấu hai chấm (:) và đợi chúng ta vào giá trị mới. Tương tự chúng ta có thể khởi tạo BX:

```
-RBX
BX 0000:D
```

Bây giờ ta hãy xem lại các thanh ghi:

```
-R
AX=0007 BX=000D CX=001C DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=176F ES=176F SS=1781 CS=177F IP=0000 NV UP EI PL NZ NA PO NC
177F:0000 E80400 CALL 0007
```

Lúc này AX và BX đã chứa các giá trị khởi đầu.

Để xem lệnh CALL ảnh hưởng đến các cờ như thế nào chúng ta sử dụng lệnh T (Trace). Nó sẽ thi hành một lệnh và hiển thị các thanh ghi:

```
-T
AX=0007 BX=000D CX=001C DX=0000 SP=00FE BP=0000 SI=0000 DI=0000
DS=176F ES=176F SS=1781 CS=177F IP=0007 NV UP EI PL NZ NA PO NC
177F:0007 50  PUSH AX
```

Chúng ta hãy chú ý hai thanh ghi bị thay đổi:

1. IP chứa 0007, offset bắt đầu của thủ tục MULTIPLY.

2. SP giảm từ 100h xuống 00FE bởi vì lệnh CALL cất địa chỉ trở về của thủ tục MAIN trong ngăn xếp.

Ta xem lại 16 byte cuối cùng của ngăn xếp:

```
-DSS:F0 FF
1781:00f0 00 00 00 00 00 00 00 6f 17-A4 13 07 00 6F 17 03 00
```

Địa chỉ trở về là 0003 nhưng lại được hiển thị là 03 00 vì DEBUG hiển thị byte thấp trước byte cao.

Ba lệnh đầu tiên của thủ tục MAIN cất AX, BX vào ngăn xếp và xoá DX. Để thực hiện chúng, ta dùng lệnh G (go). Cú pháp của lệnh này như sau:

```
G   offset
```

Lệnh này thực hiện chương trình và dừng lại tại offset xác định. Nhờ bản dịch ngược chúng ta có thể thấy ngay rằng sau lệnh XOR DX,DX là offset 000Bh.

```
-GB
AX=0007 BX=000D CX=001C DX=0000 SP=00FA BP=0000 SI=0000 DI=0000
DS=176F ES=176F SS=1781 CS=177F IP=000B NV UP EI PL ZR NA PE NC
177F:000B  F7C30100  TEST BX,01
```

Ta thấy rằng hai lệnh PUSH làm giảm SP 4 đơn vị: từ 00FEh xuống 00FAh. Ngăn xếp bây giờ như sau:

```
-DSS:F0 FF
1781:00f0 00 00 00 00 00 00 00 6f 17-A4 13 0D 00 07 00 03 00
```

Lúc này ngăn xếp chứa ba từ: các giá trị của BX (000D), AX (0007) và địa chỉ trở về (0003). Chúng được hiển thị như sau: 0D 00 07 00 03 00.

Bây giờ chúng ta hãy xem chương trình làm việc ra sao. Trước hết, ta cho chương trình chạy đến cuối vòng lặp REPEAT tại offset 0017h:

```
-G17
AX=000E BX=0006 CX=001C DX=0007 SP=00FA BP=0000 SI=0000 DI=0000
DS=176F ES=176F SS=1781 CS=177F IP=0017 NV UP EI PL NZ AC PE CY
177F:0017 75F2 JNZ 000B
```

Bởi vì giá trị khởi đầu của BX là 0Dh=1101b nên lsb của nó bằng 1, như vậy AX được cộng vào tích trong DX:

DX=DX+AX=0000h+0007h=0007h. AX dịch trái 1, tức là được nhân đôi: AX=14h=0110b, BX dịch phải hay chia đôi nó: BX=0006h =0110b.

Để tới được đỉnh vòng lặp ta dùng lệnh T:

```
-T
AX=000E BX=0006 CX=001C DX=0007 SP=00FA BP=0000 SI=0000 DI=0000
DS=176F ES=176F SS=1781 CS=177F IP=000B NV UP EI PL NZ AC PE CY
177F:000B F7C30100  TEST BX,0001
```

Và ta lại thực hiện đến cuối vòng lặp:

```
-G17
AX=000E BX=0006 CX=001C DX=0007 SP=00FA BP=0000 SI=0000 DI=0000
DS=176F ES=176F SS=1781 CS=177F IP=0017 NV UP EI PL NZ AC PE CY
177F:0017 75F2  JNZ 000B
```

Bởi vì BX=0006h=110b, lsb của nó bằng 0 nên DX giữ nguyên giá trị. AX dịch trái thành 11100b = 1Ch và BX dịch phải thành 11b = 3h.

Sau hai lần lặp nữa chúng ta thu được tích trong DX. Bạn hãy xem các thanh ghi AX, BX, CX và DX thay đổi như thế nào:

```
-T
AX=001C BX=0003 CX=001C DX=0007 SP=00FA BP=0000 SI=0000 DI=0000
DS=176F ES=176F SS=1781 CS=177F IP=000B NV UP EI PL NZ AC PE NC
177F:000B F7C30100  TEST BX,0001
-G17
AX=0038 BX=0001 CX=001C DX=0023 SP=00FA BP=0000 SI=0000 DI=0000
DS=176F ES=176F SS=1781 CS=177F IP=0017 NV UP EI PL NZ AC PE CY
177F:0017 75F2  JNZ 000B

-T
AX=0038 BX=0001 CX=001C DX=0023 SP=00FA BP=0000 SI=0000 DI=0000
DS=176F ES=176F SS=1781 CS=177F IP=000B NV UP EI PL NZ AC PE CY
177F:000B F7C30100  TEST BX,0001

-G17
AX=0070 BX=0000 CX=001C DX=005B SP=00FA BP=0000 SI=0000 DI=0000
DS=176F ES=176F SS=1781 CS=177F IP=0017 NV UP EI PL ZR AC PE CY
177F:0017 75F2  JNZ 000B
```

Lần dịch trái cuối cùng làm BX = 0, ZF = 1 và vòng lặp sẽ kết thúc. Tích là 91 = 5bh chứa trong DX.

Để kết thúc chương trình, chúng ta duyệt qua lệnh JNZ và hai lệnh POP:

```
-T
AX=0070 BX=0000 CX=001C DX=005B SP=00FA BP=0000 SI=0000 DI=0000
DS=176F ES=176F SS=1781 CS=177F IP=0019 NV UP EI PL ZR AC PE CY
177F:0019 5B  POP BX

-T
AX=0070 BX=000D CX=001C DX=005B SP=00FC BP=0000 SI=0000 DI=0000
DS=176F ES=176F SS=1781 CS=177F IP=0019 NV UP EI PL ZR AC PE CY
177F:001A 58  POP AX

-T
AX=0007 BX=000D CX=001C DX=005B SP=00FE BP=0000 SI=0000 DI=0000
DS=176F ES=176F SS=1781 CS=177F IP=0019 NV UP EI PL ZR AC PE CY
177F:001B C3  RET
```

Hai lệnh POP trả lại các thanh ghi các giá trị nguyên thuỷ của chúng. Ta hãy xem ngăn xếp:

```
-DSS:F0 FF
1781:00f0 00 00 00 00 00 00 00 6f 17-A4 13 07 00 6F 17 03 00
```

Các giá trị 000Dh và 0007 không còn trong ngăn xếp nữa. Đây không phải là kết quả của lệnh POP mà do DEBUG cũng sử dụng ngăn xếp.

Cuối cùng ta chạy lệnh RET:

```
-T
AX=0007 BX=000D CX=001C DX=005B SP=0100 BP=0000 SI=0000 DI=0000
DS=176F ES=176F SS=1781 CS=177F IP=0019 NV UP EI PL ZR AC PE CY
177F:0003 B44C  MOV AH,4C
```

Lệnh RET nạp cho IP địa chỉ trở về chương trình MAIN: IP = 0003. SP cũng được trả về giá trị nguyên thuỷ của nó: SP = 100h. Để kết thúc chương trình chúng ta chỉ cần dùng lệnh G:

```
-G
Program terminated normally.
```

và ta thoát khỏi DEBUG bằng lệnh Q (quit):

```
-Q
>C
```

TỔNG KẾT:

- Ngăn xếp là vùng chứa dữ liệu tạm thời của cả chương trình ứng dụng lẫn hệ điều hành.
- Ngăn xếp là cấu trúc dữ liệu làm việc theo nguyên tắc vào trước-ra sau. SS:SP trỏ đến đỉnh ngăn xếp.
- PUSH, PUSHF, POP và POPF là các lệnh làm thay đổi ngăn xếp. Lệnh PUSH thêm một từ mới vào ngăn xếp còn POP lấy ra đỉnh ngăn xếp. PUSHF cất thanh ghi cờ vào ngăn xếp còn POPF lấy ra đỉnh ngăn xếp đưa vào thanh ghi cờ.
- SP giảm đi 2 khi thực hiện lệnh PUSH và PUSHF và tăng lên 2 khi thực hiện lệnh POP và POPF. SP được khởi tạo trỏ đến từ đầu tiên sau đoạn ngăn xếp khi chương trình nạp vào bộ nhớ.
- Thủ tục là một chương trình con. Một chương trình Hợp ngữ thường chia thành hai thủ tục. Một trong chúng là thủ tục chính, nó chứa điểm xuất phát của chương trình. Mỗi thủ tục có thể gọi các thủ tục khác hay có thể gọi chính nó.
- Có hai loại thủ tục: NEAR và FAR. Một thủ tục NEAR nằm trong cùng một đoạn với chương trình gọi nó còn thủ tục FAR nằm trong đoạn một khác.
- Lệnh CALL dùng để gọi các thủ tục. Đối với một thủ tục NEAR, lệnh CALL cất địa chỉ offset của lệnh tiếp theo đồng thời nạp cho IP địa chỉ offset của lệnh đầu tiên của thủ tục.
- Thủ tục kết thúc bằng lệnh RET. Lệnh này lấy giá trị của IP từ ngăn xếp và điều khiển được trả về chương trình gọi. Để có được địa chỉ trở về đúng, thủ tục phải chắc chắn rằng nó đang ở đỉnh ngăn xếp khi thi hành lệnh RET.
- Trong Hợp ngữ, các thủ tục thường trao đổi số liệu thông qua các thanh ghi.

Các thuật ngữ tin học:

direct procedure call	Gọi thủ tục trực tiếp bằng lệnh CALL name
FAR procedure	Thủ tục gọi được ở tầm xa: Một thủ tục có thể được gọi bởi thủ tục nằm ở bất kỳ đoạn nào.
indirect procedure call	Gọi thủ tục gián tiếp bằng lệnh CALL addr_exp
NEAR procedure	Thủ tục gọi gần: Một thủ tục chỉ có thể được gọi bởi thủ tục khác nằm trong cùng đoạn.
top of the stack	Đỉnh stack: Từ dữ liệu cuối cùng được cất vào ngăn xếp.

Các lệnh mới học:

CALL	**POPF**	**PUSHF**
POP	**PUSH**	**RET**

Bài tập:

1. Giả sử đoạn ngăn xếp được khai báo như sau:

```
.STACK      100h
```

   a. SP bằng bao nhiêu khi bắt đầu chương trình (dạng số hex) ?

   b. Ngăn xếp chứa được nhiều nhất bao nhiêu từ ?

2. Biết rằng AX = 1234h, BX = 5678h, CX = 9ABCh và SP = 1000h. Hãy cho biết nội dung các thanh ghi AX, BX, CX và SP khi thực hiện các lệnh sau đây:

```
PUSH  AX
PUSH  BX
XCHG  AX,CX
POP   CX
PUSH  AX
POP   BX
```

3. Khi ngăn xếp điền đầy vùng cung cấp cho nó thì SP = 0. Nếu có một từ nữa lại được cất vào ngăn xếp thì SP sẽ như thế nào ?, điều gì sẽ xảy ra với chương trình ?

4. Giả thiết chương trình chứa các đòng lệnh sau đây:

```
CALL  PROC1
MOV   AX,BX
```

   Biết rằng lệnh MOV nằm ở địa chỉ 08FD:0203 và PROC1 là thủ tục NEAR bắt đầu tại địa chỉ 08FD:0300, SP = 010Ah. Cho biết nội dung của IP và SP sau mỗi lệnh. Giá trị đỉnh ngăn xếp là bao nhiêu ?

5. Giả thiết SP = 0200h và giá trị đỉnh ngăn xếp là 012Ah. Cho biết nột dung của IP và SP:

   a. Sau khi thi hành lệnh RET (của một thủ tục NEAR).

   b. Sau khi thi hành lệnh RET

a. Đưa giá trị của đỉnh ngăn xếp vào AX, không làm thay đổi ngăn xếp.

b. Đưa từ nằm sau đỉnh ngăn xếp vào AX, không làm thay đổi ngăn xếp.

c. Đổi chỗ hai từ ở đỉnh ngăn xếp. Bạn có thể sử dụng AX và BX.

7. Ngăn xếp phải được thủ tục trả về chương trình gọi giống hệt như khi nó nhận được. Tuy nhiên cũng tốt nếu như có các thủ tục làm thay đổi ngăn xếp. Ví dụ chúng ta muốn viết một thủ tục NEAR: SAVE_REGS để cất các thanh ghi BX, CX, DX, SI, DI, BP, DS và ES vào ngăn xếp. Sau khi cất các thanh ghi, ngăn xếp như sau:

nội dung ES

.

.

.

nội dung CX
nội dung BX
địa chỉ trở về

Thật không may là bây giờ SAVE_REGS không thể trở về chương trình gọi bởi lẽ địa chỉ trở về không nằm ở đỉnh ngăn xếp.

a. Hãy viết thủ tục SEVE_REGS khắc phục khó khăn nêu trên.

b. Viết thủ tục RESTORE_REGS phục hồi các thanh ghi được cất bởi chương trình SAVE_SEGS.

Các bài tập lập trình:

8. Viết chương trình để nhập vào các dòng văn bản trong đó các từ ngăn cách nhau bởi các ký tự trắng, kết thúc bằng ký tự trở về đầu dòng và hiển thị chúng theo cùng thứ tự nhưng các chữ trong mỗi từ được đảo ngược. Ví dụ: " this is a text" được đổi thành " siht si a txet". Gợi ý: sửa lại chương trình PGM8_2 trong mục 8.3.

9. Một biểu thức đại số có chứa các dấu ngặc (như ( ); [ ];{ }) được coi là hợp lệ nếu thoả mãn:

(a) số các dấu ngoặc trái và ngặc phải của mỗi loại bằng nhau

(b) các dấu ngoặc tương ứng phải cùng loại.

Ví dụ:

(a + [b -{ c * (d - e) }] + f) là hợp lệ, nhưng:

(a + [b -{ c * (d - e)] }+ f) không hợp lệ.

Công việc kiểm tra có thể thực hiện nhờ ngăn xếp. Biểu thức được quét từ trái qua phải. Nếu gặp dấu ngoặc trái ta cất nó vào ngăn xếp. Nếu gặp dấu ngoặc phải ta lấy ra từ ngăn xếp dấu ngoặc nữa (nếu ngăn xếp rỗng tức là có quá nhiều dấu ngoặc phải) và so sánh: nếu cùng loại ta "quét tiếp" còn nếu khác loại có nghĩa là biểu thức được điền các dấu ngoặc không tương ứng. Đến cuối biểu thức nếu ngăn xếp rỗng: biểu thức hợp lệ, nếu ngăn xếp còn dữ liệu tức là có quá nhiều dấu ngoặc trái.

Bạn hãy viết một chương trình để người sử dụng đánh vào một biểu thức đại số có chứa các dấu ngoặc (ngoặc tròn, vuông và nhọn) và kết thúc bằng ký tự về đầu dòng. Trong khi đánh vào biểu thức, chương trình kiểm tra mỗi ký tự. Nếu tại một thời điểm nào đó vào các dấu ngoặc không hợp lệ (quá nhiều dấu ngặc phải hay các cặp dấu ngoặc tương ứng không cùng loại), chương trình sẽ báo cho người sử dụng bắt đầu lại. Khi ký tự về đầu dòng được đánh vào, nếu biểu thức hợp lệ chương trình sẽ hiển thị thông báo "Biểu thức hợp lệ", nếu ngược lại, chương trình hiển thị thông báo "quá nhiều ngoặc trái". Trong cả hai trường hợp chương trình đều hỏi xem người sử dùng có muốn tiếp tục hay không. Nếu trả lời là 'Y', chương trình sẽ được thực hiện lại.

Chương trình của bạn không cần phải ghi lại chuỗi vào, chỉ kiểm tra biểu thức có được điền các dấu ngoặc hợp lệ hay không.

Một ví dụ khi chạy chương trình:

```
Bạn vào một biểu thức đại số:
(a+b))quá nhiều dấu ngoặc phải.Bạn vào lại!
Bạn vào một biểu thức đại số:
[a+(b-c)*d]
Biểu thức hợp lệ.
Nhấn phím 'Y' nếu bạn muốn tiếp tục:Y
Bạn vào một biểu thức đại số:
(a+b*(c-d)-e]vào sai kiểu.Bạn vào lại!
Bạn vào một biểu thức đại số:
((a+[b-{c*(d-e)}]+f) có qúa nhiều ngoặc trái.Bạn vào lại!
Bạn vào một biểu thức đại số:
tôi đã vào đủ!
```

```
Biểu thức hợp lệ.
Nhấn phím 'Y' nếu bạn muốn tiếp tục:N
```

Phương pháp sau đây có thể dùng để tạo ra các số ngẫu nhiên trong khoảng từ 1 đến 32767:

Bắt đầu bằng một số bất kỳ trong miền giới hạn.
Dịch trái một bit.
Thay bit 0 bằng kết quả phép XOR bit 14 và bit 15.
Xoá bit 15.

Viết các thủ tục sau đây:

a. Thủ tục READ để người sử dụng vào một số nhị phân và chứa nó trong AX. Bạn có thể dùng các lệnh nhập các số nhị phân được đưa ra trong mục 7.3.

b. Thủ tục RANDOM nhận một số vào AX và trả về một số ngẫu nhiên trong AX.

c. Thủ tục WRITE hiển thị AX dưới dạng nhị phân, Bạn thể sử dụng thuật toán đã nêu trong mục 7.4.

Cuối cùng, bạn hãy viết một chương trình hiển thị môt dấu hỏi chấm (?), gọi READ để đọc một số nhị phân sau đó gọi RANDOM và WRITE để tính toán và hiển thị 100 số ngẫu nhiên. Các số được hiển thị thành các dòng 4 số và hai số trong cùng một dòng ngăn cách nhau bằng 4 ký tự trắng.

Chương 9

CÁC LỆNH NHÂN VÀ CHIA

Tổng quan

Trong chương 7 chúng ta đã thấy việc nhân và chia được thực hiện ra sao bằng cách dịch các bit trong một byte hay word. Chúng ta có thể sử dụng các phép dịch trái hoặc phải để nhân hoặc chia cho một luỹ thừa của 2. Trong chương này chúng tôi sẽ giới thiệu các lệnh dùng để nhân hoặc chia cho bất kỳ số nào.

Quá trình nhân và chia là khác nhau đối với các số có dấu và không có dấu, do đó cũng có những lệnh nhân và chia khác nhau cho các số không dấu và có dấu. Ngoài ra các lệnh cũng phân biệt dạng byte và word. Các phần từ 9.1 đến 9.4 sẽ trình bày chi tiết về các lệnh.

Một trong những ứng dụng có ý nghĩa nhất của việc nhân và chia là thực hiện công việc vào ra các số thập phân. Trong phần 9.5 chúng ta sẽ viết các thủ tục thực hiện các thao tác này, ứng dụng này sẽ mở rộng rất nhiều cho khả năng vào ra trong chương trình của bạn.

9.1 Các lệnh MUL và IMUL

Sự khác nhau giữa phép nhân không dấu và có dấu.

Trong phép nhân các số nhị phân, các số có dấu và không dấu phải được "đối xử" khác nhau. Chẳng hạn chúng ta muốn nhân các số 8 bit 10000000 và 11111111 với nhau. Nếu coi chúng là các số không dấu, chúng lần lượt bằng 128 và 255. Tích số bằng :

32640 = 0111111110000000b. Nhưng khi xem chúng là các số có dấu, chúng lại bằng -128 và -1 do đó tích số sẽ bằng 128 = 0000000010000000b.

Vì phép nhân các số không dấu và có dấu dẫn đến những kết quả khác nhau cho nên có 2 lệnh nhân là: MUL (Multiply) cho các số không dấu và IMUL (Integer Multiply) cho các số có dấu. Các lệnh này làm việc với byte hoặc word. Kết quả của phép nhân 2 byte là một word và của 2 word là một từ kép (double word, 32 bit). Cú pháp của các lệnh này như sau:

```
MUL   toán hạng nguồn
```

Và

```
IMUL  toán hạng nguồn
```

Dạng byte của các lệnh nhân

Khi nhân các byte với nhau, một số được chứa trong toán hạng nguồn và số còn lại được giả thiết đã chứa trong AL. Toán hạng nguồn có thể là một thanh ghi hay một byte nhớ nhưng không thể là một hằng số.

Khi nhân các số dương (bit msb bằng 0) MUL và IMUL cho cùng một kết quả.

Tác động của MUL và IMUL đến các cờ trạng thái:

```
SF, ZF, AF, PF              Không xác định

CF/OF:
Sau lệnh MUL, CF/OF = 0  Nếu nửa cao của kết quả
                         bằng 0
                    = 1  Trong các trường hợp khác
Sau lệnh IMUL, CF/OF= 0  Nếu nửa cao của kết quả là
                    phần mở rộng dấu của nửa
                    thấp(có nghĩa là các bit
                    của nửa cao giống với bit
                    dấu của nửa thấp).
                     = 1  Trong các trường hợp khác.
```

Trong cả 2 lệnh, CF/OF = 1 có nghĩa là kết quả quá lớn để chứa trong nửa thấp của toán hạng đích (AL trong phép nhân với byte và AX trong phép nhân với word).

Các ví dụ

Để mô tả các lệnh MUL và IMUL, chúng ta sẽ làm thử vài ví dụ. Vì thông thường việc nhân các số hex thường rất khó, chúng ta sẽ đoán trước kết quả

bằng cách đổi giá trị hex của số nhân và số bị nhân ra dạng thập phân, thực hiện phép nhân các số thập phân rồi đổi lại kết quả ra dạng hex.

Ví dụ 9.1 Giả sử AX chứa 1 và BX chứa FFFFh :

Lệnh	Tích số dạng thập phân	Tích số dạng hex	DX	AX	CF/OF
MUL BX	65535	0000FFFF	0000	FFFF	0
IMUL BX	-1	FFFFFFFF	FFFF	FFFF	0

Trong lệnh MUL do DX = 0 nên CF/OF = 0. Trong lệnh IMUL do nội dung của BX dạng có dấu là -1 nên kết quả cũng bằng -1. CF/OF = 0 vì DX là sự mở rộng dấu của AX.

Ví dụ 9.2 Giả sử AX và BX cùng chứa FFFFh:

Lệnh	Tích số dạng thập phân	Tích số dạng hex	DX	AX	CF/OF
MUL BX	4294836225	FFFE0001	FFFE	0001	1
IMUL BX	-1	00000001	0000	0001	0.

Trong lệnh MUL CF/OF = 1 vì DX không bằng 0. Điều này phản ánh thực tế là kết quả FFFE0001 quá lớn để có thể chứa trong AX. Trong lệnh IMUL do AX và BX cùng chứa -1 nên kết quả là 1. DX là phần mở rộng dấu của AX nên CF/OF = 0.

Ví dụ 9.3 Giả sử AX chứa 0FFFh:

Lệnh	Tích số dạng thập phân	Tích số dạng hex	DX	AX	CF/OF
MUL AX	16769025	00FFE001	00FF	E001	1
IMUL AX	16769025	00FFE001	00FF	E001	1

Vì bit msb của AX bằng 0 nên cả 2 lệnh cho cùng một kết quả. Vì kết quả quá lớn để chứa trong AX nên CF/OF = 1.

Ví dụ 9.4 Giả sử AX chứa 0100h và CX chứa FFFFh

Lệnh	Tích số dạng thập phân	Tích số dạng hex	DX	AX	CF/OF
MUL CX	16776960	00FFFF00	00FF	FF00	1
IMUL CX	-256	FFFFFF00	FFFF	FF00	0

Trong lệnh MUL tích FFFF00 nhận được bằng cách thêm 2 chữ số 0 vào giá trị của toán hạng nguồn FFFFh. Do tích số qua lớn để có thể chứa trong AX nên CF/OF = 1. Trong lệnh IMUL chứa 256 và CX chứa -1 nên kết quả là -256 và có thể biểu diễn dưới dạng 16 bit là FF00h. DX chứa phần mở rộng của AX nên CF/OF = 0.

Ví dụ 9.5 Giả sử AL chứa 80h còn BL chứa FFh:

Lệnh	Tích số dạng thập phân	Tích số dạng hex	AH	AL	CF/OF
MUL BL	128	7F80	7F	80	1
IMUL BL	128	0080	00	80	1

Đối với phép nhân các byte kết quả 16 bit được chứa trong AX.

Trong lệnh MUL tích số là 7F80, do 8 bit cao của tích khác 0 nên CF/OF= 1.

Trong lệnh IMUL đây là một trường hợp đáng ngờ. Ta có 80h = -128, FFh =-1 nên tích số là 128 = 0080h. AH không chứa phần mở rộng của AL nên CF/OF = 1. Điều này phản ánh sự thật là AL không chứa kết quả đúng về mặt dấu, vì 80h khi được hiểu là số thập phân có dấu thì bằng -128.

9.2 Các ứng dụng đơn giản của MUL và IMUL

Để các bạn làm quen với việc lập trình với các lệnh MUL và IMUL chúng tôi sẽ chỉ ra ở đây một số thao tác đơn giản được thực hiện ra sao với các lệnh này.

Ví dụ 9.6 Hãy dịch lệnh gán A = 5 * A - 12 * B của ngôn ngữ bậc cao ra các lệnh của hợp ngữ. Coi A và B là các biến kiểu word và giả sử rằng không có hiện tượng tràn xảy ra. Sử dụng lệnh IMUL cho các phép nhân.

Lời giải:

```
MOV     AX,5        ;AX=5
IMUL    A           ;AX=5*A
MOV     A,AX        ;A=5*A
MOV     AX,12       ;AX=12
IMUL    B           ;AX=12*B
SUB     A,AX        ;A=5*A-12*B
```

Ví dụ 9.7 Viết một thủ tục mang tên FACTORIAL tính N! với N là số nguyên dương. Thủ tục nhận giá trị.N trong thanh ghi CX và trả lại kết quả trong thanh ghi AX. Giả thiết không có hiện tượng tràn.

Lời giải:

Ta đã biết định nghĩa: N! = 1 nếu N = 1

= N * (N-1) * (N-2) * ..* 1 nếu N >1.

Dưới đây là thuật toán:

```
Tích = 1
Số hạng = N
FOR N lần DO
   Tích = Tích*Số hạng
   Số hạng = Số hạng - 1
ENDFOR
```

Ta có đoạn lệnh của thủ tục như sau:

```
FACTORIAL    PROC
;Thủ tục tính N!
;Vào:    CX=N
;Ra :    AX=N!
         MOV   AX,1  ;AX chứa tích số
TOP:
         MUL   CX    ;Tích=Tích*Số hạng
         LOOP  TOP
         RET
FACTORIAL    ENDP
```

Trong trường hợp này CX đóng vai trò vừa là biến đếm vòng lặp vừa là số hạng. Lệnh LOOP tự động giảm nó trong mỗi lần lặp. Chúng ta đã giả sử tích số không vượt quá 16 bit.

9.3 Các lệnh DIV và IDIV

Mỗi khi thực hiện phép chia chúng ta nhận được 2 kết quả : thương số và số dư. Cũng như đối với phép nhân, có những lệnh riêng biệt cho phép chia không dấu và có dấu. Lệnh DIV (Divide) được dùng cho phép chia không dấu còn lệnh IDIV (Integer Divide) dùng cho các phép chia có dấu. Cú pháp của lệnh như sau:

```
   DIV      Số chia
Và
   IDIV     Số chia
```

Các lệnh này chia số 16 hoặc 32 bit cho số 8 bit hoặc 16 bit. Thương số và số dư có cùng kích thước với số chia.

Dạng Byte của các lệnh chia

Trong dạng này số chia là một thanh ghi 8 bit hay một byte bộ nhớ. Số bị chia 16 bit được giả định là đã chứa trong AX. Sau khi thực hiện phép chia thương số 8 bit chứa trong AL và số dư 8 bit chứa trong AH. Số chia không thể là hằng số.

Dạng Word của các lệnh chia

Trong trường hợp này số chia là thanh ghi 16 bit hay một từ nhớ (16 bit). Số bị chia giả định đã chứa trong DX:AX. Sau khi phép chia được thực hiện thương số 16 bit được chứa trong AX và số dư cũng 16 bit được chứa trong DX. Số chia không thể là hằng số.

Đối với phép chia có dấu, số dư có cùng kích thước với số bị chia. Nếu cả số bị chia và số chia đều dương các lệnh DIV và IDIV cho cùng một kết quả.

Tất cả các cờ trạng thái đều không xác định sau các lệnh DIV và IDIV.

Sự tràn trong phép chia

Rất có thể xảy ra trường hợp là thương số quá lớn để có thể chứa trong toán hạng đích xác định(AL hay AX). Điều này có thể xảy ra khi số chia nhỏ hơn nhiều so với số bị chia. Khi xảy ra điều này chương trình sẽ dừng lại (như các bạn sẽ thấy sau này) và hệ thống đưa ra thông báo "Divide Overflow".

Ví dụ 9.8 Giả sử AX chứa 0005h, DX chứa 0000h và BX chứa 0002h.

Lệnh	Thương số dạng thập phân	Số dư dạng thập phân	AX	DX
DIV BX	2	1	0002	0001
IDIV BX	2	1	0002	0001

Chia 5 cho 2 chúng ta nhận được 2 và dư 1. vì cả số chia và số bị chia đều là các số dương nên 2 lệnh DIV và IDIV cho cùng một kết quả.

Ví dụ 9.9 Giả sử DX chứa 0000h, AX chứa 0005h và BX chứa FFFEh.

Lệnh	Thương số dạng thập phân	Số dư dạng thập phân	AX	DX
DIV BX	0	5	0000	0005
IDIV BX	-2	1	FFFE	0001

Đối với lệnh DIV, số bị chia là 5 trong khi số chia là FFFEh = 65534. 5 chia cho 65534 được 0 và dư 5.

Đối với lệnh IDIV số bị chia là 5 trong khi số chia là FFFEh = -2. 5 chia cho -2 được -2 và dư 1.

Ví dụ 9.10 Giả sử DX chứa FFFFh, AX chứa FFFBh và BX chứa 0002.

Lệnh	Thương số dạng thập phân	Số dư dạng thập phân	AX	DX
IDIV BX	-2	-1	FFFE	FFFF
IDIV BX	DIVIDE OVERFLOW			

Trong lệnh IDIV, số bị chia lã DX:AX = FFFFFFFBh = -5, số chia là BX = 2. -5 chia cho 2 được 2 và dư -1 = FFFFh

Trong lệnh DIV, số bị chia DX:AX = FFFFFFFBh = 4294967291 và số chia là 2 thương số chính xác bằng 2147483646 = 7FFFFFFEh. Số này quá lớn để có thể chứa trong AX, do đó máy tính thông báo DIVIDE OVERFLOW và chương

trình ngừng lại. Ví dụ trên cho thấy điều gì có thể xảy ra khi số chia nhỏ hơn nhiều so với số bị chia.

Ví dụ 9.11 Giả sử AX chứa 00FBh và BL chứa FFh.

Lệnh	Thương số dạng thập phân	Số dư dạng thập phân	AX	AL
DIV BL	0	251	FB	00
IDIV BL	DIVIDE OVERFLOW			

Trong phép chia cho byte, số bị chia chứa trong AX, Thương số đặt trong AL và số dư đặt trong AH. Đối với lệnh DIV, số bị chia là 00FBh = 251 và số chia là FFh = 256. 251 chia 256 được thương là 0 và dư 251 = FBh. Đối với lệnh IDIV, số bị chia là 00FBh = 251 còn số chia là FFh = -1. Chia 251 cho -1 nhận được thương là -251, số này quá lớn để có thể chứa trong AL do đó thông báo DIVIDE OVERFLOW được đưa ra màn hình.

9.4 Sự mở rộng dấu của số bị chia

Phép chia cho word

Trong phép chia cho word, số bị chia được đặt trong DX:AX ngay cả khi số bị chia có thể chứa vừa vặn trong AX. Trong trường hợp này DX phải được chuẩn bị như sau:

1. Đối với lệnh DIV, DX phải được xoá.

2. Đối với lệnh IDIV DX phải được làm thành phần dấu mở rộng của AX. Phép mở rộng được thực hiện bằng lệnh CWD (Convert Word to Double word).

Ví dụ 9.12 Chia -1250 cho 7.

Lời giải:

```
MOV   AX,-1250    ;AX chứa số bị chia
CWD               ;mở rộng dấu vào DX
MOV   BX,7        ;BX chứa số chia
IDIV  BX          ;AX chứa thương số DX chứa số dư
```

Phép chia cho byte

Trong phép chia cho byte, số bị chia đặt trong AX. Trong trường hợp số bị chia thực sự chỉ là một byte, AH cần được chuẩn bị như sau:

1. Đối với lệnh DIV, AH cần được xoá.

2. Đối với lệnh IDIV, AH cần được làm thành phần dấu mở rộng của AL. Phép mở rộng này được thực hiện bằng lệnh CBW (Convert Byte to Word).

Ví dụ 9.13 Chia giá trị có dấu của biến kiểu byte XBYTE cho -7.

Lời giải:

```
MOV   Al,XBYTE    ;AL chứa số bị chia
CBW               ;Mở rộng dấu vào AH
MOV   BL,-7       ;BL chứa số chia
IDIV  BL     ;AL chứa thương số, AH chứa số dư.
```

Các lệnh CBW và CWD không làm ảnh hưởng tới cờ.

9.5 Các thủ tục vào ra với số thập phân

Mặc dù máy tính biểu diễn mọi thứ dưới dạng nhị phân, sẽ vẫn tiện lợi hơn cho người sử dụng khi làm các thao tác vào và ra với số thập phân. Trong phần này chúng ta sẽ viết các thủ tục quản lý việc vào/ra với số thập phân.

Trong thao tác vào, nếu chúng ta đánh 21543 thì thực chất chúng chỉ đưa vào một chuỗi ký tự. Chuỗi ký tự này phải được đổi ra (bằng chương trình) giá trị nhị phân tương đương của số nguyên thập phân 21453. Ngược lại trong thao tác ra, nội dung dạng nhị phân của một thanh ghi phải được đổi thành chuỗi ký tự biểu diễn số nguyên thập phân trước khi được in ra

Việc đưa ra các số thập phân

Chúng ta sẽ viết thủ tục OUTDEC để in ra nội dung của AX như là một số nguyên thập phân có dấu. Nếu AX >0, OUTDEC sẽ in ra dấu âm(-) rồi đổi AX thành - AX (để cho AX chứa số nguyên dương), và sau đó in ra nội dung của nó dạng thập phân. Trong trường hợp còn lại, vấn đề chỉ còn là in ra giá trị thập phân tương đương với một số dương dạng nhị phân. Dưới đây là thuật toán:

Thuật toán cho việc đưa ra số thập phân

```
1. IF    AX<0 /*AX chứa giá trị cần đưa ra*/
2. THEN
3. In ra dấu âm
4. Thay nội dung của AX bằng số bù 2 của nó.
5. END_IF
6. Lấy dạng biểu diễn thập phân các chữ số trong AX
7. Đổi các chữ số này thành các ký tự rồi đưa chúng
   ra màn hình.
```

Để thấy được dòng 6 trong thuật toán cần những gì, chúng ta giả sử nội dung của AX dưới dạng thập phân là 24168. Để nhận được từng chữ số trong dạng biểu diễn thập phân chúng ta có thể làm như sau:

Chia 24168 cho 10. Thương số là 2416, số dư là 8

Chia 2416 cho 10. Thương số là 241, số dư là 6

Chia 241 cho 10. Thương số là 24, số dư là 1

Chia 24 cho 10. Thương số là 2, số dư là 4

Chia 2 cho 10. Thương số là 0, số dư là 2.

Như vậy những chữ số mà chúng ta muốn chính là những số dư trong phép chia lặp cho 10. Nhưng chúng lại xuất hiện theo thứ tự ngược lại, để đổi chúng ngược lại chúng ta có thể chứa chúng trong ngăn xếp.

Dưới đây là dòng 6 được chia nhỏ ra thành các thao tác cơ bản:

Dòng 6

```
Đếm = 0 /* biến đếm số chữ số thập phân */
REPEAT
    Chia số bị chia cho 10
    Cất số dư vào ngăn xếp
    Đếm = Đếm +1
UNTIL   Thương số bằng 0
```

Trong đó giá trị ban đầu của thương số chính là nội dung của AX. Khi các chữ *số đã ở trong ngăn xếp, những gì chúng ta còn phải làm là lấy chúng ra*, đổi chúng thành các ký tự và in chúng ra màn hình. Dòng 7 có thể viết lại thành:

Dòng 7

```
FOR       Đếm lần  DO
    Lấy chữ số từ ngăn xếp
```

```
    Đổi nó thành ký tự
    Đưa ra màn hình ký tự
END_FOR
```

Bây giờ chúng ta có thể viết thủ tục như sau

Chương trình Nguồn PGM9_1.ASM

```
1:    OUTDEC   PROC
2:    ;in ra nội dung của AX dưới dạng số thập phân có dấu
3:    ;Vào: AX
4:    ;Ra : Không
5:             PUSH  AX          ;Cất các thanh ghi
6:             PUSH  BX
7:             PUSH  CX
8:             PUSH  DX
9:    ;Nếu AX <0
10:            OR    AX,AX       ;AX<0?
11:   JGE @END_IF1   ;NO,>0
12:   ;Thì
13:            PUSH  AX          ;Cất số cần đưa ra
14:            MOV   DL,'-'      ;Lấy ký tự '-'
15:            MOV   AH,2    ;Hàm con in ký tự ra màn hình
16:            INT   21H         ;In dấu '-'
17:            POP   AX          ;Lấy lại AX
18:            NEG   AX          ;AX=-AX
19:   @END_IF1:
20:   ;Lấy ra các chữ số thập phân
21:            XOR   CX,CX    ;CX đếm các chữ số thập phân
22:            MOV   BX,10D      ;BX chứa số chia
23:   @REPEAT1:
24:            XOR   DX,DX       ;Chuẩn bị word cao cho số
                                 ;bịchia
25:            DIV   BX          ;AX=thương số, DX=số dư
26:            PUSH  DX          ;Cất số dư vào ngăn xếp
27:            INC CX            ;Đếm=Đếm+1
28:            OR AX,AX          ;Thương số=0?
30:            JNE   @REPEAT1    ;Không!, tiếp tục
31:   ;Đổi các chữ số thành ký tự và đưa ra màn hình
32:            MOV   AH,2        ;Hàm con in ký tự
33:   ;FOR Đếm lần DO
34:   @PRINT_LOOP:
```

```
35:             POP   DX          ;Chữ số trong DL
36:             OR    DL,30H      ;Đổi nó thành ký tự
37:             INT   21H         ;In ra màn hình
38:             LOOP  @PRINT_LOOP;Lặp lại cho đến khi xong
39:     ;END_FOR
40:             POP   DX          ;Phục hồi các thanh ghi
41:             POP   CX
42:             POP   BX
43:             POP   AX
44:             RET
45:     OUTDEC  ENDP
```

Sau khi cất các thanh ghi chúng ta kiểm tra dấu của AX ở dòng 10 bằng cách hoặc AX với chính nó. Nếu AX >0, chương trình nhảy đến dòng 19, nếu AX <0, dấu âm được in ra và AX được thay bằng số bù 2 của nó. Cũng như trong trường hợp còn lại, tại dòng 19, AX chứa số dương.

Tại dòng 21, OUTDEC chuẩn bị cho phép chia. Vì việc chia cho hằng số là không hợp lệ chúng ta phải lưu số chia 10 vào một thanh ghi.

Vòng lặp REPEAT tại các dòng 23-30 sẽ nhận các chữ số và đưa nó vào ngăn xếp. Vì chúng ta làm việc với các số không dấu DX được xoá. Sau phép chia AX chứa thương số và DX chứa số dư (chính xác là số dư ở trong DL vì số dư nằm trong phạm vi từ 0 đến 9). Tại dòng 29, AX được kiểm tra xem đã bằng 0 chưa bằng cách dùng lệnh OR với chính nó. phép chia cho 10 được lặp lại cho đến khi thương số bằng 0

Vòng lặp FOR trong các dòng 34-38 lấy các chữ số trong ngăn xếp và in chúng. Trước khi in ra màn hình chúng phải được đổi ra các ký tự ASCII (dòng 36).

Toán tử giả INCLUDE

Chúng ta có thể kiểm tra lại OUTDEC bằng cách đặt nó bên trong một chương trình và chạy chương trình trong DEBUG. Để chèn OUTDEC vào chương trình mà không phải đánh lại chúng ta sử dụng toán tử giả INCLUDE. Nó có dạng sau:

```
INCLUDE  Tên file
```

Trong đó tên file xác định file (có thể có cả ổ đĩa và đường dẫn). Ví dụ file chứa OUTDEC là PGM9_1.ASM do đó chúng ta có thể viết:

```
INCLUDE  A:PGM9_1.ASM
```

Khi MASM gặp dòng này lúc biên dịch nó sẽ tìm và lấy file **PGM9_1.ASM** trong đĩa và chèn nó vào trong chương trình tại vị trí của dẫn hướng biên dịch INCLUDE.

Dưới đây là chương trình kiểm tra

Chương trình Nguồn PGM9_2.ASM

```
TITLE    PGM9_2:DECIMAL OUTPUT
.MODEL   SMALL
.STACK   100H
.CODE
MAIN     PROC
         CALL  OUTDEC
         MOV   AH,4CH
         INT   21H          ;Trở về DOS
MAIN     ENDP
INCLUDE  A:PGM9_1.ASM
         END   MAIN
```

Để kiểm tra chương trình chúng ta đánh vào DEBUG và chạy chương trình 2 lần, lần đầu cho AX = -25487 = 9C7h và lần thứ 2 cho AX = 654 = 2E8h:

```
C>DEBUG  PGM9_2.EXE
-RAX
AX 0000
:9C71
-G
-25487                ( kết quả đầu tiên)
Program terminated normally
-RAX
AX 9C71
:28E
-G
654             (kết quả thứ 2)
```

Chú ý rằng sau lần chạy đầu tiên DEBUG tự động đặt lại con trỏ lệnh IP vào đầu chương trình.

Nhập vào số thập phân

Để nhập vào các số thập phân chúng ta phải đổi chuỗi các chữ số ASCII thành dạng biểu diễn nhị phân của số thập phân. Chúng ta sẽ viết thủ tục INDEC để làm việc này.

Trong thủ tục OUTDEC để đưa ra nội dung của AX dưới dạng thập phân chúng ta đã lặp lại phép chia cho 10. Trong thủ tục INDEC chúng ta sẽ lặp lại phép nhân với 10. Dưới đây là những ý tưởng cơ bản:

Thuật toán nhập các số thập phân (version đầu tiên)

```
Tổng = 0
Đọc một chữ số ASCII
REPEAT
    Đổi ký tự ra giá trị nhị phân
    Tổng = Tổng*10+giá trị nhận được
    Đọc ký tự
UNTIL    Ký tự nhận được là CR
```

Ví dụ việc nhập vào số 123 được thực hiện như sau:

```
Tổng = 0
Đọc '1'
Đổi '1' thành 1
Tổng = 10*0+1 = 1
Đọc '2'
Đổi '2' thành 2
Tổng = 10*1+2 = 12
Đọc '3'
Đổi '3' thành 3
Tổng = 10*12+3 = 123
```

Chúng ta sẽ thiết kế INDEC sao cho nó có thể kiểm soát các số thập phân trong phạm vi từ -32768 đến 32767. Chương trình sẽ in ra dấu hỏi, và để người sử dụng đánh vào dấu tuỳ ý theo sau là chuỗi các chữ số và kết thúc là ký tự CR. Nếu người sử dụng đánh vào một ký tự nằm ngoài khoảng từ "0" đến "9", thủ tục sẽ điều khiển con trỏ tới dòng mới và bắt đầu lại từ đầu. Với những yêu cầu đó, thuật toán trên đây được đổi lại như sau:

Thuật toán nhập các số thập phân (version thứ 2)

```
In ra dấu hỏi
```

```
Tổng = 0
âm = false
Đọc ký tự
CASE     ký tự OF
    '-' : âm = true
          Đọc ký tự
    '+' : Đọc ký tự
END_CASE
REPEAT
    IF    Ký tự không nằm trong khoảng giữa '0' và '9'
       THEN
          Nhảy về đầu*
    ELSE
          Đổi các ký tự thành gia trị nhị phân
          Tổng = 10*Tổng+Giá trị
    END_IF
    Đọc ký tự
UNTIL          Ký tự = CR
IF  âm = TRUE
    THEN
          Tổng = -Tổng
END_IF
```

Chú ý : Việc nhảy như trên không mang tính chất của "lập trình cấu trúc". Tuy nhiên đôi khi cũng cần thiết phải phá vỡ các nguyên tắc về cấu trúc để đạt được tính hiệu quả, chẳng hạn khi có lỗi xuất hiện.

Từ thuật toán trên chúng ta có thể viết được chương trình như sau:

Chương trình nguồn PGM9_3.ASM

```
1:  INDEC          PROC
2:  ;Đọc vào một số trong phạm vi từ -32768 đến 32767
3:  ;Vào: Không
4:  ;Ra : AX = gía trị nhị phân tương đương của số
5:            PUSH  BX          ;Cất các thanh ghi
6:            PUSH  CX
7:            PUSH  DX
8:  ;In ra dấu nhắc
9:  @BEGIN:
10:           MOV   AH,2
11:           MOV   DL,'?'
12:           INT   21H         ;In ra dấu '?'
13: ;Tổng = 0
14:           XOR   BX,BX       ;BX chứa Tổng
```

```
15:;âm = false
16:          XOR   CX,CX       ;CX chứa dấu
17:;Đọc ký tự
18:          MOV   AH,1
19:          INT   21H         ;Ký tự trong AL
20:;CASE ký tự OF
21:          CMP   AL,'-'      ;Dấu âm?
22:          JE    @MINUS      ;Đúng! thiết lập dấu
23:          CMP   AL,'+'      ;Dấu dương?
24:          JE    @PLUS       ;Đúng! Đọc ký tự tiếp theo
25:          JMP   @REPEAT2    ;Bắt đầu xử lý các ký tự
26:@MINUS:
27:          MOV   CX,1        ;âm = TRUE
28:@PLUS:
29:          INT   21H         ;Đọc ký tự
30:;END_CASE
31:@REPEAT2:
32:;Nếu ký tự nằm trong phạm vi từ '0' đến '9'
33:          CMP   AL,'0'      ;Ký tự>='0'?
34:          JNGE  @NOT_DIGIT  ;Không! ký tự không hợp lệ
35:          CMP   AL,'9'      ;Ký tự<='9'?
36:          JNLE  @NOT_DIGIT  ;Không! ký tự không hợp lệ
37:;THEN Đổi ký tự thành chữ số
38:          AND   AX,000FH    ;Đổi thành chữ số
39:          PUSH  AX          ;Cất vào ngăn xếp
40:;Tổng = 10*Tổng+Chữ số
41:          MOV   AX,10
42:          MUL   BX          ;AX = Tổng*10
43:          POP   BX          ;Phục hồi chữ số
44:          ADD   BX,AX       ;Tổng = Tổng*10+Chữ số
45:;Đọc ký tự
46:          MOV   AH,1
47:          INT   21H
48:          CMP   AL,0DH      ;Ký tự CR?
49:          JNE   @REPEAT2    ;Không phải! tiếp tục
50:;UNTIL    CR
51:          MOV   AX,BX       ;Lưu số vào AX
52:;IF âm
53:          OR    CX,CX       ;Số âm?
54:          JE    @EXIT       ;Không! kết thúc
55:;THEN
56:          NEG   AX          ;Đúng
57:;END_IF
58:@EXIT:
```

```
59:          POP   DX          ;Phục hồi các thanh ghi
60:          POP   CX
61:          POP   BX
62:          RET               ;Và trở về
63:;Nếu không phải ký tự hợp lệ
64:@NOT_DIGIT:
65:          MOV   AH,2        ;Chuyển con trỏ đến dòng mới
66:          MOV   DL,0DH
67:          INT   21H
68:          MOV   DL,0AH
69:          INT   21H
70:          JMP   @begin      ;Làm lại từ đầu
71: INDEC    ENDP
```

Thủ tục bắt đầu bằng việc cất các thanh ghi và in ra dấu hỏi chấm. BX chứa tổng, ở dòng 14 được xoá đi để khởi tạo.

CX sử dụng để theo dõi dấu; 0 có nghĩa là số dương và 1 có nghĩa là số âm. Ban đầu chúng ta giả sử số là dương do đó CX được xoá ở dòng 16.

Ký tự đầu tiên được đọc vào ở dòng 18,19. Nó có khả năng là "+" hay "-" hay là một chữ số. Nếu nó là dấu, CX được thay đổi nếu cần thiết và ký tự khác được đọc (dòng 29) giả sử ký tự tiếp theo này là một chữ số.

Tại dòng 31 INDEC bước vào vòng lặp REPEAT, vòng lặp này xử lý ký tự hiện thời và đọc vào ký tự tiếp theo cho đến khi người sử dụng đánh ENTER.

Tại dòng 33-36 INDEC kiểm tra xem ký tự hiện thời có phải là một chữ số không, nếu không phải thủ tục nhảy đến nhãn @NOT_DIGIT (dòng 64), chuyển con trỏ đến đầu dòng mới và nhảy đến nhãn @BEGIN. Điều này có nghĩa là người sử dụng không thể thoát khỏi chương trình mà không đánh vào một số hợp lệ.

Nếu ký tự hiện thời trong AL là một chữ số thập phân, nó được đổi sang trị nhị phân (dòng 38). Tiếp theo giá trị đó được cất vào ngăn xếp (dòng 39), vì AX được dùng đến trong phép nhân tổng với 10.

Tại dòng 41 và 42, tổng trong BX được nhân với 10. Tích số sẽ được chứa trong DX:AX . Tuy nhiên DX sẽ chứa 0 trừ khi số đánh vào nằm ngoài phạm vi (chúng ta sẽ nói thêm về chúng sau này). Tại dòng 43, giá trị đã cất vào ngăn xếp sẽ được đưa ra và cộng với 10 lần tổng.

Tại dòng 51, INDEC thoát khỏi vòng lặp REPEAT với số nhận được trong BX. Sau khi chuyển nó vào AX, INDEC kiểm tra dấu trong CX. Nếu CX chứa 1, AX được đảo dấu sau khi thủ tục kết thúc.

Kiểm tra INDEC

Chúng ta có thể kiểm tra INDEC bằng cách viết một chương trình sử dụng INDEC cho việc nhập và OUTDEC cho việc xuất.

Chương trình Nguồn PGM9_4.ASM

```
TITLE     PGM9_4:DECIMAL I/O
.MODEL    SMALL
.STACK
.CODE
MAIN      PROC
;Nhập một số
          CALL  INDEC       ;Số trong AX
          PUSH  AX          ;Cất số vào ngăn xếp
;Chuyển con trỏ sang dòng mới
          MOV   AH,2
          MOV   DL,0DH
          INT   21H
          MOV   DL,0AH
          INT   21H
;Đưa ra số đã nhập
          POP   AX          ;Phục hồi số đã nhập
          CALL  OUTDEC
;Trở về DOS
          MOV   AX,4CH
          INT   21H
MAIN      ENDP

INCLUDE   A:PGM9_1.ASM      ;Bao gồm cả thủ tục INDEC
INCLUDE   A:PGM9_3.ASM      ;Bao gồm cả thủ tục OUTDEC

END       MAIN
```

Ví dụ thực hiện

```
C>PGM9_4
?21345
21345Overflow
```

Hiện tượng tràn

Thủ tục INDEC có thể quản lý việc vào có những ký tự không hợp lệ, nhưng nó không thể quản lý việc đưa ra giá trị nằm ngoài khoảng -32768 đến 32767. Chúng ta gọi chúng.là hiện tượng tràn khi nhập.

Hiện tượng tràn có thể xảy ra ở 2 chỗ trong thủ tục INDEC:

* Khi Tổng được nhân với 10

* Khi Giá trị mới được cộng vào tổng.

Để lấy ví dụ cho trường hợp tràn đầu tiên người sử dụng có thể đánh vào 99999. Hiện tượng tràn sẽ xuất hiện khi Tổng = 9999 được nhân với 10. Để lấy ví dụ cho trường hợp tràn thứ 2 người sử dụng có thể đánh vào 32769, khi đó lúc Tổng = 32760, hiện tượng tràn xuất hiện khi 9 được cộng vào Tổng. Thuật toán có thể thay đổi để kiểm tra tràn như sau:

Thuật toán nhập số thập phân (version 3)

```
In ra dấu hỏi chấm
Tổng = 0
Âm = false
Đọc 1 ký tự
CASE ký tự OF
    '-' : Âm = TRUE
    '+' : Đọc ký tự
END_CASE
REPEAT
   IF ký tự không nằm trong khoảng giữa '0' và '9'
        THEN
             Quay lại từ đầu
        ELSE
             Đổi ký tự thành giá trị số
             Tổng = 10*Tổng
             IF overflow
                THEN
                       Quay lại từ đầu
                ELSE
                       Tổng = Tổng + Giá trị
                IF overflow
                      THEN
                       Quay lại từ đầu
```

```
                    END_IF

                END_IF
        END_IF
        Đọc ký tự
    UNTIL Ký tự = CR
    IF  Âm = TRUE
        THEN
            Tổng  = -Tổng
    END_IF
```

Việc lập chương trình từ thuật toán này chúng tôi giành cho các bạn như là một bài tập.

TỔNG KẾT

Trong chương này chúng ta đã nghiên cứu những vấn đề sau:

- Các lệnh nhân là MUL cho phép nhân các số không dấu và IMUL cho các số có dấu.
- Trong phép nhân giữa các byte, AL chứa một số hạng, số hạng còn lại có thể là một thanh ghi 8 bit hay một byte bộ nhớ. Trong phép nhân các word, AX chứa một số hạng, số hạng còn lại có thể là một thanh ghi 16 bit hay một từ bộ nhớ.
- Trong phép nhân giữa các byte tích số 16 bit được chứa trong AX, còn trong phép nhân các word tích số 32 bit chứa trong DX:AX.
- Các lệnh chia là DIV cho các số không dấu và IDIV cho các số có dấu.
- Số chia có thể là một thanh ghi, một byte hay một word. Trong phép chia cho một byte AX chứa số bị chia còn trong phép chia cho một word số bị chia được đặt trong DX:AX
- Sau phép chia cho byte AH chứa số dư còn AL chứa thương số. Sau phép chia cho word AX chứa thương số còn DX chứa số dư.
- Đối với phép chia có dấu cho các word, nếu AX chứa số bị chia thì lệnh CWD được dùng để mở rộng dấu của AX vào DX. Tương tự như vậy trong phép chia có dấu cho các byte lệnh CBW sẽ mở rộng dấu của AL vào AH. Đối với phép chia không dấu cho các word, nếu AX chứa số bị chia thì DX phải được xoá cũng tương tự như vậy trong phép chia không dấu cho các byte, nếu AL chứa số bị chia thì AH phải được xoá.
- Các lệnh nhân và chia được sử dụng rất hữu hiệu trong việc thực hiện các thao tác vào/ra với số thập phân.
- Toán tử giả INCLUDE có thể sử dụng để chèn một đoạn văn bản (chương trình) từ một file bên ngoài vào trong chương trình.

Các lệnh mới học

CBW	DIV	IMUL
CWD	IDIV	MUL

Toán tử giả mới

INCLUDE

Bài tập

1. Hãy cho biết nội dung của DX, AX và các cờ CF/OF khi các lệnh sau là hợp lệ và được thực hiện.

   a. MUL BX trong đó AX chứa 0008h và BX chứa 0003h
   b. MUL BX trong đó AX chứa 00FFh và BX chứa 1000h
   c. IMUL CX trong đó AX chứa 0005h và CX chứa FFFFh
   d. IMUL WORD1 trong đó AX chứa 0005h và WORD1 chứa FFFFh.
   e. IMUL 10h trong đó AX chứa FFE0h

2. Cho biết nội dung của AX và các cờ CF/OF sau mỗi lệnh sau:

   a. MUL BX trong đó AL chứa ABh và BL chứa 10h
   b. IMUL BX trong đó AL chứa ABh và BL chứa 10h
   c. MUL AH trong đó AX chứa 01ABh
   d. IMUL BYTE1 trong đó AL chứa 02h và BYTE1 chứa FBh

3. Cho biết nội dung mới của AX và DX sau mỗi lệnh sau, hiện tượng tràn có thể xảy ra hay không ?

   a. DIV BX trong đó DX chứa 0000h, AX chứa 0007h và BX chứa 0002h
   b. DIV BX trong đó DX chứa 0000h, AX chứa FFFEh và BX chứa 0010h
   c. IDIV BX trong đó DX chứa FFFFh, AX chứa FFFCh và BX chứa 0003h
   d. DIV BX với các giá trị giống như câu c

4. Cho biết nội dung mới của AL, AH sau mỗi lệnh sau đây, hiện tượng tràn có thể xảy ra không ?

   a. DIV BL trong đó AX chứa 000Dh và BL chứa 03h
   b. IDIV BL trong đó AX chứa FFFBh và BL chứa FEh
   c. DIV BL trong đó AX chứa 00FEh và BL chứa 10h
   d. DIV BL trong đó AX chứa FFE0h và BL chứa 02h

5. Cho biết nội dung của DX sau khi thực hiện lệnh CWD nếu AX chứa các gía trị:

   a. 7E02h
   b. 8ABCh
   c. 1ABCh

6. Cho biết nội dung cuả AX sau khi thực hiện lệnh CBW nếu AL chứa:

   a. F0h
   b. 5Fh
   c. 80h

7. Viết các lệnh hợp ngữ thực hiện các lệnh gán sau đây trong ngôn ngữ bậc cao. Giả thiết rằng A, B, C là các biến kiểu word và tất cả các tích số đều chứa vừa trong 16

bit. Sử dụng lệnh IMUL trong các phép nhân. Không cần giữ lại nội dung của các biến A, B, C.

```
a.  A = 5*A-7
b.  B = (A-B)*(B+10)
c.  A = 6-9*A
D.  IF A²+B² = C²
    THEN
         Thiết lập CF
    ELSE
         Xoá CF
    END_IF
```

Bài tập lập trình

Chú ý: Một số bài tập dưới đây yêu cầu bạn sử dụng IDEC hay OUTDEC hoặc cả 2 để thực hiện các thao tác vào ra. Hãy chú ý không được dùng những nhãn đã dùng trong các thủ tục này nếu không bạn sẽ nhận được thông báo lỗi dùng nhãn 2 lần. Điều này có thể thực hiện dễ dàng vì các nhãn của những thủ tục này đều bắt đầu bằng chữ "@".

8. Sửa lại thủ tục INDEC để nó có khả năng kiểm tra hiện tượng tràn.

9. Viết một chương trình cho phép người sử dụng đánh vào thời gian ở dạng giây nhỏ hơn hay bằng 65535, sau đó đưa ra thời gian dạng giờ, phút, giây. Sử dụng INDEC và UTDEC cho các thao tác vào/ra.

11. Viết một chương trình cho phép người sử dụng đánh vào một phân số dạng M/N(M<N). Sau đó chương trình in ra thương số đến N chữ số thập phân theo thuật toán sau:

    1. In ra dấu"."
    Thực hiện các bước sau N lần:
    2. Chia 10*M cho N, nhận được thương số Q và số dư R.
    3. In ra Q.
    4. Thay M bằng R và quay lại bước 2.

    Sử dụng INDEC để đọc vào M và N

12. Viết một chương trình tìm ước số chung lớn nhất (USCLN) của 2 số nguyên M và N theo thuật toá

Chương 10

MẢNG VÀ CÁC CHẾ ĐỘ ĐỊA CHỈ.

Tổng quan.

Đôi khi trong các ứng dụng chúng ta cần phải làm việc với tập hợp các ký tự như là một nhóm. Ví dụ chúng ta cần đọc vào các điểm kiểm tra và in ra trị trung bình. Để làm việc này trước hết ta phải lưu trữ các điểm theo thứ tự tăng dần (đồng thời với việc nhập các điểm hoặc là chúng đã có sẵn trong bộ nhớ). Ưu điểm của việc sử dụng mảng để chứa dữ liệu là có thể đưa ra tên cho cả cấu trúc và mỗi phần tử có thể được truy nhập bằng cách cung cấp một chỉ số.

Mục 10.1 chúng tôi sẽ trình bày cách khai báo mảng một chiều trong Hợp ngữ . Để truy nhập các phần tử, mục 10.2 chúng tôi sẽ giới thiệu các phương pháp mới định nghĩa các toán hạng - các chế độ địa chỉ cơ sở, chỉ số và chế độ địa chỉ gián tiếp thông qua các thanh ghi. Đến mục 10.3 chúng tôi sẽ sử dùng các chế độ địa chỉ này để sắp xếp một mảng.

Mảng hai chiều là mảng một chiều mà mỗi phần tử lại là một mảng một chiều (mảng của các mảng). Trong mục 10.4 chúng tôi sẽ trình bày cách thức lưu trữ chúng. Các mảng này có hai chỉ số, chúng được tính toán thuận lợi hơn cả trong chế độ địa chỉ chỉ số cơ sở mà chúng tôi sẽ trình bày trong mục 10.5. Mục 10.6 sẽ cung cấp một ứng dụng đơn giản.

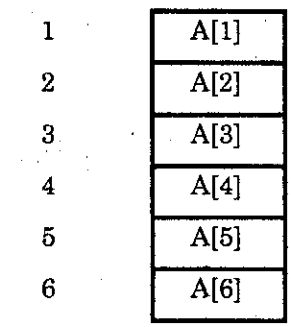
Mục 10.7 giới thiệu về lệnh XLAT (translate). Lệnh này có ích khi muốn đổi dữ liệu. Chúng ta sẽ sử dụng nó để mã hoá và giải mã một mật khẩu.

10.1. Mảng một chiều.

Mảng một chiều là một dãy có thứ tự các phần tử có cùng một kiểu.

"Thứ tự" ở đây có nghĩa là có phần tử thứ nhất, phần tử thứ hai, thứ ba v.v... Trong toán học các phần tử của mảng A thường được ký hiệu là A[1], A[2], A[3]... . Hình 10.1 biểu thị mảng A với 6 phần tử.

Hình 10.1. Mảng một chiều A



Trong chương 4 chúng ta đã sử dụng các toán tử giả DB và DW để khai báo các mảng byte và word. Ví dụ chuỗi 5 ký tự MSG:

```
MSG   DB    'abcde'
```

hay một mảng word W gồm 6 số nguyên được khởi tạo với các giá trị 10, 20, 30, 40, 50, 60:

```
W     DW    10, 20, 30, 40, 50, 60
```

Địa chỉ của biến mảng gọi là địa chỉ cơ sở của mảng (base address of the array). Nếu như địa chỉ offset gán cho W bằng 0200h thì nó sẽ được phân bố trong bộ nhớ như sau:

Địa chỉ offset	*Ký hiệu địa chỉ*	*Giá trị thập phân*
0200h	W	10
0202h	W+2h	20
0204h	W+4h	30
0206h	W+6h	40
0208h	W+8h	50
020Ah	W+Ah	60

Toán tử DUP.

Chúng ta có thể định nghĩa một mảng mà các phần tử có cùng một giá trị khởi đầu nhờ toán tử DUP (duplicate). Nó có dạng:

```
repeat_count        DUP ( value)
```

Với toán tử này, value sẽ được lặp lại một số lần xác định bởi repeat_count. Ví dụ:

```
GAMMA    DW    100   DUP   ( 0 )
```

thiết lập một mảng 100 từ với giá trị khởi đầu cho mỗi phần là 0. Tương tự:

```
DELTA    DB    212   DUP   ( ? )
```

khai báo một mảng 212 phần tử không khởi tạo. Các toán tử DUP có thể lồng nhau. Ví dụ:

```
LINE     DB    5, 4, 2  DUP ( 2, 3  DUP ( 0 ), 1 )
```

tương đương với:

```
LINE     DB    5, 4, 2, 0, 0, 0, 1, 2, 0, 0, 0, 1
```

Xác định vị trí các phần tử trong mảng.

Địa chỉ một phần tử của mảng có thể được xác định bằng cách cộng một hằng số vào địa chỉ cơ sở. Giả sử mảng A có S là số byte của một phần tử (đối với mảng byte *S = 1 còn mảng word S = 2*). *Vị trí các phần tử của mảng A được xác định như sau:*

Số thứ tự	Vị trí
1	A
2	A + 1 * S
3	A + 2 * S
.	.
.	.
.	.
N	A + (N- 1) * S

Vídụ 10.1. Hoán chuyển các phần tử thứ 10 và thứ 25 của mảng word W.

Lời giải:

W[10] ở tại địa chỉ W + 9 * 2 = W + 18 và W[25] tại W + 24 * 2 = W + 48, vì vậy ta có thể đổi chỗ như sau:

```
MOV     AX,W + 18          ; AX chứa W[10]
XCHG    W + 48,AX          ; AX chứa W[25]
MOV     W + 18,AX  ; hoàn thành việc hoán chuyển.
```

Trong rất nhiều ứng dụng chúng ta cần phải thực hiện các thao tác với từng phần tử của một mảng. Ví dụ mảng A có 10 phần tử và chúng cần tính tổng các phần tử của nó. Trong một ngôn ngữ bậc cao chúng ta có thể thực hiện như sau:

```
sum = 0
N = 1
REPEAT
   sum = sum + A[N]
   N = N + 1
ULTIL  N > 10
```

Trong Hợp ngữ , để làm việc này chúng ta cần có một phương pháp di chuyển từ phần tử này đến phần tử tiếp theo của mảng. Trong mục tiếp theo chúng tôi sẽ chỉ ra điều đó được thực hiện như thế nào nhờ việc đánh địa chỉ gián tiếp.

10.2. Các chế độ địa chỉ.

Phương thức xác định toán hạng gọi là chế độ địa chỉ (addressing mode) của nó. Các chế độ địa chỉ chúng ta đã sử dụng đó là:

(1) Chế độ địa chỉ thanh ghi (register mode): toán hạng là thanh ghi.

(2) Chế độ địa chỉ tức thời (immediate mode): toán hạng là hằng số.

(3) Chế độ địa chỉ trực tiếp (direct mode): toán hạng là biến nhớ.

Ví dụ:

```
MOV  AX,0        ;toán hạng đích AX ở chế độ
                 ;thanh ghi,toán hạnh nguồn 0
                 ;ở chế độ tức thời.
ADD  ALPHA,AX    ;toán hạng đích ALPHA ở chế độ
                 ;trực tiếp,toán hạnh nguồn AX
                 ;ở chế độ thanh ghi.
```

Bộ vi xử lý 8086 có tổng cộng 4 chế độ địa chỉ, đó là các chế độ địa chỉ thanh ghi, cơ sở, chỉ số và chỉ số cơ sở. Các chế độ này được sử dụng để đánh địa chỉ ô nhớ các toán hạng một cách gián tiếp. Trong phần này chúng tôi sẽ trình bày ba chế độ đầu, chúng hữu ích khi thao tác với các mảng một chiều. Chế độ địa chỉ chỉ số cơ sở có thể sử dụng cho mảng hai chiều, nó sẽ được trình bày trong mục 10.5.

10.2.1. Chế độ địa chỉ gián tiếp thanh ghi.

Trong chế độ này, địa chỉ offset của toán hạng được chứa trong một thanh ghi. Như vậy các thanh ghi đóng vai trò như một con trỏ trỏ đến các ô nhớ. Khuôn dạng của toán hạng ở chế độ này là:

```
[ register ]
```

Register có thể là BX, DI, SI hay BP. Với các thanh ghi BX, DI và SI, số hiệu đoạn của toán hạng được chứa trong DS. Với thanh ghi BP, SS chứa số hiệu đoạn.

Ví dụ SI chứa địa chỉ offset 0100h và word tại địa chỉ 0100h có giá trị 1234h. Khi thi hành:

```
MOV   AX,[SI]
```

CPU sẽ kiểm tra SI để suy ra địa chỉ từ nhớ là DS : 0100h, sau đó nó chuyển nội dung của từ này, tức là 1234h vào AX. Điều đó khác hẳn với:

```
MOV   AX,SI
```

chỉ đơn thuần chuyển giá trị của SI hay 100h vào AX.

Ví dụ 10.2. Giả sử:

```
BX chứa 1000h          Offset 1000h chứa 1BACh
SI chứa 2000h          Offset 2000h chứa 20FEh
DI chứa 3000h          Offset 3000h chứa 031Dh
```

Ở đây các địa chỉ offset trên nằm trong đoạn dữ liệu đánh địa chỉ bởi DS.

Hãy cho biết lệnh nào sau đây là hợp lệ. Nếu hợp lệ hãy đưa ra địa chỉ offset của toán hạng nguồn và kết quả hoặc số được dịch chuyển.

```
a. MOV       BX,[BX]
```

```
b. MOV      CX,[SI]
c. MOV      BX,[AX]
d. ADD      [SI],[DI]
e. INC      [DI]
```

Lời giải:

	offset toán hạng nguồn	kết quả
a.	1000h	1BACh
b.	2000h	20FEh
c.	thanh ghi nguồn không hợp lệ	phải là BX, SI hay DI
d.	phép cộng không hợp lệ ô nhớ - ô nhớ	
e.	3000h	031Eh

Bây giờ chúng ta hãy trở lại vấn đề tính tổng các phần tử của một mảng.

Ví dụ 10.3. Viết các lệnh đưa vào AX tổng của các phần tử của một mảng 10 phần tử W được định nghĩa như sau:

```
W     DW    10,20,30,40,50,60,70,80,90,100
```

Lời giải:

Ta thiết lập một con trỏ trỏ đến địa chỉ cơ sở của mảng, cho nó dịch chuyển đến hết lượt các phần tử của mảng đồng thời cộng vào tổng các phần tử được tham trỏ tới.

```
        XOR   AX,AX       ;AX chứa tổng
        LEA   SI,W        ;SI trỏ đến mảng W
        MOV   CX,10       ;CX chứa số các phần tử
ADDNOS:
        ADD   AX,[SI]     ;tổng=tổng+phần tử
        ADD   SI,2        ;chuyển con trỏ đến phần tử
                          ;tiếp theo
LOOP          ADDNOS      ;lặp đến khi thực hiện xong
```

Ta phải cộng 2 vào SI sau mỗi bước lặp bởi lẽ W là một mảng word (xem lại chương 4, lệnh LEA chuyển địa chỉ offset của toán hạng nguồn vào toán hạng đích).

Ví dụ tiếp theo sẽ chỉ ra chế độ địa chỉ gián tiếp thanh ghi được sử dụng như thế nào khi xử lý các mảng.

Ví dụ 10.4. Viết thủ tục REVERSE để đảo ngược một mảng N từ. Điều này có nghĩa là từ thứ N trở thành từ đầu tiên, từ thứ N-1 trở thành từ thứ hai ... và từ đầu tiên trở thành từ thứ N. Trước khi vào thủ tục, SI trỏ đến mảng, BX chứa số phần tử N.

Lời giải:

Mục đích của ta là đổi chỗ các từ thứ nhất và thứ N, từ thứ hai và thứ (N-1) v.v.. . Số lần hoán chuyển sẽ bằng *N/2 (làm tròn xuống nếu như N lẻ)*. Trong mục 10 ta đã biết rằng phần tử thứ N của mảng A có địa chỉ A + 2 * (N - 1).

Chương trình nguồn PGR10_1.ASM

```
REVERSE  PROC
;đảo ngược một mảng
;vào:    SI=địa chỉ offset của mảng
;        BX=số phần tử
;ra:     đảo ngược mảng
         PUSH  AX
         PUSH  BX
         PUSH  CX
         PUSH  SI
         PUSH  DI
;cho DI trỏ đến phần tử thứ N
         MOV   DI,SI ;DI trỏ đến phần tử đầu tiên
         MOV   CX,BX       ;CX=N
         DEC   BX          ;BX=N-1
         SHL   BX,1        ;BX=2*(N-1)
         ADD   DI,BX       ;DI trỏ đến phần tử thứ N
         SHR   CX,1        ;CX=N/2=số lần hoán chuyển
                           ;hoán chuyển
XCHG_LOOP:
         MOV   AX,[SI]     ;lấy một phần tử trong nửa
                           ;đầu của mảng
         XCHG  AX,[DI]     ;chèn nó vào nửa sau
         MOV   [SI],AX     ;hoàn thành hoán chuyển
         ADD   SI,2        ;di chuyển con trỏ
         SUB   DI,2        ;di chuyển con trỏ
         LOOP  XCHG_LOOP   ;lặp đến khi hoán chuyển hết
         POP   DI
```

```
        POP   SI
        POP   CX
        POP   BX
        POP   AX
        RET
REVERSE ENDP
```

10.2.2. Chế độ địa chỉ cơ sở và địa chỉ chỉ số.

Trong các chế độ địa chỉ này, địa chỉ offset của các toán hạng nhận được bằng cách cộng một số được gọi là độ dịch (displacement) với nội dung của một thanh ghi, trong đó độ dịch có thể là:

- địa chỉ offset của một biến.
- một hằng số (âm hoặc dương).
- địa chỉ offset của một biến cộng hoặc trừ với một hằng số .

Ví dụ các độ dịch có thể là (A là một biến):

```
A
-2
A+4
```

Toán hạng phải được viết tương đương với một trong các biểu thức sau đây:

```
[ thanh ghi  + độ dịch ]
[ độ dịch + thanh ghi  ]
[ thanh ghi  ] + độ dịch
độ dịch + [ thanh ghi ]
độ dịch [ thanh ghi  ]
```

Các thanh ghi phải là BX, BP, SI hay DI. Nếu ta dùng BX, SI hay DI, DS sẽ chứa số hiệu đoạn của địa chỉ toán hạng. Nếu BP được sử dụng, SS sẽ chứa số hiệu đoạn. Chế độ địa chỉ được gọi là chế độ địa chỉ cơ sở (based) nếu ta dùng BX (base register) hay BP (base pointer) và gọi là chế độ địa chỉ chỉ số nếu SI (source indexed) hay DI (destination indexed) được sử dụng.

Ví dụ. Giả sử W là một mảng word, BX chứa số 4. Trong lệnh sau đây:

```
MOV   AX,W[BX]
```

độ dịch là địa chỉ offset của biến W. Lệnh MOV sẽ chuyển phần tử có địa chỉ W+4 vào AX. Đây chính là phần tử thứ 3 của mảng. Lệnh trên còn có thể được viết đưới các dạng sau :

```
MOV   AX,[W+BX]
MOV   AX,[BX+W]
MOV   AX,W+[BX]
MOV   AX,[BX]+W
```

Ta xét tiếp một ví dụ khác. Giả sử SI chứa địa chỉ của mảng word W. Trong lệnh:

```
MOV   AX,[SI+2]
```

độ dịch bằng 2. Lệnh trên sẽ chuyển phần tử có địa chỉ W+2 vào AX. Đây chính là phần tử thứ hai của mảng. Ta có thể viết lại lệnh dưới các dạng sau đây:

```
MOV   AX,[2+SI]
MOV   AX,2+[SI]
MOV   AX,[SI]+2
MOV   AX,2[SI]
```

Ví dụ 10.5. Làm lại ví dụ 10.3 bằng cách sử dụng chế độ địa chỉ cơ sở.

Lời giải.

Phương hướng ở đây là xoá thanh ghi BX, cộng thêm vào nó 2 sau mỗi lần lặp:

```
        XOR   AX,AX       ;AXchứa tổng
        XOR   BX,BX       ;xoá thanh ghi cơ sở
        MOV   CX,10       ;CX chứa số phần tử
ADDNOS:
        ADD   AX,W[BX]    ;tổng=tổng+phần tử
        ADD   BX,2  ;chỉ số của phần tử tiếp theo
        LOOP  ADDNOS      ;lặp đến khi làm xong
```

Ví dụ 10.6. Giả sử ALPHA được khai báo như sau:

```
ALPHA DW 0123h, 0456h, 0789h, 0ABCh.
```

trong đoạn được đánh địa chỉ bởi DS. Biết rằng:

BX chứa 2	Offset 0002 chứa 1084h
SI chứa 4	Offset 0004 chứa 2BACh
DI chứa 1	

Hỏi rằng lệnh nào trong các lệnh sau đây là hợp lệ. Với các lệnh hợp lệ hãy cho biết địa chỉ offset của toán tử nguồn và con số được dịch chuyển.

```
a.    MOV   AX,[ALPHA + BX]
b.    MOV   BX,[BX + 2]
c.    MOV   CX,ALPHA [SI]
d.    MOV   AX,-2[SI]
e.    MOV   BX,[ALPHA + 3 + DI]
f.    MOV   AX,[BX]2
g.    ADD   BX,[ALPHA + AX]
```

Lời giải:

Offset toán hạng nguồn	Con số được dịch chuyển
a. ALPHA + 2	0456h
b. 2 + 2 = 4	2BACh
c. ALPHA + 4	0789h
d. -2 + 4 = 2	1084h
e. ALPHA +3+1=ALPHA +4	0789h
f. dạng toán hạng nguồn không hợp lệ	
g. thanh ghi nguồn không hợp lệ	

Ví dụ tiếp theo sẽ minh hoạ cho việc xử lý mảng bằng chế độ địa chỉ chỉ số và địa chỉ cơ sở.

Ví dụ 10.7. Thay thế các chữ thường trong chuỗi sau đây bằng các chữ hoa tương ứng. Bạn hãy dùng chế độ địa chỉ chỉ số.

```
MSG    DB   ' this is a messsage '
```

Lời giải:

```
        MOV   CX,17              ;số ký tự trong chuỗi
        XOR   SI,SI              ;SI chỉ đến một ký tự
TOP:
        CMP   MSG[SI],' '        ;ký tự trắng?
        JE    NEXT               ;đúng! nhảy qua
        AND   MSG[SI],0DFh ;không!đổi thành chữ hoa
NEXT:
        INC   SI          ;chỉ đến byte tiếp theo
```

```
LOOP  TOP          ;lặp đến khi hoàn thành
```

10.2.3. Các toán tử PTR và LABEL.

Hẳn các bạn còn nhớ trong chương 4 đã chỉ ra rằng các toán hạng trong một lệnh phải có cùng một kiểu, ví dụ cùng là byte hoặc word. Nếu một toán hạng là hằng số, chương trình biên dịch sẽ căn cứ vào toán hạng kia để suy ra dạng của nó. Ví dụ chương trình biên dịch xem lệnh:

```
MOV   AX,1
```

như một lệnh thao tác trên các word bởi vì AX là thanh ghi 16 bit. Tương tự nó coi:

```
MOV   BH,5
```

là lệnh thao tác với các byte. Tuy nhiên nó không thể biên dịch:

```
MOV   [BX],1       ;không hợp lệ
```

bởi lẽ nó không thể biết được BX chỉ đến byte hay word. Nếu như bạn muốn toán hạng đích là một byte, bạn có thể viết:

```
MOV   BYTE  PTR [BX],1
```

còn nếu bạn lại muốn toán hạng đích là một word, bạn hãy viết như sau:

```
MOV   WORD  PTR [BX],1
```

Ví dụ 10.8. Thay ký tự 't' của chuỗi đã cho trong ví dụ 7 bằng ký tự 'T'.

Lời giải 1:

Sử dụng chế độ địa chỉ gián tiếp thanh ghi.

```
LEA   SI,MSG              ;SI chỉ đến MSG
MOV   BYTE  PTR [SI],'T';thay 't' bằng 'T'
```

Lời giải 2:

Sử dụng chế độ địa chỉ chỉ số.

```
XOR   SI,SI               ;xoá SI
MOV   MSG[SI],'T'         ;thay 't' bằng 'T'
```

Lời giải này không cần thiết phải dùng toán tử PTR bởi lẽ MSG đã được định nghĩa là một biến byte.

Dùng PTR để định lại kiểu.

Nói chung PTR có thể dùng để định lại kiểu đã khai báo của một biểu thức địa chỉ. Cú pháp như sau:

```
type PTR  address_expression
```

Trong đó type có thể là BYTE, WORD hay DWORD (doubleword), còn biểu thức địa chỉ có kiểu DB, DW hay DD.

Ví dụ bạn có khai báo sau đây:

```
DOLLARS     DB    1Ah
CENTS       DB    52h
```

và bạn muốn chuyển giá trị của DOLLARS vào AL, giá trị của CENTS vào AH chỉ bằng một lệnh duy nhất. Lúc này:

```
MOV   AX, DOLLARS        ;không hợp lệ
```

là không hợp lệ bởi vì toán hạng đích là một word trong khi toán hạng nguồn được định kiểu như là một byte.Trong trường hợp này bạn có thể định lại kiểu khai báo nhờ WORD PTR như sau:

```
MOV   AX,WORD PTR DOLLARS   ;AL=dollars, AH=cents
```

Lệnh này sẽ chuyển 521Ah vào AX.

Toán tử giả LABEL.

Thực chất đây là một phương pháp khác để giải quyết mẫu thuẫn về kiểu trong ví dụ trước. Sử dụng toán tử giả LABEL, chúng ta có thể khai báo:

```
MONEY       LABEL WORD
DOLLARS     DB    1Ah
CENTS       DB    52h
```

Khai báo này ấn định MONEY là một biến word, DOLLARS và CENTS là các biến byte, và MONEY và DOLLARS được gán cho cùng một địa chỉ bởi trình biên dịch. Như vậy:

```
MOV   AX,MONEY    ;AL=dollars, AH=cents
```

là hợp lệ. Lệnh này cho kết quả giống hệt như:

```
MOV   AL,DOLLARS
MOV   AH,CENTS
```

Ví dụ 10.9. Cho các dữ liệu đã được khai báo như sau:

```
.DATA
A       DW          1234H
B       LABEL       BYTE
        DW          5678h
C       LABEL       WORD
C1      DB          9Ah
C2      DB          0BCh
```

Hỏi rằng lệnh nào trong các lệnh sau đây là hợp lệ. Với các lệnh hợp lệ hãy cho biết địa chỉ offset của toán tử nguồn và con số được dịch chuyển.

Các lệnh:

```
a.  MOV   AX,B
b.  MOV   AH,B
c.  MOV   CX,C
d.  MOV   BX,WORD   PTR   B
e.  MOV   DL,WORD   PTR   C
f.  MOV   AX,WORD   PTR   C1
```

Lời giải:

a. không hợp lệ - sai kiểu.
b. hợp lệ, 78h.
c. hợp lệ,0BC9Ah.
d. hợp lệ, 5678h.
e. hợp lệ, 9Ah.
f. hợp lệ, 0BC9Ah

10.2.4. Phương pháp ghi rõ đoạn.

Trong chế độ địa chỉ gián tiếp thanh ghi, các con trỏ thanh ghi BX, SI và DI xác định địa chỉ offset ứng với địa chỉ đoạn trong DS. Nhưng chúng ta cũng có thể xác định một địa chỉ offset tương ứng với một thanh ghi đoạn khác một cách

rõ ràng, tường minh. Khuôn mẫu của một toán hạng để thực hiện điều này như sau:

```
segment_register:[ pointer_register]
```

Ví dụ:

```
MOV   AX,ES:[SI]
```

Nếu SI chứa 0100h, địa chỉ của toán hạng nguồn trong trường hợp này sẽ là ES : 0100h. Có thể bạn muốn thực hiện điều này trong một chương trình có hai đoạn dữ liệu và ES chứa số hiệu đoạn của đoạn dữ liệu thứ hai.

Phương pháp ghi rõ đoạn cũng có thể dùng được với chế độ địa chỉ chỉ số và địa chỉ cơ sở.

10.2.5. Truy nhập ngăn xếp.

Như chúng tôi đã lưu ý trước đây, khi BP xác định một địa chỉ offset trong chế độ địa chỉ gián tiếp thanh ghi thì SS sẽ chứa số hiệu đoạn. Điều này có nghĩa là BP có thể được dùng để truy nhập các phần tử của ngăn xếp.

Ví dụ 10.10. Chuyển 3 từ ở đỉnh ngăn xếp vào AX, BX và CX mà không làm thay đổi ngăn xếp.

Lời giải:

```
MOV   BP,SP       ;SP trỏ đến đỉnh ngăn xếp
MOV   AX,[BP]     ;chuyển đỉnh ngăn xếp vào AX
MOV   BX,[BP+2]   ;chuyển từ thứ hai vào BX
MOV   CX,[BP+4]   ;chuyển từ thứ ba vào BX
```

Một ví dụ tương tự đó là dùng BP để truyền các giá trị cho thủ tục (xem chương 14).

10.3. Sắp xếp một mảng.

Khi một mảng đã được sắp xếp, chúng ta sẽ dễ dàng xác định vị trí các phần tử của nó. Có đến hàng tá các phương pháp sắp xếp. Phương pháp mà chúng tôi sẽ trình bày ở đây gọi là phương pháp chọn lựa. Nó là một trong những phương pháp sắp xếp đơn giản nhất.

Để sắp xếp một mảng N phần tử chúng ta thực hiện các bước sau đây:

Bước 1: Tìm phần tử lớn nhất của A[1] ... A[N]. Đổi chỗ nó và A[N]. Do phần tử lớn nhất đã được đặt ở vị trí N, chúng ta chỉ còn phải sắp xếp A[1] ... A[N-1].

Bước 2: Tìm phần tử lớn nhất của A[1] ... A[N-1]. Đổi chỗ nó và A[N-1]. Động tác này đặt phần tử "lớn thứ hai" vào đúng vị trí của nó.

.
.
.

Bước N-1: Tìm phần tử lớn nhất của A[1], A[2]. Đổi chỗ nó với A[2]. Lúc này A[2] ... A[N] đã được đặt ở vị trí thích hợp của chúng và A[1] cũng vậy. Như thế, mảng đã sắp xếp xong.

Ví dụ, mảng A chứa các số nguyên 21, 5, 16, 40, 7:

Vị trí	*1*	*2*	*3*	*4*	*5*
Giá trị ban đầu	21	5	16	40	7
Bước1	21	5	16	7	40
Bước2	7	5	16	21	40
Bước3	7	5	16	21	40
Bước4	5	7	16	21	40

Thuật toán sắp xếp chọn lựa

```
i = N
FOR  N-1  DO
   tìm vị trí k của phần tử lớn nhất của A[1]... A[i]
   (*)đổi chỗ A[k] và A[i]
   i = i - 1
END_FOR
```

Bước (*) có thể được thực hiện nhờ thủ tục SWAP. Sau đây là chương trình (ta giả thiết mảng cần sắp xếp là mảng các byte):

Chương trình PGM10_2.ASM

```
1:   SELECT   PROC
2:   ;sắp xếp một mảng byte bằng phương pháp chọn lựa.
3:   ;vào:SI=địa chỉ offset của mảng
4:   ;BX=số phần tử của mảng ra
5:   ;ra:SI=offset của mảng đã sắp xếp
```

```
6:   ;sử dụng:SWAP
7:            PUSH  BX
8:            PUSH  CX
9:            PUSH  DX
10:           PUSH  SI
11:           DEC   BX          ;N=N-1
12:           JE    END_SORT    ;thoát nếu còn một phần tử
13:           MOV   DX,SI       ;chép offset của mảng
14:  ;for N-1 times do
15:  SORT_LOOP:
16:           MOV   SI,DX       ;SI trỏ đến mảng
17:           MOV   CX,BX       ;số phép so sánh
18:           MOV   DI,SI       ;DI trỏ đến p.tử lớn nhất
19:           MOV   AL,[DI]     ;AL chứa phần tử lớn nhất
20:  ;xác định phần tử lớn nhất còn lại
21:  FIND_BIG:
22:           INC   SI          ;SI trỏ đến p.tử tiếp theo
23:           CMP [SI],AL       ;p.tử mới > p.tử lớn nhất?
24:           JNG   NEXT        ;không,tiếp tục
25:           MOV   DI,SI       ;đúng,trỏ đến nó
26:           MOV   AL,[DI]     ;AL chứa p.tử lớn nhất
27:  NEXT:
28:           LOOP  FIND_BIG    ;lặp đến khi hoán thành
29:  ;đổi chỗ phần tử lớn nhất với phần tử cuối cùng
30:           CALL  SWAP  ;đổi chỗ với p.tử cuối cùng
31:           DEC   BX          ;N=N-1
32:           JNE   SORT_LOOP   ;lặp lại nếu N<>0
33:  END_SORT:
34:           POP   SI
35:           POP   DX
36:           POP   CX
37:           POP   BX
38:           RET
39:  SELECT   ENDP
40:  SWAP     PROC
41:  ;đổi chỗ hai phần tử của mảng
42:  ;vào:SI=một phần tử
43:  ;DI=phần tử kia
44:  ;ra:đổi chỗ hai phần tử
45:           PUSH  AX          ;cất AX
46:           MOV   AL,[SI]     ;lấy A[i]
47:           XCHG  AL,[DI]     ;đưa vào A[k]
48:           MOV   [SI],AL     ;đưa A[k] vào A[i]
49:           POP   AX          ;khôi phục AX
```

```
50:             RET
51:  SWAP      ENDP
```

Trước khi vào thủ tục SELECT, SI chứa địa chỉ offset của mảng còn BX chứa số phần tử N. Thuật toán sắp xếp mảng có N-1 bước tương ứng với BX giảm dần. Nếu BX bằng 0, chúng ta chỉ còn mảng một phần tử. Đến đây chẳng còn gì để làm nữa và ta thoát khỏi thủ tục.

Trong trường hợp tổng quát, thủ tục tiến vào vòng lặp xử lý chính (dòng 15-32). Mỗi lần duyệt qua vòng lặp này, phần tử lớn nhất trong số các phần tử chưa sắp xếp còn lại được đưa vào vị trí thích hợp của nó.

Các dòng 21-28 là vòng lặp để tìm phần tử lớn nhất trong số các phần tử chưa được sắp xếp còn lại. Khi ra khỏi vòng lặp này, DI trỏ đến phần tử lớn nhất và SI trỏ đến phần tử cuối cùng của mảng. Dòng 30 gọi thủ tục SWAP để hoán chuyển các phần tử được SI và DI trỏ đến.

Ta có thể kiểm tra thủ tục bằng cách chèn nó vào một chương trình thử nghiệm.

Chương trình PGM10_3.ASM

```
TITLE     PGM10_3: Kiểm tra SELECT
.MODEL    SMALL
.STACK    100H
.DATA
ALA       DB    5, 2, 1, 3, 4
.CODE
MAIN      PROC
          MOV   Ax,@DATA
          MOV   DS,AX
          LEA   SI,ALA
          MOV   BX,5
          CALL  SELECT
          MOV   AH,4Ch
          INT   21h
MAIN      ENDP
;SELECT ở đây
          END   MAIN
```

Sau khi biên dịch và liên kết, ta vào DEBUG và thi hành đến lời gọi thủ tục (địa chỉ chỉ ra sau đây đã được xác định trong DEBUG)

```
-GC
AX=100D BX=0005 CX=0049 DX=0000 SP=0100 BP=0000 SI=0004 DI=0000
DS=100D ES=0FF9 SS=100E CS=1009 IP=000C NV UP EI PL NZ NA PO NC
1009:000C E80400 CALL 00013
```

Trước khi gọi thủ tục ta hãy xem lại mảng chưa sắp xếp:

```
-D4 8
100D:0000          05 02 01 03-04
```

Dữ liệu xuất hiện theo thứ tự 5, 2, 1, 3, 4. Bây giờ chúng ta hãy thực hiện SELECT:

```
-GF
AX=1002 BX=0005 CX=0049 DX=0000 SP=0100 BP=0000 SI=0004 DI=0005
DS=100D ES=0FF9 SS=100E CS=1009 IP=000F NV UP EI PL ZR NA PE NC
1009:000F B44C  MOV AH,4C
```

Và ta hãy xem lại mảng:

```
-D4 8
100D:0000  01 02 03 04 05
```

Mảng của chúng ta bây giờ đã được sắp xếp theo thứ tự tăng dần.

10.4. Mảng hai chiều.

Mảng hai chiều (two-dimensional array) là mảng của các mảng có nghĩa là một mảng một chiều mà các phần tử của nó lại là các mảng một chiều. Ta có thể tưởng tượng các phần tử được xếp vào các hàng và các cột. Hình 10.2 chỉ ra một mảng hai chiều B với ba hàng, bốn cột (mảng 3x4). B[i,j] là phần tử thuộc hàng i và cột j.

Mảng hai chiều được lưu trữ trong bộ nhớ như thế nào?.

Bởi vì bộ nhớ là mảng một chiều cho nên các phần tử của mảng hai chiều phải được lưu trữ liên tiếp nhau. Có hai phương pháp thường được sử dụng để lưu trữ mảng. Phương pháp thứ nhất là lưu trữ theo thứ tự hàng(row-major order), trong đó các phần tử hàng 1 được lưu trữ, tiếp theo là các phần tử hàng 2, rồi đến các phần tử hàng 3 và cứ như thế... Phương pháp thứ hai là lưu trữ theo thứ tự cột. Phương pháp này lưu trữ dữ liệu cột 1, tiếp theo là cột 2 rồi đến cột 3 v.v.. . Ví dụ, mảng B chứa 10, 20, 30, 40 trong hàng đầu tiên; 50, 60, 70, 80 trong hàng thứ hai và 90, 100, 110, 120 trong hàng thứ ba. Nó có thể được lưu trữ theo thứ tự hàng như sau:

```
B       DB      10, 20, 30, 40
        DB      50, 60, 70, 80
        DB      90,100,110,120
```

hay theo thứ tự cột:

```
B       DB      10, 50, 90
        DB      20, 60,100
        DB      30, 70,110
        DB      40, 80,120
```

Hình 10.2A. Mảng hai chiều B

Hàng \ Cột	1	2	3	4
1	B[1,1]	B[1,2]	B[1,3]	B[1,4]
2	B[2,1]	B[2,2]	B[2,3]	B[2,4]
3	B[3,1]	B[3,2]	B[3,3]	B[3,4]
4				

Hầu hết các chương trình biên dịch ngôn ngữ bậc cao lưu trữ mảng hai chiều theo thứ tự hàng. Trong Hợp ngữ chúng ta có thể lưu trữ theo cả hai cách. Nếu như các phần tử của một hàng được xử lý liên tiếp nhau thì việc lưu trữ theo thứ tự hàng chiếm ưu thế hơn bởi vì các phần tử kề nhau trong một hàng

chiếm các ô nhớ kế tiếp nhau. Ngược lại, lưu trữ theo thứ tự cột sẽ tốt hơn khi xử lý các phần tử trong cùng một cột.

Xác định vị trí một phần tử trong mảng hai chiều.

Giả thiết ta có một mảng hai chiều lưu trữ trong bộ nhớ theo thứ tự hàng, trong đó kích thước mỗi phần tử là S (mảng byte S = 1, mảng word S = 2). Để tìm vị trí của A[i,j] ta xác định:

1. Vị trí bắt đầu cột i.
2. Vị trí của phần tử thứ j trong hàng.

Sau đây là bước thứ nhất. Hàng 1 bắt đầu tại vị trí A. Do có N phần tử trong mỗi hàng và mỗi phần tử có kích thước S byte, hàng 2 sẽ bắt đầu tại vị trí A + N*S. Tương tự, hàng 3 bắt đầu tại vị trí A + 2*N*S. Một cách tổng quát ta có hàng i sẽ bắt đầu tại vị trí A + (i - 1) * N * S.

Và bây giờ đến bước thứ hai. Như chúng tôi đã trình bày về các mảng một chiều, phần tử thứ j của một hàng được lưu trữ ở vị trí (j - 1) * S byte kể từ đầu dòng.

Cộng kết quả của bước 1 và bước 2 ta được kết quả cuối cùng:

Nếu một mảng A kích thước M x N, mỗi phần tử chiếm S byte được lưu trữ theo theo thứ tự hàng thì:

A[i,j] có địa chỉ A + ((i - 1) * N + (j - 1)) * S (1)

Tương tự với các mảng lưu trữ theo theo thứ tự cột:

Nếu một mảng A kích thước M x N, mỗi phần tử chiếm S byte được lưu trữ theo theo thứ tự cột thì:

A[i,j] có địa chỉ A + ((i - 1) + (j - 1) * M) * S (2)

Ví dụ 10.12. Giả sử A là một mảng word M x N được lưu trữ trong bộ nhớ theo theo thứ tự hàng. Hãy xác định:

1. Vị trí bắt đầu hàng i.
2. Vị trí bắt đầu cột j.
3. Số byte giữa các phần tử trong một cột.

Lời giải:

1. Hàng i bắt đầu tại A[i,1], theo công thức (1) nó có địa chỉ:

A + (i - 1) * N * 2.

2. Cột j bắt đầu tại A[1,j], theo công thức (1) nó có địa chỉ: A + (j - 1) * 2.

3. Do có N cột nên sẽ có 2 * N byte giữa hai phần tử kề nhau trong một cột bất kỳ.

10.5. Chế độ địa chỉ chỉ số cơ sở.

Trong chế độ này, địa chỉ offset của một toán hạng là tổng của:

1. Nội dung của một thanh ghi cơ sở (BX hay BP).
2. Nội dung của một thanh ghi chỉ số (SI hay DI).
3. (có thể) Địa chỉ offset của một biến byte.
4. (có thể) Một hằng số (âm hay dương).

Nếu chúng ta sử dụng BX, DS sẽ chứa số hiệu đoạn của địa chỉ toán hạng, còn nếu BP được sử dụng, SS sẽ chứa số hiệu địa chỉ đoạn. Toán hạng có thể được viết ở nhiều dạng khác nhau, sau đây là một số dạng thường dùng:

1. biến nhớ [thanh ghi cơ sở][thanh ghi chỉ số]
2. [thanh ghi chỉ số + thanh ghi cơ sở + biến nhớ + hằng số]
3. biến nhớ [thanh ghi chỉ số + thanh ghi cơ sở + hằng số]
4. hằng số [thanh ghi chỉ số + thanh ghi cơ sở + biến nhớ]

Thứ tự các phần tử trong dấu ngoặc là tuỳ ý.

Ví dụ W là một biến byte, BX chứa 2 và SI chứa 4. Khi đó lệnh:

```
MOV   AX,W[BX][SI]
```

sẽ chuyển nội dung của W + 2 + 4 = W + 6 vào AX. Lệnh trên cũng có thể được viết dưới các dạng sau:

```
MOV   AX,[W + BX + SI]
```

hay

```
MOV   AX,W[BX + SI]
```

Chế độ địa chỉ chỉ số cơ sở đặc biệt hữu hiệu khi xử lý mảng hai chiều. Bạn sẽ thấy rõ điều này trong ví dụ sau đây.

Ví dụ 10.13. Giả sử A là một mảng word kích thước 5 x 7, lưu trữ số liệu theo thứ tự hàng. Bạn hãy viết các lệnh sử dụng chế độ địa chỉ chỉ số cơ sở để:

1. Xoá hàng 3.

2. Xoá cột 4.

Lời giải:

1. Trong ví dụ 10.2 chúng ta đã biết rằng với A là một mảng word M x N phần tử thì hằng i sẽ bắt đầu tại địa chỉ A + (i - 1) * N * 2. Như vậy trong mảng 5 x 7, hàng 3 bắt đầu tại địa chỉ A + (3 - 1) *7*2 = A + 28. Chúng ta có thể xoá hàng 3 như sau:

```
        MOV   BX,28              ;BX chỉ đến hàng 3
        XOR   SI,SI              ;SI sẽ chỉ đến các cột
        MOV   CX,7               ;số p.tử trong một hàng
CLEAR:
        MOV   A[BX][SI],0        ;xoá A[3,1]
        ADD   SI,2               ;trỏ đến cột kế tiếp
        LOOP  CLEAR              ;lặp đến khi xong
```

2. Cũng trong ví dụ 10.2 chúng biết được cột j bắt đầu tại địa chỉ

A + (j - 1) * 2 với A là mảng word M x N phần tử. Như vậy cột 4 sẽ bắt đầu tại địa chỉ A + (4 - 1) *2 = A + 6. Do A có 7 cột và được lưu trữ theo thứ tự hàng nên để trỏ đến phần tử tiếp theo trong cột 4 ta phải cộng thêm địa chỉ offset với 7 * 2 = 14. Chúng ta có thể xoá cột 4 như sau:

```
        MOV   SI,6          ;SI chỉ đến cột 4
        XOR   BX,BX         ;BX sẽ chỉ đến các hàng
        MOV   CX,5          ;số p.tử trong một cột
CLEAR:
        MOV   A[BX][SI],0       ;xoá A[1,4]
        ADD   BX,14             ;trỏ đến hàng kế tiếp
        LOOP  CLEAR             ;lặp đến khi xong
```

10.6. Một ứng dụng: tính điểm trung bình.

Giả sử một trong lớp có 5 học sinh vừa thi hết học kỳ 1. Kết quả 4 môn thi của họ như sau:

	Bài thi 1	Bài thi 2	Bài thi 3	Bài thi 4
Ngọc Lan	10	8	8	10
Kiều Nga	8	8	9	10
Thu Hiền	9	8	10	10
Ngọc Thuý	8	7	8	10
Thanh Hằng	10	8	6	10

Chúng ta sẽ viết một chương trình để tính điểm trung bình của mỗi môn thi. Để làm điều này, ta chỉ việc cộng các điểm trong mỗi cột rồi chi cho 5.

Thuật toán:

```
1.  j=4
2.  REPEAT
3.        tính tổng các điểm trong cột j
4.        j=j-1
5.  UNTIL j=o
```

Bắt đầu tính tổng từ cột 4 làm cho chương trình của chúng ta ngắn hơn một chút. Bước 3 có thể được viết chi tiết hơn như sau:

```
sum[j]=0
i=1
FOR 5 DO
      sum[j]=sum[j]+score[i,j]
      i=i+1
END_FOR
```

Chương trình PGM10_4.ASM

```
0:  TITLE     PGM10_4: Tính điểm trung bình.
1:  .MODEL    SMALL
2:  .STACK    100H
3:  .DATA
4:  FIVE      DW    5
5:  SCORE     DW    10,8,8,10          ;ngọc lan
6:            DW    8,8,9,10           ;kiều nga
7:            DW    9,8,10,10          ;thu hiền
8:            DW    8,7,8,10           ;ngọc thuý
9:            DW    10,8,6,10          ;thanh hằng
10: AVG       DW    4 DUP (0)
11: .CODE
12: MAIN      PROC
13:           MOV   AX,@DATA
14:           MOV   DS,AX        ;khởi tạo DS
15: ;J=4
16:           MOV   SI,6         ;khởi tạo chỉ đến cột 4
17: REPEAT:
18:           MOV   CX,5         ;số p.tử trong 1 cột
19:           XOR   BX,BX        ;khởi tạo chỉ hàng 1
```

```
20:             XOR   AX,AX          ;khởi tạo tổng =0
21:;tính tổng các điểm trong cột j
22:FOR:
23:             ADD   AX,SCORES[BX+SI];tổng=tổng+điểm
24:             ADD   BX,8           ;chỉ đến hàng tiếp theo
25:             LOOP  FOR            ;cộng hết một cột
26:;end_for
27:;tính trị trung bình cột j
28:             XOR   DX,DX          ;xoá phần cao số bị chia
29:             DIV   FIVE           ;AX=trị trung bình
30:             MOV   AVG[SI],AX ;chứa vào mảng
31:             SUB   SI,2           ;đến cột tiếp theo
32:;until j=0
33:             JNL   REPEAT         ;lặp khi SI>=0
34:;trở về DOS
35:             MOV   AH,4Ch
36:             INT   21h
37:MAIN         ENDP
38:             END   MAIN
```

Các điểm thi được lưu trữ trong một mảng hai chiều (dòng 5-9).

Các dòng 22-25 tính tổng một cột và chứa kết quả vào trong mảng AVG. Chương trình lấy kết quả này chia cho 5 để tính điểm trung bình ở các dòng 28-30.

Các hàng và các cột của mảng SCORE được tham trỏ tương ứng bởi BX và SI. Chúng ta bắt đầu tính tổng từ cột 4 và cột này có địa chỉ bắt đầu là SCORE + 6 do đó SI được khởi tạo bằng 6. Sau khi tính tổng một cột, SI được giảm đi 2 (để trỏ đến cột tiếp theo) đến khi nó bằng 0 thì kết thúc.

Việc thi hành chương trình có thể được xem nhờ DEBUG. Chúng ta thi hành đến trước lệnh trở về DOS và liệt kê mảng AVG (các địa chỉ sử dụng ở đây đã được xác định trong lần làm việc trước với DEBUG).

```
-G29
AX=4C4B BX=0028 CX=0000 DX=0002 SP=0100 BP=0000 SI=FFFE DI=0000
DS=100B ES=0FF9 SS=100F CS=1009 IP=0029 NV UP EI NG ZR AC PO CY
100B:0029 CD21  INT 21

-D30 3D
100B:0030        09 00-07 00 08 00 08 00
```

Các điểm trung bình được lưu trữ: 0009h, 0007h, 0008h và 000Ah, hay viết dưới dạng nhị phân là 9, 7, 8 và 10.

10.7. Lệnh XLAT. MOV AL, [AL] [BX]

Đôi khi trong các ứng dụng chung ta phải chuyển dữ liệu từ một đạng này sang dạng khác. Ví dụ IBM PC dùng mã ASCII cho các ký tự nhưng các máy chủ IBM lại sử dụng EBCDIC (Extended Binary Coded Decimal Interchange Code). Để chuyển đổi một chuỗi ký tự từ mã ASCII sang mã EBCDIC chương trình của chúng ta phải thay mã ASCII của mỗi ký tự trong chuỗi bằng mã EBCDIC tương ứng.

XLAT (translate) là một lệnh không toán hạng, nó có thể được sử dụng để đổi giá trị của một byte sang một giá trị khác được lấy từ một bảng. Byte được đổi phải chứa trong AL và BX chứa địa chỉ offset của bảng chuyển đổi. Lệnh XLAT thực hiện các công việc sau:(1)Cộng nội dung AL với địa chỉ trong BX để nhận được một địa chỉ trong bảng và (2) thay thế nội dung của AL bằng giá trị tìm được ở địa chỉ trên.

Ví dụ: giả sử nội dung của AL trong khoảng từ 0 đến Fh và ta muốn thay nó bằng mã ASCII của chữ số hex tương ứng. Chẳng hạn 6h thay bởi 036h = '6' hay Bh thay bởi 042h = 'B'. Bảng chuyển đổi lúc này sẽ là

```
TABLE    DB 030h,031h,032h,033h,034h,035h,036h,037h
         DB 038h,039h,041h,042h,043h,044h,045h,046h
```

Ví dụ, để đổi 0Ch thành 'C ' ta có thể viết:

```
MOV   AL,0Ch          ;số cần đổi
LEA   BX,TABLE        ;BX chứa offset của bảng
XLAT                  ;AL chứa 'C '
```

Lệnh XLAT ở đây tính địa chỉ TABLE + Ch = TABLE + 12, sau đó nó thay nội dung của AL bởi giá trị tại địa chỉ vừa tìm được, tức là 043h = 'C '.

Trong ví dụ này nếu AL không chứa giá trị trong khoảng (0 - 15), XLAT sẽ thay thế nó bằng một giá trị vô nghĩa.

Ví dụ: mã hoá và giả mã một mặt khẩu.

Trong ví dụ sau đây, chương trình sẽ nhắc người sử dụng đánh vào một dòng chữ, mã hoá nó và in ra dòng chữ đã mã hoá. Cuối cùng dịch ngược trở lại rồi in ra dòng giải mã.

Một ví dụ khi thực hiện chương trình:

```
ENTER A MESSAGE:
GATHER YOUR FORCES AND ATTACK AT DAWT ( dòng chữ vào )
ZXKBGM WULM HUMPGN XJO XKKXPD XK OXSJ..( đã mã hoá )
GATHER YOUR FORCES AND ATTACK AT DAWT( đã giải mã )
```

Thuật toán:

```
in ra lời nhắc
đọc và mã hoá dòng chữ
xuống dòng mới
in ra dòng đã mã hoá
xuống dòng mới
giải mã và in ra kết quả
```

Chương trình PGM10_5.ASM

```
0:  TITLE     PGM10_5: Mật khẩu
1:  .MODEL    SMALL
2:  .STACK    100H
3:  .DATA
4:  ;alphabet                    ABCDEFGHIJKLMNOPQRSTUVWXYZ
5:  CODE_KEY DB 65 DUP (' ')
             DB                'XQPOGHZBCADEIJUVFMNKLRSTWY'
6:           DB 37 DUP (' ')
7:  DECODE_KEY DB 65 DUP (' ')
             DB    'JHIKLQEFMNTURSDCBVWXOPYAZG'
8:           DB 37 DUP (' ')
9:  CODED    DB 80 DUP ('$')
10: PROMPT   DB    'ENTER A MESSAGE',0Dh,0Ah,'$'
11: CRLF     DB    0Dh,0Ah,'$'
12: .CODE
13: MAIN     PROC
14:          MOV   AX,@DATA          ;khởi tạo DX
15:          MOV   DS,AX
16: ;in ra lời nhắc
17:          MOV   AH,9              ;hàm hiển thị chuỗi
18:          LEA   DX,PROMPT         ;DX trỏ đến lời nhắc
```

```
19:             INT   21h                ;in ra lời nhắc
20:;đọc và mã hoá một dòng chữ
21:             MOV   AH,1               ;hàm đọc ký tự
22:             LEA   BX,CODE_KEY  ;BX trỏ đến bảng mã hoá
23:             LEA   DI,CODED      ;DI trỏ đến dòng mã hoá
24:WHILE:
25:             INT   21h                ;đọc một ký tự
26:             CMP   AL,0Dh             ;CR?
27:             JE    ENDWHILE   ;đúng,in ra dòng mã hoá
28:             XLAT                     ;không,mã hoá ký tự
29:             MOV   [DI],AL    ;cất vào dòng mã hoá
30:             INC   DI         ;chuyển con trỏ
31:             JMP   WHILE_     ;xử lý ký tự tiếp theo
32:END_WHILE:
33:;xuốmg dòng mới
34:             MOV   AH,9
35:             LEA   DX,CRLF
36:             INT   21h        ;dòng mới
37:;in ra dòng chữ đã mã hoá
38:             LEA   DX,CODED   ;DX tro đến dòng mã hoá
39:             INT   21h        ;in ra dòng mã hoá
40:;xuốmg dòng mới
41:             LEA   DX,CRLF
42:             INT   21h        ;dòmg mới
43:;mã hoá dòng chữ và in ra kết quả
44:             MOV   AH,2       ;hàm in ký tự
45:             LEA   BX,DECODE_KEY;BX trỏ đến bảng giải mã
46:             LEA   SI,CODED   ;SI trỏ đến dòng mã hoá
47:WHILE1:
48:             MOV   AL,[SI]    ;lấy một ký tự
49:             CMP   AL,'$'     ;ký tự kết thúc chuỗi
50:             JE    ENDWHILE1  ;đúng,kết thúc
51:             XLAT             ;không,giải mã ký tự
52:             MOV   DL,AL      ;đưa vào DL
53:             INT   21h        ;in ra
54:             INC   SI         ;chuyển con trỏ
55:             JMP   WHILE1     ;xử lý ký tự tiếp theo
56:ENDWHILE1
57:             MOV   AH,4Ch
58:             INT   21h        ;trở về DOS
59:MAIN         ENDP
60:             END   MAIN
```

Có ba mảng được khai báo trong đoạn dữ liệu:

1. CODE_KEY dùng để mã hoá.

2. CODED chứa dòng chữ đã được mã hoá, được khởi tạo bằng một chuỗi các ký tự '$ '. Như vậy khi ta đưa chuỗi (chiều dài nhỏ hơn chuỗi khởi tạo) vào vị trí này, nó sẽ được kết thúc bằng ký tự '$ ' và ta có thể in được bằng hàm 9 của ngắt 21h.

3. DECODE_KEY dùng để chuyển đổi dòng chữ đã được mã hoá trở về dạng nguyên thuỷ của nó.

Các bạn sẽ dễ dàng nhận ra cách thức mã hoá và giải mã các ký tự nhờ bảng chữ cái trong dòng lời bình 4 được viết tương ứng với CODE_KEY và DECODE_KEY. Các dòng 24-32 đọc các ký tự rồi mã hoá chúng và kết thúc khi gặp ký tự về đầu dòng. AL nhận mã ASCII của ký tự được đánh vào, XLAT cộng giá trị này với địa chỉ của CODE_KEY trong BX để tìm ra địa chỉ của kết quả chuyển đổi trong bảng CODE_KEY.

Mảng CODE_KEY tạo thành từ 65 ký tự trắng, tiếp theo là các chữ cái, là các giá trị mã hoá của các ký tự từ 'A' đến 'Z' và kết thúc với 37 ký tự trắng nữa để được tổng cộng 128 byte (128 byte là cần thiết bởi vì giới hạn các ký tự ASCII chuẩn từ 0 đến 127). Ví dụ ta đánh vào chữ cái 'A', mã A SCII của nó là 65. XLAT sẽ đưa giá trị của byte có địa chỉ CODE_KEY mà ở đây này là 'X', vào AL. Đến dòng 29, giá trị này được chuyển vào mảng CODED. Một cách tương tự, 'B' đổi thành 'Q', 'C' thành 'P' ... 'Z' thành 'Y' (bảng mã hoá là tuỳ ý). Các ký tự không phải chữ hoa (bao gồm cả ký tự trắng) có mã ASCII trong khoảng từ 0 đến 64 hoặc từ 92 đến 127, được đổi thành ký tự trắng. Các dòng 38-39 in ra dòng chữ đã được mã hoá.

DECODE_KEY cũng bắt đầu và kết thúc tương ứng với 65 và 37 ký tự trắng. Vị trí các chữ cái trong mảng được xác định như sau (nằm dưới bảng chữ cái dòng 4): Do 'A' được mã hoá thành 'X' nên ký tự ở vị trí 'X' trong bảng giải mã phải là 'A'. Tương tự, bởi vì 'B' được mã hoá thành 'Q', chữ cái 'B' phải ở vị trí 'Q' và cứ như vậy.. .

Các dòng 47-56 dịch dòng chữ đã mã hoá. Sau khi đưa địa chỉ của DECODE_KEY và CODED một cách tương ứng vào BX và SI, chương trình chuyển một byte của dòng mã hoá vào AL. Nếu là ký tự '$ ', có nghĩa dòng chữ đã được dịch xong và chương trình được thoát ra. Ngược lại, XLAT cộng AL với địa chỉ DECODE_KEY để nhận được một địa chỉ trong bảng giải mã, sau đó đưa ký tự tìm được vào AL. Dòng 52 chuyển ký tự này vào DL, như vậy nó có thể được in ra bằng hàm 2 của ngắt 21h.

TỔNG KẾT.

- Mảng một chiều là một chuỗi các phần tử có cùng kiểu. Các toán tử giả DB và DW được dùng để khai báo các mảng byte và word.
- Ta có thể xác định địa chỉ một phần tử của mảng bằng cách cộng một hằng số với địa chỉ cơ sở.
- Chế độ địa chỉ của một toán hạng là phương thức xác định địa chỉ của nó. Các chế độ địa chỉ gồm có chế độ địa chỉ thanh ghi, tức thời, trực tiếp, gián tiếp thanh ghi, cơ sở, chỉ số và chế độ địa chỉ chỉ số cơ sở.
- Trong chế độ địa chỉ gián tiếp thanh ghi, toán hạng có dạng [thanh_ghi], ở đây thanh_ghi có thể là BX, SI, DI hay BP. Offset của toán hạng được xác định bằng cách cộng độ dịch với nội dung của thanh_ghi. Với BX, SI và DI, DS chứa số hiệu đoạn; còn với BP, số hiệu đoạn trong SS.
- Các toán tử BYTE PTR và WORD PTR đứng trước một toán hạng dùng để định lại kiểu đã khai báo của toán hạng.
- Toán tử giả LABEL có thể dùng để xác định kiểu dữ liệu cho một biến.
- Mảng hai chiều là mảng một chiều mà các phần tử là các mảng một chiều. Mảng hai chiều có thể được lưu trữ "hàng nối tiếp hàng" (theo theo thứ tự hàng) hay "cột nối tiếp cột" (theo thứ tự cột).
- Trong chế độ địa chỉ chỉ số cơ sở, địa chỉ offset của toán hạng là tổng của: (1)BX hay BP; (2) SI hay DI; (3) Địa chỉ offset của một ô nhớ (...);(4) Một hằng số(...). Ví dụ một dạng hợp lệ: [thanh ghi cơ sở + thanh ghi chỉ số + ô nhớ +hằng số]. DS chứa số hiệu đoạn nếu ta sử dụng BX, còn nếu dùng BP, SS sẽ chứa số hiệu đoạn.
- Chế độ địa chỉ chỉ số cơ sở thường dùng để xử lý mảng hai chiều.
- Lệnh XLAT dùng để đổi giá trị của một byte sang một giá trị khác lấy từ một bảng. AL chứa giá trị được đổi và BX chứa địa chỉ của bảng. Lệnh này cộng AL với địa chỉ offset chứa trong BX để tìm ra được một địa chỉ trong bảng. Nội dung của AL sẽ được thay bằng giá trị tại địa chỉ này.

Các thuật ngữ tin học.

addressing mode	Chế độ địa chỉ: phương thức xác định địa chỉ toán hạng.
base address of array	Địa chỉ cơ sở của mảng: địa chỉ của biến mảng
based addressing mode	Chế độ địa chỉ cơ sở: là chế độ địa chỉ gián tiếp trong đó nội dung của BX hay BP được cộng với độ dịch tạo thành địa chỉ của toán hạng.
column-maior order	Thứ tự cột: cột tiếp theo cột
direct mode	Chế độ địa chỉ trực tiếp: toán hạng là biến nhớ.
displacement	Độ dịch: số được cộng vào nội dung của thanh ghi để xác định địa chỉ offset của toán hạng trong chế độ địa chỉ chỉ số hay địa chỉ cơ sở.
immediate mode	Chế độ địa chỉ tức thời: toán hạng là hằng số.
indexed addressing mode	Chế độ địa chỉ chỉ số: là chế độ địa chỉ gián tiếp trong đó nội dung của SI hay DI được cộng với độ dịch tạo thành địa chỉ offset của toán hạng.
one-domensional array	Mảng một chiều: chuỗi các phần tử có cùng kiểu.
pointer	Con trỏ: thanh ghi chứa địa chỉ offset của một toán hạng.
register mode	Chế độ địa chỉ thanh ghi: toán hàng là một thanh ghi.
row-major order	Thứ tự hàng: hàng nối tiếp hàng.
tow-dimensional array	Mảng hai chiều: Mảng một chiều mà các phần tử là các mảng một chiều.

Các lệnh mới:

XLAT

Các toán tử giả mới:

DUP **LABEL** **PTR**

Bài tập:

1. Giả thiết:

AX chứa 0500h	offset 1000 chứa 0100h
BX chứa 1000h	offset 1500 chứa 0150h
SI chứa 1500h	offset 2000 chứa 0200h
DI chứa 2000h	offset 2500 chứa 0250h
	offset 3000 chứa 0300h

và BETA là một biến word nằm ở địa chỉ offset 1000h.

Với mỗi trong các lệnh sau đây, nếu hợp lệ, hãy cho biết địa chỉ offset của toán hạng nguồn hay thanh ghi và kết quả được lưu trữ trong toán hạng đích.

```
a. MOV   DI,SI
b. MOV   DI,[DI]
c. ADD   AX,[SI]
d. SUB   BX,[DI]
e. LEA   BX,BETA[BX]
f. ADD   [SI],[DI]
g. ADD   BH,[BL]
h. ADD   AH,[SI]
i. MOV   AX,[BX+DI+BETA]
```

2. Cho các khai báo sau đây:

```
A     DW    1,2,3
B     DB    4,5,6
C     LABEL WORD
MSG   DB    'ABC'
```

và giả thiết BX chứa địa chỉ offset của C. Hãy cho biết lệnh nào trong các lệnh sau đây là hợp lệ. Với lệnh hợp lệ hãy cho biết con số được chuyển dịch.

```
a. MOV   AH,BYTE   PTR   A
b. MOV   AX,WORD   PTR   B
c. MOV   AX,C
d. MOV   AX,MSG
e. MOV   AH,BYTE   PTR   C
```

3. Sử dụng BP và chế độ địa chỉ cơ sở thực hiện các thao tác với ngăn xếp sau đây (Bạn có thể dùng một thanh ghi khác nhưng không được dùng các lệnh PUSH và POP):

a. Thay nội dung của hai từ đỉnh ngăn xếp bằng 0.

b. Sao chép một ngăn xếp gồm 5 từ sang một mảng word ST_A RR sao cho ST_ARR chứa đỉnh ngăn xếp, ST_ARR+2 chứa từ tiếp theo của đỉnh ngăn xếp, và cứ như vậy.

4. Viết các lệnh thực hiện các thao tác sau đây trên mảng word A 10 phần tử hay mảng byte B 15 phần tử:

a. Chuyển A[i+1] vào vị trí i với i = 1 ... 9 và A[1] đến vị trí 10.

b. Dùng DX đếm số số 0 của mảng A.

c. Giả sử mảng byte B chứa một chuỗi ký tự. Hãy kiểm tra lần xuất hiện đầu tiên của chữ cái 'E' trong B. Nếu tìm thấy, hãy đặt SI trỏ đến ô nhớ đó. Ngược lại, hãy lập cờ CF.

5. Viết thủ tục FIND_IJ trả về địa chỉ offset của phần tử trong hàng i, cột j của một mảng word hai chiều A có M x N phần tử, được lưu trữ theo thứ tự hàng. Thủ tục nhận i trong AX, j trong BX, N trong CX và địa chỉ offset của A trong DX. Nó trả về địa chỉ offset của phần tử trong DX. Chú ý: bạn có thể bỏ qua khả năng tràn.

6. Để sắp xếp một mảng A gồm N phần tử bằng phương pháp nổi bọt, chúng ta thực hiện các bước sau:

Bước1: Cho j chạy từ 2 đến N, nếu A[j] < A[j-1] ta đổi chỗ A[j] và A[j-1]. Bước này sẽ đưa phần tử lớn nhất đến vị trí N.

Bước2: Cho j chạy từ 2 đến N-1, nếu A[j] < A[j-1] ta đổi chỗ A[j] và A[j-1]. Bước này sẽ đưa phần tử lớn thứ hai đến vị trí N-1.

BướcN-1: Nếu A[2] <A[1] ta đổi chỗ A[2] và A[1]. Lúc này mảng đã được sắp xếp.

Ví dụ:

Giá trị ban đầu	7	5	3	9	1
Bước1	5	3	7	1	9
Bước2	3	5	1	7	9
Bước3	3	1	5	7	9
Bước4	1	3	5	7	9

Bạn hãy viết thủ tục BUBLE để sắp xếp một mảng byte bằng thuật toán nổi bọt. Thủ tục nhận địa chỉ offset của mảng trong SI và số các phần tử trong BX. Sau đó bạn hãy viết một chương trình để người sử dụng đánh vào một chuỗi các số có một chữ số ngăn cách nhau bằng một ký tự trắng và gọi BUBLE để sắp xếp chúng, cuối cùng in ra chuỗi đã sắp xếp trên dòng tiếp theo. Ví dụ:

```
? 2 4 3 7 1 5 6
1 2 3 4 5 6 7
```

Chương trình của bạn phải làm việc được với mảng chỉ có một phần tử.

7. Giả thiết các bản ghi trong ví dụ phần 10.4.3 được lưu trữ như sau:

```
CLASS
        DW    'ngọc lan  ', 10,8,8,10
        DW    'kiều nga  ', 8,8,9,10
        DW    'thu hiền  ', 9,8,10,10
        DW    'ngọc thuý ', 8,7,8,10
        DW    'thanh hằng', 10,8,6,10
```

Mỗi tên chiếm 12 byte. Bạn hãy viết một chương trình in ra tên của mỗi học sinh cùng với với điểm trung bình của cô ta (làm tròn thành số nguyên) trong 4 bài thi.

8. Viết một chương trình bắt đầu bằng một mảng byte không có giá trị ban đầu có kích thước tối đa là 100 và cho phép người sử dụng thêm vào các ký tự sao cho mảng luôn được lưu trữ theo thứ tự tăng dần. Chương trình phải in ra một dấu chấm hỏi, để người sử dụng đánh vào một ký tự và hiển thị mảng có ký tự mới được chèn vào. Kết thúc nhập khi người sử dụng đánh vào phím ESC. Ký tự lặp lại được bỏ qua.

Một ví dụ khi thi hành:

```
?A
A
?D
AD
?B
ABD
?a
ABDa
?D
ABDa
?<ESC>
```

9. Viết một chương trình sử dụng lệnh XLAT để (1) đọc vào một dòng văn bản và (2) in ra dòng tiếp theo với tất cả các chữ thường được đổi thành chữ hoa. Trong dòng vào có thể chứa các ký tự bất kỳ: chữ thường, chữ hoa, chữ số, dấu chấm câu và v.v...

10. Viết thủ tục PRINTHEX sử dụng lệnh XLAT để hiển thị nội dung của BX dưới dạng số hex có 4 chữ số. Kiểm tra nó bằng một chương trình dùng thuật toán nhập số hex trong mục 7.4 để người sử dụng nhập vào một số hex có 4 chữ số, chứa vào BX và gọi PRINTHEX để in nó ra trên dòng tiếp theo.

Chương 11

CÁC LỆNH THAO TÁC CHUỖI

Tổng quan

Trong chương này chúng sẽ xem xét một nhóm lệnh đặc biệt gọi là các lệnh thao tác chuỗi. Trong Hợp ngữ 8086 khái niệm chuỗi bộ nhớ hay chuỗi đơn giản là các mảng byte hay word. Do đó các lệnh thao tác chuỗi được thiết kế cho các thao tác với dữ liệu kiểu mảng.

Dưới đây là một số ví dụ về các công việc có thể thực hiện bằng các lệnh thao tác chuỗi :

- Chép một chuỗi sang chuỗi khác.
- Tìm một byte hay một word xác định trong một chuỗi.
- Lưu các ký tự vào trong chuỗi.
- So sánh các chuỗi theo thứ tự al pha bê của các ký tự.

Các công việc thực hiện bằng các lệnh thao tác chuỗi có thể sử dụng các chế độ địa chỉ gián tiếp thanh ghi mà chúng ta đã học trong chương 10. Tuy nhiên các lệnh thao tác chuỗi có một số ưu điểm bên trong. Chẳng hạn nó có thể tự động điều chỉnh thanh ghi con trỏ và cho phép có những thao tác giữa bộ nhớ với bộ nhớ.

11.1 Cờ định hướng

Trong chương 5 chúng ta đã biết thanh ghi cờ bao gồm 6 cờ trạng thái và 3 cờ điều khiển. Chúng ta cũng biết rằng các cờ trạng thái phản ánh kết quả của một thao tác mà bộ xử lý thực hiện trong khi đó cờ điều khiển được sử dụng để điều khiển các thao tác của bộ xử lý.

Một trong những cờ điều khiển đó là cờ định hướng (DF- Direction Flag). Công dụng của nó là để xác định hướng cho các thao tác chuỗi. Các thao tác này được thực hiện bằng 2 thanh ghi chỉ số SI và DI. Ví dụ, chẳng hạn có một chuỗi được khai báo như sau:

```
STRING1   DB          'ABCDE'
```

Và chuỗi này được lưu trong bộ nhớ bắt đầu tại offset 0200h:.

Địa chỉ offset	Nội dung	Ký tự ASCII
0200h	041h	A
0201h	042h	B
0202h	043h	C
0203h	044h	D
0204h	045h	E

Nếu DF = 0, SI và DI được xử lý theo chiều tăng của các địa chỉ bộ nhớ: từ trái qua phải trong chuỗi. Ngược lại nếu DF = 1, SI và DI được xử lý theo chiều giảm dần các địa chỉ bộ nhớ : từ phải qua trái trong các chuỗi.

Trong DEBUG DF = 0 được ký hiệu là UP còn DF = 1 được ký hiệu là DN.

Các lệnh CLD và STD

Để làm cho DF = 0, chúng ta sử dụng lệnh CLD(CLear Direction flag):

```
CLD   ;xoá cờ định hướng.
```

Để làm cho DF = 1 chúng ta sử dụng lệnh STD (SeT Direction flag):

```
STD   :thiết lập cờ định hướng
```

Các lệnh CLD và STD không làm ảnh hưởng tới các cờ khác.

11.2 Lệnh chuyển một chuỗi

Giả sử chúng ta đã định nghĩa 2 chuỗi như sau:

```
.DATA
STRING1   DB     'HELLO'
STRING2   DB     5 DUP (?)
```

Và bây giờ chúng ta muốn chuyển nội dung của chuỗi STRING1(chuỗi nguồn) vào chuỗi STRING2 (chuỗi đích). Thao tác này rất cần thiết cho các công việc xử lý chuỗi như là lặp chuỗi hay nối chuỗi (gắn một chuỗi vào cuối chuỗi khác).

Lệnh MOVSB

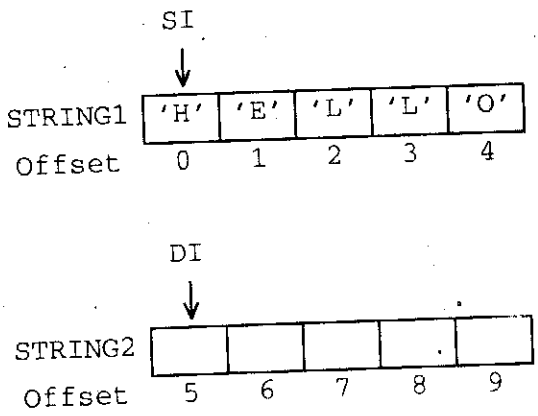
```
MOVSB           ;chuyển một chuỗi các byte
```

sẽ chép nội dung của byte được định địa chỉ bởi DS:SI đến byte được định địa chỉ bởi ES:DI. Nội dung của byte nguồn không bị thay đổi. Sau khi byte được chuyển cả 2 thanh ghi SI và DI đều tự động tăng lên 1 nếu DF = 0 hay giảm đi 1 nếu DF = 1. Ví dụ nếu muốn chuyển 2 byte đầu tiên của STRING1 đến STRING2 chúng ta có thể thực hiện những lệnh sau:

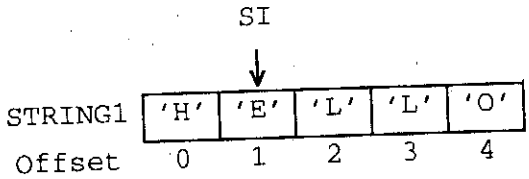
```
MOV   AX,@DATA
MOV   DS,AX         ;Khởi tạo DS
MOV   ES,AX         ;và ES
LEA   SI,STRING1 ;SI trỏ đến chuỗi nguồn
LEA   DI,STRING2 ;DI trỏ đến chuỗi đích
CLD                 ;Xoá DF
MOVSB       ;Chuyển byte đầu tiên
MOVSB       ;rồi đến byte thứ 2
```

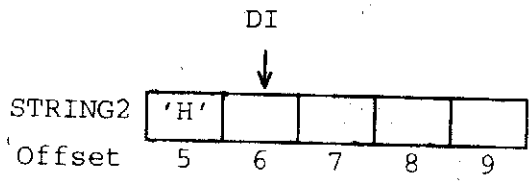
Xem hình 11.1

Trước lệnh MOVSB

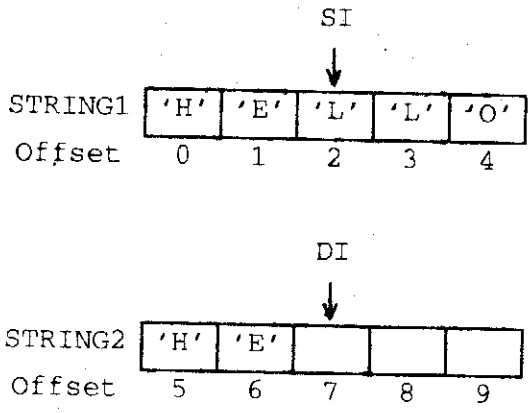


Sau lệnh MOVSB thứ nhất.





Sau lệnh MOVSB thứ hai.



Hình 11.1 Lệnh MOVSB

Lệnh MOVSB là lệnh đầu tiên cho phép thao tác với bộ nhớ mà chúng ta học, Đây cũng là lệnh đầu tiên liên quan đến thanh ghi ES.

Lệnh khởi tạo REP

Lệnh MOVSB chỉ chuyển 1 byte từ chuỗi nguồn đến chuỗi đích. Để chuyển cả chuỗi đầu tiên chúng ta phải khởi tạo CX với số N bằng số byte trong chuỗi nguồn và thực hiện lệnh sau:

```
REP   MOVSB
```

Lệnh khởi tạo REP có tác dụng làm cho MOVSB được thực hiện N lần. Sau mỗi lệnh MOVSB, CX được giảm đi 1 cho đến khi nó bằng 0. Ví dụ để chuyển chuỗi STRING1 của ví dụ trước vào chuỗi STRING2, chúng ta có thể làm như sau:

```
CLD
LEA   SI,STRING1
LEA   DI,STRING2
MOV   CX,5          ;Số ký tự trong chuỗi STRING1
```

```
        REP   MOVSB
```

Ví dụ 11.1 Viết các lệnh chép chuỗi STRING1 trong phần trước vào chuỗi STRING2 theo thứ tự ngược lại

Lời giải:

Ý tưởng ở đây là cho SI trỏ đến cuối chuỗi STRING1 và DI trỏ đến đầu chuỗi STRING2 sau đó chuyển các ký tự trong khi SI di chuyển dọc theo chuỗi STRING1 từ trái qua phải.

```
        LEA   SI,STRING1+4;SI trỏ đến cuối chuỗi STRING1
        LEA   DI,STRING2  ;DI trỏ đến đầu chuỗi STRING2
        STD               ;Xử lý từ phải qua trái
        MOV   CX,5
MOVE:
        MOVSB             ;Chuyển từng byte
        ADD   DI,2
        LOOP  MOVE
```

Ở trên chúng ta phải cộng thêm 2 vào DI sau mỗi lệnh MOVSB bởi vì khi DF = 1 thì sau mỗi lệnh MOVSB, SI và DI tự động giảm đi 1 trong khi chúng ta lại muốn tăng DI.

Lệnh MOVSW

Có dạng word cho lệnh MOVSB đó là:

```
        MOVSW         ;chuyển một chuỗi các word
```

Lệnh MOVSW chuyển từng word từ chuỗi nguồn đến chuỗi đích. Giống như lệnh MOVSB nó yêu cầu DS:SI trỏ đến chuỗi nguồn và ES:DI trỏ đến chuỗi đích. Sau khi một word của chuỗi được chuyển cả SI và DI cùng tăng lên 2 đơn vị nếu DF = 0 hoặc cùng giảm đi 2 nếu DF = 1.

MOVSB và MOVSW đều không làm ảnh hưởng tới cờ

Ví dụ 11.2 Cho mảng sau đây:

```
ARR DW    10,20,40,50,60,?
```

Hãy viết các lệnh để chèn 30 vào giữa 20 và 40 (giả thiết rằng DS và ES đã chứa địa chỉ đoạn dữ liệu).

Lời giải:

Chúng ta sẽ chuyển 40, 50 và 60 dịch lên một vị trí trong mảng rồi chèn 30 vào.

```
STD                          ;Xử lý từ phải qua trái
LEA   SI,ARR+8h              ;SI trỏ đến 60
LEA   DI,ARR+Ah              ;DI trỏ đến ?
MOV   CX,3                   ;Chuyển 3 phần tử
REP   MOVSW                  ;Chuyển 40, 50 ,60
MOV   WORD PTR [DI],30 ;Chèn 30
```

Chú ý : toán tử PTR đã trình bày trong phần 10.2.3

11.3 Lệnh lưu chuỗi

Lệnh STOSB:

```
STOSB          ;Lưu chuỗi các byte
```

Có tác dụng chuyển nội dung của thanh ghi AL đến byte được định địa chỉ bởi ES:DI. Sau khi lệnh được thực hiện DI tăng 1 nếu DF = 0 hoặc giảm 1 nếu DF = 1. Tương tự như vậy lệnh STOSW:

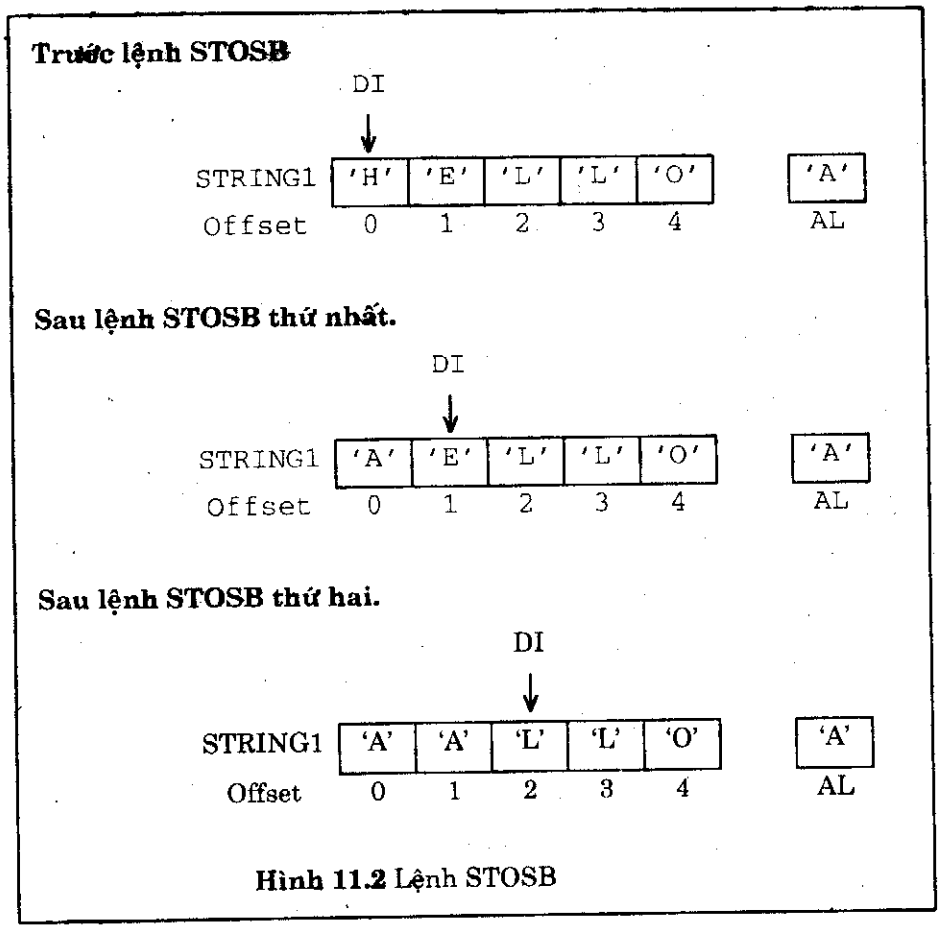
```
STOSW          ;Lưu chuỗi các word
```

Chuyển nội dung của thanh ghi AX vào word được định địa chỉ bởi ES:DI và tăng hay gảm DI 2 đơn vị tuỳ theo trạng thái cờ DF.

Các lệnh STOSB và STOSW không ảnh hưởng tới cờ. Để làm ví dụ về lệnh STOSB, đoạn lệnh dưới đây sẽ lưu 2 chữ "A" vào chuỗi STRING1:

```
MOV   AX,@DATA
MOV   ES,AX          ;Khởi tạo ES
LEA   DI,STRING1 ;DI trỏ đến STRING1
CLD                  ;Xử lý từ trái sang phải
MOV   AL,'A'         ;AL chứa ký tự cần lưu
STOSB                ;Lưu chữ cái "A"
STOSB                ;Lưu chữ thứ 2
```

Xem hình 11.2



Hình 11.2 Lệnh STOSB

Đọc và lưu một chuỗi ký tự

Hàm con số 1 của ngắt 21h đọc một ký tự từ bàn phím vào trong AL. Bằng cách lặp lại công việc này và kết hợp với lệnh STOSB chúng ta có thể đọc và lưu một chuỗi. Thêm vào đó chúng ta còn có thể xử lý các ký tự trước khi lưu chúng.

Thủ tục READ_STR dưới đây đọc và lưu các ký tự trong một chuỗi, cho đến khi người sử dụng đánh ENTER. Thủ tục được bắt đầu với địa chỉ offset của chuỗi ở trong DI. Sau khi thực hiện xong nó trả về địa chỉ offset chuỗi trong DI và số ký tự nhập vào trong BX. Nếu người sử dụng đánh nhầm một ký tự và dùng phím Backspace để xoá thì thủ tục cũng sẽ xoá ký tự đó khỏi chuỗi.

Thủ tục này tương tự như hàm 0Ah ngắt 21h của DOS (xem bài tập 11.11)

Thuật giải cho READ_STR

```
chars_read = 0
Đọc một ký tự
WHILE  ký tự không phải CR  DO
   IF  ký tự là Backspace
       THEN
        chars_read = chars_read-1
        Xoá ký tự trước đó khỏi chuỗi
       ELSE
        Lưu ký tự vào chuỗi
        chars_read = chars_read+1
   END_IF
   Đọc tiếp ký tự
END_WHILE
```

Chương trình nguồn PGM11_1.ASM

```
1:  READ_STR PROC  NEAR
2:  ;đọc và lưu một chuỗi
3:  ;Vào:DI chứa địa chỉ offset của chuỗi
4:  ;RA :DI chứa địa chỉ offset của chuỗi
5:  ;BX chứa số ký tự nhập được
6:           PUSH  AX
7:           PUSH  DI
8:           CLD                ;Xử lý từ bên trái sang
9:           XOR   BX,BX        ;Số ký tự nhập được
10:          MOV   AH,1         ;Hàm đọc một ký tự
11:          INT   21H          ;Đọc 1 ký tự vào AL
12:WHILE1:
13:          CMP   AL,0DH       ;Ký tự CR?
14:          JE    END_WHILE1   ;Đúng! kết thúc
15:;Nếu ký tự là Backspace
16:          CMP   AL,8H        ;Backspace?
17:          JNE   ELSE1        ;Không! lưu nó vào chuỗi
18:;Thì
19:          DEC   DI           ;Đúng!,lùi con trỏ chuỗi
20:          DEC   BX     ;Giảm bộ đếm số ký tự nhập được
21:          JMP   READ         ;và đọc ký tự khác
22:ELSE1:
23:          STOSB              ;Lưu ký tự vào chuỗi
24:          INC   BX           ;Tăng bộ đếm số KT
25:READ:
26:          INT   21H          ;Đọc ký tự vào AL
```

```
27:             JMP   WHILE1      ;Và tiếp tục vòng lặp
28: END_WHILE1:
29:             POP   DI
30:             POP   AX
31:             RET
32: READ_STR    ENDP
```

Tại dòng 23 thủ tục sử dụng lệnh STOSB để lưu ký tự nhập được vào chuỗi. STOSB tự động tăng DI và tại dòng 24, bộ đếm số ký tự nhập được trong BX được tăng lên 1.

Thủ tục còn có khả năng quản lý các lỗi khi đưa ký tự vào. Nếu người sử dụng đánh phím Backspace thì tại dòng 19 thủ tục giảm DI và BX, bản thân ký tự Backspace cũng không được lưu. Khi ký tự tiếp theo được đọc nó sẽ thay thế ký tự bị lỗi trước đó. Chú ý rằng nếu phím Backspace được đánh ngay trước khi đánh ENTER thì ký tự lỗi vẫn nằm trong chuỗi nhưng số ký tự hợp lệ nhận được trong BX sẽ được điều chỉnh đúng.

Chúng ta sẽ sử dụng READ_STR cho việc nhập một chuỗi trong phần sau.

11.4 Nạp một chuỗi

Lệnh LODSB

```
LODSB ; nạp một chuỗi các byte
```

chuyển byte tại địa chỉ được chỉ ra bởi DS:SI vào AL. SI sau đó được tăng thêm 1 nếu DF = 0 và ngược lại bị giảm đi 1 nếu DF = 1. Dạng word của lệnh như sau:

```
LODSW ;nạp một chuỗi các word
```

Lệnh này chuyển word tại địa chỉ được chỉ ra bởi DS:SI vào AX. Sau đó SI được tăng hay giảm 2 tuỳ theo trạng thái cờ DF. Lệnh LODSB có thể sử dụng để kiểm tra các ký tự của một chuỗi như chúng ta sẽ thấy sau này.

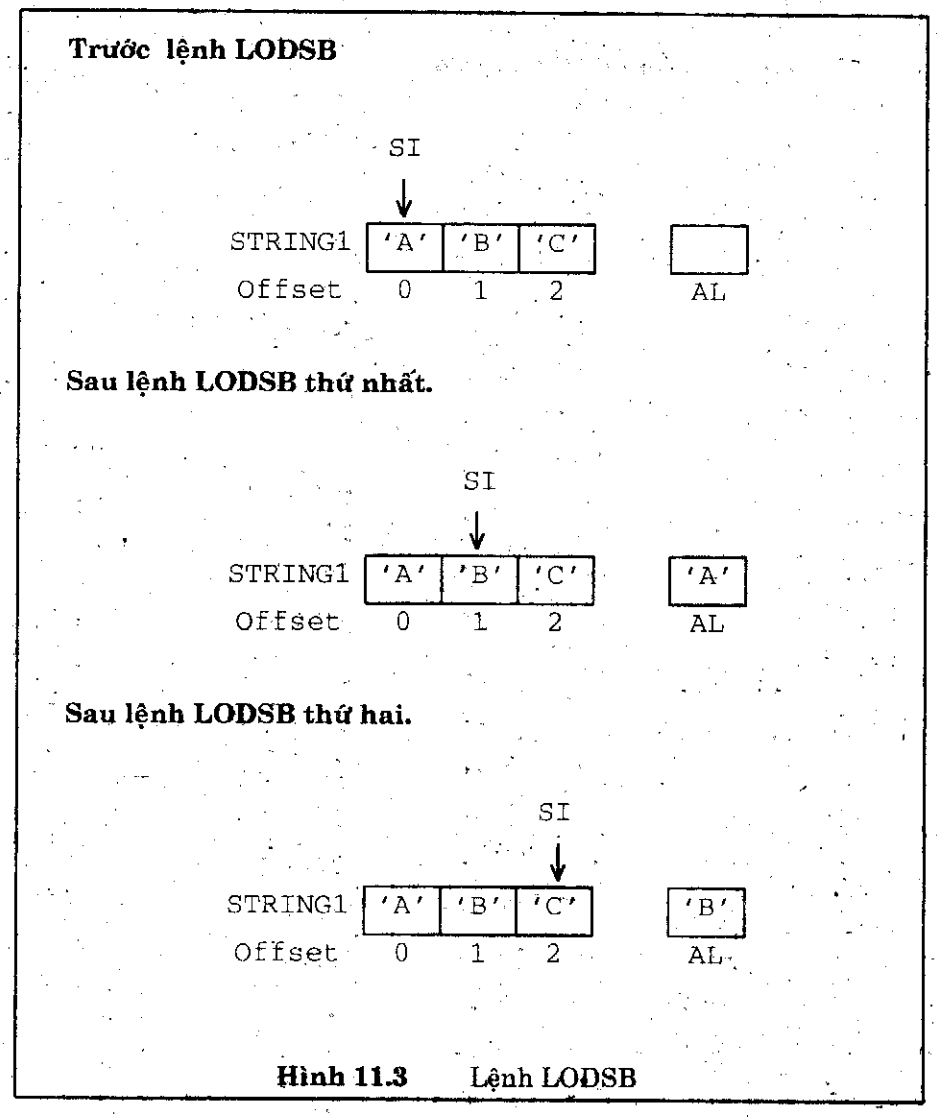
Các lệnh LODSB và LODSW đều không tác động đến cờ. Để miêu tả lệnh LODSB chúng ta giả sử có chuỗi được định nghĩa như sau:

```
STRING1     DB     'ABC'
```

Đoạn lệnh dưới đây lần lượt nạp các byte thứ nhất và thứ 2 vào trong AL

```
MOV   AX,@DATA
MOV   DS,AX              ;Khởi tạo DS
LEA   SI,STRING1         ;SI trỏ tới STRING1
CLD                      ;Xử lý từ trái sang phải
LODSB                    ;Nạp ký tự đầu tiên và AL
LODSB                    ;Nạp ký tự thứ 2 vào AL
```

Xem hình 11.3



Hình 11.3 Lệnh LODSB

Hiển thị một chuỗi ký tự

Thủ tục DISP_STR dưới đây hiển thị chuỗi được trỏ đến bởi SI với số ký tự trong BX. Thủ tục này có thể sử dụng để hiển thị toàn bộ hay một phần của chuỗi.

Thuật toán cho DISP_STR

```
FOR Đếm lần   DO /*Đếm = số ký tự cần hiển thị*/
   Nạp chuỗi ký tự vào AL
   Chuyển vào DL
   Hiển thị ký tự
END_FOR
```

Chương trình nguồn PGM11_2.ASM

```
DISP_STR PROC  NEAR
;Hiển thị một chuỗi ký tự
;Vào :     SI = offset chuỗi
;          BX = số các ký tự cần hiện thị
;ra : none
          PUSH  AX
          PUSH  BX
          PUSH CX
          PUSH  DX
          PUSH  SI
          MOV   CX,BX       ; khởi tạo bộ đếm trong DX
          JCXZ  P_EXIT ;Kếtthúc nếu không có ký tự nào
          CLD               ;Xử lý từ trái sang phải
          MOV   AH,2        ;Chuẩn bị đưa ký tự ra
                            ;thiết bị xuất lỗi chuẩn
TOP:
          LODSB             ;đưa 1 ký tự vào AL
          MOV   DL,AL       ;Chuyển nó vào trong DL
          INT   21H         ;Hiển thị ký tự
          LOOP  TOP         ;Lặp lại cho đến khi
                            ;hiển thị hết
P_EXIT:
          POP   SI
          POP   DX
          POP   CX
          POP   BX
          POP   AX
          RET
DISP_STR ENDP
```

Để xem READ_STR và DISP_STR hoạt động ra sao chúng ta viết một chương trình đọc một chuỗi ký tự (dài tối đa 80 ký tự) và hiển thị 10 ký tự đầu tiên của nó ở dòng tiếp theo.

Chương trình Nguồn PGM11_3.ASM

```
    TITLE    PGM11_3:Kiểm tra 2 thủ tục READ_str Và
disp_str
    .MODEL   SMALL
    .STACK
    .DATA
    STRING   DB    80 DUP(0)
    CRLF     DB    0DH,0AH,'$'
    .CODE
    MAIN     PROC
             MOV   AX,@DATA
             MOV   DS,AX
             MOV   ES,AX
    ;Đọc vào một chuỗi
             LEA   DI,STRING  ;DI trỏ tới chuỗi
             CALL  READ_STR   ;BX = số ký tự nhập được
    ;Nhảy đến đầu dòng mới
             LEA   DX,CRLF    ; SI trỏ tới chuỗi
             MOV   AH,09H     ; hiển thị 10 kí tự
             INT   21H
    ;Hiển thị chuỗi
             LEA   SI,STRING
             MOV   BX,10
             CALL  DISP_STR
    ;Trở về DOS
             MOV   AX,4CH
             INT   21H
    MAIN     ENDP
    ;RAED_STR đặt ở đây
    ;DISP_STR đặt ở đây
             END   MAIN
```

Ví dụ chạy chương trình :

```
C>PGM11_3
THIS PROGRAM TETS TWO PROCEDURES
THIS PROGR
```

11.5 Lệnh duyệt chuỗi (Scan string)

Lệnh:

```
SCASB        ;Duyệt một chuỗi các byte
```

có thể được sử dụng để tìm một byte (tạm gọi là byte đích- target byte) trong một chuỗi. Byte đích được chứa trong AL. Lệnh SCASB lấy nội dung của AL trừ đi từng byte trong chuỗi và sử dụng kết quả để thiết lập các cờ. Kết quả không được lưu lại và sau mỗi lần thực hiện phép trừ DI được tăng 1 nếu DF = 0 và giảm 1 nếu DF = 1.

Dạng word của lệnh như sau:

```
SCASW        ;Duyệt chuỗi các word
```

Trong trường hợp này word đích đặt trong AX. SCASW trừ nội dung của AX cho từng word của chuỗi và thiết lập cờ. DI được tăng 2 nếu DF = 0 và giảm 2 nếu DF =1.

Tất cả các cờ đều bị ảnh hưởng bởi 2 lệnh này.

Ví dụ nếu chuỗi :

```
STRING1     DB     'ABC'
```

được định nghĩa thì các lệnh sau sẽ kiểm tra xem 2 byte đầu của chuỗi có phải là chữ cái 'B' hay không

```
MOV   AX,@DATA
MOV   ES,AX              ;Khởi tạo ES
CLD                      ;Xử lý từ trái sang
LEA   DI,STRING1         ;DI trỏ đến chuỗi STRING1
MOV   AL,'B'             ;Ký tự đích
SCASB                    ;Duyệt byte đầu tiên
SCASB                    ;Duyệt byte thứ 2
```

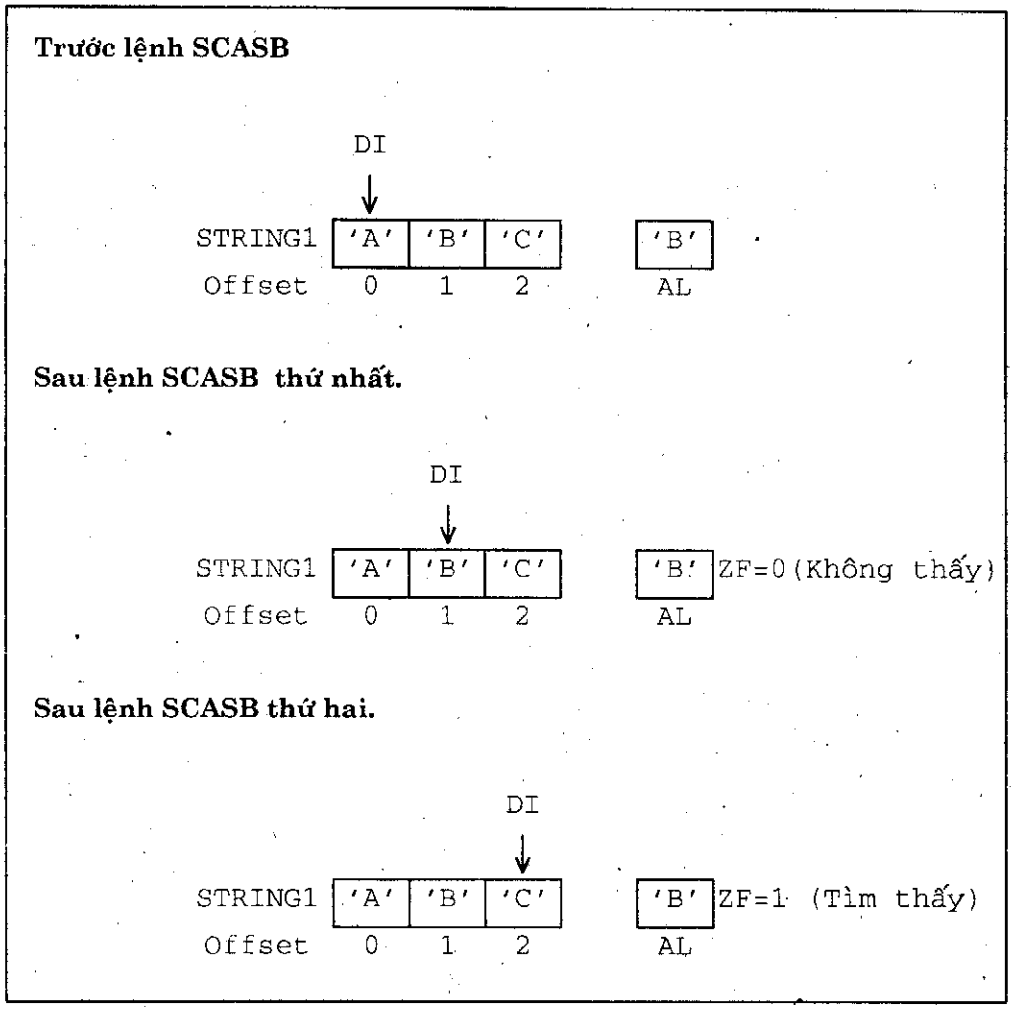
Chúng ta hãy xem hình 11.4 chú ý rằng khi đích 'B' được tìm thấy, cờ ZF được thiết lập và do DI tự động được tăng thêm 1 nên DI trỏ tới ký tự đứng sau ký tự cần tìm.

Khi tìm một byte đích trong chuỗi, chuỗi sẽ được duyệt cho đến khi byte được tìm thấy hoặc hết chuỗi. Nếu CX nhận giá trị đầu là số byte trong chuỗi,

```
REPNE  SCASB       ;lặp lại khi chưa bằng(chưa đến đích)
```

sẽ lặp lại phép trừ nội dung của AL đi từng byte trong chuỗi, điều chỉnh DI, giảm CX cho đến khi có kết quả 0 (khi tìm thấy byte đích) hoặc CX = 0 (khi chuỗi kết thúc).

Chú ý : Lệnh REPNZ (REPeat while Not Zero) tạo ra mã máy giống như lệnh REPNE (REPeat while Not Equal).



Hình 11.4 Lệnh SCASB

Để làm ví dụ chúng hãy viết một chương trình đếm số nguyên âm và phụ âm trong một chuỗi.

Thuật toán đếm số nguyên âm và phụ âm của một chuỗi

Khởi tạo bộ đếm số nguyên âm(SNA)và số phụ âm(SPA)bằng 0

```
Đọc và lưu một chuỗi
REPEAT
    Nạp một ký tự của chuỗi
    IF    nó là nguyên âm
        THEN  tăng SNA
          ELSE  IF    nếu nó là phụ âm
                      THEN tăng SPA
                END_IF
    END_IF
UNTIL           hết chuỗi
Hiển thị số nguyên âm
Hiển thị số phụ âm
```

Chúng ta sẽ sử dụng thủ tục READ_STR để đọc một chuỗi. Thủ tục trở về với DI trỏ tới chuỗi và BX chứa số ký tự đọc được. Để hiển thị số nguyên âm và số phụ âm trong chuỗi chúng ta sẽ sử dụng thủ tục OUTDEC trong chương 9. Thủ tục này hiển thị nội dung của AX dưới dạng một số thập phân có dấu. Để đơn giản chúng ta giả thiết các ký tự được nhập đều là các chữ in hoa.

Chương trình Nguồn PGM11_4.ASM

```
0:  TITLE    PGM11_4:Đếm số nguyên âm và phụ âm
1:  .MODEL   SMALL
2:  .STACK   100H
3:  .DATA
4:  STRING   DB    80 DUP(0)
5:  NA       DB    'AEIOU'
6:  PA       DB    'BCDFGHJKLMNPQRSTVWXYZ'
7:  TB1      DB    0DH,0AH,'So nguyen am = $'
8:  TB2      DB    ', So phu am = $'
9:  SNA      DW    0
10: SPA      DW    0
11: .CODE
12: MAIN     PROC
13:          MOV   AX,@DATA
14:          MOV   DS,AX          ;Khởi tạo DS
15:          MOV   ES,AX          ; và ES
16:          LEA   DI,STRING      ;DI trỏ đến chuỗi
17:          CALL  READ_STR       ;BX chứa số ký tự đọc được
18:          MOV   SI,DI          ;SI trỏ tới chuỗi
19:          CLD                  ;Xử lý từ trái qua phải
20: REPEAT:
21: ;Nạp chuỗi ký tự
22:          LODSB                ;Ký tự trong AL
```

```
23: ;Nếu nó là nguyên âm
24:         LEA   DI,NA ;DI trỏ tới chuỗi các nguyên âm
25:         MOV   CX,5        ;Có tất cả 5 nguyên âm
26:         REPNE SCASB;Ký tự có phải là nguyên âm?
27:         JNE   PHU_AM  ;Không!,có thể là một phụ âm
28: ;Thì tăng số nguyên âm
29:         INC   SNA
30:         JMP   UNTIL
31: ;Hay nếu nó là phụ âm
32: PHU_AM:
33:         LEA   DI,PA ;DI trỏ tới chuỗi các phụ âm
34:         MOV   CX,21       ;Có tất cả 21 phụ âm
35:         REPNE SCASB       ;Ký tự có phải là phụ âm?
36:         JNE   UNTIL       ;Không!
37: ;Thì tăng số phụ âm
38:         INC   SPA
39: UNTIL:
40:         DEC   BX          ;BX chứa số ký tự còn lại
                              ;trong chuỗi
41:         JNE   REPEAT      ;Lặp lại nếu vẫn còn ký tự
43                            ;trong chuỗi
42: ;Đưa ra số nguyên âm
43:         MOV   AH,9        ;Chuẩn bị đưa ra màn hình
44:         LEA   DX,TB1;Lấy thông báo về số nguyên âm
45:         INT   21H         ;Hiển thị nó
46:         MOV   AX,SNA      ;Lấy số nguyên âm
47:         CALL  OUTDEC      ;Hiển thị nó
48: ;Đưa ra số phụ âm
49:         MOV   AH,9        ;Chuẩn bị đưa ra màn hình
50:         LEA   DX,TB2  ;Lấy thông báo về số phụ âm
51:         INT   21H         ;Hiển thị nó
52:         MOV   AX,SPA      ;Lấy số phụ âm
53:         CALL  OUTDEC      ;Hiển thị nó
54: ;Trở về DOS
55:         MOV   AX,4CH
56:         INT   21H
57: MAIN    ENDP
58: ;READ_STR đặt ở đây
59: ;OUTDEC đặt ở đây
60:         END   MAIN
```

Vì chương trình sử dụng cả 2 lệnh :LODSB để nạp byte tại địa chỉ DS:SI và SCASB để duyệt byte tại địa chỉ ES:DI, cả DS và ES đều phải khởi tạo để chúng chứa địa chỉ đoạn dữ liệu. BX được dùng như là bộ đếm vòng lặp và nó nhận giá

trị ban đầu bằng số ký tự trong chuỗi (CX được sử dụng ở nơi khác trong chương trình).

Dòng 22. Lệnh LODSB đưa ký tự vào AL và cho SI trỏ đến ký tự tiếp theo.

Dòng 26. Để xem ký tự trong AL có phải là nguyên âm hay không chương trình duyệt chuỗi NA bằng lệnh REPNE SCASB. Lệnh này trừ nội dung của AL cho mỗi byte của chuỗi NA và thiết lập cờ. Lệnh sẽ trả về cờ ZF = 1 nếu ký tự là nguyên âm và ZF = 0 nếu không phải.

Dòng 35. Nếu ký tự trong AL không phải nguyên âm, chương trình duyệt chuỗi PA theo cách giống như đã làm với chuỗi NA.

Ví dụ thực hiện

```
C>PGM11_4
A,E,I,O,U LA CAC NGUYEN AM
SO NGUYEN AM = 11,    SO PHU AM = 7
```

11.6 Lệnh so sánh chuỗi

Lệnh CMPSB

```
CMPSB          ;so sánh chuỗi các byte
```

trừ byte tại địa chỉ DS:SI cho byte tại địa chỉ ES:DI và thiết lập các cờ. Kết quả không được lưu lại. Và sau đó cả SI và DI cùng tăng 1 nếu DF = 0 hay giảm 1 nếu DF = 1.

Dạng word của lệnh CMPSB là:

```
CMPSW          ;so sánh chuỗi các word
```

Lệnh này trừ word tại địa chỉ DS:SI cho word tại địa chỉ ES:DI và thiết lập các cờ. Nếu DF =0 SI và DI được tăng 2 và ngược lại nếu DF = 1 SI và DI bị giảm đi 2. Lệnh CMPSW rất hữu hiệu khi dùng để so sánh các mảng số kiểu word.

Tất cả các cờ đều bị tác động bởi 2 lệnh CMPSB và CMPSW. Ví dụ chúng ta có:

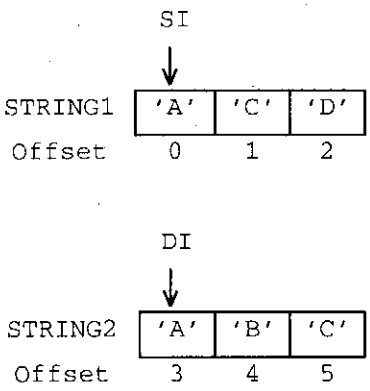
```
.DATA
STRING1    DB    'ACD'
STRING2    DB    'ABC'
```

Đoạn lệnh dưới đây sẽ so sánh 2 byte đầu tiên của 2 chuỗi trên:

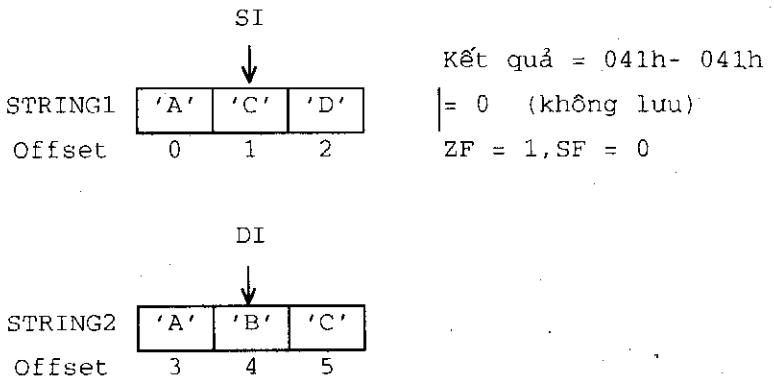
```
MOV   AX,@DATA
MOV   DS,AX              ;Khởi tạo DS
MOV   ES,AX              ;và ES
CLD                      ;Xử lý từ trái qua phải
LEA   SI,STRING1         ;SI trỏ tới STRING1
LEA   DI,STRING2         ;DI trỏ tới STRING2
CMPSB                    ;So sánh byte đầu tiên
CMPSB                    ;So sánh byte thứ 2
```

Xem hình 11.5

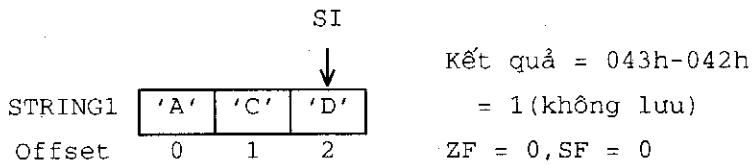
Trước lệnh CMPSB

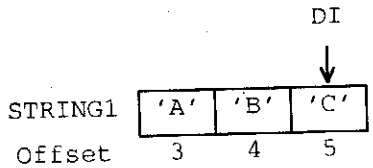


Sau lệnh CMPSB thứ nhất.



Sau lệnh CMPSB thứ hai.





Hình 11.5 Lệnh CMPSB

Các lệnh khởi tạo REPE và REPZ

Việc so sánh chuỗi có thể được thực hiện bằng cách gắn thêm lệnh khởi tạo REPE (Repeat While Equal) hay REPZ (Repeat While Zero) vào các lệnh CMPSB và CMPSW. CX nhận giá trị ban đầu là số byte trong chuỗi ngắn hơn sau đó ta có:

```
REPE CMPSB ;So sánh chuỗi các byte khi còn bằng
           nhau
```

hay

```
REPE CMPSW ;So sánh chuỗi các word khi còn bằng
           nhau
```

sẽ lặp lại việc thực hiện các lệnh CMPSB hay CMPSW rồi giảm CX cho đến khi

1. Có 2 byte tương ứng ở 2 chuỗi không bằng nhau

hoặc 2. CX = 0.

Các cờ được thiết lập tuỳ theo kết quả của phép so sánh cuối cùng.

Lệnh CMPSB có thể sử dụng để so sánh 2 chuỗi xem chuỗi nào đứng trước theo thứ tự al pha bê, chúng có giống nhau hay chuỗi này có là chuỗi con của chuỗi kia không (có nghĩa là một chuỗi được chứa trong chuỗi còn lại như là một chuỗi các ký tự liên tiếp nhau).

Để làm ví dụ chúng ta giả sử STR1 và STR2 là 2 chuỗi có chiều dài 10. Đoạn lệnh sau đây sẽ đặt 0 vào ÁX nếu 2 chuỗi giống nhau, đặt 1 nếu chuỗi STR1 đứng trước chuỗi STR2 và đặt 2 nếu chuỗi STR2 đứng trước chuỗi STR1 (giả thiết rằng DS và ES đã được khởi tạo giá trị thích hợp).

```
MOV     CX,10       ;Chiều dài của các chuỗi
LEA     SI,STR1     ;SI trỏ đến STR1
LEA     DI,STR2     ;DI trỏ tới STR2
CLD                 ;Xử lý từ trái qua phải
REPE    CMPSB       ;So sánh chuỗi các byte.
```

```
        JL      STR1_FIRST ;STR1 đứng trước STR2
        JG      STR2_FIRST ;STR2 đứng trước STR1
;Đây là trường hợp 2 chuỗi giống nhau
        MOV     AX,0       ;Đặt 0 vào AX
        JMP     EXIT       ;Và kết thúc
;Đây là trường hợp STR1 đứng trước STR2
STR1_FIRST:
        MOV     AX,1       ;Đặt 1 vào AX và
        JMP     EXIT       ;Kết thúc
;Đây là trường hợp STR2 đứng trước STR1
STR2_FIRST:
        MOV     AX,2       ;Đặt 2 vào AX
EXIT:
```

11.6.1 Tìm một chuỗi con của một chuỗi

Có một số cách để xác định xem một chuỗi có phải là chuỗi con của chuỗi khác hay không. Phương pháp mà chúng tôi trình bày dưới đây có lẽ là một phương pháp đơn giản nhất. Giả sử chúng ta khai báo :

```
SUB1    DB    'ABC'
SUB2    DB    'CAB'
MAINST  DB    'ABABCA'
```

và chúng ta muốn biết SUB1 và SUB2 có phải là chuỗi con của MAINST hay không. Hãy bắt đầu từ chuỗi SUB1. Chúng ta có thể so sánh các ký tự tương ứng trong 2 chuỗi :

```
SUB1              A  B  C
                  |  |  +
MAINST            A  B  A  B  C  A
```

Vì có sự khác nhau trong phép so sánh thứ 3 chúng ta quay lại và thử so sánh SUB1 với phần của chuỗi MAINST từ vị trí MAINST+1trở đi:

```
SUB1                 A  B  C
                     +
MAINST            A  B  A  B  C  A
```

Vì có sự khác biệt ngay từ phép so sánh đầu tiên chúng ta lại bắt đầu lại từ đầu tại vị trí MAINST+2

```
SUB1                        A  B  C
                            |  |  |
MAINST              A  B  A  B  C  A
```

Lần này chúng ta đã thành công :SUB1 là chuỗi con của MAINST

Bây giờ hãy thử với chuỗi SUB2. Quá trình tìm kiếm cũng giống như trên cho đến khi chúng ta gặp trường hợp sau :

```
SUB2                           C  A  B
                               |
MAINST              A  B  A  B  C  A
```

Có sự khác nhau trong phép so sánh nhưng ở đây chúng ta không cần làm tiếp vì nếu chúng ta cứ tiếp tục chúng ta sẽ phải so sánh 3 ký tự của chuỗi SUB2 với 2 ký tự còn lại của MAINST. Như vậy SUB2 không phải là chuỗi con của MAINST.

Thực tế chúng ta có thể đoán trước vị trí cuối cùng trong phép tìm kiếm. Đó là:

```
STOP = MAINST + chiều dài  MAINST - chiều dài SUB2
     =      MAINST + 6 - 3 =  MAINST + 3.
```

Dưới đây là thuật toán và chương trình tìm chuỗi con SUBST trong chuỗi MAINST.

Thuật toán tìm chuỗi con

```
Thông báo cho người sử dụng đưa vào chuỗi con SUBST
Đọc SUBST
Thông báo cho người sử dụng đưa vào chuỗi  MAINST
Đọc  MAINST
IF  (chiều dài của MAINST = 0)
        OR (chiều dài của SUBST = 0)OR(SUBST dài
        hơn MAINST)
   THEN
        SUBST không phải là chuỗi con của  MAINST
   ELSE
        Tính vị trí dừng STOP
```

```
            Vị trí bắt đầu START = offset  MAINST
    ENDIF
    REPEAT
        So sánh các ký tự tương ứng trong MAINST bắt đầu
            từ START và SUBST
        IF tất cả các ký tự đều giống nhau
            THEN
                SUBST được tìm thấy trong MAINST
            ELSE
                START = START + 1
    END_IF
    UNTIL (SUBST được tìm thấy trong MAINST)
                OR  (START > STOP)
    Hiển thị kết quả
```

Sau khi đọc vào 2 chuỗi SUBST, MAINST, kiểm tra xem chúng khác rỗng và SUBST không dài hơn MAINST. Tại dòng 44 - 50 chương trình tính vị trí dừng STOP(vị trí trong MAINST để dừng lại việc tìm kiếm), và khởi tạo vị trí bắt đầu START(vị trí bắt đầu tìm kiếm) bằng đầu chuỗi MAINST.

Chương trình Nguồn PGM11_5.ASM

```
1:  TITLE    PGM11_5:Tìm chuỗi con
2:  .MODEL   SMALL
3:  .STACK   100H
4:  .DATA
5:  MSG1     DB    'VAO CHUOI CON SUBST ',0AH,0DH,'$'
6:  MSG2     DB    0AH,0DH,'VAO CHUOI CHINH MAINST '
             DB    0AH,0DH,'$'
7:  MAINST   DB    80 DUP(0)
8:  SUBST    DB    80 DUP(0)
9:  STOP     DW    ?       ;Vị trí cuối cùng để tìm kiếm
10: START    DW    ?       ;Vị trí tiếp tục tìm kiếm
11: SUB_LEN  DW    ?       ;Chiều dài chuỗi con
12: YESMSG   DB    0AH,0DH,'SUBST la chuoi con cua'
             DB    'MAINST$'
13: NOMSG    DB    0AH,0DH,'SUBST khong la chuoi con cua'
             DB    ' MAINST$'
14: .CODE
15: MAIN     PROC
16:          MOV   AX,@DATA
17:          MOV   DS,AX
18:          MOV   ES,AX
19: ;Thông báo cho chuỗi con
```

```
20:          MOV   AH,9          ;Hàm con in một chuỗi ký tự
21:          LEA   DX,MSG1       ;Thông báo vào chuỗi con
22:          INT   21H
23:;Đọc chuỗi SUBST
24:          LEA   DI,SUBST      ;DI trỏ vào chuỗi con
25:          CALL  READ_STR      ;BX chứa chiều dài chuỗi con
26:          MOV   SUB_LEN,BX ;Cất nó trong biến SUB_LEN
27:;Thông báo cho chuỗi chính
28:          LEA   DX,MSG2       ;Thông báo vào chuỗi  MAINST
29:          INT   21H
30:;Đọc chuỗi MAINST
31:          LEA   DI,MAINST  ;DI trỏ tới chuỗi MAINST
32:          CALL  READ_STR ;BX chứa chiều dài chuỗi MAINST
33:;Kiểm tra xem các chuỗi có rỗng hay SUBST có dài
   ;hơn MAINST không?
34:          OR    BX,BX         ;MAINST rỗng?
35:          JE    NO            ;Đúng!SUBST không phải là
                                 ;chuỗi con của  MAINST
36:          CMP   SUB_LEN,0  ;SUBST rỗng?
37:          JE    NO    ;Đúng!SUBST không phải chuỗicon
38:          CMP   SUB_LEN,BX ;SUBST dài hơn MAINST?
39:          JG    NO    ;Đúng!SUBST không phải chuỗicon
40:;Kiểmtra xem SUBST có phải là chuỗi con của MAINST
   ;không
41:          LEA   SI,SUBST    ;SI trỏ tới SUBST
42:          LEA   DI,MAINST   ;DI trỏ tới MAINST
43:          CLD               ;Xử lý từ trái qua phải
44:;Tính vị trí dừng STOP
45:          MOV   STOP,DI     ;STOP chứa địa chỉ MAINST
46:          ADD   STOP,BX     ;cộng thêm chiều dài MAINST
47:          MOV   CX,SUB_LEN
48:          SUB   STOP,CX     ;trừ đi chiều dài SUBST
49:;Khởi tạo START
50:          MOV   START,DI    ;Vị trí đầu tiên để tìm
51:REPEAT:
52:;So sánh các ký tự
53:          MOV   DI,START    ;Định lại DI
55:          LEA   SI,SUBST    ;Định lại SI
56:          REPE  CMPSB       ;So sánh các ký tự
57:          JE    YES         ;Tìm thấy SUBST
58:;Chưa tìm thấy SUBST
59:          INC   START       ;Điều chỉnh lại START
60:;Xem START còn nhỏ hơn STOP hay không
61:          MOV   AX,START
```

```
62:             CMP   AX,STOP      ;START < STOP?
63:             JNLE  NO          ;Không!kết thúc ngay
64:             JMP   REPEAT      ;Tiếp tục
65:;Hiển thị kết quả
66:YES:
67:             LEA   DX,YESMSG    ;DX trỏ tới chuỗi YESMSG
68:             JMP   DISPLAY
69:NO:
70:             LEA   DX,NOMSG
71:DISPLAY:
72:             MOV   AH,9
73:             INT   21H         ;Hiển thị kết quả
74:;Trở về DOS
75:             MOV   AX,4CH
76:             INT   21H
77:MAIN         ENDP
78:;READ_STR đặt ở đây
79:             END   MAIN
```

Tại dòng 51 chương trình đi vào vòng lặp REPEAT trong đó các ký tự của SUBST được so sánh với một phần của MAINST từ vị trí START. Trong các dòng 53-56 CX được thiết lập bằng chiều dài của SUBST, SI trỏ tới SUBST, DI trỏ tới START và các ký tự tương ứng được so sánh bằng lệnh REPE CMPSB. Nếu ZF = 1, thì việc tìm kiếm thành công và chương trình nhảy tới dòng 66 tại đó nó hiển thị thông báo "SUBST LA CHUOI CON CUA MAINST". Nếu ZF = 0 nghĩa là đã có 2 ký tự khác nhau trong quá trình so sánh và START tăng thêm 1 ởdòng 59. Quá trình cứ tiếp tục như vậy cho đến khi SUBST giống một phần của MAINST hay START > STOP. Trong trường hợp sau thông báo "SUBST KHONG LA CHUOI CON CUA MAINST" được hiển thị.

Ví dụ thực hiện:

```
C>PGM11_5
VAO CHUOI CON SUBST
ABC
VAO CHUOI CHINH MAINST
XYZAABC
SUBST LA CHUOI CON CUA MAINST

C>PGM11_5
VAO CHUOI CON SUBST
ABD
VAO CHUOI CHINH MAINST
ABACADACD
SUBST KHONG LA CHUOI CON CUA MAINST
```

11.7 Dạng tổng quát của các lệnh thao tác chuỗi

Chúng ta hãy tổng kết các dạng byte và word của các lệnh thao tác chuỗi :

Lệnh	Toán hạng đích	Toán hạng nguồn	Dạng byte	Dạng word
Chuyển chuỗi	ES:DI	DS:SI	MOVSB	MOVSW
So sánh chuỗi*	ES:DI	DS:SI	CMPSB	CMPSW
Lưu chuỗi	ES:DI	AL hay AX	STOSB	STOSW
Nạp chuỗi	AL hay AX	DS:SI	LODSB	LODSW
Duyệt chuỗi	ES:DI	AL hay AX	SCASB	SCASW

**Kết quả không lưu lại.*

Các toán hạng của các lệnh này là ngầm định bởi vậy bản thân chúng không còn là một phần của lệnh nữa. Tuy nhiên cũng có một số dạng của các lệnh thao tác chuỗi trong đó các toán hạng có mặt một cách rõ ràng. Đó là:

Lệnh	Ví dụ
MOVS chuỗi đích,chuỗi nguồn	MOVSB
CMPS chuỗi đích,chuỗi nguồn	CMPSB
STOS chuỗi đích	STOS STRING2
LODS chuỗi nguồn	LODS STRING1
SCAS chuỗi đích	SCAS STRING2

Khi trình biên dịch gặp các dạng tổng quát này chúng sẽ kiểm tra xem:

1. Chuỗi nguồn có nằm trong đoạn định địa chỉ bởi DS và chuỗi đích có nằm trong đoạn định địa chỉ bởi ES hay không.

2. Trong trường hợp của lệnh MOVS và CMPS nếu các chuỗi có cùng kiểu nghĩa là cùng là chuỗi byte hay chuỗi word. Thì khi đó lệnh sẽ được mã hoá như là dạng byte (chẳng hạn MOVSB) hay dạng word (chẳng hạn MOVSW) tuỳ theo dạng dữ liệu mà các chuỗi được khai báo. Ví dụ DS và ES chứa địa chỉ đoạn của đoạn sau đây:

```
.DATA
STRING1  DB    'ABCDE'
STRING2  DB    'EFGH'
STRING3  DB    'IJKL'
STRING4  DB    'MNOP'
STRING5  DW    1,2,3,4,5
STRING6  DW    7,8,9
```

Thì những cặp lệnh sau đây là tương đương:

```
MOVS    STRING1,STRING2     MOVSB
MOVS    STRING6,STRING5     MOVSW
LODS    STRING4             LODSB
LODS    STRING5             LODSW
SCAS    STRING1             SCASB
STOS    STRING6             STOSW
```

Cần phải lưu ý các bạn một lần nữa là khi sử dụng dạng tổng quát vẫn cần phải đảm bảo cho DS:SI trỏ đến chuỗi nguồn và ES:DI trỏ đến chuỗi đích. Dạng tổng quát của các lệnh thao tác chuỗi có những ưu điểm cũng như những nhược điểm của nó. Ưu điểm của nó là ở chỗ do các toán hạng xuất hiện khá rõ ràng trong lệnh, chương trình trở lên dễ đọc hơn. Nhược điểm của nó là chỉ khi xác định dạng dữ liệu khai báo mới biết được dạng lệnh là byte hay word. Trong thực tế các toán hạng có mặt trong dạng tổng quát của lệnh thao tác chuỗi chưa chắc đã là các toán hạng được sử dụng khi lệnh được thực hiện! Để làm ví dụ các bạn hãy xem đoạn lệnh dưới đây:

```
LEA     SI,STRING1          ;SI trỏ đến STRING1
LEA     DI,STRING2          ;DI trỏ đến STRING2
MOVS    STRING3,STRING4
```

Mặc dù các toán hạng nguồn và đích đã được xác định là STRING3 và STRING4. Nhưng khi lệnh MOVS được thực hiện thì byte đầu tiên của chuỗi STRING1 được chuyển đến byte đầu tiên của chuỗi STRING2 vì trình biên dịch đã dịch lệnh `MOVS STRING3,STRING4` thành mã máy của lệnh MOVSB trong khi SI và DI lại trỏ đến các byte đầu tiên của các chuỗi STRING1 và STRING2 một cách tương ứng.

TỔNG KẾT

Trong chương này chúng ta đã nghiên cứu những vấn đề sau:

- Các lệnh thao tác chuỗi là một nhóm đặc biệt của các lệnh xử lý dữ liệu kiểu mảng. Trạng thái cờ DF xác định chiều của thao tác chuỗi. Nếu DF = 0 thao tác được thực hiện từ trái qua phải trong chuỗi, và ngược lại nếu DF = 1 thao tác được thực hiện từ phải qua trái. Lệnh CLD xoá cờ DF(DF = 0) và lệnh STD thiết lập cờ DF(DF = 1).

- Lệnh MOVSB chuyển một byte tại địa chỉ DS:SI đến Bbyte tại địa chỉ ES:DI và sau đó DI, SI được điều chỉnh tuỳ theo giá trị cờ DF. Lệnh MOVSB có dạng word là MOVSW. Các lệnh này có thể sử dụng với tiền tố REP để lặp lại các thao tác nói trên CX lần.

- REPE và REPNE là các tiền tố điều kiện có thể dùng với các lệnh thao tác chuỗi. Với tiền tố REPE lệnh được lặp lại CX lần nếu ZF vẫn bằng 1. Với *tiền tố REPNE lệnh được lặp lại CX lần nếu ZF vẫn bằng 0. REPZ* và REPNZ cũng có tác dụng giống như REPE và REPNE.

- Lệnh STOSB chuyển nội dung của Al vào byte được định địa chỉ bởi ES:DI và sau đó điều chỉnh DI tuỳ theo trạng thái của cờ DF. STOSB có dạng word là STOSW. STOSB có thể sử dụng để đọc một chuỗi ký tự vào một mảng.

- Lệnh LODSB chuyển vào Al một byte được định địa chỉ bởi DS:SI và sau đó điều chỉnh SI tuỳ theo trạng thái của cờ DF. LODSB có dạng word là LODSW. LODSB có thể sử dụng để kiểm tra nội dung của một chuỗi ký tự.

- Lệnh SCASB trừ nội dung của AL cho byte được chỉ ra bởi ES:DI và sử dụng kết quả để thiết lập các cờ. Kết quả không được lưu lại và DI được điều chỉnh tuỳ theo trạng thái của DF. SCASW là dạng word của lệnh SCASB. Lệnh này trừ nội dung của AX cho word được chỉ ra bởi ES:DI, thiết lập các cờ và điều chỉnh DI tuỳ theo trạng thái của DF. Kết quả cũng không được lưu lại. Các lệnh này có thể sử dụng để tìm một byte hay một *word đích (chứa trong AL hay AX) trong một chuỗi.*

- Lệnh CMPSB trừ byte được chỉ ra bởi DS:SI cho byte được chỉ ra bởi ES:DI, thiết lập các cờ và điều chỉnh cả SI lẫn DI tuỳ theo trạng thái của DF. Kết quả không được lưu lại. CMPSB có dạng word là CMPSW. Các lệnh này có thể sử dụng để so sánh 2 chuỗi theo thứ tự al pha bê, để kiểm tra xem chúng có bằng nhau hoặc chuỗi này có là chuỗi con của chuỗi kia hay không.

- Các lệnh thao tác chuỗi có dạng tổng quát, trong đó các toán hạng được xác định rõ ràng. Trình biên dịch chỉ sử dụng các toán hạng để quyết định mã hoá lệnh theo dạng byte hay word.

Thuật ngữ tiếng Anh

(Memory) string	Chuỗi - Một mảng các byte hay word

Các lệnh mới

CLD	LODSW	SCASW
CMPS	MOVS	STD
CMPSB	MOVSB	STOS
CMPSW	MOVSW	STOSB
LODS	SCAS	STOSW
LODSB	SCASB	

Các lệnh thao tác chuỗi :

REP	REPNE	REPZ
REPE	REPNZ	

Bài tập

1.Giả sử

SI chứa 100h	byte 100h chứa 10h
DI chứa 200h	byte 101h chứa 15h
AX chứa 4142h	byte 200h chứa 20h
DF = 0	byte 201h chứa 25h

Hãy cho biết toán hạng đích, toán hạng nguồn và các giá trị được chuyển trong mỗi lệnh sau đây, đồng thời cho biết giá trị mới của SI và DI.

```
a. MOVSB
b. MOVSW
c. STOSB
d. STOSW
e. LODSB
f. LODSW
```

2. Giả sử có các khai báo sau đây:

```
STRING1  DB    'FGHIJ'
STRING2  DB    'ABCDE'
DB  5 DUP(?)
```

Hãy viết các lệnh chuyển STRING1 vào cuối STRING2 để tạo ra chuỗi "ABCDEFGHIJ".

3. Viết các lệnh đổi chuỗi STRING1 và chuỗi STRING2 trong bài tập 2. Các bạn có thể sử dụng 5 byte sau chuỗi STRING2 làm vùng nhớ trung gian.

4. Chuỗi ASCIIZ là một chuỗi kết thúc bằng byte 0, ví dụ:

```
STR   DB    'THIS IS A  ASCIIZ STRING',0
```

Hãy viết thủ tục LENTH nhận vào địa chỉ của một chuỗi ASCIIZ trong DX và trả về chiều dài của chuỗi trong CX.

5. Hãy sử dụng các chế độ địa chỉ trong chương 10 để viết những lệnh tương đương với các lệnh thao tác chuỗi dưới đây. Giả sử rằng ở những chỗ cần thiết SI đã chứa offset của chuỗi nguồn và DI đã chứa offset của chuỗi đích đồng thời DF = 0. Các bạn có thể dùng AL làm vùng nhớ trung gian. Đối với các lệnh SCASB và CMPSB các cờ phải phản ánh kết quả của phép so sánh.

a. MOVSB
b. STOSB
c. LODSB
d. SCASB
e. CMPSB

6. Giả sử đã khai báo chuỗi sau đây:

```
STRING    DB    'TH*S* G*S* AR* b*ASTS'
```

Hãy viết các lệnh đổi các dấu hoa thị thành các chữ 'E'.

7. Giả sử các chuỗi sau đã được khai báo:

```
STRING1   DB    'T H I S I S A T E S T'
TRING2    DB    11 DUP (?)
```

Viết các lệnh chép chuỗi STRING1 vào chuỗi STRING2 và bỏ đi những khoảng trống.

Các bài tập lập trình

8. Một "Palindrome" là một chuỗi ký tự mà khi đọc ngược hay xuôi đều giống nhau. Khi nói một chuỗi là "Palindrome" chúng ta đã bỏ qua những khoảng trống, các dấu và các chữ hoa. Ví dụ như "Madam, I'm Adam" hay " A man, a plan, Panama!"

Viết một chương trình :

1. Cho phép người sử dụng đưa vào một chuỗi
2. In nó ngược và xuôi không có các dấu và khoảng trống trên dòng tiếp theo.
3. Kiểm tra xem nó có phải là một "Palindrome" hay không và in ra kết luận.

9. Trong một số ứng dụng ..., rất cần phải hiển thị các số căn theo biên phải trong một khoảng nhất định. Ví dụ các số dưới đây được căn theo biên phải trong một khoảng bằng 10 ký tự:

```
12345
  23423345
  123
```

Hãy viết chương trình nhập 10 số có tối đa 10 chữ số và hiển thị chúng như trên đây.

10. Chuỗi STRING1 được gọi là đứng trước chuỗi STRING2 theo thứ tự al pha bê nếu một trong 3 khả năng sau xảy ra:

a. Ký tự đầu tiên của STRING1 đứng trước ký tự đầu tiên của STRING2 theo thứ tự al pha bê

b. N-1 ký tự đầu tiên của 2 chuỗi giống nhau nhưng ký tự thứ N của STRING1 đứng trước ký tự thứ N của STRING theo thứ tự al pha bê.

c. STRING1 giống phần đầu tiên của STRING2 nhưng STRING 2 dài hơn.

Hãy viết chương trình cho phép người sử dụng đưa vào 2 chuỗi trên 2 dòng khác nhau. Kiểm tra xem chuỗi nào đứng trước theo thứ tự al pha bê hay 2 chuỗi có giống nhau không.

11. Hàm con 0Ah của ngắt 21h có thể dùng để nhập một chuỗi ký tự. Ký tự đầu tiên của mảng chứa chuỗi (bộ đệm chuỗi) phải được khởi tạo bằng số ký tự lớn nhất muốn nhập. Sau khi thực hiện hàm byte thứ 2 chứa số ký tự thực sự được nhập vào. Việc nhập kết thúc bằng ký tự CR, ký tự này cũng được lưu nhưng nó không được tính vào số các ký tự được nhập. Nếu người sử dụng đưa vào số ký tự nhiều hơn số ký tự mong đợi máy tính sẽ phát ra các tiếng bip.

    Hãy viết chương trình in ra dấu "?"; đọc chuỗi nhiều nhất là 20 ký tự (sử dụng hàm 0Ah của ngắt 21h) và hiển thị chuỗi nhập được ở dòng tiếp theo. Khởi tạo bộ đệm như sau:

```
STRING    LABEL  BYTE
MAX_LEN   DB     20          ;Số ký tự lớn nhất mong
                             ;muốn
ACT_LEN   DB     ?           ;Số ký tự thực sự nhập
                             ;được
CHARS     DB     21 DUP (?)  ;20 byte cho chuỗi và
                             ;1 byte cho ký tự CR
                             ; Return
```

12. Viết một thủ tục INSERT chèn chuỗi STRING1 vào chuỗi STRING2 tại một vị trí nhất định.

    Vào

    SI chứa địa chỉ offset của STRING1
    DI chứa địa chỉ offset của STRING2
    BX chứa chiều dài của STRING1
    CX chứa chiều dài của STRING2
    AX chứa địa chỉ offsettại đó STRING1 được chèn vào

    Ra

    DI chứa địa chỉ offset của chuỗi mới
    BX chứa chiều dài mới của chuỗi

    Thủ tục có thể giả sử rằng không có chuỗi nào có chiều dài bằng 0 và địa chỉ trong AX nằm trong chuỗi STRING2.

Hãy viết một chương trình vào 2 chuỗi STRING1 và STRING2, một số không âm N (0<= N <=40), chèn chuỗi STRING1 vào chuỗi STRING2 sau N byte tính từ đầu chuỗi. Hiển thị chuỗi nhận được. Các bạn có thể giả thiết rằng N<= chiều dài của STRING2 và chiều dài của mỗi chuỗi nhỏ hơn 40.

13. Viết một thủ tục DELETE xoá N byte từ vị trí xác định trong một chuỗi và bỏ đi các chỗ trống mới xuất hiện(trong bộ nhớ).

Vào

DI chứa địa chỉ offset của chuỗi.
BX chứa chiều dài của chuỗi.
CX chứa số ký tự cần xoá
SI chứa địa chỉ offset mà các byte từ vị trí đó bị xoá đi.

Ra

DI chứa địa chỉ offset của chuỗi mới.
BX chiều dài của chuỗi mới.

Thủ tục có thể giả thiết rằng chuỗi có chiều dài khác 0, số byte cần xoá nhỏ hơn chiều dài của chuỗi, và địa chỉ cho trong SI nằm bên trong chuỗi.

Viết chương trình đọc một chuỗi STRING, số nguyên thập phân S chứa vị trí trong chuỗi STRING và số nguyên thập phân N các byte sẽ bị xoá khỏi chuỗi (cả 2 số đều nằm trong khoảng từ 0 đến 80). Chương trình gọi DELETE để xoá N byte từ vị trí S trong chuỗi STRING và hiển thị chuỗi nhận được. Bạn có thể giả thiết 0 <= N <= L - S, trong đó L là chiều dài của STRING.

2.5 Biểu diễn các ký tự

Bảng 2.5 Bảng mã ASCII

DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR
0	00	<CC>	32	20	SP	64	40	@	96	60	
1	01	<CC>	33	21	!	65	41	A	97	61	a
2	02	<CC>	34	22	"	66	42	B	98	62	b
3	03	<CC>	35	23	#	67	43	C	99	63	c
4	04	<CC>	36	24	$	68	44	D	100	64	d
5	05	<CC>	37	25	%	69	45	E	101	65	e
6	06	<CC>	38	26	&	70	46	F	102	66	f
7	07	<CC>	39	27	'	71	47	G	103	67	g
8	08	<CC>	40	28	(	72	48	H	104	68	h
9	09	<CC>	41	29	)	73	49	I	105	69	i
10	0A	<CC>	42	2A	*	74	4A	J	106	6A	j
11	0B	<CC>	43	2B	+	75	4B	K	107	6B	k
12	0C	<CC>	44	2C	,	76	4C	L	108	6C	l
13	0D	<CC>	45	2D	-	77	4D	M	109	6D	m
14	0E	<CC>	46	2E	.	78	4E	N	110	6E	n
15	0F	<CC>	47	2F	/	79	4F	O	111	6F	o
16	10	<CC>	48	30	0	80	50	P	112	70	p
17	11	<CC>	49	31	1	81	51	Q	113	71	q
18	12	<CC>	50	32	2	82	52	R	114	72	r
19	13	<CC>	51	33	3	83	53	S	115	73	s
20	14	<CC>	52	34	4	84	54	T	116	74	t
21	15	<CC>	53	35	5	85	55	U	117	75	u
22	16	<CC>	54	36	6	86	56	V	118	76	v
23	17	<CC>	55	37	7	87	57	W	119	77	w
24	18	<CC>	56	38	8	88	58	X	120	78	x
25	19	<CC>	57	39	9	89	59	Y	121	79	y
26	1A	<CC>	58	3A	:	90	5A	Z	122	7A	z
27	1B	<CC>	59	3B	;	91	5B	[	123	7B	{
28	1C	<CC>	60	3C	<	92	5C	\	124	7C	|
29	1D	<CC>	61	3D	=	93	5D	]	125	7D	}
30	1E	<CC>	62	3E	>	94	5E	^	126	7E	~
31	1F	<CC>	63	3F	?	95	5F		127	7F	<CC>

Chú thích:

<CC> : ký tự điều khiển(Control Character)

SP : ký tự khoảng trống (blank space)

MỤC LỤC