

1.1.2 Trường toán tử (operation field)

Nói với 1 lệnh trường toán tử chứa ký hiệu (symbol) của mã phép toán (operation code = OPCODE). ASM sẽ chuyển ký hiệu mã phép toán thành mã máy. Thông thường ký hiệu mã phép toán mà ta chọn của phép toán, ví dụ: ADD, SUB, INC, DEC, INT ...

Nói với chế độ của ASM, trường toán tử chứa một opcode giả (pseudo operation code = pseudo-op). ASM không chuyển pseudo-op thành mã máy mà dùng chế độ ASM thực hiện một việc gì đó vì nó tạo ra một thuật ngữ, hình ảnh các biến ...

1.1.3 Trường các toán hạng (operand(s) field)

Trong một lệnh trường toán hạng chứa các số liệu tham gia trong lệnh đó. Một lệnh có thể không có toán hạng, có 1 hoặc 2 toán hạng. Ví dụ:

NOP ; không có toán hạng

INC AX ; 1 toán hạng

ADD WORD1,2 ; 2 toán hạng cộng 2 với nội dung của từ nhớ WORD1

Trong các lệnh 2 toán hạng toán hạng đầu là toán hạng đích (destination operand). Toán hạng đích thông thường là thanh ghi hoặc vị trí nhớ chứa nội dung sẽ lưu trữ kết quả. Toán hạng thứ hai là toán hạng nguồn. Toán hạng nguồn thông thường không bị thay đổi sau khi thực hiện lệnh.

Nói với một chế độ của ASM, trường toán hạng chứa một hoặc nhiều thông tin mà ASM dùng để thực thi chế độ.

1.1.4 Trường chú thích (comment field)

Trường chú thích là một tùy chọn của mệnh đề trong ngôn ngữ ASM. Lập trình viên dùng trường chú thích để thuyết minh về các lệnh. Nhiều máy cần thiết vì ngôn ngữ ASM là ngôn ngữ cấp thấp (low level) vì vậy sẽ rất khó hiểu chương trình nếu không có chú thích một cách này nọ và rõ ràng. Tuy nhiên không nên có chú thích quá với mỗi dòng của chương trình, kể cả những lệnh mà ý nghĩa của nó đã rất rõ ràng rồi:

NOP ; không làm gì cả

Ngôn ngữ ta dùng dấu chấm phẩy (;) để bắt đầu trường chú thích.

ASM cũng cho phép dùng toàn bộ một dòng cho chú thích để tạo một khoảng trống ngăn cách các phần khác nhau của chương trình, ví dụ:

```
;
; khởi tạo các thanh ghi
;
```

MOV AX,0

MOV BX,0

1.2 Các kiểu số liệu trong chương trình hộp ngỗng

CPU chỉ làm việc với các số nhị phân. Vì vậy ASM phải chuyển tất cả các loại số liệu thành số nhị phân. Trong một chương trình hộp ngồ cho phép biểu diễn số liệu dưới dạng nhị phân, thập phân hoặc thập lục phân và thậm chí là cả ký tự nữa.

1.2.1 Các số

Một số nhị phân là một dãy các bit 0 và 1 và 2 phải kết thúc bằng h hoặc H

Một số thập phân là một dãy các chữ số thập phân và kết thúc bởi d hoặc D (có thể không cần)

Một số hex phải bắt đầu bởi 1 chữ số thập phân và phải kết thúc bởi h hoặc H.

Sau đây là các biểu diễn số hộp lẻ và không hộp lẻ trong ASM :

Số	Loại
10111	thập phân
10111b	nhị phân
64223	thập phân
-2183D	thập phân
1B4DH	hex
1B4D	số hex không hộp lẻ
FFFFH	số hex không hộp lẻ
0FFFFH	số hex

1.2.2 Các ký tự

Ký tự và một chuỗi các ký tự phải nằm trong giữa hai dấu ngoặc đơn hoặc hai dấu ngoặc kép. Ví dụ 'A' và "HELLO". Các ký tự nếu nằm ngoài chuyển thành mã ASCII bởi ASM. Do đó trong một chương trình ASM sẽ xem khai báo 'A' và 41h (mã ASCII của A) là giống nhau.

1.3 Các biến (variables)

Trong ASM biến đóng vai trò nhỏ trong ngôn ngữ cấp cao. Mỗi biến có một loại dữ liệu và nó được gán một địa chỉ nhớ sau khi dịch chương trình. Bảng sau đây liệt kê các toán tử giải dụng nên nhớ nghĩa các loại số liệu.

PSEUDO-OP	STANDS FOR
DB	define byte
DW	define word (doublebyte)
DD	define doubleword (2 từ liên tiếp)
DQ	define quadword (4 từ liên tiếp)
DT	define tenbytes (10 bytes liên tiếp)

1.3.1. Biến byte

Chức năng của ASM nên hình nghĩa biến byte có dạng như sau :

NAME	DB	initial_value
------	----	---------------

Ví dụ :

ALPHA DB 4

Chẳng hạn này sẽ giải thích ALPHA cho một byte nội dung trong bộ nhớ mà giải thích bạn nhận được của nó là 4. Nếu giá trị của byte là không xác định thì nó sẽ đưa ra dấu chấm hỏi (?) vào giá trị bạn nhận được. Ví dụ:

BYT DB ?

Nội với biến byte vùng giao trò khai độ ma nôi lờu trờ i n h 128 n h
127 n h i với số coida i u v a 0 n h 255 n h i với số k h o i g d a i u .

1.3.2 Bieñ töö

Chưa đầy đủ ASM nên hình thức mới biến tấu sau :

NAME	DW	initial_value
------	----	---------------

Ví dụ :

WRD DW -2

Cũng có thể dùng dấu ? để thay thế cho biến từ có giá trị không xác định . Vùng giá trị của biến từ là -32768 đến 32767 nối với số có dấu vai 0 đến 56535 nối với số không dấu .

1.3.3 Mảng (arrays)

Trong ASM mỗi mảng là một loại các byte nhỏ hoặc lớn liên tiếp nhau. Ví dụ nếu định nghĩa một mảng 3 byte gọi là B_ARRAY mà giá trị ban đầu của nó là 10h, 20h và 30h chúng ta có thể viết :

B ARRAY DB 10h,20h,30h

B_ARRAY là tên ñööïc gài cho byte ñàu tiên

B_ARRAY+1 là tên của byte cuối cùng

B_ARRAY+2 là tên của byte thứ ba

Neu ASM gain nua chæ offset lai 0200h cho mang B_ARRAY thi noi dung boi nhõu se nhõ sau :

SYMBOL	ADDRESS	CONTENTS
B_ARRAY	200h	10h
B_ARRAY+1	201h	20h
B_ARRAY+2	202h	30h

Chẳng hạn sau này sẽ hình thành một mảng 4 phần tử còn lại `W_ARRAY`:

```
W_ARRAY    DW    1000,40,29887,329
```

Giaisöumaing bat ñau taii 0300h thì boanhôiseinhö sau:

SYMBOL	ADDRESS	CONTENTS
W_ARRAY	300h	1000d
W_ARRAY+2	302h	40d
W_ARRAY+4	304h	29887d
W_ARRAY+6	306h	329d

Byte thấp và byte cao của một từ

Nói khi chúng ta cần truy xuất tới byte thấp và byte cao của một biến t0. Giải thích chúng ta nhìn nghĩa :

WORD1 DW 1234h

Byte thấp của WORD1 chứa 34h , còn byte cao của WORD1 chứa 12h

Kyihieü ñòa chà cuìa byte thap la WORD1 con kyihieü ñòa chà cuìa byte cao la WORD1+1 .

Chuoà cáì kyütöi (character strings)

Mỗi mảng các mã ASCII có thể hiểu như hình ảnh của một chuỗi các ký tự.

Ví dụ :

LETTERS DW 41h,42h,43h

tông ñông vôi

LETTERS DW 'ABC'

Bein trong một chuỗi , ASM sẽ phân biệt chữ hoa và chữ thường . Vì vậy chuỗi 'abc' sẽ ñược chuyển thành 3 bytes : 61h ,62h và 63h.

Trong ASM cũng có thể tích hợp các ký hiệu và các số trong một hình ảnh. Ví dụ:

MSG DB 'HELLO', 0AH, 0DH, '\$'

tông ñông vôi

MSG DB 48H,45H,4CH,4Ch,4FH,0AH,0DH,24H

1.4 Các hằng (constants)

Trong một chương trình các hàng có thể được đặt tên như các danh EQU (equates) . Cuối cùng của EQU là:

NAME EQU constant

ví duň :

LF EQU 0AH

sau khi coilkhai báo trên thì LF hoặc dùng thay cho 0Ah trong chương trình . Vì vậy ASM sẽchuyển các lệnh :

MOV DL,0Ah

```
va0    MOV    DL,LF
```

thành cung một mai mai .

Cũng có thể dùng EQU để định nghĩa một chuỗi , ví dụ:

PROMPT EQU 'TYPE YOUR NAME '

Sau khi khai báo này , thay cho

MSG DB 'TYPE YOUR NAME '

chúng ta có thể viết

MSG DB PROMPT

1.5 Các lệnh cơ bản

CPU 8086 có hàng trăm lệnh , trong chương này , chúng ta sẽ xem xét 7 lệnh nền tảng của 8086 mà chúng thông dụng với các thao tác di chuyển số liệu và thực hiện các phép toán số học .

Trong phần sau này , WORD1 và WORD2 là các biến từ , BYTE1 và BYTE2 là các biến byte .

1.5.1 Lệnh MOV và XCHG

Lệnh MOV dùng để chuyển số liệu giữa các thanh ghi , giữa 1 thanh ghi và một vị trí nhớ hoặc để chuyển trực tiếp một số đến một thanh ghi hoặc một vị trí nhớ. Cú pháp của lệnh MOV là:

MOV Destination , Source

Sau đây là vài ví dụ:

MOV AX,WORD1 ; lấy nội dung của từ nhớ WORD1 đưa vào thanh ghi AX

MOV AX,BX ; AX lấy nội dung của BX , BX không thay nội

MOV AH,'A' ; AX lấy giá trị 41h

Bảng sau cho thấy các trường hợp cho phép hoặc cấm của lệnh MOV

Destination operand

source operand	General Reg	Segment Reg	Memory Location	Constant
General Reg	Y	Y	Y	NO
Segment Reg	Y	NO	Y	NO
Memory Location	Y	Y	NO	NO
Constant	Y	NO	Y	NO

Lệnh XCHG (Exchange) dùng để trao đổi nội dung của 2 thanh ghi hoặc của một thanh ghi và một vị trí nhớ. Ví dụ: XCHG AH,BL

XCHG AX,WORD1 ; trao đổi nội dung của thanh ghi AX và từ nhớ WORD1.

Cũng nhớ lệnh MOV có một số hạn chế đối với lệnh XCHG nhớ bảng sau :

Destination operand

Source operand	General Register	Memory Locatin
General Memory	Y	Y
Memory Location	Y	No

1.5.2 Lệnh ADD, SUB, INC , DEC

Lệnh ADD và SUB nhữc dung nhữc cõng và trõ nhữc dung của 2 thanh ghi , của một thanh ghi và một vò trí nhữc, hoặc cõng (trõ) một số vớ (khõ) một thanh ghi hoặc một vò trí nhữc. Cũipháp lã:

ADD Destination , Source

SUB Destination , Source

Ví dũ:

ADD WORD1, AX

ADD BL , 5

SUB AX,DX ; AX=AX-DX

Vì lýdo kỹthuã , lệnh ADD vàSUB cũng bõ một sốhãn cheánhõ bãng sau:

Destination operand

Source operand	General Reg	Memory Loacation
Gen Memory	Y	Y
Memory Location	Y	NO
Constant	Y	Y

Viẽc cõng hoặc trõtrõc tiếp giữa 2 vò trí nhữc làkhõng nhữc phép . Nhữc giã quyết vãn nhữc này ngõõ ta phải di chuyẽn byte (tõ) nhữc nhữc một thanh ghi sau nhữc mõi cõng hoặc trõthanh ghi này vớ một byte (tõ) nhữc khác . Ví dũ:

MOV AL, BYTE2

ADD BYTE1, AL

Lệnh INC (incremrent) nhữc cõng thêm 1 vào nhữc dung của một thanh ghi hoặc một vò trí nhữc. Lệnh DEC (decrement) nhữc giãm bõ 1 khõ một thanh ghi hoặc 1 vò trí nhữc. Cũipháp của chũng lã:

INC Destination

DEC Destination

Ví dũ:

INC WORD1

INC AX

DEC BL

1.5.3 Lệnh NEG (negative)

Lệnh NEG sẽ đổi dấu (lấy bù 2) của một thanh ghi hoặc một vị trí nhớ. Cú pháp :

NEG destination

Ví dụ : NEG AX ;

Giả sử AX=0002h sau khi thực hiện lệnh NEG AX thì AX=FFFEh

LƯU Ý: 2 toán hạng trong các lệnh trên này phải cùng loại (cùng là byte hoặc từ)

1.6 Chuyển ngôn ngữ cấp cao thành ngôn ngữ ASM

Giả sử A và B là 2 biến từ.

Chúng ta sẽ chuyển các mệnh đề sau trong ngôn ngữ cấp cao ra ngôn ngữ ASM .

1.6.1 Mệnh đề B=A

MOV AX,A ; gán A vào AX

MOV B,AX ; gán AX vào B

1.6.2 Mệnh đề A=5-A

MOV AX,5 ; gán 5 vào AX

SUB AX,A ; AX=5-A

MOV A,AX ; A=5-A

cách khác :

NEG A ; A=-A

ADD A,5 ; A=5-A

1.6.3 Mệnh đề A=B-2*A

MOV AX,B ; AX=B

SUB AX,A ; AX=B-A

SUB AX,A ; AX=B-2*A

MOV A,AX ; A=B-2*A

1.7 Cấu trúc của một chương trình hợp ngữ

Một chương trình ngôn ngữ máy bao gồm mã (code) , số liệu (data) và ngăn xếp (stack) . Mỗi một phần chiếm một đoạn nhớ. Mỗi một đoạn chương trình lại được chuyển thành một đoạn nhớ bởi ASM .

1.7.1 Các kiểu nhớ (memory models)

Nhiệm vụ của mã và số liệu trong một chương trình được quy định bởi chế độ MODEL nhằm xác định kiểu nhớ dùng với chương trình . Cú pháp của chế độ MODEL như sau :

.MODEL memory_model

Bảng sau cho thấy các kiểu nhớ:

MODEL	DESCRIPTION
SMALL	code và data nằm trong 1 đoạn
MEDIUM	code nhiều hơn 1 đoạn , data trong 1 đoạn
COMPACT	data nhiều hơn 1 đoạn , code trong 1 đoạn
LARGE	code và data lớn hơn 1 đoạn , array không quá 64KB
HUGE	code ,data lớn hơn 1 đoạn , array lớn hơn 64KB

1.7.2 Đoạn số liệu

Đoạn số liệu của chương trình chứa các khai báo biến , khai báo hằng ... Nếu bắt đầu đoạn số liệu chúng ta dùng chữ dẫn DATA với cú pháp như sau :

.DATA

; khai báo tên các biến , hằng và mảng

ví dụ :

.DATA

WORD1 DW 2

WORD2 DW 5

MSG DB 'THIS IS A MESSAGE '

MASK EQU 10010010B

1.7.3 Đoạn ngăn xếp

Mục đích của việc khai báo đoạn ngăn xếp là dành một vùng nhớ (vùng stack) để lưu trữ cho stack . Cú pháp của lệnh như sau :

.STACK size

nếu không khai báo size thì 1KB được dành cho vùng stack .

.STACK 100h ; dành 256 bytes cho vùng stack

1.7.4 Đoạn mã

Đoạn mã chứa các lệnh của chương trình . Bắt đầu đoạn mã bằng chữ dẫn CODE như sau :

.CODE

Trong đoạn mã các lệnh thông thường có thể trở thành thủ tục (procedure) mà cấu trúc của một thủ tục như sau :

name PROC

; body of the procedure

name ENDP

Sau đây là cấu trúc của một chương trình hộp ngổ mà phần CODE là thủ tục có tên là MAIN

```

.MODEL      SMALL
.STACK      100h
.DATA
; ñhnh nghóa số liệu tại ñây
.CODE
MAIN        PROC
;thủ của thủ tục MAIN
MAIN        ENDP
; các thủ tục khác nếu có
END         MAIN

```

1.8 Các lệnh vào ra

CPU thông tin với các ngoại vi thông qua các cổng IO . Lệnh IN và OUT của CPU cho phép truy xuất đến các cổng này . Tuy nhiên hầu hết các ứng dụng không dùng lệnh IN và OUT vì 2 lý do:

- các nhà chế tạo thay đổi tùy theo loại máy tính
- có thể lập trình cho các IO đang tồn tại các chương trình con (routine) để cung cấp bởi các hãng chế tạo máy tính

Có 2 loại chương trình phục vụ IO là: các routine của BIOS (Basic Input Output System) và các routine của DOS .

Lệnh INT (interrupt)

Để gọi các chương trình con của BIOS và DOS có thể dùng lệnh INT với cú pháp như sau :

```
INT    interrupt_number
```

ở đây interrupt_number là một số mà nó chỉ định một routine . Ví dụ INT 16h gọi routine thực hiện việc nhập số liệu từ Keyboard .

1.8.1 Lệnh INT 21h

INT 21h được dùng để gọi một số lớn các hàm (function) của DOS . Tùy theo giá trị mà chúng ta đặt vào thanh ghi AH , INT 21h sẽ gọi chạy một routine tương ứng .

Trong phần này chúng ta sẽ quan tâm đến 2 hàm sau đây :

FUNCTION NUMBER	ROUTINE
1	Single key input
2	Single character output

FUNCTION 1 : Single key input

Input : AH=1

Output:AL= ASCII code if character key is pressed

AL=0 if non character key is pressed

Nếu gọi routine này thực hiện các lệnh sau :

MOV AH,1 ; input key function

INT 21h ; ASCII code in AL and display character on the screen

FUNCTION 2 : Display a character or execute a control function

Input : AH=2

DL=ASCII code of the the display character or control

character

Output:AL= ASCII code of the the display character or control

character

Các lệnh sau sẽ in lên màn hình dấu ?

MOV AH,2

MOV DL,'?' ; character is '?'

INT 21H ; display character

Hàm 2 cũng có thể dùng để thực hiện chức năng như khi in. Nếu DL chứa ký tự
tối thiểu khi in thì khi gọi INT 21h, ký tự tối thiểu khi in sẽ được thực hiện.

Các ký tự tối thiểu khi in thông dụng lại:

ASCII code (Hex)	SYMBOL	FUNCTION
7	BEL	beep
8	BS	backspace
9	HT	tab
A	LF	line feed
D	CR	carriage return

1.9 Chương trình mẫu tiếp

Chúng ta sẽ viết một chương trình hộp ngỏ nhận một ký tự từ bàn phím
và in nó trên màn hình ngay.

TITLE PGM1: ECHO PROGRAM

```

.MODEL    SMALL
.STACK    100H
.CODE
MAIN      PROC
; display dấu nhac
        MOV  AH,2
        MOV  DL,'?'
        INT  21H
; nhap 1 kyitoi
        MOV  AH,1  ; ham noi kyitoi
        INT  21H    ; kyitoi noi noi vao AL
        MOV  BL,AL  ; cat kyitoi trong BL
; nhay nen dong moi
        MOV  AH,2  ; ham xuat 1 kyitoi
        MOV  DL,0DH ; kyitoi carriage return
        INT  21H    ; thoc hien carriage return
        MOV  DL,0AH ; kyitoi line feed
        INT  21H    ; thoc hien line feed
; xuat kyitoi
        MOV  DL,BL ; noi kyitoi vao DL
        INT  21H    ; xuat kyitoi
; troi ve DOS
        MOV  AH,4CH ; ham thoat ve DOS
        INT  21H    ; exit to DOS
MAIN ENDP
        END  MAIN

```

1.10 Tao ra va chay mot chong trinh hop ngoi

Coi 4 bococ nea tao ra va chay mot chong trinh hop ngoi la:

- Dung mot trinh soan thao van ban nea tao ra tap tin chong trinh nguon (source program file) .
- Dung mot trinh bien dich (Assembler) nea tao ra tap tin noi toong (object file) ngoi ngoi mai
- Dung trinh LINK nea lien ket mot hoac nhieu tap tin noi toong roi tao ra file thoc thi noioc .
- Cho thoc hien tap tin EXE hoac COM .

Bococ 1 : Tao ra chong trinh nguon

Dùng một trình soạn thảo văn bản (NC chẳng hạn) để tạo ra chương trình nguồn. Ví dụ đặt tên là PGM1.ASM. Phần môi trường ASM là phần môi trường quy định để Assembler nhận ra chương trình nguồn.

Bước 2 : Biên dịch chương trình

Chúng ta sẽ dùng MASM (Microsoft Macro Assembler) để chuyển tập tin nguồn PGM1.ASM thành tập tin nối tổng nguồn gọi là PGM1.OBJ bằng lệnh sau :

MASM PGM1;

Sau khi in thông tin về bản quyền MASM sẽ kiểm tra file nguồn để tìm lỗi cú pháp. Nếu có lỗi thì MASM sẽ in ra số dòng bị lỗi và một mô tả ngắn về lỗi đó. Nếu không có lỗi thì MASM sẽ chuyển PGM1.ASM thành tập tin nối tổng nguồn gọi là PGM1.OBJ.

Đầu chấm phải sau lệnh MASM PGM1 có nghĩa là chúng ta không muốn tạo ra một tập tin nối tổng có tên khác với PGM1. Nếu không có đầu chấm phải sau lệnh thì MASM sẽ yêu cầu chúng ta gõ vào tên của một số tập tin mà nó có thể tạo ra như hình dưới đây :

Object file name [PGM1.OBJ]:

Source listing [NUL.LIST] : **PGM1**

Cross-reference [NUL.CRF] : **PGM1**

Tên mặc định là NUL có nghĩa là không tạo ra file tổng ở dạng trống khi lập trình viên gõ vào tên tập tin.

Tập tin danh sách nguồn (source listing file) : là một tập tin Text có thể nhìn thấy được, trong đó mã lệnh nguồn và mã nguồn nằm cạnh nhau. Tập tin này thông dụng để gỡ rối chương trình nguồn vì MASM thông báo lỗi theo số dòng.

Tập tin tham chiếu chéo (Cross -Reference File) : là 1 tập tin chứa danh sách các tên mà chúng xuất hiện trong chương trình kèm theo số dòng mà tên ấy xuất hiện. Tập tin này có thể dùng để tìm các biến và nhãn trong một chương trình lớn.

Bước 3 : Liên kết chương trình

Tập tin nối tổng tạo ra ở bước 2 là một tập tin nguồn gọi mà không thể chạy được vì chưa có dạng thích hợp của 1 file chạy. Hơn nữa nó chưa biết chương trình nào nạp vào và trí nhớ trên board như thế nào. Một số nhà chế tạo đã đưa mã mà có thể bỏ thiếu.

Trình LINK sẽ liên kết một hoặc nhiều file nối tổng thành một file chạy duy nhất (*.EXE). Tập tin này có thể nạp vào board và thi hành.

Nội dung kết chương trình ta gọi:

LINK PGM1;

Nếu không có dấu chấm phẩy ASM sẽ yêu cầu chúng ta gõ vào tên tập tin thối thì .

Bước 4 : Chạy chương trình

Tôi đã nhấn lệnh để chạy chương trình bằng cách gõ tên rồi nhấn ENTER .

1.11 Xuất một chuỗi ký tự

Trong chương trình PGM1 trên này chúng ta đã dùng INT 21H hàm 2 và 4 để nhập và xuất một ký tự . Hàm 9 ngắt 21H để dùng để xuất một chuỗi ký tự .

INT 21H , Function 9 : Display a string

Input : DX=offset address of string

The string must end with a '\$' character

Ký tự \$ ở cuối chuỗi sẽ không được in lên màn hình . Nếu chuỗi có chứa ký tự nhiều khi thì chờ đợi nhiều khi thì tổng cộng sẽ được thực hiện .

Chúng ta sẽ viết 1 chương trình in lên màn hình chuỗi "HELLO!" . Thông điệp HELLO được định nghĩa như sau trong đoạn số liệu :

MSG DB 'HELLO!\$'

Lệnh LEA (Load Effective Address)

LEA destination , source

Ngắt 21h , hàm số 9 sẽ xuất một chuỗi ký tự ra màn hình với nhiều kiến thức về hiệu dụng của biến chuỗi phải ở trên DX . Có thể thực hiện như này bởi lệnh :

LEA DX,MSG ; đưa địa chỉ offset của biến MSG vào DX

Program Segment Prefix (PSP) : Phần đầu của đoạn chương trình

Khi một chương trình được nạp vào bộ nhớ máy tính , DOS dành ra 256 byte cho cái gọi là PSP . PSP chứa một số thông tin về chương trình đang được nạp trong bộ nhớ . Để cho các chương trình có thể truy xuất tới PSP , DOS đặt số phần nhớ của nó (PSP) trong cái DS và ES trước khi thực thi chương trình . Kết quả là thanh ghi DS không chứa số nhớ của đoạn số liệu của chương trình . Để khắc phục điều này , một chương trình có chứa đoạn số liệu phải được bắt đầu bởi 2 lệnh sau này :

MOV AX,@DATA

```
MOV DS,AX
```

Ồ này @DATA là tên của đoạn số liệu nhớ dành cho DATA .
Assembler sẽ chuyển @DATA thành số đoạn .

Sau này là chương trình hoàn chỉnh sẽ xuất chuỗi ký tự HELLO!

```
TITLE      PGM2: PRINT STRING PROGRAM
.MODEL     SMALL
.STACK     100H
.DATA
MSG DB     'HELLO!$'
.CODE
MAIN       PROC
; initialize DS
        MOV AX,@DATA
        MOV DS,AX
; display message
        LEA DX,MSG
        MOV AH,9
        INT 21H
; return to DOS
        MOV AH,4CH
        INT 21H
MAIN       ENDP
END MAIN
```

1.12 Chương trình nổi chữ hoa sang chữ hoa

Chúng ta sẽ viết 1 chương trình yêu cầu người dùng gõ vào một ký tự bất kỳ. Chương trình sẽ nổi nó sang dạng chữ hoa rồi in ra ôi dòng tiếp theo .

```
TITLE      PGM3: CASE COVERT PROGRAM
.MODEL     SMALL
.STACK     100H
.DATA
        CR EQU 0DH
        LF EQU 0AH
MSG1 DB     'ENTER A LOWER CASE LETTER:$'
MSG2 DB     0DH,0AH,'IN UPPER CASE IT IS :'
```

```
CHAR      DB    '?,$' ; ñhnh nghĩa biến CHAR cõigiatrò ban ñầu chõa  
                                ;xác ñhnh  
.  
.CODE  
MAIN      PROC  
; INITIALIZE      DS  
      MOV      AX,@DATA  
      MOV      DS,AX  
;PRINT PROMPT USER  
      LEA      DX,MSG1 ; lấy thông ñiệp số1  
      MOV      AH,9  
      INT      21H      ; xuất ñõra màn hình  
;nhập vào một kýitõithông và ñõa ñõithành kýitõihoa  
      MOV      AH,1      ; nhập vào 1 kýitõ  
      INT      21H      ; cấ ñõitrõng AL  
      SUB      AL,20H      ; ñõa ñõithành chõõhoa và cấ ñõitrõng AL  
      MOV      CHAR, AL ; cấ kýitõitrõng biến CHAR  
; xuất kýitõitrõn dòng tiếp theo  
      LEA      DX,MSG2 ; lấy thông ñiệp số2  
      MOV      AH,9  
      INT      21H      ; xuất chuỗi kýitõithõihai , vì MSG2 không kết  
;thức bởi kýitõ $ ñõn ñõitiếp tục xuất kýitõitrõn biến CHAR  
;dos exit  
      MOV      AH,4CH  
      INT      21H      ; dos exit  
MAIN      ENDP  
      END      MAIN
```

Chương 2 : Trạng thái của vi xử lý và các thanh ghi cơ

Trong chương này chúng ta sẽ xem xét các thanh ghi cơ của vi xử lý và ảnh hưởng của các lệnh máy đến các thanh ghi cơ như thế nào . Trạng thái của các thanh ghi là căn cứ để chương trình có thể thực hiện lệnh nhảy , rẽ nhánh và lặp .

Một phần của chương này sẽ giới thiệu chương trình DEBUG của DOS .

2.1 Các thanh ghi cơ(Flags register)

Nhiệm vụ khác biệt quan trọng của máy tính so với các thiết bị điện tử khác là khả năng cho các quyết định . Một mạch cần biết trong CPU có thể làm các quyết định này bằng cách căn cứ vào trạng thái hiện hành của CPU . Có một thanh ghi cần biết cho biết trạng thái của CPU đó là thanh ghi cơ.

Bảng 2.1 cho thấy thanh ghi cơ 16 bit của 8086

11	10	9	8	7	6	5	4	3	2	1	0
O	D	IF	T	S	Z		A		P		C
F	F		F	F	F		F		F		F

Bảng 2.1 : Thanh ghi cơ của 8086

Mức ních của các thanh ghi của hệ thống ra trạng thái của CPU. Có hai loại của hệ thống trạng thái (status flags) và của hệ thống điều khiển (control flags). Hệ thống trạng thái phản ánh các kết quả thực hiện lệnh của CPU. Bảng 2.2 cho ra tên và ký hiệu các thanh ghi của hệ thống 8086.

Bit	Name	Symbol
0	Carry flag	CF
2	Parity flag	PF
4	Auxiliary carry flag	AF
6	Zero flag	ZF
7	Sign flag	SF
11	Overflow flag	OF
8	Trap flag	TF
9	Interrupt flag	IF
10	Direction flag	DF

Bảng 2.2 : Các của hệ thống 8086

Mỗi bit trên thanh ghi của hệ thống phản ánh 1 trạng thái của CPU.

Các của hệ thống trạng thái (status flags)

Các cờ trạng thái phản ánh kết quả của các phép toán. Ví dụ sau khi thực hiện lệnh SUB AX,AX cờ ZF = 1, nghĩa là kết quả của phép trừ là zero.

Cờ nhôi (Carry Flag - CF) : CF=1 nếu xuất hiện bit nhôi (carry) từ vị trí MSB trong khi thực hiện phép cộng hoặc có bit mượn (borrow) tại MSB trong khi thực hiện phép trừ. Trong các trường hợp khác CF=0. Cờ CF cũng bị ảnh hưởng bởi lệnh dịch (Shift) và quay (Rotate) số liệu.

Cờ chẵn lẻ (Parity Flag - PF) : PF=1 nếu byte thập của kết quả có tổng số con số 1 là một số chẵn (even parity). PF=0 nếu byte thập là chẵn lẻ (old parity). Ví dụ nếu kết quả là FF FEh thì PF=0

Cờ nhôi phụ (Auxiliary Carry Flag - AF) : AF = 1 nếu có nhôi (mượn) từ bit thứ 3 trong phép cộng (trừ).

Cờ Zero (Zero Flag - ZF) : ZF=1 nếu kết quả là số 0.

Cờ dấu (Sign Flag - SF) : SF=1 nếu MSB của kết quả là 1 (kết quả là số âm). SF=0 nếu MSB=0

Cờ tràn (Overflow Flag - OF) : OF=1 nếu xảy ra tràn số trong khi thực hiện các phép toán. Sau này chúng ta sẽ phân tích các trường hợp xảy ra tràn trong khi thực hiện tính toán. Hiện tổng tràn số liên quan đến việc biểu diễn số trong máy tính với một số hữu hạn các bit. Các số thập phân có dấu biểu diễn bởi 1 byte là -128 đến +127. Nếu biểu diễn bằng 1 từ (16 bit) thì các số thập phân có thể biểu diễn là -32768 đến +32767. Nói với các số không dấu, dải các số có thể biểu diễn trong

một từ là 0 đến 65535, trong một byte là 0 đến 255. Nếu kết quả của một phép toán vượt ra ngoài dải số có thể biểu diễn thì xảy ra tràn số. Khi có tràn số kết quả thu được sẽ sai.

2.2 Tràn (overflow)

Có 2 loại tràn số: Tràn có dấu (signed overflow) và tràn không dấu (unsigned overflow). Khi thực hiện phép cộng số hoặc phép trừ, sẽ xảy ra 4 khả năng sau đây:

- 1) không tràn
- 2) tràn có dấu
- 3) tràn không dấu
- 4) tràn có dấu và tràn không dấu

Ví dụ của tràn không dấu là phép cộng ADD AX, BX với AX=0FFFFh, BX=0001h. Kết quả двої dạng như sau:

$$\begin{array}{r}
 1111\ 1111\ 1111\ 1111 \\
 0000\ 0000\ 0000\ 0001 \\
 \hline
 10000\ 0000\ 0000\ 0000
 \end{array}$$

Nếu diễn giải kết quả двої dạng không dấu thì kết quả là 10000h (=65536). Những kết quả này vượt quá giới hạn của từ. Bit 1 (bit nhớ từ và trí

MSB) nữa xảy ra và kết quả trên AX = 0000h là sai. Số tràn nhỏ thế là tràn không dấu. Nếu xem rằng phép cộng trên này là phép cộng hai số có dấu thì kết quả trên AX = 0000h là đúng, vì FFFFh = -1, còn 0001h = +1, do đó kết quả phép cộng là 0. Vậy trong trường hợp này số tràn dấu không xảy ra.

Ví dụ về số tràn dấu : giả sử AX = BX = 7FFFh, lệnh ADD AX, BX sẽ cho kết quả nhỏ sau :

```

0111 1111 1111 1111
0111 1111 1111 1111

```

1111 1111 1111 1110 = FFFE h

Biểu diễn có dấu và không dấu của 7FFFh là 32767₁₀. Nhỏ vậy là nhờ với phép cộng có dấu cũng nhờ không dấu thì kết quả vẫn là 32767 + 32767 = 65534. Số này (65534) đã vượt ngoài dải giá trị mà 1 số 16 bit có dấu có thể biểu diễn. Hơn nữa FFFEh = -2. Do vậy số tràn dấu đã xảy ra.

Trong trường hợp xảy ra tràn, CPU sẽ biểu thị số tràn nhỏ sau :

- CPU sẽ set OF = 1 nếu xảy ra tràn dấu
- CPU sẽ set CF = 1 nếu xảy ra tràn không dấu

Sau khi có tràn, một chương trình hợp lý sẽ nhờ thực hiện nếu số sai kết quả ngay lập tức. Các lập trình viên sẽ cần phải quan tâm tới cờ OF hoặc CF nếu biểu

diện số của hai số có dấu hay không dấu một cách tổng
 ồng .

Vậy thì làm thế nào để CPU biết nó có tràn ?

- Tràn không dấu sẽ xảy ra khi có một bit nhồi (hoặc mượn) từ MSB
- Tràn dấu sẽ xảy ra trong các trường hợp sau :

a) Khi cộng hai số cùng dấu , sẽ tràn dấu xảy ra khi tổng có dấu khác với hai toán hạng ban đầu . Trong ví dụ 2 , cộng hai số 7FFFh + 7FFFh (hai số dương) nhưng kết quả là FFFFh (số âm)

b) Khi trừ hai số khác dấu (giống như cộng hai số cùng dấu) kết quả phải có dấu hợp lý . Nếu kết quả cho dấu không như mong muốn thì có nghĩa là đã xảy ra tràn dấu . Ví dụ $8000h - 0001h = 7FFFh$ (số dương) . Do đó $OF=1$.

Vậy làm thế nào để CPU nhận ra rằng có tràn ?

- $OF=1$ nếu tràn dấu
- $CF=1$ nếu tràn không dấu

Làm thế nào để CPU biết là có tràn ?

- Tràn không dấu xảy ra khi có số nhồi (carry) hoặc mượn (borrow) từ MSB
- Tràn dấu xảy ra khi cộng hai số cùng dấu (hoặc trừ 2 số khác dấu) mà kết quả với dấu khác với dấu mong muốn . Phép cộng hai số có dấu khác nhau không thể xảy ra tràn . Trên thực tế CPU dùng phương pháp sau : đặt $OF=1$ nếu số nhồi vào và số nhồi ra từ MSB là không phù hợp :

ngĩa là còn hữu vào những không còn hữu ra hoặc còn hữu ra những không còn hữu vào .

Cờ hiệu khiển (control flags)

Có 3 cờ hiệu khiển trong CPU , đó là:

- Cờ hướng (Direction Flag = DF)
- Cờ bẫy (Trap flag = TF)
- Cờ ngắt (Interrupt Flag = IF)

Các cờ hiệu khiển nội dung của cờ hiệu khiển hoạt động của CPU

Cờ hướng (DF) nội dung trong các lệnh xử lý chuỗi của CPU . Mục đích của DF là dùng để cờ hiệu khiển hướng của một chuỗi nội dung xử lý. Trong các lệnh xử lý chuỗi hai thanh ghi DI và SI nội dung để đưa ra địa chỉ bắt đầu chuỗi . Nếu DF=0 thì lệnh xử lý chuỗi sẽ tăng địa chỉ bắt đầu sao cho chuỗi nội dung xử lý từ trái sang phải Nếu DF=1 thì địa chỉ bắt đầu sẽ nội dung xử lý theo hướng từ phải sang trái .

2.3 Các lệnh ảnh hưởng đến cờ hiệu

Tại một thời điểm , CPU thực hiện 1 lệnh , các cờ lần lượt phản ánh kết quả thực hiện lệnh . Đó nên có một số lệnh không làm thay đổi một cờ nào cả hoặc thay đổi chỉ 1 vài cờ hoặc làm cho một vài cờ có trạng thái

không xác định . Trong phần này chúng ta sẽ xét ảnh hưởng của các lệnh (những lệnh cơ bản) lên các cờ nhô thế nào .

Bảng sau này cho thấy ảnh hưởng của các lệnh lên các cờ:

INSTRUCTION	AFFECTS FLAGS
MOV/XCHG	NONE
ADD/SUB	ALL
INC/DEC	ALL trừ CF
NEG	ALL
	(CF=1 trừ khi kết quả bằng 0 , OF=1 nếu kết quả là 8000H)

Nếu thấy rõ ảnh hưởng của các lệnh lên các cờ chúng ta sẽ lấy vài ví dụ .

Ví dụ 1 : ADD AX,AX trong nhò AX=BX=FFFFh

$$\begin{array}{r} \text{FFFFh} \\ + \quad \text{FFFFh} \\ \hline \end{array}$$

1FFFEh

Kết quả chò trên AX là FFFEh = 1111 1111 1111 1110

SF=1 vì MSB=1

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

ZF=0 vì kết quả khác 0

CF=1 vì có 1 carry từ MSB

OF=0 vì dấu của kết quả giống với dấu của 2 số hạng ban đầu .

Ví dụ 2 : ADD AL,BL trong đó AL= BL= 80h

$$\begin{array}{r} 80h \\ + 80h \\ \hline 100h \end{array}$$

Kết quả trên AL = 00h

SF=0 vì MSB=0

PF=1 vì tất cả các bit đều bằng 0

ZF=1 vì kết quả bằng 0

CF=1 vì có 1 carry từ MSB

OF=1 vì có 2 toàn hàng là số âm nhưng kết quả là số dương (có 1 carry từ MSB nhưng không có carry vào) .

Ví dụ 3 : SUB AX,BX trong đó AX=8000h và BX=0001h

$$\begin{array}{r} 8000h \\ - 0001h \end{array}$$

$7FFFFh = 0111\ 1111\ 1111\ 1111$

SF=0 vì MSB=0

PF=1 vì có 8 (chẵn) số 1 trong byte thấp của kết quả

ZF=0 vì kết quả khác 0

CF=0 vì không có overflow

OF=1 vì trừ một số âm cho 1 số dương (tức là cộng 2 số âm) mà kết quả lại một số dương .

Ví dụ 4 : INC AL trong ñiều kiện AL=FFh

Kết quả trên AL=00h = 0000 0000

SF=0 vì MSB=0

PF=1

ZF=1 vì kết quả bằng 0

CF không bị ảnh hưởng bởi lệnh INC mặc dù có ñiều kiện 1 tới MSB

OF=0 vì hai số khác dấu ñược cộng với nhau (có số ñưa vào MSB và cũng có số ñưa ra từ MSB)

Ví dụ 5: MOV AX,-5

Kết quả trên BX = -5 = FFFBh

Không có ảnh hưởng bởi lệnh MOV

Ví dụ 6: NEG AX trong ñoù AX=8000h

$$\begin{array}{r}
 8000h = 1000\ 0000\ 0000\ 0000 \\
 bu\ 1 = 0111\ 1111\ 1111\ 1111 \\
 \hline
 +1
 \end{array}$$

$$1000\ 0000\ 0000\ 0000 = 8000h$$

Kết quả trên AX=8000h

SF=1 vì MSB=1

PF=1 vì có số chẵn con số 1 trong byte thấp của kết quả

ZF=0 vì kết quả khác 0

CF=1 vì lệnh NEG làm cho CF=1 trở khi kết quả bằng 0

OF=1 vì dấu của kết quả giống với dấu của toán hạng nguồn .

2.4 Chương trình DEBUG.EXE

Debug là một chương trình của DOS cho phép chạy thời các chương trình hộp ngỗng. Người dùng có thể cho chạy chương trình tổng lệnh 1 từ ñầu ñến cuối ,trong quá trình ñó có thể thấy nội dung các thanh ghi thay ñổi nhờ thế nào . Debug cho phép nhập vào mã hộp ngỗng trực tiếp sau ñó DEBUG sẽ chuyển thành mã máy và lưu trữ trong bộ nhớ. DEBUG cung cấp khả năng xem nội dung của tất cả các thanh ghi có trong CPU.

Sau này chúng ta sẽ dùng DEBUG để mô tả cách
thực mà các lệnh ảnh hưởng đến các cờ nhỏ thế nào .

Giaỉ sôì chúng ta có chương trình hộp ngỗng sau :

```
TITLE PGM2_1: CHECK - FLAGS
; dùng DEBUG để kiểm tra các cờ
.MODEL    SMALL
.STACK    100H
.CODE
    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    ; HẠM THOÁT VỀ DOS
    INT     21H        ; EXIT TO DOS
END

MAIN     ENDP
        END MAIN
```

Sau khi dịch chương trình , giaỉ sôì file chạy là CHECK-
FL.EXE trên ổ cứng đĩa

C:\ASM . Để chạy debug chúng ta gõ lệnh sau :

C:\> **DEBUG C:\ASM\CHECK-FL.EXE**

tổn này trôini dấu nhac la của debug (dấu “_”) , ngôôi sôidùng còitheĩa vào các leñh debug tồdấu nhac này .

Trôôi hết còitheĩa xem nôi dung các thanh ghi bảng leñh R(Register) , man hình se còinôi dung nhô sau :

-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
```

Chúng ta thấy tên các thanh ghi vào nôi dung của chúng (dôi dĩ dĩng HEX) trên 3 dòng ãàu .

Dòng thôii 4 là trảng thấi các thanh ghi theo cách biêu thò của debug.

Bảng 2-3 là cách mao Debug biêu thò trảng thấi của các thanh ghi cò của CPU .

Flag s	Set (1) Symbol	Clear (0) Symbol
CF	CY (carry)	NC (no carry)
PF	PE (even parity)	PO (odd parity)
AF	AC (auxiliary carry)	NA (no auxiliary carry)
ZF	ZR (zero)	NZ (non zero)
SF	NG (negative)	PL (plus)
OF	OV (overflow)	NV (no overflow)
DF	DN (down)	UP (up)
IF	EI (enable)	DI (disable)

	interrupts)	interrupts)
--	-------------	-------------

Bảng 2.3 : Biểu đồ trạng thái các cờ của DEBUG

Dòng cuối cùng cho biết giá trị hiện hành của PC (nó chứa của lệnh sẽ được thực hiện dưới dạng nó chứa logic) mã máy của lệnh và nội dung của lệnh tiếp theo. Khi chạy chương trình này trên 1 máy tính khác thì sẽ thấy một vài khác nhau khác.

Chúng ta sẽ dùng lệnh T(Trace) để thi hành từng lệnh của chương trình bắt đầu từ lệnh `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
```

Sau khi thực hiện lệnh `MOV AX,4000` các cờ không bị thay đổi, chỉ có AX=4000h. Bây giờ chúng ta 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
```

Kết quả của phép cộng là 8000h, do đó SF=1(NG),
OF=1(OV) và PF=1(PE)

Bây giờ chúng ta thực hiện lệnh SUB AX,0FFFh

-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=8000H-FFFFH=8001H

Còn OF=0(NV) nhưng CF=1(CY) vì có một đơn vị MSB

Còn PF=0(PO) vì byte thấp chỉ còn 1 con số 1.

Lệnh tiếp theo sẽ là 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 lấy bù 2 của 8001h là 7FFFh. CF=1(CY) vì lệnh NEG
cho kết quả khác 0.

OF=0(NV) vì kết quả khác 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,4CH

OF=1(OV) vì có 2 số dương mà kết quả là 1 số âm
 CF=1(CY) vì lệnh INC không ảnh hưởng tới cờ này.

Nếu thực hiện toàn bộ chương trình chúng ta gọi G(Go)

-G

Program terminated normally

Nếu thoát khỏi debug gọi Q(Quit)

-Q

C:\>

Bảng sau đây cho biết một số lệnh debug thông dụng ,
 các tham số trong ngoặc là tùy chọn

COMMAND	ACTION
D(start (end) (range))	Liệt kê nội dung các byte dưới dạng HEX
D 100	Liệt kê 80h bytes bắt đầu từ DS:100h
D CS:100 120	Liệt kê các bytes từ DS:100h đến DS:120
D(DUMP)	Liệt kê 80h bytes từ byte cuối cùng của

	nhỏ hiển thị
G(=start) (addr1 addr2...addrn)	Chạy (go) lệnh từ vị trí Start với các lệnh dòng tại addr1,addr2,addrn
G G=100 G=100 150	Thực thi lệnh từ CS:IP đến hết Thực thi lệnh từ CS:100h đến hết Thực thi lệnh tại CS:100h dòng tại CS:150h
Q	Quit debug and return to DOS
R(register)	Xem/ thay nội dung của thanh ghi
R R AX	Xem nội dung tại các thanh ghi và Xem và thay nội dung của thanh ghi AX
T(=start)(value)	Quét "value" lệnh từ vị trí start
T T=100 T=100 5 T 4	Trace lệnh tại CS:IP Trace lệnh tại CS:100h Trace 5 lệnh bắt đầu từ CS:100h Trace 4 lệnh bắt đầu từ CS:IP

U(start)(value)	Unassemble vùng nhớ chứa thanh lệnh asm
U CS:100 110 U 200 L 20 U	Unassemble từ CS:100h đến CS:110h Unassemble 20 lệnh từ CS:200h Unassemble 32 bytes từ bytes cuối cùng nhỏ hiển thị
A(start)	Nhà vào mã hộp ngỏ cho 1 nhớ chứa hoặc 1

A A CS:100h	vùng ñiải chæ Ñòa vào mãihộp ngỗngtại CS:IP Ñòa vào mãihộp ngỗngtại CS:100h
----------------	---

Chương 3 : CÁC LỆNH ĐIỀU KHIỂN

Một chương trình thông thường sẽ thực hiện lần lượt các lệnh theo thời gian chúng được viết ra . Tuy nhiên trong một vài trường hợp cần phải chuyển điều khiển đến 1 phần khác của chương trình . Trong phần này chúng ta sẽ nghiên cứu các lệnh nhảy và lệnh lặp có tính đến cấu trúc của các lệnh này trong các ngôn ngữ cấp cao .

3.1 Ví dụ về lệnh nhảy

Nếu hình dung nội dung lệnh nhảy làm việc nhớ thế nào chúng ta hãy viết chương trình in ra toàn bộ tập các ký tự IBM .

```

TITLE PGR3-1:IBM CHARACTER DISPLAY
.MODEL      SMALL
.STACK     100H
.CODE
MAIN          PROC
    MOV  AH,2      ; hàm xuất ký tự
    MOV  CX,256    ; số ký tự cần xuất
    MOV  DL,0      ; DL giữ mã ASCII của ký tự NUL
; PRINT_LOOP :
    INT  21H       ;display character
    INC  DL
    DEC  CX
    JNZ  PRINT_LOOP ;nhảy nếu print_loop nếu CX# 0
;DOS EXIT
    MOV  AH,4CH
    INT  21H
MAIN ENDP
END MAIN
    
```

Trong chương trình chúng ta đã dùng lệnh điều khiển ***Jump if not zero (JNZ)*** để quay trở lại đầu chương trình xuất ký tự còn lại nữa chưa phải là PRINT_LOOP

3.2 Nhảy có điều kiện

Lệnh JNZ là một lệnh nhảy có điều kiện. Cấu pháp của một lệnh nhảy có điều kiện là:

Jxxx destination-label

Nếu điều kiện của lệnh được thỏa mãn thì lệnh tại Destination-label sẽ được thực hiện, nếu điều kiện không thỏa thì lệnh tiếp theo lệnh nhảy sẽ được thực hiện. Nói với lệnh JNZ thì điều kiện là kết quả của lệnh trước phải bằng 0.

Phạm vi của lệnh nhảy có điều kiện.

Câu trúc mã của lệnh nhảy có điều kiện yêu cầu destination-label nên (precede) lệnh nhảy phải không quá 126 bytes.

Làm thế nào để CPU thực hiện một lệnh nhảy có điều kiện?

Để thực hiện một lệnh nhảy có điều kiện CPU phải theo dõi thanh ghi cờ. Nếu điều kiện cho lệnh nhảy (được biểu diễn bởi một tổ hợp trạng thái các cờ) là đúng thì CPU sẽ thay đổi IP về destination-label sao cho lệnh tại địa chỉ destination-label được thực hiện. Nếu điều kiện nhảy không thỏa thì IP sẽ không thay đổi, nghĩa là lệnh tiếp theo lệnh nhảy sẽ được thực hiện.

Trong chương trình trên đây, CPU thực hiện lệnh JNZ PRINT_LOOP bằng cách kiểm tra các cờ ZF. Nếu ZF=0 điều kiện chuyển tới PRINT_LOOP. Nếu ZF=1 lệnh MOV AH,4CH sẽ được thực hiện.

Bảng 3-1 cho thấy các lệnh nhảy có điều kiện. Các lệnh nhảy được chia thành 3 loại:

- nhảy có dấu (dùng cho các diễn dịch có dấu nói với kết quả)
- nhảy không dấu (dùng cho các diễn dịch không dấu nói với kết quả)
- nhảy một cờ (dùng cho các thao tác ảnh hưởng lên 1 cờ)

Một số lệnh nhảy chỉ 2 Opcode. Chúng ta có thể dùng một trong 2 Opcode, nhưng kết quả thực hiện lệnh là như nhau.

Nhảy có dấu

SYMBOL	DESCRIPTION	CONDITION FOR JUMPS
JG/JNLE	jump if greater than jump if not less than or equal to	ZF=0 and SF=OF
JGE/JNL	jump if greater than or equal to jump if not less or equal to	SF=OF
JL/JNGE	jump if less than jump if not greater or equal	SF<>OF

JLE/JNG	jump if less than or equal jump if not greater	ZF=1 or SF<>OF
---------	---	----------------

Nhảy có điều kiện không dấu

SYMBOL	DESCRIPTION	CONDITION FOR JUMPS
JA/JNBE	jump if above jump if not below or equal	CF=0 and ZF=0
JAE/JNB	jump if above or equal jump if not below	CF=0
JB/JNA	jump if below jump if not above or equal	Cf=1
JBE/JNA	jump if below or equal jump if not above	CF=1 or ZF=1

Nhảy 1 chữ

SYMBOL	DESCRIPTION	CONDITION FOR JUMPS
JE/JZ	jump if equal jump if equal to zero	ZF=1
JNE/JNZ	jump if not equal jump if not zero	ZF=0
JC	jump if carry	CF=1
JNC	jump if no carry	CF=0
JO	jump if overflow	OF=1
JNO	jump if not overflow	OF=0
JS	jump if sign negative	SF=1
SYMBOL	DESCRIPTION	CONDITION FOR JUMPS

JNS	jump if nonnegative sign	SF=0
JP/JPE	jump if parity even	PF=1
JNP/JPO	jump if parity odd	PF=0

Lệnh CMP (Compare)

Các lệnh nhảy thông lấy kết quả của lệnh Compare nhờ vào điều kiện . Cú pháp của lệnh CMP là:

CMP destination, source

Lệnh này so sánh toán hạng nguồn và toán hạng đích bằng cách tính hiệu Destination - Source . Kết quả sẽ không được cất giữ. Nhờ vậy là lệnh CMP giống như lệnh SUB , khác là trong lệnh CMP toán hạng đích không thay đổi .

Gia sư có chương trình cho các lệnh sau :

CMP AX,BX ;trong đó AX=7FFF và BX=0001h
JG BELOW

Kết quả của lệnh CMP AX,BX là 7FFEh . Lệnh JG được thỏa mãn vì ZF=0=SF=OF do đó điều kiện được chuyển nên nhận BELOW.

Diễn dịch lệnh nhảy có điều kiện

Ví dụ trên đây về lệnh CMP cho phép lệnh nhảy sau nó chuyển điều kiện nên nhận BELOW . Đây là ví dụ cho thấy CPU thực hiện lệnh nhảy nhờ thế nào . Chúng thực hiện bằng cách khám xét trạng thái các cờ. Lập trình viên không cần quan tâm đến các cờ, mà chỉ thể dùng tên của các lệnh nhảy để chuyển điều kiện nên một nhận nào đó. Các lệnh

CMP AX,BX
JG BELOW

cùng hóa là nếu AX > BX thì nhảy nên nhận BELOW

Mục đích của lệnh CMP được thiết kế cho các lệnh nhảy . Những lệnh nhảy có thể đứng trước 1 lệnh khác , chẳng hạn :

DEC AX
JL THERE

cùng hóa là nếu AX trong diện dịch có dấu < 0 thì điều kiện được chuyển cho THERE .

Nhảy có dấu so với nhảy không dấu

Một lệnh nhảy có dấu tổng cộng với 1 nhảy không dấu . Ví dụ lệnh nhảy có dấu JG và lệnh nhảy không dấu JA . Việc sử dụng JG hay JA lại tùy thuộc vào đích có dấu hay không dấu . Bảng 3-1 cho thấy các lệnh nhảy có dấu phụ thuộc vào trạng thái của các cờ ZF,SF,OF . Các lệnh nhảy không dấu phụ thuộc vào trạng thái của các cờ ZF và CF . Sử dụng lệnh nhảy không hợp lý sẽ tạo ra kết quả sai .

Giải thích chúng ta đích có dấu . Nếu AX=7FFFh và BX=8000h , các lệnh :

```
CMP AX,BX
```

```
JA below
```

sẽ cho kết quả sai mặc dù $7FFFh > 8000h$ (lệnh JA không thực hiện nhảy vì $7FFFFh < 8000h$ trong đích không dấu)

Sau này chúng ta sẽ lấy ví dụ để minh họa việc sử dụng các lệnh nhảy

Ví dụ: Giải thích AX và BX chứa các số có dấu . Viết đoạn code để kiểm tra số lớn nhất vào CX .

Giải :

```
MOV CX,AX ; đặt AX vào CX
```

```
CMP BX,CX ; BX lớn hơn CX?
```

```
JLE NEXT ; không thì tiếp tục
```

```
MOV CX,BX ; yes , đặt BX vào CX
```

```
NEXT:
```

3.3 Lệnh JMP

Lệnh JMP (jump) là lệnh nhảy không điều kiện . Cú pháp của JMP là
JMP destination

Trong đó destination là một nhãn ở trong cùng 1 đoạn với lệnh JMP .

Lệnh JMP dùng để thực hiện chế độ của các lệnh nhảy có điều kiện (không quá 126 bytes kể từ vị trí của lệnh nhảy có điều kiện)

Ví dụ chúng ta có đoạn chương trình sau :

```
TOP:
```

```
; thành vòng lặp
```

```
DEC CX
```

```
JNZ TOP ; nếu CX>0 tiếp tục lặp
```

```
MOV AX,BX
```

giải thích thành vòng lặp chứa nhiều lệnh mà nội dung chỉ 126 bytes trước lệnh

JNZ TOP . Có thể giải quyết tình trạng này bằng các lệnh sau :

```
TOP:
```

```

        ; thành vòng lặp
        DEC  CX
        JNZ  BOTTOM    ; nếu CX>0 tiếp tục lặp
        JMP  EXIT
BOTTOM:
        JMP  TOP
EXIT:
        MOV  AX,BX
    
```

3.4 Cấu trúc của ngôn ngữ cấp cao

Chúng ta sẽ dùng các lệnh này để thể hiện các cấu trúc tổng thể nhỏ trong ngôn ngữ cấp cao

3.4.1 Cấu trúc rẽ nhánh

Trong ngôn ngữ cấp cao cấu trúc rẽ nhánh cho phép một chương trình rẽ nhánh nên những đoạn khác nhau tùy thuộc vào các điều kiện. Trong phần này chúng ta sẽ xem xét 3 cấu trúc

a) IF-THEN

Cấu trúc IF-THEN có thể diễn đạt như sau :

```

IF condition is true
    THEN
        execute true branch statements
END IF
    
```

Ví dụ: Thay thế giá trị trên AX bằng giá trị tuyệt đối của nó

Thuật toán như sau :

```

IF AX<0
    THEN
        replace AX by -AX
END-IF
    
```

Có thể mã hóa như sau :


```

; if AX<0
        CMP  AX,0
        JNL  END_IF      ; no , exit
;then
        NEG  AX          , yes , change sign
END_IF :
    
```

b) IF_THEN_ELSE

```

IF condition is true
    THEN
        execute true branch statements
    ELSE
        execute false branch statements
END_IF
    
```

Ví dụ: giả sử AL và BL chứa ASCII code của 1 ký tự. Hãy xuất ra màn hình ký tự trước (theo thứ tự ký tự)

```

Thuật toán
    IF  AL<= BL
        THEN
            display AL
        ELSE
            display character in BL
    END_IF
    
```

Có thể mã hóa như sau :

```

        MOV  AH,2          ; chuẩn bị xuất ký tự
;if      AL<=BL
        CMP  AL,BL          ;AL<=BL?
        JNBE ELSE_         ; no, display character in BL
;then
        MOV  DL,AL
        JMP  DISPLAY
ELSE_:
        MOV  DL,BL

DISPLAY:
        INT  21H
END_IF :
    
```

c) CASE

Case là một cấu trúc rẽ nhánh nhiều hướng . Có thể dùng để test một thành phần hay , biến nào đó thay một biểu thức mà giá trị của nó nằm trong 1 vùng các giá trị . Cấu trúc của CASE như sau :

```
CASE expression
    value_1 : Statements_1
    value_2 : Statements_2
    .
    .
    value_n : Statements_n
```

Ví dụ : Nếu AX âm thì đặt -1 vào BX
 Nếu AX bằng 0 thì đặt 0 vào BX
 Nếu AX dương thì đặt 1 vào BX

Thuật toán :

```
CASE      AX
    < 0 put -1 in BX
    = 0 put 0 in BX
    > 0 put 1 in BX
```

Có thể mã hóa như sau :

```
; case AX
        CMP  AX,0      ;test AX
        JL   NEGATIVE  ;AX<0
        JE   ZERO      ;AX=0
        JG   POSITIVE   ;AX>0
NEGATIVE:
        MOV  BX,-1
        JMP  END_CASE
ZERO:
        MOV  BX,0
        JMP  END_CASE
POSITIVE:
        MOV  BX,1
        JMP  END_CASE
END_CASE :
```

Rẽ nhánh với một tổ hợp các điều kiện

Nếu khi tình trạng rẽ nhánh trong các lệnh IF ,CASE cần một tổ hợp các điều kiện dưới dạng :

Condition_1 AND Condition_2
Condition_1 OR Condition_2

Ví dụ về điều kiện AND : Nếu một ký tự và nếu nó là ký tự hoa thì in nó ra màn hình

Thuật toán :
Read a character (into AL)
IF ('A' <= character) AND (character <= 'Z')
THEN
display character
END_IF

Sau đây là code

```
;read a character
    MOV AH,2
    INT 21H          ; character in AL
; IF ( 'A' <= character ) AND ( character <= 'Z' )
    CMP AL,'A'       ; char >='A'?
    JNGE END_IF      ;no, exit
    CMP AL,'Z'       ; char <='Z'?
    JNLE END_IF      ; no exit
; then display it
    MOV DL,AL
    MOV AH,2
    INT 21H
END_IF :
```

Ví dụ về điều kiện OR : Nếu một ký tự, nếu ký tự nó là 'Y' hoặc 'y' thì in nó lên màn hình , ngược lại thì kết thúc chương trình .

Thuật toán
Read a character (into AL)
IF (character = 'Y') OR (character='y')
THEN
display it
ELSE
terminate the program
END_IF

Code

```

;read a character
    MOV AH,2
    INT 21H          ; character in AL
; IF ( character = 'y' ) OR ( character = 'Y' )
    CMP AL,'y'       ; char = 'y'?
    JE THEN          ;yes , goto display it
    CMP AL,'Y'       ; char = 'Y'?
    JE THEN          ; yes , goto display it
    JMP ELSE_        ;no , terminate
THEN :
    MOV DL,AL
    MOV AH,2
    INT 21H
    JMP END_IF
ELSE_:
    MOV AH,4CH
    INT 21h
END_IF :

```

4.3.2 Cấu trúc lặp

Một vòng lặp gồm nhiều lệnh nối tiếp nhau, số lần lặp phụ thuộc nhiều điều kiện.

a) Vòng **FOR**

Lệnh LOOP có thể dùng để thực hiện vòng FOR. Cấu pháp của lệnh LOOP như sau :

```

    LOOP destination_label

```

Số đếm cho vòng lặp là thanh ghi CX mà ban đầu nó được gán 1 giá trị nào đó. Khi lệnh LOOP được thực hiện CX sẽ tự động giảm đi 1. Nếu CX còn bằng 0 thì vòng lặp được thực hiện tiếp tục. Nếu CX=0 lệnh sau lệnh LOOP được thực hiện. Dùng lệnh LOOP, vòng FOR có thể thực hiện như sau :

```

; gán cho CX số lần lặp
TOP:
; thân của vòng lặp
LOOP TOP
Ví dụ: Dùng vòng lặp in ra 1 hàng 80 dấu '*'
    MOV CX,80      ; CX chứa số lần lặp
    MOV AH,2       ; hàm xuất ký tự

```

```

        MOV  DL,'*'      ;DL chứa ký tự '*'
TOP:
        INT  21h         ; in dấu '*'
        LOOP TOP         ; lặp 80 lần
    
```

Lưu ý rằng vòng FOR cũng nhờ lệnh LOOP thực hiện ít nhất là 1 lần . Do đó nếu ban đầu CX=0 thì vòng lặp sẽ làm cho CX=FFFFH , tức là thực hiện lặp đến 65535 lần . Nếu trình tình trạng này , lệnh JCXZ (Jump if CX is zero) phải được dùng trước vòng lặp . Lệnh JXCZ có cú pháp như sau :

```

        JCXZ destination_label

        Nếu CX=0 nên khiến được chuyển cho destination_label . Các lệnh sau này
        sẽ nằm bên ngoài vòng lặp không thực hiện nếu CX=0
        JCXZ SKIP
TOP :
        ; thân vòng lặp
        LOOP TOP
SKIP :
    
```

b) Vòng WHILE

Vòng WHILE phụ thuộc vào 1 điều kiện . Nếu điều kiện đúng thì thực hiện vòng WHILE . Vì vậy nếu điều kiện sai thì vòng WHILE không thực hiện gì cả.

Ví dụ: Viết đoạn mã để nhập số ký tự được nhập vào trên cùng một hàng .

```

        MOV  DX,0        ; DX chứa số ký tự
        MOV  AH,1        ; hàm đọc 1 ký tự
        INT  21h         ; đọc ký tự vào AL
WHILE_:
        CMP  AL,0DH      ; có phải là ký tự CR?
        JE   END_WHILE   ; đúng , thoát
        INC  DX           ; tăng DX lên 1
        INT  21h         ; đọc ký tự
        JMP  WHILE_      ; lặp
END_WHILE :
    
```

c) Vòng REPEAT

Cấu trúc của REPEAT là

```

repeat statements
until condition
    
```

Trong cấu trúc repeat mệnh đề được thực hiện trong thời gian điều kiện được kiểm tra . Nếu điều kiện đúng thì vòng lặp kết thúc .

Ví dụ: viết đoạn mã để nhập vào các ký tự cho đến khi gặp ký tự trống .

```

        MOV  AH,1          ; nhập ký tự
REPEAT:
        INT  21h           ; ký tự trên AL
;until
        CMP  AL,' '        ; AL=' '?
        JNE  REPEAT
    
```

Lưu ý: việc sử dụng REPEAT hay WHILE là tùy theo chuỗi quan của mỗi người . Tuy nhiên có thể thấy rằng REPEAT phải tiến hành ít nhất 1 lần , trong khi với WHILE có thể không tiến hành lần nào cả nếu ngay từ đầu nhiều kiện đã sai .

3.5 Lập trình với cấu trúc cấp cao

Bài toán : Viết chương trình nhắc người dùng gõ vào một dòng văn bản . Trên 2 dòng tiếp theo in ra ký tự viết hoa đầu tiên và ký tự viết hoa cuối cùng theo thứ tự alphabetical . Nếu người dùng gõ vào một ký tự thông thường , máy sẽ thông báo 'No capitals'

Kết quả chạy chương trình sẽ như sau :

Type a line of text :

TRUONG DAI HOC DALAT

First capital = A

Last capital = U

Nếu giải bài toán này ta dùng kỹ thuật lập trình TOP-DOWN , nghĩa là chia nhỏ bài toán thành nhiều bài toán con . Có thể chia bài toán thành 3 bài toán con nhỏ sau :

1. Xuất 1 chuỗi ký tự (lời nhắc)
2. Nhập vào và xử lý 1 dòng văn bản
3. In kết quả

Bước 1: Hiển thị nhắc .

Bước này có thể mã hóa như sau :

```

MOV  AH,9          ; hàm xuất chuỗi
LEA  DX,PRMOPT     ; lấy địa chỉ chuỗi vào DX
INT  21h           ; xuất chuỗi
Đầu nhắc có thể mã hóa như sau trong đoạn số liệu .
    
```

PROMPT DB 'Type a line of text :',0DH,0AH,'\$'

Bước 2 : Viết chương trình

Bước này thực hiện hầu hết các công việc của chương trình : đọc các ký tự từ bàn phím , tìm ra ký tự đầu và ký tự cuối , nhắc nhở người dùng nếu ký tự gõ vào không phải là ký tự hoa .

Có thể biểu diễn bước này bởi thuật toán sau :

```

Read a character
WHILE character is not a carriage return DO
  IF character is a capital (*)
    THEN
      IF character precedes first capital
        Then
          first capital= character
        End_if
      IF character follows last character
        Then
          last character = character
        End_if
    END_IF
  Read a character
END_WHILE
    
```

Trong nội dung (*) có nghĩa là nếu ký tự là hoa là nếu ký tự AND IF ('A' <= character) AND (character <= 'Z')

Bước 2 có thể mã hóa như sau :

```

MOV AH,1 ; đọc ký tự
INT 21H ; ký tự trên AL
WHILE :
;trong khi ký tự gõ vào không phải là CR thì thực hiện
CMP AL,0DH ; CR?
JE END_WHILE ;yes, thoát
; nếu ký tự là hoa
CMP AL,'A' ; char >= 'A'?
JNGE END_IF ;không phải ký tự hoa thì nhảy nên END_IF
CMP AL,'Z' ; char <= 'Z'?
JNLE END_IF ; không phải ký tự hoa thì nhảy nên END_IF
; thì
    
```

```

; nếu ký tự nằm trước biến FIRST ( giá trị ban đầu là '[' : ký tự sau Z )
    CMP AL,FIRST ; char < FIRST ?
    JNL CHECK_LAST; >=
; thì ký tự viết hoa đầu tiên = ký tự
    MOV FIRST,AL ; FIRST=character
;end_if
CHECK_LAST:
; nếu ký tự là sau biến LAST ( giá trị ban đầu là '@' : ký tự trước A)
    CMP AL, LAST ; char > LAST ?
    JNG END_IF ; <=
;thì ký tự cuối cùng = ký tự
    MOV LAST, AL ;LAST = character
;end_if
END_IF :
; nếu một ký tự
    INT 21H ; ký tự trên AL
    JMP WHILE_ ; lặp
END_WHILE:
Các biến FIRST và LAST được định nghĩa như sau trong đoạn mã :
    FIRST DB '[ $' ; '[' là ký tự sau Z
    LAST DB '@ $ ' ; '@' là ký tự trước A

```

Bước 3 : In kết quả

Thuật toán

IF no capital were typed

THEN

display 'No capital'

ELSE

display first capital and last capital

END_IF

Bước 3 sẽ phải in ra các thông báo :

- NOCAP_MSG nếu không phải chữ in
- CAP1_MSG chữ in đầu tiên
- CAP2_MSG chữ in cuối cùng

Chúng ta định nghĩa như sau trong đoạn mã .

```

NOCAP_MSG DB 0DH,0AH,'No capitals $'
CAP1_MSG DB 0DH,0AH,'First capital= '
FIRST DB '[ $ '
CAP2_MSG DB 0DH,0AH,'Last capital='

```



```

                LAST                DB    '@ $'

Bước 3 có thể mã hóa như sau :
; in kết quả
    MOV  AH,9    ; hàm xuất ký tự
; IF không có chữ hoa nào nhập thì FIRST = '['
    CMP  FIRST,'['    ; FIRST='[' ?
    JNE  CAPS        ; không , in kết quả
; THEN
    LEA  DX, NOCAP_MSG
    INT  21H
CAPS:
    LEA  DX,CAP1_MSG
    INT  21H
    LEA  DX,CAP2_MSG
    INT  21H
; end_if
    
```

Chương trình có thể viết như sau :

```

TITLE  PGM3-1 : FIRST AND LAST CAPITALS
.MODEL  SMALL
.STACK  100h
.DATA
PROMPT    DB    'Type a line of text', 0DH, AH, '$'
NOCAP_MSG    DB    0DH,0AH, 'No capitals $'
CAP1_MSG     DB    0DH,0AH, 'First capital='
FIRST        DB    '[' $'
CAP2_MSG     DB    'Last capital = '
LAST         DB    '@ $'

.CODE
MAIN        PROC
; khởi tạo DS
    MOV  AX,@DATA
    MOV  DS,AX
; in dấu nhắc

    MOV  AH,9    ; hàm xuất chuỗi
    LEA  DX,PROMPT ; lấy địa chỉ chuỗi vào DX
    INT  21H    ; xuất chuỗi
    
```

; đọc vào dữ liệu 1 dòng văn bản

```
MOV AH,1 ; đọc ký tự
INT 21H ; ký tự trên AL
```

WHILE :

; trong khi ký tự gõ vào không phải là CR thì thực hiện

```
CMP AL,0DH ; CR?
JE END_WHILE ; yes, thoát
```

; nếu ký tự là hoa

```
CMP AL,'A' ; char >= 'A'?
JNGE END_IF ; không phải ký tự hoa thì nhảy nên END_IF
CMP AL,'Z' ; char <= 'Z'?
JNLE END_IF ; không phải ký tự hoa thì nhảy nên END_IF
```

; thì

; nếu ký tự nằm trước biến FIRST

```
CMP AL,FIRST ; char < FIRST?
JNL CHECK_LAST ; >=
```

; thì ký tự viết hoa nếu tiền = ký tự

```
MOV FIRST,AL ; FIRST=character
```

;end_if

CHECK_LAST:

; nếu ký tự là sau biến LAST

```
CMP AL,LAST ; char > LAST?
JNG END_IF ; <=
```

; thì ký tự cuối cùng = ký tự

```
MOV LAST,AL ; LAST = character
```

;end_if

END_IF :

; đọc một ký tự

```
INT 21H ; ký tự trên AL
JMP WHILE_ ; lặp
```

END_WHILE:

; in kết quả

```
MOV AH,9 ; hàm xuất ký tự
```

; IF không có chữ hoa nào được nhập thì FIRST = '['

```
CMP FIRST,'[' ; FIRST='['?
JNE CAPS ; không , in kết quả
```

;Then

```
LEA DX,NOCAP_MSG
INT 21H
```

CAPS:

```
LEA DX,CAP1_MSG
```

```
        INT    21H
        LEA    DX,CAP2_MSG
        INT    21H
    ; end_if
; dos exit
        MOV    AH,4CH
        INT    21h
MAIN ENDP
        END    MAIN
```

Chương 4 : CÁC LỆNH LOGIC , DÒCH VÀ QUAY

Trong chương này chúng ta sẽ xem xét các lệnh mà chúng có thể dùng để thay đổi từng bit trên một byte hoặc một từ số liệu . Khả năng quản lý trên từng bit thông thường có trong các ngôn ngữ cấp cao (trừ C) và đây là lý do giải thích tại sao hộp ngũ và những vai trò quan trọng trong khi lập trình .

4.1 Các lệnh logic

Chúng ta có thể dùng các lệnh logic để thay đổi từng bit trên byte hoặc trên một từ số liệu .

Khi một phép toán logic nào đó áp dụng cho toán hạng 8 hoặc 16 bit thì có thể áp dụng phép toán logic đó trên từng bit nếu thu được kết quả cuối cùng .

Ví dụ : Thực hiện các phép toán sau :

1. 10101010 AND 1111 0000
2. 10101010 OR 1111 0000
3. 10101010 XOR 1111 0000
4. NOT 10101010

Giải :

$$\begin{array}{rcl}
 1. & & 10101010 \\
 & & \text{AND } 1111\ 0000 \\
 & \hline
 & = & 1010\ 0000
 \end{array}$$

$$\begin{array}{rcl}
 2. & & 10101010 \\
 & & \text{OR } 1111\ 0000 \\
 & \hline
 & = & 1111\ 1010
 \end{array}$$

$$\begin{array}{rcl}
 3. & & 1010\ 1010 \\
 & & \text{XOR } 1111\ 0000 \\
 & \hline
 & & 0101\ 1010
 \end{array}$$

$$\begin{array}{rcl}
 4. & \text{NOT} & 10101010 \\
 & = & 01010101
 \end{array}$$

4.1.1 Lệnh AND, OR và XOR

Lệnh AND, OR và XOR thực hiện các chức năng như sau:

Cú pháp của chúng là:

AND destination, source

OR destination, source

XOR destination, source

Kết quả của lệnh sẽ lưu trữ trong toàn hàng đích do đó chúng phải là thanh ghi hoặc vị trí nhớ. Toàn hàng nguồn là có thể là hàng số, thanh ghi hoặc vị trí nhớ. Dĩ nhiên hai toàn hàng nếu là vị trí nhớ là không được phép.

Ảnh hưởng đến các cờ:

Các cờ SF, ZF và PF phản ánh kết quả

AF không xác định

CF=OF=0

Nếu thay đổi tổng bit theo ý muốn chúng ta xây dựng toàn hàng nguồn theo kiểu mặt nạ (mask). Nếu xây dựng mặt nạ chúng ta sử dụng các tính chất sau đây của các phép toán AND, OR và XOR:

$b \text{ AND } 1 = b$	$b \text{ OR } 0 = b$	$b \text{ XOR } 0 = b$
$b \text{ AND } 0 = 0$	$b \text{ OR } 1 = 1$	$b \text{ XOR } 1 = \text{not } b$

- Lệnh AND có thể dùng để xóa (clear) toàn hàng đích nếu mặt nạ bằng 0
- Lệnh OR có thể dùng để đặt (set) 1 cho toàn hàng đích nếu mặt nạ bằng 1
- Lệnh XOR có thể dùng để lấy đảo toàn hàng đích nếu mặt nạ bằng 1. Lệnh XOR cũng có thể dùng để xóa nội dung một thanh ghi (XOR với chính nó)

Ví dụ: Xóa bit đầu của AL trong khi các bit khác không thay đổi

Giải: Dùng lệnh AND với mặt nạ 01111111=7Fh

AND AL,7Fh ; xóa bit đầu (bit 7) của AL

Ví dụ: Set 1 cho các bit MSB và LSB của AL, các bit khác không thay đổi.

Giải: Dùng lệnh OR với mặt nạ 10000001=81h

OR AL,81h ; set 1 cho LSB và MSB của AL

Ví dụ: Thay đổi bit đầu của DX

Giải: Dùng lệnh XOR với mặt nạ 1000000000000000=8000h

XOR DX,8000h

Các lệnh logic là các lệnh khi thực hiện các nhiệm vụ sau:

Đổi một số nhị phân ASCII thành một số

Giải thích chúng ta có một ký tự từ bàn phím bằng hàm 1 byte 21h. Khi nó AL chứa mã ASCII của ký tự. Nếu nay cũng như ký tự đó là một số (digital character). Ví dụ nếu chúng ta gõ số 5 thì AL = 35h (ASCII code for 5)

Nếu còn 5 trên AL chúng ta dùng lệnh :

SUB AL,30h

Có một cách khác để làm việc này là dùng lệnh AND để xóa nửa byte cao (high nibble = 4 bit cao) của AL :

AND AL,0Fh

Vì các số từ 0-9 có mã ASCII từ 30h-39h, nên cách này dùng để đưa mỗi số ASCII ra thập phân.

Chương trình hợp nguồn đưa mỗi số thập phân thành mã ASCII của chúng rồi xem nhớ bài tập.

Đổi chữ thường thành chữ hoa

Mã ASCII của các ký tự thường từ a-z là 61h-7Ah và mã ASCII của các ký tự hoa từ A-Z là 41h-5Ah. Giải quyết DL cho các ký tự thường, để đưa nó thành chữ hoa ta dùng lệnh :

SUB DL,20h

Nếu chúng ta so sánh mã nhị phân tổng cộng của ký tự thường và ký tự hoa thì thấy rằng chỉ cần xóa bit thứ 5 thì sẽ đưa ký tự thường sang ký tự hoa.

Character	Code	Character	Code
a(61h)	01100001	A (41h)	01000001
b (62h)	01100010	B (42h)	01000010
.			
.			
z (7Ah)	01111010	Z (5Ah)	01011010

Có thể xóa bit thứ 5 của DL bằng cách dùng lệnh AND với mã nhị phân 11011111= DF h

AND DL,0DFh ; đưa ký tự thường trong DL sang ký tự hoa

Xóa nội dung thanh ghi

Chúng ta có thể dùng lệnh sau để xóa nội dung thanh ghi AX :

MOV AX,0

hoặc SUB AX,AX

XOR AX,AX

Lệnh trên cần 3 bytes trong khi hai lệnh sau chỉ cần 2 bytes. Những lệnh MOV phải được dùng để đưa 1 vào vị trí nhớ.

Kiểm tra nội dung thanh ghi có bằng 0 ?

Thay cho lệnh
CMP AX,0
Ngươi ta dùng lệnh
OR CX,CX
để kiểm tra xem CX có bằng 0 hay không vì nó làm thay đổi cờ ZF (ZF=0 nếu CX=0)

4.1.2 Lệnh NOT

Lệnh NOT dùng để lấy bù 1 (đảo) toàn hàng nhị . Cú pháp là:
NOT destination
Không có cờ nào bị ảnh hưởng bởi lệnh NOT
Ví dụ : Lấy bù 1 AX
NOT AX

4.1.3 Lệnh TEST

Lệnh TEST thực hiện phép AND giữa toàn hàng nhị và toàn hàng nguồn nhưng không làm thay đổi toàn hàng nhị . Mục đích của lệnh TEST là để set các cờ trạng thái . Cú pháp của lệnh test là:

TEST destination,source
Các cờ bị ảnh hưởng của lệnh TEST :
SF,ZF và PF phản ánh kết quả
AF không xác định
CF=OF=0

Lệnh TEST có thể dùng để kiểm tra 1 bit trên toàn hàng . Mặt này phải chứa bit 1 tại vị trí cần kiểm tra , các bit khác thì bằng 0 . Kết quả của lệnh :

TEST destination,mask
sẽ là 1 tại bit cần test nếu nhớ toàn hàng nhị chứa 1 tại bit test . Nếu toàn hàng nhị chứa 0 tại bit test thì kết quả sẽ bằng 0 và do đó ZF=1 .

Ví dụ : Nhảy tới nhãn BELOW nếu AL là một số chẵn

Giải : Số chẵn có bit thứ 0 bằng 0 , lệnh

TEST AL,1 ; AL chẵn ?
JZ BELOW ; nếu , nhảy đến BELOW

4.2 Lệnh SHIFT

Lệnh dịch và quay sẽ dịch các bit trên trên toàn hàng ních một hoặc nhiều vị trí sang trái hoặc sang phải . Khác nhau của lệnh dịch và lệnh quay là ở chỗ: các bit bỏ dịch ra (trong lệnh dịch) sẽ bỏ mất . Trong khi đó với lệnh quay , các bit bỏ dịch ra trở về đầu của toàn hàng sẽ nối lại nữa kia của nó.

Có 2 khả năng viết nối với lệnh dịch và quay :

OPCODE destination, 1

OPCODE destination, CL

trong cách viết thì hai thành ghi CL chứa N là số lần dịch hay quay . Toàn hàng ních có thể là một thành ghi 8 hoặc 16 bit , hoặc một vị trí nhớ.

Các lệnh dịch và quay thông dụng nên nhận và chia các số nhị phân . Chúng cũng được dùng cho các hoạt động nhập xuất nhị phân và hex .

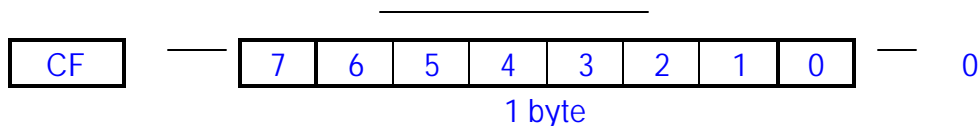
4.2.1 Lệnh dịch trái (left shift)

Lệnh SHL dịch toàn hàng ních sang trái . Cú pháp của lệnh như sau :

SHL destination , 1 ; dịch trái dest 1 bit

SHL destination , CL ; dịch trái N bit (CL chứa N)

Còn một lần dịch trái , một số 0 được thêm vào LSB .



Các cờ báo ảnh hưởng :

SF, PF, ZF phản ánh kết quả

AF không xác định

CF = bit cuối cùng được dịch ra

OF = 1 nếu kết quả thay đổi dấu vào lần dịch cuối cùng

Ví dụ : Giả sử DH = 8Ah và CL = 3 . Hỏi giá trị của DH và CF sau khi lệnh

SHL DH, CL được thực hiện ?

Kết quả DH = 01010000 = 50h , CF = 0

Nhân bằng lệnh SHL

Chúng ta hãy xét số 235 decimal . Nếu dịch trái 235 một bit và thêm 0 vào bên phải chúng ta sẽ có 2350 . Nói cách khác , khi dịch trái 1 bit chúng ta nhân 10.

Nếu với số nhị phân , dịch trái 1 bit cũng hóa ra nhân với 2. Ví dụ

AL = 00000101 = 5d

SHL AL, 1 ; AL = 00001010 = 10d

SHL AL, CL ; nếu CL = 2 thì AL = 20d sau khi thực hiện lệnh

Lệnh dịch trái số học (SAL = Shift Arithmetic Left)

Lệnh SHL có thể dùng để nhân một toán hạng với hệ số 2. Tuy nhiên trong trường hợp người ta muốn nhân mình nên tính chất số học của phép toán thì lệnh SAL sẽ tốt hơn thay cho SHL. Các lệnh này tạo ra cùng một mã máy.

Một số máy cũng có thể nhân 2 bằng cách dịch trái. Ví dụ: Nếu $AX = \text{FFFFh} = -1$ thì sau khi dịch trái 3 lần $AX = \text{FFF8h} = -8$

Tran

Khi chúng ta dùng lệnh dịch trái để nhân thì có thể xảy ra số tràn. Nếu với lệnh dịch trái 1 lần, CF và OF phản ánh chính xác số tràn dấu và tràn không dấu. Tuy nhiên các cờ sẽ không phản ánh chính xác kết quả nếu dịch trái nhiều lần bởi vì dịch nhiều lần thực chất là một chuỗi các dịch 1 lần liên tiếp và vì vậy các cờ CF và OF chỉ phản ánh kết quả của lần dịch cuối cùng. Ví dụ: $BL = 80\text{h}$, $CL = 2$ thì lệnh

SHL BL, CL

sẽ làm cho $CF = OF = 0$ mặc dù trên thực tế đã xảy ra tràn dấu và tràn không dấu.

Ví dụ: viết đoạn mã nhân AX với 8. Giải thích rằng không có tràn.

MOV CL, 3 ; $CL = 3$

SHL AX, CL ; $AX * 8$

4.2.2 Lệnh dịch phải (Right Shift)

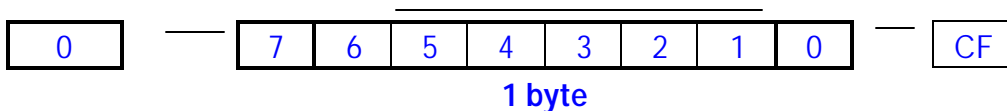
Lệnh SHR dịch phải toán hạng ních 1 hoặc N lần.

SHR destination, 1

SHR destination, CL

Cứ mỗi lần dịch phải, một số 0 sẽ được thêm vào MSB

Các cờ bị ảnh hưởng giống như lệnh SHL



Ví dụ: giả sử $DH = 8Ah$, $CL = 2$

Lệnh SHR DH, CL ; dịch phải DH 2 lần sẽ cho kết quả như sau :

Kết quả trên $DH = 22\text{h}$, $CF = 1$

Cũng như lệnh SAL, lệnh SAR (dịch phải số học) hoạt động giống như SHR, chỉ có 1 khác là MSB vẫn giữ giá trị nguyên thủy (bit dấu giữ nguyên) sau khi dịch.

Chia bảng lệnh dịch phải

Lệnh dịch phải sẽ chia 2 giá trị của toán hạng đích . Nếu nay nếu nối với số chain . Nếu với số lẻ, lệnh dịch phải sẽ chia 2 và làm tròn xuống số nguyên gần nhất . Ví dụ, nếu BL = 00000101=5 thì khi dịch phải BL=00000010 =2 .

Chia có dấu và không dấu

Nếu thực hiện phép chia bằng lệnh dịch phải , chúng ta phải phân biệt giữa số có dấu và số không dấu . Nếu dịch không dấu thì dùng lệnh SHR , còn nếu dịch có dấu thì dùng SAR (bit dấu giống nguyên) .

Ví dụ: dùng lệnh dịch phải để chia số không dấu 65143 cho 4 . Thông số đặt trên AX .

```
MOV AX,65143
MOV CL,2
SHR AX,CL
```

Ví dụ: Nếu AL = -15 , cho biết AL sau khi lệnh

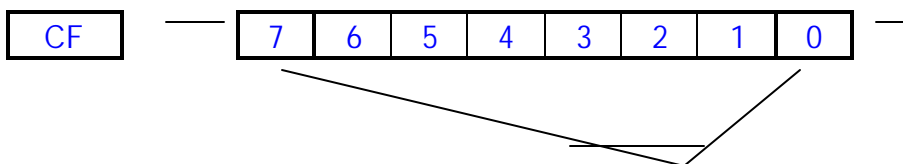
SAR AL,1 được thực hiện

Giải : AL= -15 = 11110001b

Sau khi thực hiện SAR AL ta có AL = 11111000b = -8

4.3 Lệnh quay (Rotate)

Quay trái (rotate left) = ROL sẽ quay các bit sang trái , LSB sẽ được thay bằng MSB . Còn CF=MSB

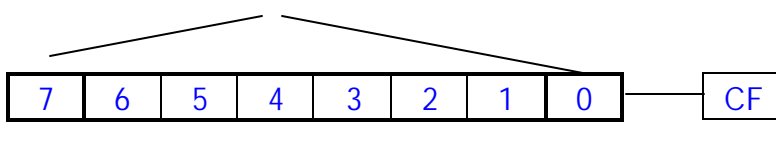


Cú pháp của ROL như sau :

ROL destination,1

ROL destination,CL

Quay phải (rotate right) = ROR sẽ quay các bit sang phải , MSB sẽ được thay bằng LSB . Còn CF=LSB



Cú pháp của lệnh quay phải là

```
ROR destination,1
ROR destination,CL
```

Trong các lệnh quay phải và quay trái CF chứa bit bị quay ra ngoài .

Ví dụ sau này cho thấy cách nhả các bit trên một byte hoặc 1 từ mà không làm thay đổi nội dung của nó.

Ví dụ : Dùng ROL để nhả số bit 1 trên BX mà không thay đổi nội dung của nó. Kết quả cất trên AX .

Giải :

```
XOR AX,AX      ; xoá AX
MOV CX,16      ; số lần lặp = 16 ( một từ )
```

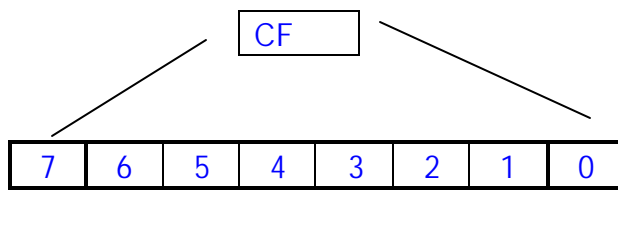
TOP:

```
ROL BX,1       ; CF = bit quay ra
JNC NEXT       ; nếu CF =0 thì nhảy đến vòng lặp
INC AX         ; ngược lại (CF=1) , tăng AX
```

NEXT:

```
LOOP TOP
```

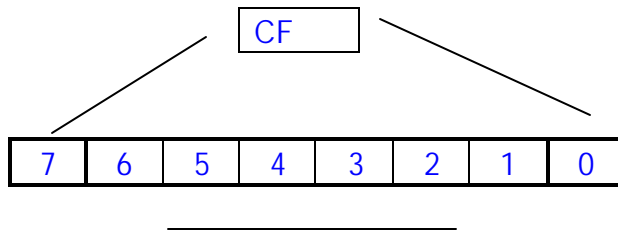
Quay trái qua carry (rotate through carry left) = RCL . Lệnh này giống với lệnh ROL chỉ khác là carry nằm giữa MSB và LSB trong vòng kín của các bit



Cú pháp của lệnh RCL như sau :

```
RCL destination,1
RCL destination,CL
```

Quay phải qua carry (rotate through carry right) = RCR . Lệnh này giống với lệnh ROR chỉ khác là carry nằm giữa MSB và LSB trong vòng kín của các bit .



Cú pháp của lệnh RCR như sau :

RCR destination,1

RCR destination,CL

Ví dụ : Giả sử DH = 8Ah ,CF=1 và CL=3 . Tìm giá trị của DH,CF sau khi lệnh

RCR DH,CL được thực hiện

Giải :

	CF	DH
Giá trị ban đầu	1	10001010
Sau khi quay 1 lần	0	11000101
Sau khi quay 2 lần	1	01100010
Sau khi quay 3 lần	0	10110001=B1H

Ảnh hưởng của lệnh quay lên các cờ

SF,PF và ZF phản ánh kết quả

CF-bit cuối cùng được dịch ra

OF=1 nếu kết quả thay đổi dấu vào lần quay cuối cùng

Ứng dụng : Đảo ngược các bit trên một byte hoặc 1 từ . Ví dụ AL =10101111 thì sau khi đảo ngược AL=11110101 .

Có thể lặp 8 lần công việc sau :**Dùng SHL để dịch bit MSB ra CF , Sau đó dùng RCR để đưa nó vào BL .**

Now mình làm việc này như sau :

MOV CX,8 ; số lần lặp

REVERSE :

SHL AL,1 ; dịch MSB ra CF

RCR BL,1 ; đưa CF (MSB) vào BL

LOOP REVERSE

MOV AL,BL ; AL chứa các bit đã đảo ngược

4.4 Xuất nhập số nhị phân và số hex

Các lệnh dịch và quay thông thường được sử dụng trong các hoạt động xuất nhập số nhị phân và số hex.

4.4.1 Nhập số nhị phân

Giải thích cần nhập một số nhị phân từ bàn phím, kết thúc là phím CR. Số nhị phân là một chuỗi các bit 0 và 1. Mỗi một ký tự gõ vào phải được biến đổi thành một bit giá trị (0 hoặc 1) rồi tích lũy chúng trong 1 thanh ghi. Thuật toán sau đây sẽ nối một số nhị phân từ bàn phím vào các nội dung trên thanh ghi BX.

Clear BX

input a character ('0' or '1')

WHILE character <> CR DO

convert character to binary value

left shift BX

insert value into LSB of BX

input a character

END_WHILE

Nhìn mã thối hiện thuật toán trên như sau :

```

        XOR  BX,BX      ; Xoá BX
        MOV  AH,1       ; ham nối 1 ký tự
        INT  21h        ; ký tự trên AL
WHILE_:
        CMP  AL,0DH     ; ký tự là CR?
        JE   END_WHILE  ; nếu , kết thúc
        AND  AL,0Fh     ; convert to binary value
        SHL  BX,1       ; dịch trái BX 1 bit
        OR   BL,AL      ; nối giá trị vào BX
        INT  21h        ; nối ký tự tiếp theo
        JMP  WHILE_     ; lặp
END_WHILE:
    
```

4.4.2 Xuất số nhị phân

Giải thích cần xuất số nhị phân trên BX (16 bit). Thuật toán có thể viết như sau

FOR 16 times DO

rotate left BX (put MSB into CF)

IF CF=1

```

        then
        output '1'
        else
        output '0'
    END_IF
END_FOR

```

Nó in mã nhị phân của số nhô phân cõ thể xem nhô bài tập .

4.4.3 Nhập số HEX

Nhập số hex bao gồm các số từ 0 đến 9 và các ký từ A đến F . Kết quả chõa trong BX .

Nếu chõn giao diện chúng ta giao sõ như sau :

- chõ cõ ký từ hoa ñõ cõ dung
- ngõ ñõ dung nhập vào không quá 4 ký từ hex

Thuật toán nhõ sau :

Clear BX

input character

WHILE character<> CR DO

convert character to binary value(4 bit)

left shift BX 4 times

insert value into lower 4 bits of BX

input character

END_WHILE

Nó in mã cõ thể viết nhõ sau :

```
        XOR    BX,BX        ; clear BX
        MOV    CL,4         ; counter for 4 shift
        MOV    AH,1         ; input character
                                ; function
        INT 21h             ; input a chracter AL
WHILE_:
        CMP    AL,0Dh       ; character <>CR?
        JE     END_WHILE_   ; yes , exit
; convert character to binary value
        CMP    AL,39H       ; a character?
        JG     LETTER       ; no , a letter
; input is a digit
        AND    AL,0Fh       ; convert digit to binary
value
        JMP    SHIFT        ; go to insert BX
LETTER:
        SUB    AL,37h       ; convert letter to binary
value
SHIFT:
        SHL    BX,CL        ; make room for new value
; insert value into BX
        OR     BL,AL         ; put value into low 4 bits of
BX
        INT 21H             ; input a character
        JMP    WHILE_
END_WHILE:
```

4.4.4 Xuất số HEX

Nếu xuất số hex trên BX (16 bit = 4 digit hex) rồi thì bắt đầu từ 4 bit bên trái , chuyển chúng thành một số hex rồi xuất ra màn hình .

Thuật toán như sau :

FOR 4 times DO
 move BH to DL



Shift DL 4 times to right

IF DL < 10

then

convert to character in '0' ... '9'

else

convert to character in 'A'..'F'

END_IF

output character (HAM 2 NGAT 21H)

rotate BX left 4 times

END_FOR

Phần code cho thuật toán này xem như bài tập .

Chương 5 : NGĂN XẾP VÀ THỦ TỤC

Ngăn xếp (stack segment) trong chương trình là một vùng nhớ có giới hạn thời gian và nội dung. Trong chương này chúng ta sẽ xem xét cách tổ chức stack và sử dụng nó để thực hiện các thủ tục (procedure) .

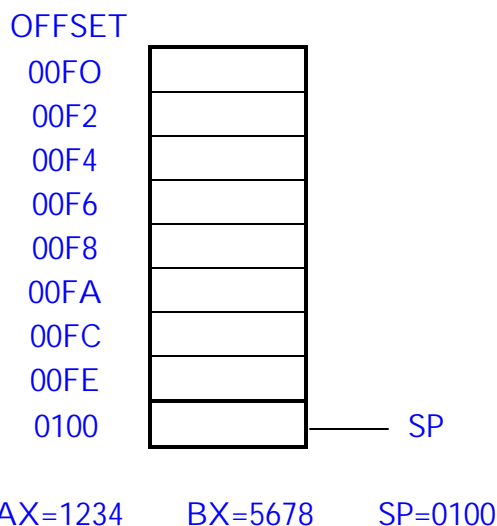
5.1 Ngăn xếp

Ngăn xếp là cấu trúc dữ liệu 1 chiều . Nhiều nội dung khác nhau của dữ liệu được đưa vào và lấy ra khỏi stack tại đầu cuối của stack theo nguyên tắc LIFO (last in first out) . Vị trí tại nơi dữ liệu được đưa vào hay lấy ra gọi là đỉnh của ngăn xếp (top of stack) . Có thể hình dung stack nhờ một chồng đĩa . Đĩa đưa vào sau cùng nằm tại đỉnh của chồng đĩa . Khi lấy ra , đĩa trên cùng sẽ được lấy ra trước . Một chương trình phải dành ra một khối nhớ cho ngăn xếp . Chúng ta dùng chữ đại

.STACK 100h

để khai báo kích thước vùng stack là 256 bytes .

Khi chương trình được dịch và nạp vào bộ nhớ thành ghi SS (stack segment) sẽ chứa địa chỉ của ngăn xếp . Con SP (stack pointer) chứa địa chỉ đỉnh của ngăn xếp . Trong khai báo stack 100h trên đây , SP nhận giá trị 100h . Nhiều nay có nghĩa là stack trống rỗng (empty) nhờ hình 4-1.



Hình 4.1 : STACK EMPTY

Lehnh PUSH và PUSHF

Nếu thêm một từ mới vào stack chúng ta dùng lệnh :

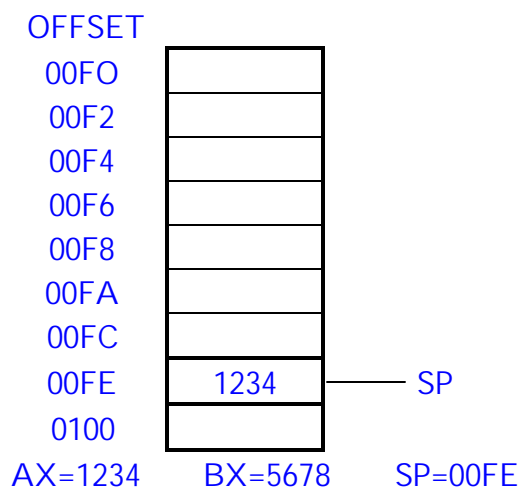
PUSH source ; đưa một thanh ghi hoặc từ nhỏ 16 bit vào stack

Ví dụ PUSH AX . Khi lệnh này được thực hiện thì :

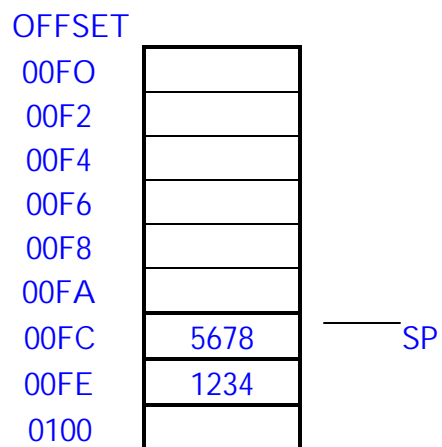
- SP giảm đi 2
- một bản copy của toàn hàng nguồn được chuyển đến địa chỉ SS:SP còn toàn hàng nguồn không thay đổi .

Lệnh PUSHF không có toàn hàng . Nội dung của nó được đưa vào stack .

Sau khi thực hiện lệnh PUSH thì SP sẽ giảm 2 . Hình 5-2 và 5-3 cho thấy lệnh PUSH làm thay đổi trạng thái stack như thế nào .



Hình 5-2 : STACK sau khi thực hiện lệnh PUSH AX



Hình 5-3 : STACK sau khi thực hiện lệnh PUSH BX

Leñh POP vaøPOPF

Để lấy số liệu tại đỉnh stack ra khỏi stack ,chúng ta dùng lệnh :

POP destination ; lấy số liệu tài nguyên stack ra destination

Destination có thể là 1 thanh ghi hoặc tổng cộng 16 bit. Ví dụ:

POP BX ; Lấy số liệu trong stack ra thành ghi BX .

Khi thực hiện lệnh POP :

- nội dung của nhanh stack (nửa chẵn SS:SP) được di chuyển nên ních.
- SP tăng 2

Leaving POPF seiláy ñanh stack ñoã vào thanh ghi cô.

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

Lưu ý: Lệnh PUSH, POP là lệnh 2 bytes vì vậy các lệnh 1 byte nhỏ :

PUSH DL ; lệnh không hộp lea

PUSH 2 ; lệnh không hộp lea

Ngoài chồi nạng lều trổ số liệu và nhà chạ của chồong trình do người sôidùng viết , stack con nồic dung bôii heñhiều hanh ñeñlều trổitrang thail của chồong trình chính khi coingat .

5.2 Ứng dụng của stack

Bởi vì nguyên tắc làm việc của stack là LIFO nên các nội dung được lấy ra khỏi stack contra tới ngược lại với trả tới mà chúng được nhả vào stack . Chương trình sau đây sẽ nhả một chuỗi ký tự rồi in chúng trên dòng mới với trả tới ngược lại .

Thuật toán cho chương trình nhỏ sau :

Display a '?'

Initialize count to 0

Read a character

WHILE character is not CR DO

PUSH chracter onto stack

Increment count

Read a character

```
END_WHILE;
```

Goto a new line

FOR count times DO

POP a chracter from the stack

Display it ;

END_FOR

Sau đây là chương trình :

```

TITLE PGM5-1 : REVERSE INPUT
.MODEL      SMALL
.STACK     100H
.CODE
MAIN       PROC
; in dấu nhắc
        MOV  AH,2
        MOV  DL,'?'
        INT  21H
; xoá biến nhớ CX
        XOR  CX,CX
; đọc 1 ký tự
        MOV  AH,1
        INT  21H
; Trong khi character không phải là CR
WHILE_:
        CMP  AL,0DH
        JE   END_WHILE
; cất AL vào stack tăng biến nhớ
        PUSH AX ; đây AX vào stack
        INC  CX ; tăng CX
; đọc 1 ký tự
        INT  21h
        JMP  WHILE_
END_WHILE:
; Xuống dòng mới
        MOV  AH,2
        MOV  DL,0DH
        INT  21H
        MOV  DL,0AH
        INT  21H
        JCXZ EXIT ; thoát nếu CX=0 ( không có ký tự nào nữa nhập)
; lặp CX lần
TOP:
; lấy ký tự từ stack
        POP  DX
; xuất nó
        INT  21H
        LOOP TOP ; lặp nếu CX>0
    
```

```
; end_for
EXIT:
    MOV AH,4CH
    INT 21H
MAIN ENDP
END MAIN
```

Giải thích thêm về chương trình : vì số ký tự nhập là không biết vì vậy dùng thanh ghi CX để đếm số ký tự nhập . CX cũng dùng cho vòng FOR để xuất các ký tự theo thứ tự ngược lại . Mặc dù ký tự chỉ có ở trên AL nhưng phải lấy cả thanh ghi AX vào stack . Khi xuất ký tự chúng ta dùng lệnh POP DX để lấy nội dung trên stack ra. Mã ASCII của ký tự ở trên DL , sau đó gọi INT 21h để xuất ký tự .

5.3 Thuật ngữ (Procedure)

Trong chương 3 chúng ta đã biết về lập trình top-down . Ý tưởng này có nghĩa là một bài toán nguyên thủy được chia thành các bài toán con mà chúng ta giải quyết hơn bài toán nguyên thủy . Trong các ngôn ngữ cấp cao người ta dùng thuật ngữ để giải các bài toán con , và chúng ta cũng làm như vậy trong hộp gọi . Nhờ vậy là một chương trình hộp gọi có thể được xây dựng bằng các thuật ngữ .

Một thuật ngữ gọi là thuật ngữ chính sẽ chứa nội dung chủ yếu của chương trình . Nếu thực hiện một công việc nào đó, thuật ngữ chính gọi (CALL) một thuật ngữ con . Thuật ngữ con cũng có thể gọi một thuật ngữ con khác .

Khi một thuật ngữ gọi một thuật ngữ khác , điều khiển được chuyển tới (control transfer) thuật ngữ được gọi và các lệnh của thuật ngữ được gọi sẽ được thi hành . Sau khi thi hành hết các lệnh trong nó, thuật ngữ được gọi sẽ trả lại điều khiển (return control) cho thuật ngữ gọi nó. Trong ngôn ngữ cấp cao , lập trình viên không biết và không thể biết cấu trúc của việc chuyển và trả lại điều khiển giữa thuật ngữ chính và thuật ngữ con. Nhưng trong hộp gọi có thể thấy rõ cấu trúc này (xem phần 5.4) .

Khai báo thuật ngữ

Cú pháp của lệnh tạo một thuật ngữ như sau :

```
name PROC          type
; body of procedure
RET
name ENDP
```

Name do người dùng định nghĩa là tên của thuật ngữ .

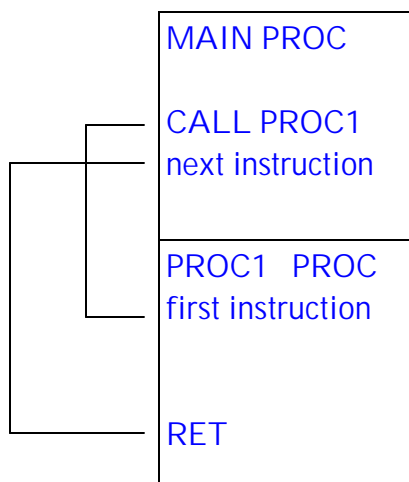
Type có thể là NEAR (có thể không khai báo) hoặc FAR .

NEAR coi như là thủ tục nội gọi nằm cùng một đoạn với thủ tục gọi . FAR coi như là thủ tục nội gọi và thủ tục gọi nằm khác đoạn . Trong phần này chúng ta sẽ chỉ mô tả thủ tục NEAR .

Lệnh RET trả lại cho thủ tục gọi . Tất cả các thủ tục phải kết thúc bởi RET trở về thủ tục chính .

Chú thích cho thủ tục : Nếu người đọc để hiểu thủ tục người ta thường sẽ dùng chú thích cho thủ tục dưới dạng sau :

- ; (mô tả các công việc mà thủ tục thi hành)
- ; input: (mô tả các tham số có tham gia trong chương trình)
- ; output : (cho biết kết quả sau khi chạy thủ tục)
- ; uses : (liệt kê danh sách các thủ tục mà nó gọi)



Hình 5-1 : Gọi thủ tục và trở về

5.4 CALL & RETURN

Lệnh CALL nội dung để gọi một thủ tục . Có 2 cách gọi một thủ tục là gọi trực tiếp và gọi gián tiếp .

CALL name ; gọi trực tiếp thủ tục có tên là name

CALL address-expression ; gọi gián tiếp thủ tục trong đó address-expression chỉ định một thanh ghi hoặc một vị trí nhớ mà nó chứa địa chỉ của thủ tục .

Khi lệnh CALL nội dung thi hành thì :

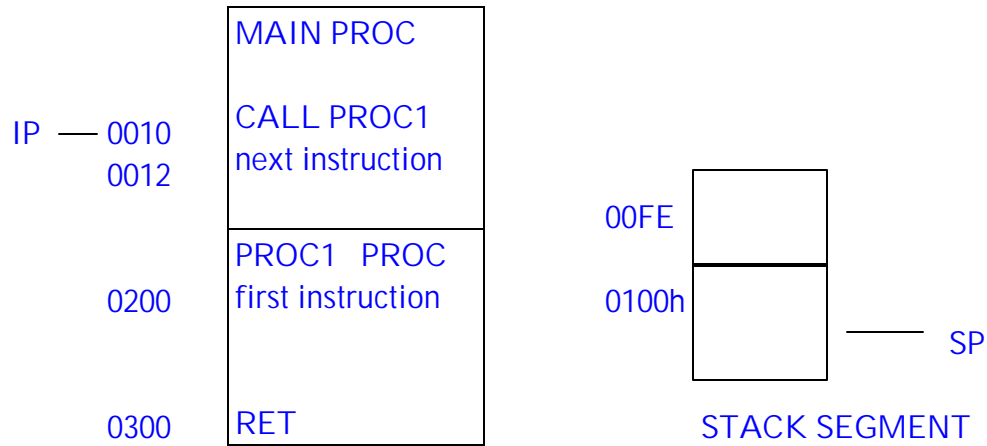
- Địa chỉ quay về của thủ tục gọi nội dung cất vào stack . Địa chỉ này chính là offset của lệnh tiếp theo sau lệnh CALL .
- IP lấy địa chỉ offset của lệnh nêu trên thủ tục nội dung gọi , cộng với địa chỉ hiện nội dung chuyển đến thủ tục .

Nếu trả lại cho thủ tục chính , lệnh

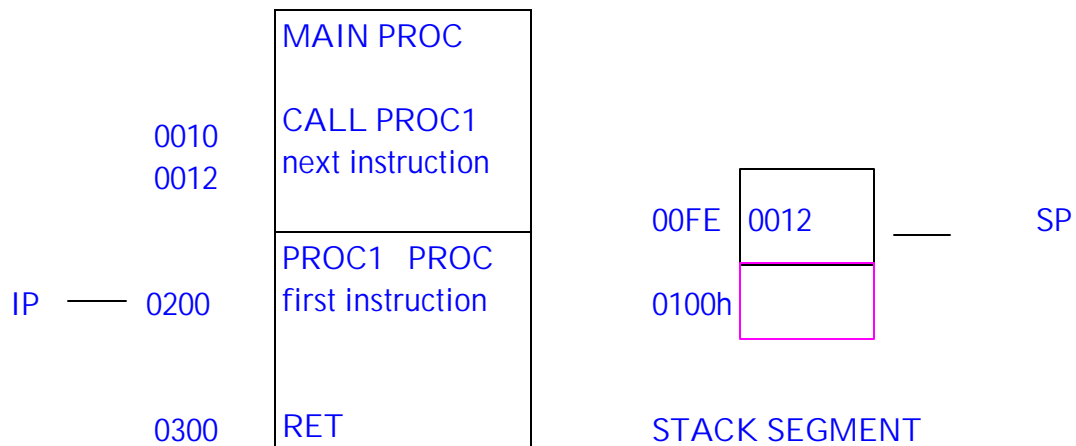
RET pop-value

thực hiện . Pop-value (một số nguyên N) là tùy chọn . Nếu với thuật ngữ NEAR ,
lệnh RET sẽ lấy giá trị trong SP đưa vào IP . Nếu pop-value là ra một số N thì
 $IP = SP + N$

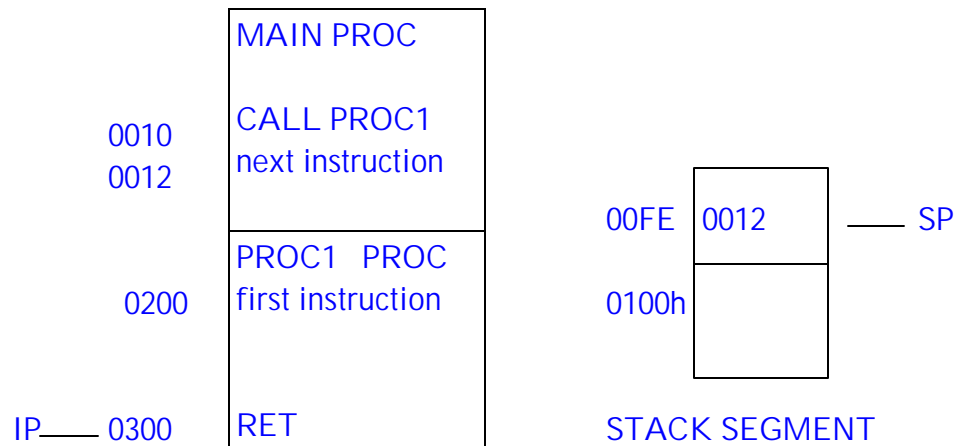
Trong cả 2 trường hợp thì CS:IP chứa địa chỉ trở về chương trình gọi và nếu
khiến thực thể cho chương trình gọi (xem hình 5-2)



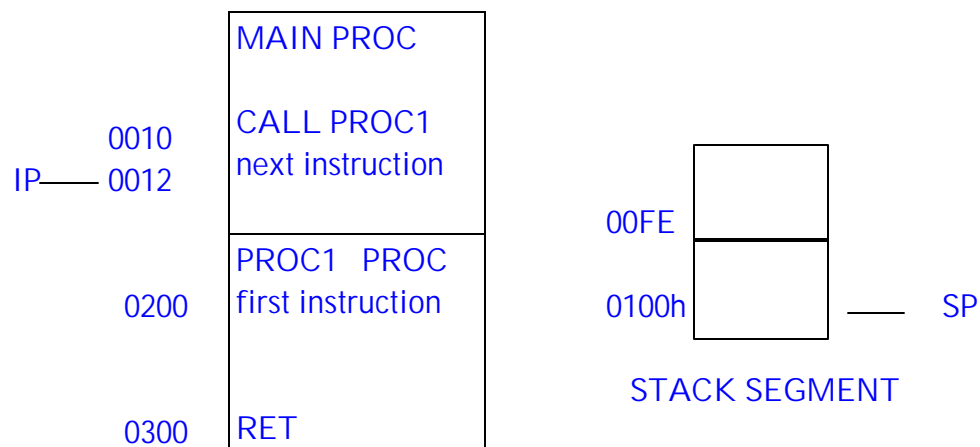
Hình 5-2 a : Trước khi CALL



Hình 5-2 b : Sau khi CALL



Hình 5-2 c : Trước khi RET



Hình 5-2 d : Sau khi RET

5.5 Ví dụ về thuật toán

Chúng ta sẽ viết chương trình tính tích của 2 số nguyên A và B bằng thuật toán cộng (ADD) và dịch (SHIFT)

Thuật toán nhỏ sau :

```

Product = 0
REPEAT
    IF lsb of B is 1
    THEN
        product=product+A
    END_IF
    shift left A
    shift right B
UNTIL B=0

```

Trong chương trình sau này chúng ta sẽ mô tả thuật toán nhân với tên là MULTIPLY. Chương trình chính không cần nhập xuất, thay vào đó chúng ta dùng DEBUG để nhập xuất.

```

TITLE PGM5-1: MULTIPLICATION BY ADD AND SHIFT
.MODEL    SMALL
.STACK 100H
.CODE
MAIN PROC
; thực hiện bằng DEBUG. Đặt A = AX, B=BX
CALL MULTIPLY
;DX chứa kết quả
    MOV AH,4CH
    INT 21H
MAIN ENDP
MULTIPLY PROC
; input : AX=A, BX=B, AX và BX có giá trị trong khoảng 0...FFH
; output : DX= kết quả
    PUSH AX
    PUSH BX
    XOR DX,DX
REPEAT:
; Nếu lsb của B =1
    TEST BX,1    ;lsb=1?
    JZ END_IF    ; không, nhảy đến END_IF
; thì
    ADD DX,AX    ; DX=DX+AX
END_IF :
    SHL AX,1    ; dịch trái AX 1 bit
    SHR BX,1    ; dịch phải BX 1 bit
; cho đến khi BX=0

```

```

JNZ REPEAT ; nếu BX chứa bằng 0 thì lặp
POP BX ; lấy lại BX
POP AX ; lấy lại AX
RET ; trả về khi kết thúc chương trình chính
MULTIPLY ENDP
END MAIN

```

Sau khi dịch chương trình, có thể dùng DEBUG để chạy thử nội dung chương trình và kiểm tra giá trị của AX và BX.

Dùng lệnh **U(unassembler)** để xem nội dung của bộ nhớ tương ứng với các lệnh hộp ngỏ.

Có thể xem nội dung của stack bằng lệnh D(dump)

DSS:F0 FF ; xem 16 bytes trên cùng của stack

Dùng lệnh **G(go) offset** để chạy tiếp nhóm lệnh từ CS:IP hiện hành CS:offset.

Trong quá trình chạy DEBUG có thể kiểm tra nội dung các thanh ghi. Lưu ý rằng biết nên IP để xem cách chuyển và trả về khi gọi và thực hiện một thủ tục.

Chương 6 : LỆNH NHÂN VÀ CHIA

Trong chương 5 chúng ta đã nói đến các lệnh dịch mà chúng có thể dùng để nhân và chia với hệ số 2. Trong chương này chúng ta sẽ nói đến các lệnh nhân và chia một số bất kỳ.

Quá trình xử lý của lệnh nhân và chia đối với số có dấu và số không dấu là khác nhau do đó có lệnh nhân có dấu và lệnh nhân không dấu.

Một trong những ứng dụng thông dụng nhất của lệnh nhân và chia là thực hiện các thao tác nhập xuất thập phân. Trong chương này chúng ta sẽ viết thuật toán cho nhập xuất thập phân mà chúng được sử dụng nhiều trong các hoạt động xuất nhập thông tin.

6.1 Lệnh MUL và IMUL

Nhân có dấu và nhân không dấu

Trong phép nhân nhị phân số có dấu và số không dấu phải được phân biệt một cách rõ ràng. Ví dụ chúng ta muốn nhân hai số 8 bit 1000000 và 1111111. Trong diễn dịch không dấu, chúng là 128 và 255. Tích số của chúng là 32640 = 0111111110000000b. Trong diễn dịch có dấu, chúng là -128 và -1. Do đó tích của chúng là 128 = 0000000010000000b.

Vì nhân có dấu và không dấu dẫn đến các kết quả khác nhau nên có 2 lệnh nhân:

MUL (multiply) nhân không dấu

IMUL (integer multiply) nhân có dấu

Các lệnh này nhân 2 toán hạng byte hoặc từ. Nếu 2 toán hạng byte được nhân với nhau thì kết quả là một từ 16 bit. Nếu 2 toán hạng từ được nhân với nhau thì kết quả là một double từ 32 bit. Cú pháp của chúng là:

MUL source ;

IMUL source ;

Toán hạng nguồn là thanh ghi hoặc vị trí nhớ không được là một hằng

Phép nhân kiểu byte

Nếu với phép nhân mà toán hạng là kiểu byte thì

AX=AL*SOURCE ;

Phép nhân kiểu từ

Nếu với phép nhân mà toán hạng là kiểu từ thì

DX:AX=AX*SOURCE

Anh hưởng của các lệnh nhân lên các cờ.

SF,ZF,AF,PF : không xác định

sau lệnh MUL CF/OF = 0 nếu nửa trên của kết quả (DX) bằng 0
= 1 trong các trường hợp khác

sau lệnh IMUL CF/OF = 0 nếu nửa trên của kết quả có bit dấu
giống với bit dấu của nửa thấp.
= 1 trong các trường hợp khác

Sau đây chúng ta sẽ lấy vài ví dụ.

Ví dụ 1 : Giả sử rằng AX=1 và BX=FFFFh

INSTRUCTION	Dec product	Hex Product	DX	AX	CF/OF
MUL BX	65535	0000FFFF	0000	FFFF	0
IMUL BX	-1	FFFFFFFF	FFFF	FFFF	0

Ví dụ 2 : Giả sử rằng AX=FFFFh và BX=FFFFh

INSTRUCTION	Dec product	Hex Product	DX	AX	CF/OF
MUL BX	4294836225	FFFE0001	FFFE	0001	1
IMUL BX	1	00000001	0000	0001	0

Ví dụ 3 : Giả sử rằng AX=0FFFh

INSTRUCTION	Dec product	Hex Product	DX	AX	CF/OF
MUL AX	16769025	00FFE001	00FF	E001	1
IMUL AX	16769025	00FFE001	00FF	E001	1

Ví dụ 4 : Giả sử rằng AX=0100h và CX=FFFFh

INSTRUCTION	Dec product	Hex Product	DX	AX	CF/OF
MUL CX	16776960	00FFFF00	00FF	FF00	1
IMUL CX	-256	FFFFFF00	FFFF	FF00	0

Ví dụ 5 : Giả sử rằng AL=80h và BL=FFh

INSTRUCTION	Dec product	Hex Product	AH	AL	CF/OF
MUL BL	128	7F80	7F	80	1
IMUL BL	128	0080	00	80	1

6.2 Ứng dụng nâng cao của lệnh MUL và IMUL

Sau này chúng ta sẽ lấy một số ví dụ minh họa việc sử dụng lệnh MUL và IMUL trong chương trình.

Ví dụ 1 : Chuyển toán chương trình sau trong ngôn ngữ cấp cao thành mã hợp ngữ : $A = 5x A - 12x B$. Giả sử rằng A và B là 2 biến từ và không xảy ra sai lầm.

Code :

```
MOV  AX,5      ; AX=5
IMUL A         ; AX=5xA
MOV  A,AX      ; A=5xA
MOV  AX,12     ; AX=12
IMUL B         ; AX=12xB
SUB  A,AX      ; A=5xA-12xB
```

Ví Dụ 2 : viết thuật toán FACTORIAL để tính $N!$ cho một số nguyên dương. Thuật toán phải chứa N trên CX và trả về $N!$ trên AX. Giả sử không có sai lầm.

Giải : Định nghĩa của $N!$ là

$N! = 1$ nếu $N=1$

$= N \times (N-1) \times (N-2) \times \dots \times 1$ nếu $N > 1$

Thuật toán để tính $N!$ như sau :

Product = 1

Term = N

FOR N times DO

Product = product x term

term=term -1

ENDFOR

Code :

```
FACTORIAL PROC
; computes N!
; input : CX=N
; output : AX=N!
        MOV  AX,1      ; AX=1
        MOV  CX,N      ; CX=N
TOP:
        MUL  CX         ; Product = product x term
        LOOP TOP        ;
        RET
FACTORIAL ENDP
```

6.3 Lệnh DIV và IDIV

Cũng nhớ lệnh nhân, coi 2 lệnh chia DIV và IDIV cho số không dấu và cho số có dấu. Cú pháp của chúng là:

DIV divisor

IDIV divisor

Toàn hàng byte

Lệnh chia toàn hàng byte sẽ chia số bù chia 16 bit (dividend) trên AX cho số chia (divisor) là 1 byte. Divisor phải là 1 thanh ghi 8 bit hoặc 1 byte nhớ.

Thông số ở trên AL còn số dư trên AH.

Toàn hàng từ

Lệnh chia toàn hàng từ sẽ chia số bù chia 32 bit (dividend) trên DX:AX cho số chia (divisor) là 1 từ. Divisor phải là 1 thanh ghi 16 bit hoặc 1 từ nhớ. Thông số ở trên AX còn số dư trên DX.

Anh hưởng của các cờ: các cờ có trạng thái không xác định.

Divide Overflow

Khi thực hiện phép chia kết quả có thể không chứa hết trên AL hoặc AX nếu số chia bé hơn rất nhiều so với số bù chia. Trong trường hợp này trên màn hình sẽ xuất hiện thông báo : "**Divide overflow**"

Ví dụ 1: Giả sử DX = 0000h, AX = 0005h và BX = 0002h

Instruction	Dec Quotient	Dec Remainder	AX	DX
DIV BX	2	1	0002	0001
IDIV BX	2	1	0002	0001

Ví dụ 1: Giả sử DX = 0000h, AX = 0005h và BX = FFFEh

Instruction	Dec Quotient	Dec Remainder	AX	DX
DIV BX	0	5	0000	0005
IDIV BX	-2	1	FFFE	0001

Ví dụ 3: Giả sử DX = FFFFh, AX = FFFBh và BX = 0002h

Instruction	Dec Quotient	Dec Remainder	AX	DX
IDIV BX	-2	-1	FFFE	FFFF
DIV BX	OVERFLOW			

Ví dụ 4 : Giả sử AX = 00FBh và BL = FFh

Instruction	Dec Quotient	Dec Remainder	AX	DX
DIV BL	0	251	FB	00
IDIV BL	OVERFLOW			

6.4 Môi trường đầu của số bị chia

Phep chia với toàn hàng từ

Trong phép chia với toàn hàng từ, số bị chia phải nằm trên DX:AX ngay cả khi số bị chia có thể nằm trên AX. Trong trường hợp này, cần phải sửa soạn như sau

- Nếu với lệnh DIV, DX phải bỏ xoá
- Nếu với lệnh IDIV, DX phải ở môi trường đầu của AX. Lệnh CWD (Convert Word to Doubleword) sẽ thực hiện việc này.

Ví dụ : Chia -1250 cho 7

```
MOV AX,-1250 ; AX= -1250
CWD          ; môi trường đầu của AX vào DX
MOV BX,7     ; BX=7
IDIV BX      ; chia DX:AX cho BX , kết quả trên AX , số dư
              ; trên DX
```

Phep chia với toàn hàng byte

Trong phép chia với toàn hàng byte, số bị chia phải nằm trên AX ngay cả khi số bị chia có thể nằm trên AL. Trong trường hợp này, cần phải sửa soạn như sau

- Nếu với lệnh DIV, AH phải bỏ xoá
- Nếu với lệnh IDIV, AH phải ở môi trường đầu của AL. Lệnh CBW (Convert Byte to Doublebyte) sẽ thực hiện việc này.

Ví dụ : Chia một số có dấu trong biến byte XBYTE cho -7

```
MOV AL,XBYTE ; AL giữ số bị chia
CBW          ; môi trường đầu của AL vào AH
MOV BL,-7    ; BX= -7
IDIV BL      ; chia AX cho BL , kết quả trên AL , số dư
              ; trên AH
```

Khoảng có dấu của số bị chia ảnh hưởng bởi lệnh CWD và CBW.

6.5 Thuituc nhap xuat so thap phan

Mặc dù trong PC tất cả số liệu đều được biểu diễn dưới dạng binary . Nhưng việc biểu diễn dưới dạng thập phân sẽ thuận tiện hơn cho người dùng . Trong phần này chúng ta sẽ viết các thuật ngữ nhập xuất số thập phân .

Khi nhập số liệu , nếu chúng ta gọi 21543 chẳng hạn thì thực chất là chúng ta gõ vào một chuỗi ký tự, bên trong PC , chúng được biến đổi thành các giá trị nhớ phần tổng cộng của 21543 . Ngược lại khi xuất số liệu , nội dung nhớ phần của thanh ghi hoặc bộ nhớ phải được biến đổi thành một chuỗi ký tự để biểu diễn một số thập phân trước khi chúng được in ra .

Xuat số thập phân (Decimal Output)

Chúng ta sẽ viết một thuật ngữ OUTDEC nhận nội dung của thanh ghi AX và in ra một số nguyên thập phân có dấu . Nếu $AX > 0$, OUTDEC sẽ in nội dung của AX dưới dạng thập phân . Nếu $AX < 0$, OUTDEC sẽ in dấu trừ (-) , thay $AX = -AX$ (biến thành số dương) rồi in số dương này sau dấu trừ (-) . Nhờ vậy là trong cái 2 trường hợp , OUTDEC sẽ in giá trị thập phân tổng cộng của một số dương . Sau đây là thuật toán :

Algorithm for Decimal Output

1. IF $AX < 0$ / AX hold output value /
2. THEN
3. PRINT a minus sign
4. Replace AX by its two's complement
5. END_IF
6. Get the digits in AX's decimal representation
7. Convert these digits to characters and print them .

Nếu hiểu chi tiết bước 6 cần phải làm việc gì , chúng ta giải thích nội dung của AX là một số thập phân , ví dụ 24618 thập phân . Có thể lấy các digits thập phân của 24618 bằng cách chia lặp lại cho 10 để theo thuật ngữ sau :

Divide 24618 by 10 . Quotient = 2461 , remainder = 8
 Divide 2461 by 10 . Quotient = 246 , remainder = 1
 Divide 246 by 10 . Quotient = 24 , remainder = 6
 Divide 24 by 10 . Quotient = 2 , remainder = 4
 Divide 2 by 10 . Quotient = 0 , remainder = 2

Các digits thu được bằng cách lấy các số dư theo trật tự ngược lại .

Bước 7 của thuật toán có thể thực hiện bằng vòng FOR như sau :

```

FOR count times DO
    pop a digit from the stack
    convert it to a character
    output the character
END_FOR

```

Code cho thuật toán OUTDEC như sau :

```

OUTDEC    PROC
; Print  AX  as a signed decimal integer
; input : AX
; output : none
        PUSH        AX    ; save registers
        PUSH        BX
        PUSH        CX
        PUSH        DX
; IF AX<0
        OR          AX,AX    ; AX < 0 ?
        JGE         @END_IF1 ; NO , AX>0
; THEN
        PUSH        AX    ; save AX
        MOV         DL,'-' ; GET '-'
        MOV         AH,2
        INT         21H    ; print '-'
        POP         AX     ; get AX back
        NEG         AX     ; AX = -AX
@END_IF1:
; get decimal digits
        XOR         CX,CX    ; clear CX for counts digit
        MOV         BX,10d    ; BX has divisor
@REPEAT1:
        XOR         DX,DX    ; clear DX
        DIV         BX       ; AX:BX ; AX = quotient , DX=
remainder
        PUSH        DX       ; push remainder onto stack
        INC         CX       ; increment count
;until
        OR          AX,AX    ; quotient = 0?
        JNE         @REPEAT1 ; no keep going
; convert digits to characters and print
        MOV         AH,2     ; print character function

```

```

; for count times do
@PRINT_LOOP:
    POP    DX          ; digits in DL
    OR     DL,30h      ; convert digit to character
    INT    21H         ; print digit
    LOOP   @PRINT_LOOP
;end_for
    POP    DX          ; restore registers
    POP    CX
    POP    BX
    POP    AX
    RET
OUTDEC    ENDP

```

Toán tử giải INCLUDE

Chúng ta có thể thay nội OUTDEC bằng cách đặt nó bên trong một chương trình ngắn và chạy chương trình trong DEBUG . Nếu nó thực hiện OUTDEC vào trong chương trình mà không cần gọi nó, chúng ta dùng toán tử giải INCLUDE với cú pháp như sau :

```
INCLUDE filespec
```

ở đây filespec dùng để chỉ đường dẫn tập tin (bao gồm cả đường dẫn của nó) .

Ví dụ tập tin chứa OUTDEC là PGM6_1.ASM ở địa chỉ A: . Chúng ta có thể viết :

```
INCLUDE A:\PGM6_1.ASM
```

Sau đây là chương trình để test thực hiện OUTDEC

```
TITLE PGM6_2 : DECIMAL OUTPUT
```

```
.MODEL SMALL
```

```
.STACK 100h
```

```
.CODE
```

```

MAIN PROC
CALL OUTDEC
MOV AH,4CH
INT 21H

```

```
MAIN ENDP
```

```
INCLUDE A:\PGM6_1.ASM
```

```
END MAIN
```

Sau khi dịch , chúng ta dùng DEBUG nhập số liệu và chạy chương trình .

Nhập Thập phân (Decimal input)

Nếu nhập số thập phân chúng ta cần biến nối một chuỗi các digits ASCII thành biểu diễn nhị phân của một số nguyên thập phân . Chúng ta sẽ viết thuật ngữ INDEC để làm việc này .

Trong thuật ngữ OUTDEC chúng ta chia lại cho 10d . Trong thuật ngữ INDEC chúng ta sẽ nhân lại với 10d .

Decimal Input Algorithm

```
Total = 0
read an ASCII digit
REPEAT
    convert character to a binary value
    total = 10x total +value
    read a chracter
UNTIL chracter is a carriage return
```

Ví dụ : nếu nhập 123 thì xử lý như sau :

```
total = 0
read '1'
convert '1' to 1
total = 10x 0 +1 =1
read '2'
convert '2' to 2
total = 10x1 +2 =12
read '3'
convert '3' to 3
total = 10x12 +3 =123
```

Sau này chúng ta sẽ xây dựng thuật ngữ INDEC sao cho nó chấp nhận được các số thập phân có dấu trong vùng - 32768 đến +32767 (một từ) . Chương trình sẽ in ra một dấu “?” nếu như người dùng gõ vào dấu + hoặc - , theo sau đó là một chuỗi các digit và kết thúc bằng ký tự CR . Nếu người dùng gõ vào một ký tự không phải là 0 đến 9 thì thuật ngữ sẽ nhập xuống dòng mới và bắt đầu lại từ đầu . Với những yêu cầu như trên này thuật ngữ nhập thập phân phải viết lại như sau :

```
Print a question mask
Total = 0
negative = false
Read a character
CASE character OF
    '-' : negative = true
        read a chracter
    '+' ;
read a charcter
```

```

END_CASE
REPEAT
    IF character not between '0' and '9'
        THEN
            goto beginning
    ELSE
        convert character to a binary value
        total = 10xtotal + value
    END_IF
    read a character
UNTIL character is a carriage return
IF negative = true
    then
        total = - total
    END_IF

```

Thuật toán thể hiện mã nguồn sau (ghi vào đĩa A : với tên là PGM6_2.ASM)

```

INDEC      PROC
; read a number in range -32768 to +32767
; input : none
; output : AX = binary equivalent of number
    PUSH     BX      ; Save register
    PUSH     CX
    PUSH     DX
; print prompt
@BEGIN:
    MOV      AH,2
    MOV      DL,'?'
    INT      21h      ; print '?'
; total = 0
    XOR      BX,BX      ;      CX holds total
; negative = false
    XOR      CX,CX      ;      cx holds sign
; read a character
    MOV      AH,1
    INT      21h      ; character in AL
; CASE character of
    CMP      AL,'-'      ; minus sign
    JE       @MINUS
    CMP      AL,'+'      ; Plus sign

```

```

        JE      @PLUS
        JMP     @REPEAT2 ; start processing characters
@MINUS:
        MOV     CX,1
@PLUS:
        INT     21H
@REPEAT2:
; if character is between '0' to '9'
        CMP     AL,'0'
        JNGE    @NOT_DIGIT
        CMP     AL,'9'
        JNLE    @NOT_DIGIT
; THEN convert character to digit
        AND     AL,000FH ; convert to digit
        PUSH    AX ; save digit on stack
; total =10x total + digit
        MOV     AX,10
        MUL     BX ; AX= total x10
        POP     BX ; Retrieve digit
        ADD     BX,AX ; TOTAL = 10XTOTAL + DIGIT
;read a character
        MOV     AH,1
        INT     21h
        CMP     AL,0DH
        JNE     @REPEAT
; until CR
        MOV     AX,BX ; restore total in AX
; if negative
        OR      CX,CX ; negative number
        JE      @EXIT ; no exit
;then
        NEG     AX
; end_if
@EXIT:
        POP     DX
        POP     CX
        POP     BX
        RET
; HERE if illegal character entered
@NOT_DIGIT
        MOV     AH,2

```

```

        MOV DL,0DH
        INT 21h
        MOV DL,0Ah
        INT 21h
        JMP @BEGIN
INDEC   ENDP

```

TEST INDEC

Có thể test thủ tục INDEC bằng cách tạo ra một chương trình dùng INDEC cho nhập thập phân và OUTDEC cho xuất thập phân nhỏ sau :

```

        TITLE      PGM6_4.ASM
.MODEL    SMALL
.STACK    100h
.CODE
        MAIN      PROC
; input a number
        CALL      INDEC
        PUSH      AX    ; save number

; move cursor to a new line
        MOV AH,2
        MOV DL,0DH
        INT 21h
        MOV DL,0Ah
        INT 21H
; output a number
        POP       AX
        CALL      OUTDEC

; dos exit
        MOV       AH,4CH
        INT       21H
MAIN     ENDP
INCLUDE  A:\PGM6_1.ASM    ; include outdec
INCLUDE  A:\PGM6-2.ASM    ; include indec
END MAIN

```

Chương 7: MẢNG VÀ CÁC THAO TÁC TRÊN MẢNG

Trong chương này chúng ta sẽ tìm hiểu về mảng một chiều và các kỹ thuật xử lý mảng trong Assembly. Phần còn lại của chương này sẽ trình bày các thao tác khác.

7.1 Mảng một chiều

Mảng một chiều là một danh sách các phần tử cùng loại và có thể truy cập. Có thể truy cập theo địa chỉ của phần tử đầu tiên, phần tử thứ hai, phần tử thứ ba ... Trong toán học, nếu A là một mảng thì các phần tử của mảng được ký hiệu là A[1], A[2], A[3] ... Hình vẽ dưới đây là mảng A có 6 phần tử.

Index	
1	A[1]
2	A[2]
3	A[3]
4	A[4]
5	A[5]
6	A[6]

Trong chương 1 chúng ta đã dùng toán tử giải DB và DW để khai báo mảng byte và mảng từ. Ví dụ, một chuỗi 5 ký tự có thể là MSG

MSG DB 'abcde'

hoặc một mảng từ W gồm 6 số nguyên mà giá trị ban đầu của chúng là 10,20,30,40,50 và 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). Trong mảng W thì địa chỉ cơ sở là 10. Nếu địa chỉ offset của W là 0200h thì trong bộ nhớ mảng 6 phần tử nói trên sẽ như sau :

Offset address	Symbolic address	Decimal content
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 (Duplicate)

Có thể hình dung một mảng các phần tử của nó có cùng một giá trị ban đầu bằng phép DUP như sau :

repeat_count DUP (value)
lặp lại một số (VALUE) n lần (n = repeat_count)

Ví dụ :

GAMMA DW 100 DUP (0) ; tạo một mảng 100 từ giá trị ban đầu là 0 .

DELTA DB 212 DUP (?) ; tạo một mảng 212 byte giá trị chưa xác định

DUP có thể lồng nhau , ví dụ :

LINE DB 5,4,3 DUP (2, 3 DUP (0) ,1)

tương ứng với :

LINE DB 5,4,2,0,0,0,1,2,0,0,0,1,2,0,0,0,1

Vị trí các phần tử của một mảng

Địa chỉ của một phần tử của mảng có thể được xác định bằng cách cộng một hằng số với địa chỉ cơ sở. Giả sử A là một mảng và S là số byte của một phần tử của mảng (S=1 nói với mảng byte và S=2 nói với mảng từ). Vị trí của các phần tử của mảng A có thể tính như sau :

Position	Location
1	A
2	A+1xS
3	A+2xS
.	.
.	.
.	.
N	A+(N-1)xS

Ví dụ : Trao đổi phần tử thứ 10 và thứ 25 của mảng từ W .

Phần tử thứ 10 là W[10] có địa chỉ là $W+9 \times 2 = W+18$

Phần tử thứ 25 là W[25] có địa chỉ là $W+24 \times 2 = W+48$

Vì vậy có thể trao đổi chúng như sau :

MOV AX,W+18 ; AX = W[10]

XCHG W+48,AX ; AX= W[25]

MOV W+18, AX ; complete exchange

7.2 Các chế độ địa chỉ (addressing modes)

Cách thức của ra toán hạng trong lệnh gọi là chế độ寻址 của x86. Các chế độ寻址 của x86 thông dụng là:

- Chế độ寻址 của x86 bằng thanh ghi (register mode) : toán hạng là thanh ghi
- Chế độ寻址 của x86 tức thời (immediate mode) : toán hạng là hằng số
- Chế độ寻址 của x86 trực tiếp (direct mode) : toán hạng là biến

Ví dụ:

MOV AX,0 ; AX là register mode còn 0 là immediate mode

ADD ALPHA,AX ; ALPHA là direct mode

Ngoài ra còn có 4 chế độ寻址 của x86 khác là:

- Chế độ寻址 của x86 gián tiếp bằng thanh ghi (register indirect mode)
- Chế độ寻址 của x86 cơ sở (based mode)
- Chế độ寻址 của x86 chỉ số (indexed mode)
- Chế độ寻址 của x86 chỉ số cơ sở (based indexed mode)

7.2.1 Chế độ寻址 của x86 gián tiếp bằng thanh ghi

Trong chế độ寻址 của x86 gián tiếp bằng thanh ghi, địa chỉ offset của toán hạng nằm ở trong 1 thanh ghi. Chúng ta nói rằng thanh ghi là con trỏ (pointer) của vị trí nhớ. Dạng toán hạng là [register]. Trong nội register là các thanh ghi BX, SI, DI, BP. Nói với các thanh ghi BX, SI, DI thì thanh ghi nhớ là DS. Còn thanh ghi nhớ của BP là SS.

Ví dụ: giả sử rằng SI = 100h và nội dung tại địa chỉ DS:0100h có nội dung là 1234h. Lệnh MOV AX,[SI] sẽ copy 1234h vào AX.

Giả sử rằng nội dung các thanh ghi và nội dung của bộ nhớ tổng cộng là như sau:

Thanh ghi	nội dung	offset	nội dung bộ nhớ
AX	1000h	1000h	1BACH
SI	2000h	2000h	20FFh
DI	3000h	3000h	031Dh

Ví dụ 1:

Hãy cho biết lệnh nào sau đây là hộp lý, offset nguồn và kết quả của các lệnh hộp lý.

- MOV BX,[BX]
- MOV CX,[SI]
- MOV BX,[AX]
- ADD [SI],[DI]
- INC [DI]

Lời giải :

	Source offset	Result
a.	1000h	1BACH
b.	2000h	20FFh
c.	illegal source register	(must be BX,SI,DI)
d.	illegal memory-memory add	
e.	3000h	031Eh

Ví dụ 2 : Viết đoạn mã để cộng vào AX 10 phần tử của một mảng W như hình vẽ sau :

W DW 10,20,30,40,50,60,70,80,90,100

Giai :

```

XOR  AX,AX      ; xóa AX
LEA  SI,W        ; SI trỏ tới địa chỉ ( base) của mảng W .
MOV  CX,10       ; CX chứa số phần tử của mảng
ADDITION:
ADD  AX,[SI]     ; AX=AX + phần tử hiện tại
ADD  SI,2        ; tăng con trỏ lên 2
LOOP ADDITION   ; lặp
    
```

Ví dụ 3 : Viết thuật toán để đảo ngược một mảng n tử. Nhiều nay có thể hóa là phần tử đầu tiên sẽ trở thành phần tử cuối, phần tử cuối sẽ trở thành phần tử đầu tiên ... Chúng ta sẽ dùng SI trỏ tới đầu của mảng còn BX chứa số phần tử của mảng (n tử) .

Giai : Số lần trao đổi là N/2 lần . Như vậy phần tử đầu của mảng có địa chỉ A+2x(N-1)

Đoạn mã như sau :

```

REVERSE  PROC
; input: SI= offset of array
;        BX= number of elements
; output : reverse array
        PUSH    AX      ; cất các thanh ghi
        PUSH    BX
        PUSH    CX
        PUSH    SI
        PUSH    DI
; DI chứa địa chỉ phần tử đầu
    
```

```

MOV DI,SI      ; DI trỏ tới đầu tiên của mảng
MOV CX,BX      ; CX=BX=n : số phần tử
DEC BX        ; BX=n-1
SHL BX,1       ; BX=2x(n-1)
ADD DI,BX      ; DI = 2x(n-1) + offset của mảng : chỉ tới phần tử
                ; cuối
SHR CX,1       ; CX=n/2 : số lần trao đổi
; trao đổi các phần tử
XCHG_LOOP:
MOV AX,[SI]    ; lấy 1 phần tử ở nửa thấp của mảng
XCHG AX,[DI]   ; nửa ở nửa cao của mảng
MOV [SI],AX    ; hoàn thành trao đổi
ADD SI,2       ; SI chỉ tới phần tử tiếp theo của mảng
SUB DI,2       ; DI chỉ tới phần tử cuối-1
LOOP XCHG_LOOP
POP DI
POP SI
POP CX
POP BX
POP AX
RET
REVERSE ENDP

```

7.2.2 Thao tác trên mảng có chỉ số

Trong các thao tác trên mảng này, nửa chỉ số offset của toàn bộ chỉ số được bằng cách cộng một số gọi là displacement với nội dung của một thanh ghi .

Displacement có thể là:

- nửa chỉ số offset của một biến , ví dụ A
- một hằng (âm hoặc dương), ví dụ -2
- nửa chỉ số offset của một biến cộng với một hằng số , ví dụ A+4

Cách pháp của một toàn bộ chỉ số có thể là một trong các kiểu tổng cộng sau :

[register + displacement]

[displacement + register]

[register]+ displacement

[displacement]+ register

displacement[register]

Các thanh ghi phải là BX , SI , DI (nửa chỉ số phải là thanh ghi DS)

và BP (thanh ghi SS chứa nửa chỉ số)

Thao tác trên mảng được gọi là có chỉ số (based) nếu thanh ghi BX (base register) hoặc BP (base pointer) được dùng .

Chế độ寻址 chế độ寻址 có chỉ số (indexed) nếu thanh ghi SI (source index) hoặc DI (destination index) được dùng .

Ví dụ: Giả sử rằng W là mảng từ địa chỉ BX chứa 4 . Trong lệnh

MOV AX,W[BX]

displacement là nửa chế độ offset của biến W . Lệnh này sẽ di chuyển phần từ có địa chỉ W+4 vào thanh ghi AX . Lệnh này cũng có thể viết dưới các dạng tổng quát sau :

MOV AX, [W+BX]

MOV AX, [BX+W]

MOV AX, W+[BX]

MOV AX, [BX]+W

Lấy ví dụ khác , giả sử rằng SI chứa nửa chế độ của mảng từ W . Trong lệnh

MOV AX,[SI+2]

displacement là 2 . Lệnh này sẽ di chuyển nội dung của từ nhớ W+2 tới AX . Lệnh này cũng có thể viết dưới các dạng khác :

MOV AX,[2+SI]

MOV AX,2+[SI]

MOV AX,[SI]+2

MOV AX,2[SI]

Với chế độ寻址 chế độ寻址 có số có thể viết lại code cho bài toán tính tổng 10 phần tử của mảng nhớ sau :

XOR AX,AX ; xóa AX

XOR BX,BX ; xóa BX (thanh ghi có số)

MOV CX,10 ; CX= số phần tử=10

ADDITION:

ADD AX,W[BX] ; sum=sum+element

ADD BX,2 ; trỏ tới phần tử tiếp theo

LOOP ADDITION

Ví dụ: Giả sử rằng ALPHA được khai báo như sau :

ALPHA DW 0123h,0456h,0789h,0ADCDh

trong đó nội dung của bộ DS và giả sử rằng :

BX =2 [0002]= 1084h

SI=4 [0004]= 2BACH

DI=1

Cho ra các lệnh sau đây là hộp địa chỉ offset nguồn và số được 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. MOV BX,[ALPHA+AX]

Giải :

	Source offset	Number moved
a.	ALPHA+2	0456h
b.	2+2	2BACH
c.	ALPHA+4	0789h
d.	-2+4=+2	1084h
e.	ALPHA+3+1=ALPHA+4	0789h
d.	illegal form source operand ...[BX]2	
g.	illegal ; thanh ghi AX là không được phép	

Ví dụ sau này cho thấy một mảng số nguyên có thể được xử lý bởi các thao tác xử lý số nguyên.

Ví dụ : Nội các ký tự viết thông trong chuỗi sau thành ký tự viết hoa .

MSG DB 'co ty lo lo ti ca '

Giải :

```
MOV CX,17 ; số ký tự chứa trong CX=17
XOR SI,SI ; SI chứa số cho ký tự
```

TOP:

```
CMP MSG[SI], ' ' ; blank?
JE NEXT ; yes , skip
AND MSG[SI],0DFH ; nội thành chữ hoa
```

NEXT:

```
INC SI ; chứa số ký tự tiếp theo
LOOP TOP ; lặp
```

7.2.3 Toán tử PTR và toán tử giả LABEL

Trong các chương trước chúng ta đã biết rằng các toán hạng của một lệnh phải cùng loại, tức là cùng là byte hoặc cùng là từ. Nếu một toán hạng là hằng số thì ASM sẽ chuyển chúng thành loại tổng cộng với toán hạng kia. Ví dụ, ASM sẽ thực hiện lệnh `MOV AX,1` nhờ là lệnh toán hạng từ. Tổng từ, ASM sẽ thực hiện lệnh `MOV BH,5` nhờ là lệnh byte. Tuy nhiên, lệnh

`MOV [BX],1` là không hợp lệ vì ASM không biết toán hạng chữ bởi thanh ghi BX là toán hạng byte hay toán hạng từ. Có thể khắc phục điều này bằng toán từ PTR nhờ sau :

`MOV BYTE PTR [BX],1` ; toán hạng đích là toán hạng byte

`MOV WORD PTR [BX],1` ; toán hạng đích là toán hạng từ

Ví dụ: Thay ký từ t thành T trong chuỗi nội dung như sau :

`MSG DB 'this is a message'`

Cách 1: Dùng chuỗi nội dung chữ gián tiếp thành ghi :

`LEA SI,MSG` ; SI trỏ tới MSG

`MOV BYTE PTR [SI], 'T'` ; thay t bằng T

Cách 2 : Dùng chuỗi nội dung chữ trực tiếp :

`XOR SI,SI` ; xóa SI

`MOV MSG[SI], 'T'` ; thay t bởi T

Ồn này không cần dùng PTR vì MSG là biến byte.

Nếu chúng ta dùng PTR nội dung để khai báo loại (type) của toán hạng. Cuối cùng chúng ta có như sau:

Type PTR address_expression

Trong đó Type : byte , word , Dword

Address_expression : là các biến nội dung khai báo bởi DB,DW, DD .

Ví dụ chúng ta có 2 khai báo biến như sau :

`DOLLARS DB 1AH`

`CENTS DB 52H`

Và chúng ta muốn di chuyển DOLLARS vào AL, di chuyển CENTS vào AH bằng một lệnh MOV duy nhất. Có thể dùng lệnh sau :

`MOV AX, WORD PTR DOLLARS` ; AL=DOLLARS và AH=CENTS

Toán từ và LABEL

Có một cách khác để giải quyết vấn đề xung đột về loại toán hạng nhờ trên bằng cách dùng toán từ và LABEL nhờ sau này :

`MONEY LABEL WORD`

`DOLLARS DB 1AH`

`CENTS DB 52H`

Các lệnh trên này khai báo biến MONEY là biến từ với 2 thanh ghi là DOLLARS và CENTS. Trong đó DOLLARS có cùng nội dung với MONEY. Lệnh

`MOV AX, MONEY`

Tổng cộng với 2 lệnh :

MOV AL, DOLLARS

MOV AH, CENTS

Ví dụ: Giả sử rằng số liệu nước khai báo như sau :

.DATA

A DW 1234h

B LABEL BYTE
DW 5678h

C LABEL WORD

C1 DB 9Ah

C2 DB 0bch

Hãy cho biết các lệnh nào sau đây là hộp lẻ và kết quả của lệnh .

- MOV AX,B
- MOV AH,B
- MOV CX,C
- MOV BX,WORD PTR B
- MOV DL,WORD PTR C
- MOV AX, WORD PTR C1

Giải :

- không hộp lẻ
- hộp lẻ, 78h
- hộp lẻ, 0BC9Ah
- hộp lẻ, 5678h
- hộp lẻ, 9Ah
- hộp lẻ , 0BC9Ah

7.2.4 Chế độ đoạn (segment override)

Trong chế độ địa chỉ gián tiếp bằng thanh ghi , các thanh ghi con trỏ BX,SI hoặc DI chứa địa chỉ offset con thanh ghi đoạn là DS . Cũng có thể chứa một thanh ghi đoạn khác theo cú pháp sau :

segment_register : [pointer_register]

Ví dụ: MOV AX, ES:[SI]

nếu SI=0100h thì địa chỉ của toàn hàng nguồn là ES:0100h

Việc chế độ đoạn cũng có thể dùng với chế độ địa chỉ chứa số và chế độ địa chỉ cô sô.

7.2.5 Truy xuất đoạn stack

Nhờ chúng ta nhớ rằng khi BP chứa một địa chỉ offset trong chế độ địa chỉ gián tiếp bằng thanh ghi , SS sẽ cung cấp số đoạn . Việc này có nghĩa là có thể dùng BP để truy xuất stack .

Ví dụ : Di chuyển 3 từ tại đỉnh stack vào AX,BX,CX mà không làm thay đổi nội dung của stack .

```
MOV BP,SP      ; BP chỉ tới đỉnh stack
MOV AX,[BP]     ; copy đỉnh stack vào AX
MOV BX,[BP+2]   ; copy từ thứ hai trên stack vào BX
MOV CX,[BP+4]   ; copy từ thứ ba vào CX
```

7.3 Sắp xếp số liệu trên mảng

Việc tìm kiếm một phần tử trên mảng sẽ dễ dàng nếu nhờ mảng được sắp xếp (sort) . Để sort mảng A gồm N phần tử có thể tiến hành qua N-1 bước như sau :

Bước 1: Tìm số lớn nhất trong số các phần tử A[1]...A[N] . Gán số lớn nhất cho A[N] .

Bước 2 : Tìm số lớn nhất trong các số A[1]...A[N-1]. Gán số lớn nhất cho A[N-1]

·
·
·

Bước N-1 : Tìm số lớn nhất trong 2 số A[1] và A[2]. Gán số lớn nhất cho A[2]

Ví dụ : giả sử mảng A chứa 5 phần tử là các số nguyên như sau :

Position	1	2	3	4	5
initial	21	5	16	40	7
bước 1	21	5	16	7	40
bước 2	7	5	16	21	40
bước 3	7	5	16	21	40
bước 4	5	7	16	21	40

Thuật toán

```
      i =N
FOR N-1 times DO
    find the position k of the largest element among A[1]..A[i]
    Swap A[i] and A[k]      ( uses procedure SWAP )
i=i-1
END_FOR
```

Sau này là chương trình để sort các phần tử trong một mảng . Chúng ta sẽ dùng thuật toán SELECT để chọn phần tử trên mảng . Thuật toán SELECT sẽ gọi thuật toán SWAP để sắp xếp . Chương trình chính sẽ như sau :

```

TITLE      PGM7_3: TEST SELECT
.MODEL     SMALL
.STACK     100H
.DATA
    A      DB    5,2,,1,3,4
.CODE
    MAIN    PROC
    MOV     AX,@DATA
    MOV     DS,AX
    LEA     SI,A
    MOV     BX,5    ; số phần tử của mảng chứa trong BX
    CALL    SELECT
    MOV     AH,4CH
    INT     21H
MAIN ENDP
INCLUDE C:\ASM\SELECT.ASM
END MAIN

```

Tập tin SELECT.ASM chứa thủ tục SELECT và thủ tục SWAP được viết như sau tại C:\ASM .

```

SELECT     PROC
; sắp xếp mảng byte
; input: SI = địa chỉ offset của mảng
        BX= số phần tử ( n ) của mảng
; output: SI = địa chỉ offset của mảng đã sắp xếp .
; uses : SWAP
    PUSH    BX
    PUSH    CX
    PUSH    DX
    PUSH    SI
    DEC     BX    ; N = N-1
    JE      END_SORT ; Nếu N=1 thì thoát
    MOV     DX,SI    ; địa chỉ offset của mảng vào DX

; lặp N-1 lần
SORT_LOOP:
    MOV     SI,DX    ; SI trỏ tới mảng A
    MOV     CX,BX    ; CX = N -1 số lần lặp
    MOV     DI,SI    ; DI chỉ phần tử cuối nhất

```

```

        MOV     AL,[DI]      ; AL chứa phần tử nhỏ nhất
; tìm phần tử lớn nhất
FIND_BIG:
        INC     SI           ; SI trỏ tới phần tử tiếp theo
        CMP     [SI],AL      ; phần tử tiếp theo > phần tử nhỏ nhất
        JNG     NEXT        ; không, tiếp tục
        MOV     DI,SI        ; DI chứa địa chỉ của phần tử lớn nhất
        MOV     AL,[DI]      ; AL chứa phần tử lớn nhất
NEXT:
        LOOP    FIND_BIG
; swap phần tử lớn nhất với phần tử cuối cùng
        CALL    SWAP
        DEC     BX           ; N= N-1
        JNE     SORT_LOOP   ; lặp nếu N > 0
END_SORT:
        POP     SI
        POP     DX
        POP     CX
        POP     BX
        RET
SELECT   ENDP
SWAP     PROC
; hoán đổi 2 phần tử của mảng
; input : SI= phần tử nhỏ nhất
;        DI = phần tử lớn nhất
; output : các phần tử đã trao đổi
        PUSH    AX          ; cất AX
        MOV     AL,[SI]     ; lấy phần tử A[i]
        XCHG    AL,[DI]     ; đặt nó trên A[k]
        MOV     [SI],AL     ; đặt A[k] trên A[i]
        POP     AX          ; lấy lại AX
        RET
SWAP     ENDP

```

Sau khi dịch chương trình, có thể dùng DEBUG để chạy thử và test kết quả.

7.4 Mảng 2 chiều

Mảng 2 chiều là một mảng của một mảng, nghĩa là một mảng 1 chiều mà các phần tử của nó là một mảng 1 chiều khác. Có thể hình dung mảng 2 chiều như một ma trận chữ nhật. Ví dụ mảng B gồm có 3 hàng và 4 cột (mảng 3x4) như sau :

ROW \ COLUMN	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]

Bởi vì mảng là 1 chiều vì vậy các phần tử của mảng 2 chiều phải được lưu trữ trên bộ nhớ theo kiểu lain lổit . Có 2 cách như sau :

- Cách 1 là lưu trữ theo thứ tự dọc : trên mảng lưu trữ các phần tử của dòng 1 rồi đến các phần tử của dòng 2 ...
- Cách 2 là lưu trữ theo thứ tự ngang : trên mảng lưu trữ các phần tử của cột 1 rồi đến các phần tử của cột 2...

Giả sử mảng B chứa 10,20,30,40 trên dòng 1
 chứa 50,60,70,80 trên dòng 2
 chứa 90,100,110,120 trên dòng 3

Theo thứ tự hàng chúng ta có được lưu trữ như sau :

B DW 10,20,30,40
 DW 50,60,70,80
 DW 90,100,110,120

Theo thứ tự cột chúng ta có được lưu trữ như sau :

B DW 10,50,90
 DW 20,60,100
 DW 30,70,110
 DW 40,80,120

Hầu hết các ngôn ngữ cấp cao biến dịch mảng 2 chiều theo thứ tự dọc .

Trong ASM , chúng ta có thể dùng một trong 2 cách :

Nếu các thành phần của một hàng được xử lý liên lổit thì cách lưu trữ theo thứ tự hàng là thích hợp . Ngược lại thì dùng cách lưu trữ theo thứ tự cột .

Xác định một phần tử trên mảng 2 chiều :

Giả sử mảng A gồm MxN phần tử lưu trữ theo thứ tự dọc . Gọi S là số lần của một phần tử: S=1 nếu phần tử là byte , S=2 nếu phần tử là từ . Để tìm phần tử ở vị trí A[i,j] thì cần tìm : hàng i và tìm phần tử ở vị trí j trên hàng này . Nhờ vậy phải tiến hành qua 2 bước :

Bước 1: Hàng 1 bắt đầu tại vị trí A . Vì mỗi hàng có N phần tử, do đó

Hàng 2 bắt đầu tại A + NxS .

Hàng 3 bắt đầu tại A + 2xNxS .

Hàng thứ i bắt đầu tại A + (i-1)xNxS .

Bước 2: Phần tử ở vị trí j trên một hàng cách vị trí đầu hàng (j-1)xS byte

Từ 2 bước trên suy ra rằng trong mảng 2 chiều NxM phần tử mà chúng ta có được lưu trữ theo thứ tự hàng thì phần tử A[i,j] có địa chỉ như sau :

$$A + ((i-1) \times N + (j-1)) \times S \quad (1)$$

Tổng thì nếu lờu trữ theo trậi thì phần tử $A[i,j]$ cũi ở địa chỉ như sau :

$$A + (i-1) + (j-1) \times M \times S \quad (2)$$

Ví dụ : Giả sử ở A là mảng $M \times N$ phần tử kiểu từ ($S=2$) ở địa chỉ lờu trữ theo kiểu trậi từ hàng . Hỏi :

Hàng i bắt đầu tại địa chỉ nào ?

Cột j bắt đầu tại địa chỉ nào ?

Hai phần tử trên một cột cách nhau bao nhiêu bytes

Giải :

Hàng i bắt đầu tại $A[i,1]$ theo công thức (1) thì địa chỉ địa chỉ là: $A + (i-1) \times N \times 2$

Cột j bắt đầu tại $A[1,j]$ theo công thức (1) thì địa chỉ địa chỉ là: $A + (j-1) \times 2$

Vì có N cột nên 2 phần tử trên cùng một cột cách nhau $2 \times N$ byte .

7.5 Thao tác trên mảng và các cơ sở

Trong chương này , địa chỉ offset của toán hạng là tổng của :

1. nội dung của thanh ghi cơ sở (BX or BP)
2. nội dung của thanh ghi chỉ số (SI or DI)
3. địa chỉ offset của 1 biến (tùy chọn)
4. một hằng âm hoặc dương (tùy chọn)

Nếu thanh ghi BX ở địa chỉ thì DS chứa số nhân của địa chỉ toán hạng . Nếu BP ở địa chỉ thì SS chứa số nhân . Toán hạng được viết theo 4 cách dưới đây:

1. $\text{variable}[\text{base_register}][\text{index_register}]$
2. $[\text{base_register} + \text{index_register} + \text{variable} + \text{constant}]$
3. $\text{variable} [\text{base_register} + \text{index_register} + \text{constant}]$
4. $\text{constant} [\text{base_register} + \text{index_register} + \text{variable}]$

Trậi của các thành phần trong dấu ngoặc là tùy ý.

Ví dụ , giả sử W là biến từ , $BX=2$ và $SI = 4$. Lệnh

`MOV AX, W[BX][SI]`

se di chuyển nội dung của mảng tại địa chỉ $W+2+4 = W+6$ vào thanh ghi AX

Lệnh này cũng cũi viết theo 2 cách sau :

`MOV AX, [W+BX+SI]`

`MOV AX, W[BX+SI]`

Thao tác trên mảng và các cơ sở : thông thường được dùng để xử lý mảng 2 chiều như ví dụ sau : Giả sử ở A là mảng 5×7 từ ở địa chỉ lờu trữ theo trậi từ hàng . Viết đoạn mã dùng để thao tác trên mảng số nguyên:

- 1) xóa dòng 3
- 2) xóa cột 4

Giải :

1) Dòng i bắt đầu tại $A + (i-1) \times N \times 2$. Như vậy dòng 3 bắt đầu tại $A + (2-1) \times 7 \times 2 = A + 28$. Cũi xóa dòng 3 như sau :

`MOV BX, 28 ; BX chứa địa chỉ dòng 3`

```

        XOR  SI,SI           ; SI sẽ chứa mức cột
        MOV  CX,7           ; CX= số phần tử của một hàng
CLEAR:
        MOV  A[BX][SI],0    ; xóa A[3,1]
        ADD  SI,2           ; đến cột tiếp theo
        LOOP CLEAR
    
```

2) Cột j bắt đầu tại địa chỉ $A + (j-1) \times 2$. Vậy cột 4 bắt đầu tại địa chỉ $A + (4-1) \times 2 = A + 6$. Hai phần tử trên một cột cách nhau $N \times 2$ byte, ở đây $N=7$, vậy 2 phần tử cách nhau 14 byte. Có thể xóa cột 4 như sau :

```

        MOV  SI,6           ; SI chứa địa chỉ cột 4
        XOR  BX,BX         ; BX chứa địa chỉ hàng
        MOV  CX,5           ; CX= 5 : số phần tử trên một cột
CLEAR:
        MOV  A[BX][SI],0    ; Xóa A[i,4]
        ADD  BX,1           ; đến dòng tiếp theo
        LOOP CLEAR
    
```

7.6 Ứng dụng tính trung bình

Giải một lớp gồm 5 sinh viên và có 4 môn thi. Kết quả cho bởi mảng 2 chiều như sau :

Tên Sinh viên	TEST1	TEST2	TEST3	TEST4
MARY	67	45	98	33
SCOTT	70	56	87	44
GEORGE	82	72	89	40
BETH	80	67	95	50
SAM	78	76	92	60

Chúng ta sẽ viết 1 chương trình tính điểm trung bình cho mỗi bài thi. Nếu làm như vậy có thể tổng theo cột rồi chia cho 5.

Thuật toán :

1. $j = 4$
2. repeat
3. Sum the scores in column j
4. divide sum by 5 to get average in column j
5. $j = j - 1$
5. Until $j = 0$

Trong nội bộ ô 3 có thể làm như sau :

```
Sum[j]= 0
i = 1
FOR 5 times DO
Sum[j]= Sum[j]+ Score[i, j]
i = i + 1
END_FOR
```

Chương trình có thể viết như sau :

```
TITLE PGM7_4 : CLASS AVERAGE
.MODEL      SMALL
.STACK      100H
.DATA
        FIVE DB 5
        SCORES DW 67,45,98,33 ; MARY
                DW 70,56,87,44 ; SCOTT
                DW 82,72,89,40 ; GEORGE
                DW 80,67,,95,50 ; BETH
                DW 78,76,92,60 ; SAM
        AVG DW 5 DUP (0)
.CODE
MAIN PROC
        MOV AX,@DATA
        MOV DS,AX
;J=4
REPEAT:
MOV SI,6 ; SI chứa nên cột thôi4
        XOR BX,BX ; BX chứa hàng thôi nhất
        XOR AX,AX ; AX chứa tổng theo cột
; Tổng điểm trên cột j
FOR:
        ADD AX , SCORES[BX+SI]
        ADD BX,8 ; BX chứa nên hàng thôi2
        LOOP FOR
; end_for
; tính trung bình cột j
        XOR DX,DX ; xóa phần cao của số bị chia (DX:AX)
        DIV FIVE ; AX = AX/5
        MOV AVG[SI],AX ; cất kết quả trên mảng AVG
        SUB SI,2 ; nên cột tiếp
; un til j=0
        JNL REPEAT
```



```

;DOS EXIT
        MOV  AH,4CH
        INT  21H
MAIN    ENDP
        END  MAIN
    
```

Sau khi biên dịch chương trình có thể dùng DEBUG để chạy và xem kết quả bằng lệnh DUMP.

7.7 Lệnh XLAT

Trong một số ứng dụng cần phải chuyển số liệu từ dạng này sang dạng khác. Ví dụ IBM PC dùng ASCII code cho các ký tự trong khi IBM Mainframes dùng EBCDIC (Extended Binary Coded Decimal Interchange Code) . Nếu chuyển một chuỗi ký tự nào đó từ mã hóa bằng ASCII thành EBCDIC , một chương trình phải thay mã ASCII của từng ký tự trong chuỗi thành mã EBCDIC tương ứng .

Lệnh XLAT (không có toán hạng) dùng để lấy một giá trị byte thành một giá trị khác chứa trong một bảng .

AL phải chứa byte cần biến đổi

DX chứa địa chỉ offset của bảng cần biến đổi

Lệnh XLAT sẽ:

1) cộng nội dung của AL với địa chỉ trên BX để tìm ra địa chỉ trong bảng

2) thay thế giá trị của AL với giá trị tìm thấy trong bảng

Ví dụ , giả sử rằng nội dung của AL là trong vùng 0 đến Fh và chúng ta muốn thay nội dung mã ASCII của số hex tương ứng đó, tức là thay 6h bằng 036h='6' , thay Bh bằng 042h="B" . Bảng biến đổi là:

```

TABLE   DB    030h,031h,032h,033h,034h,035h,036h,037h,038h,039h
         DB    041h,042h,043h,044h,045h,046h
    
```

Ví dụ , nếu 0Ch thành "C" , chúng ta thực hiện các lệnh sau :

```
MOV  AL,0Ch    ; số cần biến đổi
```

```
LEA  BX,TABLE  ; BX chứa địa chỉ offset của bảng
```

```
XLAT    ; AL chứa "C"
```

Ở đây XLAT tính TABLE + Ch = TABLE + 12 và thay thế AL bởi 043h . Nếu AL chứa một số không ở trong khoảng 0 đến 15 thì XLAT sẽ cho một giá trị sai .

Ví dụ : Mã hóa và giải mã một thông điệp mã

Chương trình này sẽ:

Nhập những dữ liệu vào một thông điệp

Mã hóa nội dung bằng bảng mã định sẵn

In ra nội dung tiếp theo

Dịch ngược trở lại bằng bản mã rồi in ra nội dung tiếp theo

Khi chạy chương trình sẽ có dạng sau :

```

ENTER A MESSAGE :
DAI HOC DA LAT ; input
OXC BUC OX EXK ; encode
DAI HOC DA LAT ; translated
    
```

Thuật toán như sau :

- Print prompt
- Read and encode message
- Go to a new line
- Print encoded message
- go to a new line
- translate and print message

```

TITLE PGM7_5 : SECRET MESSAGE
.MODEL      SMALL
.STACK      100H
.DATA
;ALPHABET      ABCDEFGHIJKLMNOPQRSTUVWXYZ
CODE_KEY DB 65 DUP ( ' ' ), '
XQPOGHZBCADEIJUVFMNKLIRSTWY'
          DB 37 DUP ( ' ' ) ; 128 ký tự của bảng mã ASCII
CODED      DB 80 dup ( '$' ) ; 80 ký tự để ghi vào
DECODE_KEY DB 65 DUP ( ' ' ),
'JHIKLQEFMNTURSDCBVWXOPYAZG'
          DB 37 DUP ( ' ' )
PROMPT      DB 'ENTER A MESSAGE :',0DH,0AH,'$'
CRLF        DB 0DH,0AH,'$'
.CODE
MAIN        PROC
            MOV     AX,@DATA
            MOV     DS,AX
; in dấu nhắc
            MOV     AH,9
            LEA     DX,PROMPT
            INT     21H
; nối và mã hóa ký tự
            MOV     AH,1

            LEA     BX,CODE_KEY      ; BX chứa tới CODE_KEY
            LEA     DI,CODED          ; DI chứa tới thông điệp cần mã hóa
    
```

Trong chương trình có nội dung số liệu với các khai báo sau :

Cho biết bảng chữ cái tiếng Anh

```
CODE_KEY DB 65 DUP(' '), 'XQPOGHZBCADEIJUVFMNKLIRSTWY'
```

DB 37 DUP (' ')

Khai báo 128 ký tự của bảng mã ASCII, trong đó có các ký tự hoa và thường.

CODED DB 80 dup ('\$')

80 ký tự của bảng mã, giá trị ban đầu là \$ nếu có thể in bảng hàng 9 ngày 21h

DECODE_KEY DB 65 DUP (' '), 'JHIKLQEFMNTURSDCBVWXOPYAZG'

DB 37 DUP (' ')

Bảng giải mã được thiết lập theo cách mã hóa, nghĩa là trong phần mã hóa chúng ta mã hóa 'A' thành 'X' vì vậy khi giải mã 'X' phải giải mã thành 'A' ...

Các ký tự của bảng mã không phải là ký tự hoa nếu nó được chuyển thành ký tự trong.
