

## BÀI 1: NGÔN NGỮ ASSEMBLY VÀ CÁCH LẬP TRÌNH

### Mục tiêu:

Trong bài này, Anh/Chị cần đạt được những mục tiêu sau:

1. Các đặc tính của ngôn ngữ Assembly.
2. Cách cài đặt chương trình dịch.
3. Quy trình 4 bước để thực hiện một chương trình Assembly.
4. Các khái niệm và sự hỗ trợ của hệ thống cho lập trình Assembly.
5. Lướt qua một số lệnh hay dùng trong 6 nhóm lệnh mnemonic (instruction set) của ngôn ngữ Assembly, những lệnh sinh ra mã máy để chạy chương trình.
6. Các lệnh điều khiển hỗ trợ khi dịch chương trình (directive), đặc biệt là 2 dạng lệnh điều khiển segment dạng đơn giản và chuẩn.

### Nội dung:

## Chương I. NGÔN NGỮ ASSEMBLY VÀ CÁCH LẬP TRÌNH

### 1.1. MỞ ĐẦU

Ngôn ngữ lập trình Assembly (hợp ngữ) là một ngôn ngữ bậc thấp được sử dụng nhiều, đặc biệt đối với các công việc gần với phần cứng, ví dụ: các chương trình có trong ROM BIOS của hầu hết các máy tính PC, các chương trình tạo và diệt VIRUS, lập trình cho đo lường và điều khiển,... Qua việc lập trình bằng ngôn ngữ Assembly, người lập trình càng hiểu thêm về kiến trúc máy tính cùng các cơ chế hoạt động của máy tính. Ngôn ngữ Assembly rất gần với ngôn ngữ máy do vậy có những ưu điểm sau:

- Chương trình được viết bằng ngôn ngữ Assembly chạy nhanh. Một ví dụ cụ thể và rõ nét là bảng tính LOTUS 1-2-3 (công cụ tương tự EXCEL chạy trên môi trường DOS), một công cụ đã một thời được ưa chuộng và sử dụng nhiều nhất đã được viết bằng ngôn ngữ Assembly do vậy chạy rất nhanh,

- Chương trình được thể hiện bằng ngôn ngữ Assembly tiết kiệm bộ nhớ,
- Ngôn ngữ Assembly giúp việc thâm nhập vào các thiết bị phần cứng cũng như vùng nhớ, các cổng, các thanh ghi,... dễ dàng hơn.

Tuy nhiên ngôn ngữ Assembly là ngôn ngữ bậc thấp do vậy cũng có những nhược điểm:

- Lập trình bằng ngôn ngữ Assembly tương đối khó khăn vì đòi hỏi người lập trình phải am hiểu khá sâu phần cứng,
- Việc kiểm tra lỗi và gỡ rối một chương trình bằng ngôn ngữ Assembly khó khăn hơn một chương trình được viết bằng ngôn ngữ bậc cao,
- Khó khăn trong việc chuyển giao chương trình được viết lên các máy tính có cấu trúc khác nhau.

Với những ưu điểm và nhược điểm của mình, từ trước đến nay ngôn ngữ Assembly là ngôn ngữ được dùng nhiều và chưa bao giờ bị coi nhẹ.

Hiện nay có hai chương trình dịch Assembler được sử dụng nhiều nhất và rộng rãi cho máy PC, đó là:

- MASM của hãng Microsoft, và
- TASM của hãng Borland.

Tuy có một số khác biệt nhỏ song hai chương trình dịch này có thể coi là gần tương thích với nhau.

## 1.2. CÀI ĐẶT CHƯƠNG TRÌNH DỊCH TASM

Để một chương trình được viết bằng ngôn ngữ Assembly có thể chạy được, chúng ta phải dịch và liên kết để tạo ra tệp có đuôi .EXE hoặc đuôi .COM. Để nhất cần 4 tệp được coi là lõi của chương trình dịch, đó là các tệp TASM.EXE (chương trình dịch ASM - chuyển một tệp nguồn có đuôi .ASM sang tệp mã có đuôi .OBJ), TLINK (chương trình thực hiện việc liên kết - chuyển một tệp mã có đuôi .OBJ sang tệp có đuôi .EXE hoặc .COM) và hai tệp hỗ trợ là RTM.EXE và DPMI16BI.OVL. Các tệp này có thể nhận được nhờ 2 cách:

- *Cách 1:* Nếu có bộ đĩa cài gồm 5 đĩa, các bước tiến hành sẽ như sau (phần chữ có gạch chân do người cài đặt đánh vào):

Đưa đĩa số 1 (gồm các chương trình hệ thống của Turbo Assembler) vào ổ đĩa A và tiến hành cài đặt:

A:\> install <Enter>

thì trên màn hình sẽ hiện ra phần quảng cáo về hãng và lưu ý về bản quyền:

Phần giới thiệu về hãng và  
những  
lưu ý về vấn đề bản quyền  
.....  
.....

Ấn Enter để tiếp tục quá trình cài đặt và trên màn hình sẽ hiện:

Enter the SOURCE drive to

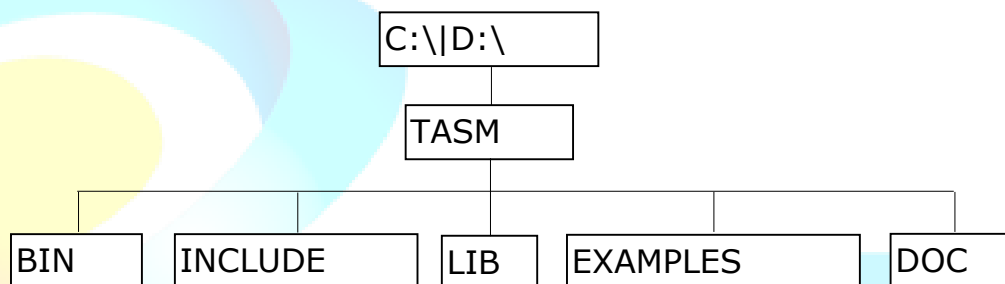
Ấn Enter để tiếp tục và trên màn hình sẽ hiện:

**Turbo Assembler Directory:**  
**C:\TASM**  
Turbo Assembler Example Directory:  
C:\TASM  
Unzip Example Files: Yes  
**Start Inslation**

Việc tạo các thư mục trong bảng trên là mặc định (default). Nếu chúng ta muốn thay đổi các thư mục theo ý muốn của mình thì đưa thanh sáng đến phần mặc định của thư mục đó và ấn ENTER, các dòng hướng dẫn sẽ hiện ra cho phép chúng ta vào tên thư mục theo yêu cầu. Sau khi đã thay đổi các thư mục hoặc đồng ý với các tên thư mục mặc định, chúng ta đưa thanh sáng xuống mục Start Instalation và chọn bằng cách ấn ENTER. Quá trình cài đặt được bắt đầu với đĩa số 1. Sau khi xong phần cài đặt ở đĩa số 1, trên màn hình sẽ hiện:

Please insert your  
TURBO ASSEMBLER EXAMPLES  
disk into drive A:  
Press any key to continue

Yêu cầu đưa đĩa số 2 (gồm các ví dụ) vào ổ A: và ấn phím bất kỳ để tiếp tục. Bước này sẽ được lặp đi lặp lại cho đĩa số 3, 4 và 5. Sau khi kết thúc việc cài đặt sẽ có thông báo trên màn hình và quá trình cài đặt như vậy là đã hoàn tất với cấu trúc như sau (các thư mục chương trình cài đặt sẽ tạo), ví dụ với sự lựa chọn mặc định:



Sau khi cài đặt thì trong thư mục BIN có rất nhiều tệp, trong đó có các tệp chính sau (các tệp lỗi của chương trình dịch TASM):

TASM.EXE . . . chương trình dịch Turbo Assembler  
TLINK . . . chương trình liên kết  
MAKE.EXE . . . chương trình tiện dụng

và hai tệp hỗ trợ là RTM.EXE và DPMI16BI.OVL

*Chú ý:* Ngày nay chương trình cài đặt được bán trên một đĩa CD với thư mục Setup. Chỉ cần chạy SETUP.EXE trong thư mục Setup cũng được kết quả tương tự như cài đặt được trình bày theo cách 1.

- *Cách 2:*

Như đã đề cập ở trên là để dịch và liên kết một chương trình được viết bằng ngôn ngữ Assembly phải có tối thiểu 2 tệp TASM.EXE và TLINK.EXE cùng 2 tệp hỗ trợ là RTM.EXE và DPMI16BI.OVL. Do vậy để tiết kiệm bộ nhớ và nếu không có bộ đĩa cài đặt thì cách dễ nhất là sao 4 tệp lỗi của chương trình dịch Turbo

Assembler kể trên ở một máy đã được cài đặt theo cách 1 vào máy của chúng ta.

### 1.3. CÁC BƯỚC THỰC HIỆN MỘT CHƯƠNG TRÌNH ĐƯỢC VIẾT BẰNG NGÔN NGỮ ASSEMBLY TRÊN MÁY PC

Muốn thực hiện một chương trình được viết bằng ngôn ngữ Assembly trên máy PC và các máy tương thích phải tiến hành theo trình tự sau:

– *Soạn thảo chương trình:*

Không giống như các ngôn ngữ bậc cao, chương trình dịch Assembler (cả TASM và MASM) không có môi trường soạn thảo riêng. Do vậy phải dùng một chương trình soạn thảo bất kỳ (ở chế độ programming mode), ví dụ như : edit, NC, Notepad, TC, Turbo,... để soạn thảo chương trình. Tập của chương trình được soạn thảo nhất thiết phải có phần mở rộng (đuôi) .ASM,

– *Dịch chương trình:*

Sau khi chương trình đã soạn thảo xong hãy dịch chương trình từ dạng nguồn (với phần mở rộng .ASM) sang dạng mã máy (với phần mở rộng .OBJ). Chú ý là tập có phần mở rộng .OBJ chỉ được tạo nếu khi dịch không có sai sót về cú pháp (syntax). Cú pháp của lệnh dịch có dạng tổng quát sau (với chương trình dịch TASM của hãng Borland):

TASM [option] SOURCEfile [, OBJfile] [, LSTfile] [, XRFfile]  
trong đó:

SOURCEfile . . . tên tệp nguồn, có phần mở rộng .ASM

OBJfile . . . tên tệp đích, có phần mở rộng .OBJ

LSTfile . . . tên tệp để in ra, có phần mở rộng .LST (trong tệp này sẽ in ra dòng lệnh, địa chỉ ô nhớ, mã máy của chương trình, lệnh dạng mnemonic và ghi chú).

*Ví dụ:* Tệp có đuôi .LST có dạng như sau:

HELLO.ASM

1 .DOSSEG

2 0000 .MODEL small

3 0000 .STACK 100h

```
4 0100      .DATA

5 0000 48 65 6C 6C 6F 2C 20 + Message DB 'Hello, world',

6      77 6F 72 6C 64 0D 0A + 13,10,12

7      0C

8      = 00F MESSAGE_LENGTH EQU $-Message

9 000F      .CODE

10 0000 B8 000a    mov AX,@data

11 0003 8E D8      mov DS,AX      ; Set DS to data seggment
```

.....

XRFfile ... tên tệp dùng để in, có đuôi .XRF (các phần in ra giống tệp .LST, ngoài ra còn có bảng qui chiếu nơi các nhãn được khai báo cũng như nơi các nhãn được sử dụng trong chương trình).

- *Liên kết:*

Dùng TLINK (hoặc LINK đối với MASM) để liên kết, tạo từ tệp có phần mở rộng .OBJ sang tệp có phần mở rộng .EXE bằng cú pháp sau:

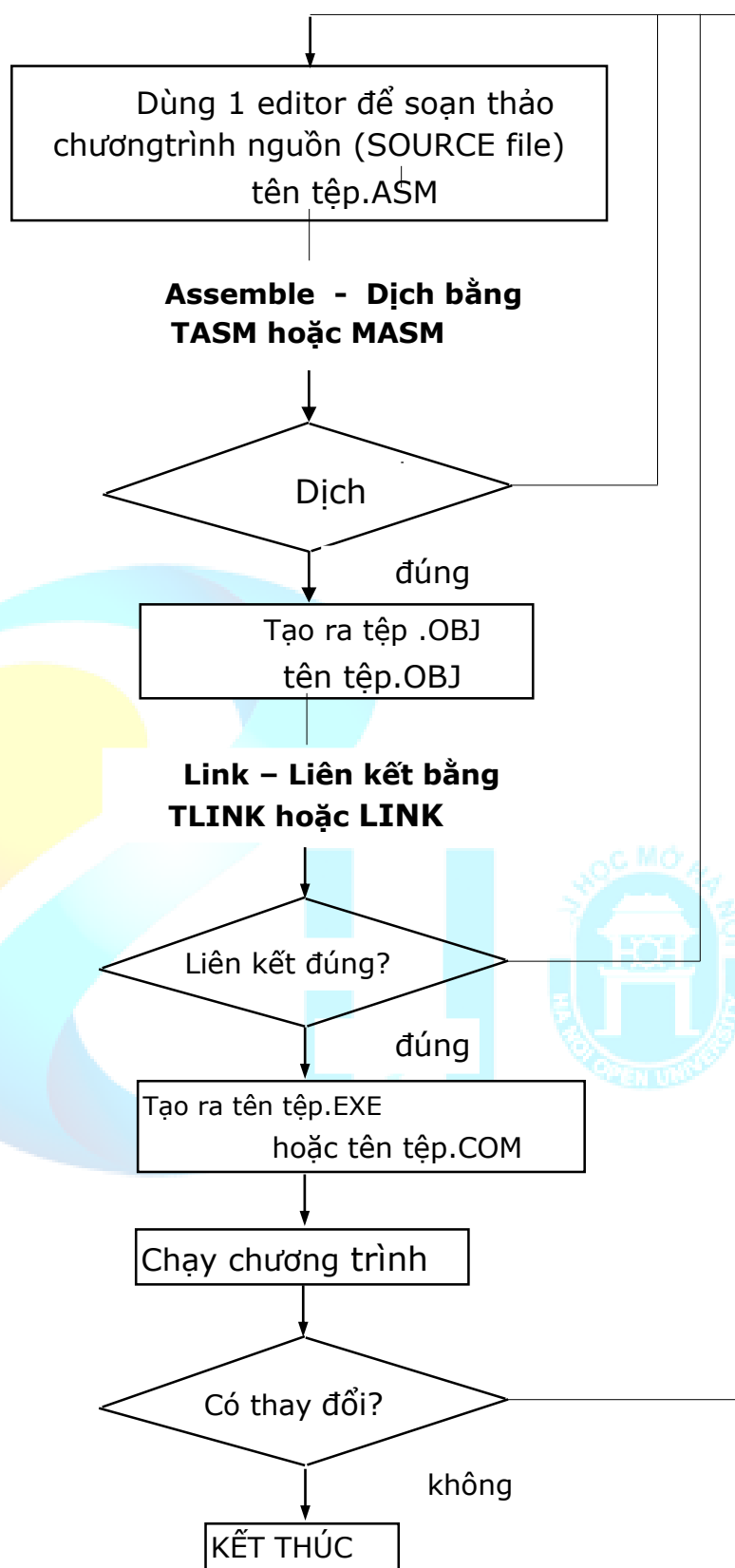
```
TLINK [option] OBJfile [, EXEfile] [, MAPfile] [, LIBfile]
```

Chú ý là nếu có sai sót khi liên kết thì chương trình liên kết sẽ không tạo ra tệp có phần mở rộng .EXE.

- *Chạy thử chương trình:*

Đến đây chúng ta đã có tệp chương trình dạng thực hiện được (có phần mở rộng .EXE) và có thể chạy thử bằng cách đánh tên tệp.

Các bước tiến hành trên được biểu diễn một cách đơn giản theo sơ đồ sau:



## 1.4. TỔNG QUAN VỀ MÔI TRƯỜNG LẬP TRÌNH BẰNG NGÔN NGỮ ASSEMBLY

### 1.4.1. Các thanh ghi

CPU 80x86 có các thanh ghi. Các thanh ghi là một vùng nhớ đặc biệt dạng RAM nằm trong CPU. Việc thâm nhập (ghi hoặc đọc dữ liệu) vào các thanh ghi thông qua tên thanh ghi chứ không phải thông qua địa chỉ như khai báo biến (khai báo biến tức là xin cấp phát ô nhớ RAM). Sở dĩ chúng ta phải tìm hiểu tương đối kỹ các thanh ghi vì người lập trình bằng ngôn ngữ Assembly thường sử dụng các thanh ghi làm toán hạng sau các lệnh của Assembly thay vì các biến và do vậy chương trình sẽ chạy nhanh hơn. Một trong những nguyên tắc của lập trình bằng ngôn ngữ Assembly là cố gắng với khả năng có thể sử dụng các thanh ghi làm toán hạng. Chỉ với những trường hợp bất khả kháng (số thanh ghi đã dùng hết hoặc kích cỡ của toán hạng không cho phép dùng thanh ghi) thì mới dùng đến khai báo biến. Các thanh ghi có những đặc tính chung song cũng có những đặc thù riêng.

Có thể tạm chia các thanh ghi của PC ra làm 4 nhóm:

- thanh ghi cờ (flag register),
- thanh ghi con trỏ lệnh (instruction pointer register),
- các thanh ghi đa năng (general purpose registers) và
- các thanh ghi phân đoạn (segment registers).

Máy tính kiến trúc 16 bit có 14 thanh ghi (không kể thanh ghi xử lý lệnh) và máy tính 32 bit có 16 thanh ghi (không kể thanh ghi xử lý lệnh).

Hãy xem xét cụ thể các nhóm thanh ghi trên.

#### a) Máy tính 16 bit

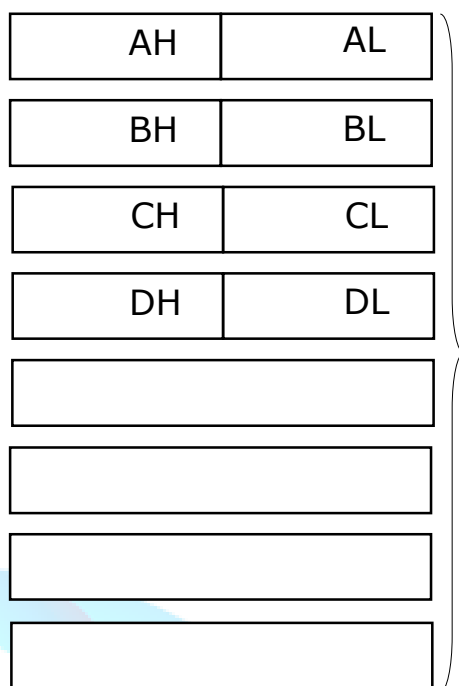
Các thanh ghi được biểu diễn ở hình sau:

FLAGS

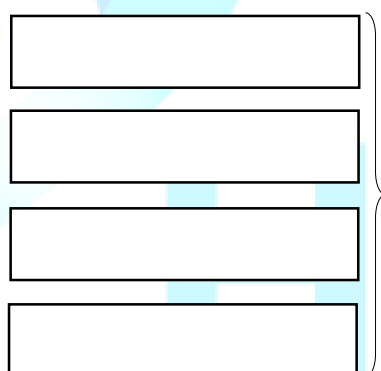
thanh ghi cờ

thanh ghi con  
trỏ lệnh





các thanh ghi  
đa năng



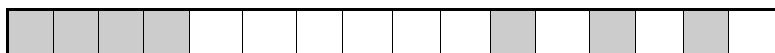
các thanh ghi  
segment

Như vậy 4 thanh ghi đa năng là AX, BX, CX và DX có 3 mode truy nhập, ví dụ với thanh ghi AX có thể truy nhập AL (byte thấp của thanh ghi AX), AH (byte cao của thanh ghi AX) hoặc AX (cả 2 byte).

Hãy xem xét kỹ vai trò từng nhóm thanh ghi một:

- *Thanh ghi cờ (flag register):*

16 bit thanh ghi cờ chứa các thông tin về trạng thái của bộ xử lý trung tâm và kết quả của lệnh vừa thực hiện (đối với những lệnh có tác động làm thay đổi trạng thái các bit cờ). Vị trí các bit cờ trong thanh ghi cờ được phân bố như sau:



trong đó:

O... cờ tràn (Overflow)    T... cờ bẫy (Trap)  
A...cờ phụ (Auxiliary)  
D... cờ hướng (Direction) S... cờ dấu (Sign)  
P... cờ chẵn lẻ (Parity)  
I ... cờ ngắt (Interruption) Z... cờ zero (Zero)  
C... cờ Carry (Carry)

Các phép tính số học và các toán tử logic thường tác động đến các cờ Z, O, A, P, S và C và trên cơ sở trạng thái của các cờ cho phép thực hiện các lệnh nhảy có điều kiện. Các cờ khác điều khiển các mode hoạt động của 80x86, ví dụ như cờ hướng D (Direction flag) phục vụ việc định hướng cho các lệnh làm việc với các chuỗi ký tự, cờ ngắt I (Interrupt flag) kiểm tra liệu có ngắt cứng từ bên ngoài, cờ bẫy T (Trap flag) phục vụ cho chương trình gỡ rối (DEBUG),...

Chúng ta không thể trực tiếp thay đổi hoặc đọc giá trị các cờ. Thay vào đó, thông qua các lệnh đặc biệt, ví dụ CLD, STD, STC, CLC, CMC, STI và CLI hoặc thông qua việc thực hiện các lệnh số học và logic, chúng ta có thể thay đổi được giá trị các cờ (phần này sẽ được đề cập sau trong khi xem xét các lệnh một cách cụ thể ở phần phụ lục).

– *Thanh ghi con trỏ lệnh (instruction pointer register):*

Thanh ghi con trỏ lệnh IP luôn chứa phần địa chỉ OFFSET (phần địa chỉ tương đối so với giá trị nằm trong thanh ghi phân đoạn mã lệnh CS (code segment)) của vùng nhớ (ROM hoặc RAM) chứa mã lệnh của lệnh tiếp theo lệnh đang thực hiện.

– *Các thanh ghi đa năng (general purpose registers):*

80x86 có 8 thanh ghi đa năng 16 bit sử dụng cho các toán hạng (toán hạng nguồn và toán hạng đích) trong các thao tác của hầu hết các lệnh. Các thanh ghi AX, BX, CX và DX có thể được sử dụng như một thanh ghi 16 bit (ví dụ AX) hoặc 2 thanh ghi 8 bit (ví dụ AH là thanh ghi 8 bit phần cao và AL là thanh ghi 8 bit phần thấp).

Bên cạnh các chức năng chung, mỗi một thanh ghi còn được sử dụng với những chức năng riêng. Hãy xem xét qua những đặc thù riêng đó của từng thanh ghi:

- *Thanh ghi AX:*

Đây là thanh ghi được sử dụng nhiều nhất. Thanh ghi này được sử dụng trong các phép tính số học, logic cũng như các thao tác trong các lệnh chuyển đổi dữ liệu. Nó còn là toán hạng “ẩn” trong một số lệnh (nhân, chia,...) hoặc là toán hạng bắt buộc trong các lệnh đọc/ghi các cổng (lệnh IN, OUT).

- *Thanh ghi BX:*

Ngoài các chức năng chung, thanh ghi BX thường được sử dụng làm con trỏ OFFSET (BX chứa phần phân địa chỉ OFFSET) của một ô nhớ).

- *Thanh ghi CX:*

Ngoài các chức năng chung, thanh ghi CX thường được sử dụng làm số đếm lần trong các lệnh dịch bit, các lệnh quay vòng, các lệnh lặp (các lệnh LOOP), các tiền tố REP cũng như lệnh nhảy JCXZ.

- *Thanh ghi DX:*

Ngoài các chức năng chung, thanh ghi DX thường đảm đương một số nhiệm vụ sau:

- \* Trở đến địa chỉ các cổng vào/ra trong các lệnh đọc (IN) và đưa số liệu ra cổng (OUT) khi địa chỉ cổng vượt quá giá trị nằm trong 1 byte (lớn hơn 255)
- \* Được sử dụng trong các lệnh nhân, chia 16 bit và 32 bit và lệnh chuyển một số có dấu dạng 16 bit sang dạng 32 bit.

- *Thanh ghi SI:*

Giống thanh ghi BX, thanh ghi SI dùng để trở đến phần địa chỉ OFFSET của các ô nhớ, đặc biệt trong các lệnh làm việc với xâu ký tự. Thường được đi với thanh ghi segment DS (có nghĩa DS:SI).

- *Thanh ghi DI:*

Giống thanh ghi SI, thanh ghi DI dùng để trở đến phần địa chỉ OFFSET của các ô nhớ, đặc biệt trong các lệnh làm việc với xâu ký tự. Thường được đi với thanh ghi segment ES (có nghĩa ES:SI).

- *Thanh ghi BP:*

Thanh ghi này thường được dùng để trỏ đến phần địa chỉ OFFSET của vùng nhớ song có một điều khác là vùng nhớ này thường là ngăn xếp, trỏ tương đối so với địa chỉ segment nằm trong thanh ghi segment SS.

- *Thanh ghi SP:*

Thanh ghi này dùng để trỏ đến đỉnh của ngăn xếp.

- *Các thanh ghi phân đoạn (segment registers):*

CPU 80x86 có 4 thanh ghi phân đoạn, đó là CS, DS, ES và SS.

- *Thanh ghi CS:*

Đây là thanh ghi phân đoạn cho vùng nhớ chứa mã lệnh. Giá trị của thanh ghi này trỏ đến phần địa chỉ segment khởi đầu của vùng nhớ chứa mã lệnh. Phần địa chỉ OFFSET của vùng nhớ chứa mã lệnh được trỏ bởi IP.

- *Thanh ghi DS:*

Đây là thanh ghi phân đoạn cho vùng nhớ dữ liệu (data segment). Giá trị của thanh ghi này trỏ đến phần địa chỉ segment khởi đầu của vùng nhớ RAM chứa dữ liệu (vùng nhớ cấp phát cho các biến).

*Chú ý:* Trong lệnh di chuyển dữ liệu (lệnh MOV) với các toán hạng là biến nhớ thì mặc định phần địa chỉ segment của vùng nhớ được cấp phát cho biến nhớ sẽ nằm trong thanh ghi DS và vì vậy 2 cách viết sau là tương đương:

`MOV AX, [value]`      và      `MOV AX, DS:[value]`

- *Thanh ghi ES:*

Đây là thanh ghi phân đoạn mở rộng cho vùng nhớ chứa dữ liệu (extra data segment) – vùng nhớ chứa dữ liệu thứ 2. Giá trị của thanh ghi này trỏ đến phần địa chỉ segment khởi đầu của vùng nhớ RAM chứa dữ liệu mở rộng.

*Chú ý:* Như đã đề cập ở phần trên, trong lệnh di chuyển dữ liệu (lệnh MOV) với các toán hạng là biến nhớ nằm trong vùng nhớ mở rộng thì mặc định phần địa chỉ segment của vùng nhớ được cấp phát cho biến nhớ sẽ nằm trong thanh ghi DS và vì vậy với các toán hạng là biến nhớ nằm trong vùng nhớ mở rộng phải khẳng định phần địa chỉ segment nằm trong thanh ghi ES, ví dụ:

`MOV AX, ES:[value]`

- *Thanh ghi SS:*

Đây là thanh ghi phân đoạn cho ngăn xếp (stack). Giá trị của thanh ghi này trỏ đến phần địa chỉ segment khởi đầu của vùng nhớ của ngăn xếp và ngăn xếp sẽ

sắp xếp tăng dần theo chiều giảm dần của vùng nhớ (có nghĩa là đáy của ngăn xếp nằm ở địa chỉ cao và đỉnh của ngăn xếp nằm ở địa chỉ thấp). Như vậy thanh ghi SS cùng với thanh ghi BP và SP bảo đảm toàn bộ sự hoạt động mềm dẻo của ngăn xếp. Ngăn xếp có một vai trò rất quan trọng trong cơ chế hoạt động của chương trình con và các lệnh PUSH-POP.

### ***b) Máy tính 32 bit***

Máy tính 32 bit thì có 16 thanh ghi, trong đó: thanh ghi cờ, 8 thanh ghi đa năng và thanh ghi con trỏ lệnh có cấu trúc 32 bit với tên các thanh ghi có chữ E đứng trước, ví dụ: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP. Song các thanh ghi phân đoạn (segment) của máy tính 32 bit vẫn là 16 bit và có thêm 2 thanh ghi phân đoạn nữa, đó là FS và GS để cùng với DS và ES tăng cường cho phần dữ liệu. Cấu trúc cụ thể của các thanh ghi đó như sau:



EFL

thanh ghi cờ

	A	AL
	AX	

	B	BL
	BX	

	C	CL
	CX	

	D	DL
	DX	

--	--

--	--

--	--

--	--

--	--

các thanh ghi  
đa năngthanh ghi  
con trỏ lệnhcác thanh ghi  
segment

– Thanh ghi cờ (*flag register*):

16 bit phần thấp của thanh ghi cờ giống như thanh ghi cờ của các hệ máy tính 16 bit. Riêng phần 16 bit cao của thanh ghi cờ có 2 bit cờ mới. Một trong 2 bit cờ này dùng để chỉ liệu máy tính 32 bit hiện hành chạy theo hệ lệnh nào: hệ lệnh 16 bit hay hệ lệnh 32 bit. Bit cờ còn lại được sử dụng cho các công cụ để thực hiện việc ghi trong chương trình gỡ rối DEBUG. Cả 2 bit cờ này không được dùng cho các phần mềm ứng dụng.

– Các thanh ghi đa năng (*general purpose registers*):

Như chúng ta thấy ở trên, các thanh ghi đa năng đều là 32 bit. Việc sử dụng 16 bit thấp của các thanh ghi này là điều khá quen thuộc với chúng ta khi đề cập đến hệ lệnh 8086. Như vậy là các phần của thanh ghi, ví dụ EAX có thể qui chiếu bởi các tên khác nhau là: thanh ghi 32 bit EAX, thanh ghi 16 bit AX và thanh ghi 8 bit AH và AL. Các thanh ghi EBX, RCX và EDX cũng tương tự như vậy.

Các thanh ghi đa năng được sử dụng ở bất kỳ nơi nào trong chương trình và về cơ bản không khác những gì chúng ta sử dụng các thanh ghi đa năng 16 bit trong hệ lệnh 8086. Ví dụ muốn gán giá trị 1 vào thanh ghi EAX, xóa thanh ghi EBX bằng 0 và sau đó cộng nội dung 2 thanh ghi đó lại thì có thể viết như sau:

```
...  
mov     EAX,1  
sub     EBX,EBX  
add     EBX,EAX  
...
```

Tuy nhiên có một điều cần lưu ý là không thể sử dụng trực tiếp 16 bit phần cao của các thanh ghi này như là một thanh ghi 16 bit độc lập. Để thâm nhập vào 16 bit phần cao của thanh ghi 32 bit (ví dụ đưa một giá trị vào 16 bit cao của thanh ghi EDX) chỉ có thể dùng lệnh quay vòng như sau:

```
...  
mov     AX,[value16Bit]    ; Đưa giá trị vào AX  
ror     EDX,16             ; Chuyển 16 bit phần cao và thấp
```

của EDX

; cho nhau

mov

DX,AX ; Đưa giá trị vào phần thấp EDX

ror

DX,16 ; Đưa giá trị vào 16 bit phần cao

của EDX và

; hồi phục lại giá trị cũ của 16 bit phần thấp

...

; EDX

– Thanh ghi con trỏ lệnh (*instruction pointer register*):

Thanh ghi con trỏ lệnh là 32 bit. Điều này sẽ mở rộng phần mã máy máy tính tuyến tính đến 4 GB.

*Chú ý:* Có một số lưu ý khi sử dụng lệnh nhảy không điều kiện đối với ngôn ngữ Assembly 32 bit:

- Với lệnh: `jmp [FWORD PTR JumpVector]`

thì hiển nhiên là lệnh nhảy xa với 16 bit segment và 32 bit offset

- Với lệnh: `jmp [DWORD PTR JumpVector]`

thì một câu hỏi đặt ra là lệnh nhảy này là lệnh nhảy gián tiếp gần 32 bit hay là lệnh nhảy gián tiếp xa với 16 bit segment và 16 bit offset? Có thể trả lời là cả 2 cách nhảy đều có thể thực hiện bằng lệnh nhảy này với việc ứng dụng toán tử LARGE và SMALL. Có nghĩa là:

`jmp SMALL [DWORD PTR JumpVector]`

sẽ được dịch như lệnh nhảy xa gián tiếp đến địa chỉ trỏ bởi 16 bit segment và 16 bit offset được cất ở JumpVector, còn

`jmp LARGE [DWORD PTR JumpVector]`

sẽ được dịch như lệnh nhảy gần gián tiếp đến địa chỉ trỏ bởi CS hiện tại và 32 bit offset được cất ở JumpVector.

Có một điều cũng cần lưu ý là toán tử SMALL và LARGE có thể xuất hiện bên ngoài hoặc bên trong dấu ngoặc []. Nên phân biệt rằng khi SMALL và LARGE xuất hiện bên ngoài dấu ngoặc thì nó có tác dụng đến kích cỡ của toán hạng, còn nếu nó đứng bên trong dấu ngoặc thì có tác dụng đến kích cỡ của địa chỉ.



– Các thanh ghi segment (segment registers):

Các máy tính 32 bit có trang bị thêm 2 thanh ghi segment, đó là FS và GS. Hai thanh ghi này không dành riêng cho một chức năng đặc biệt nào và không một lệnh nào cũng như ở chế độ địa chỉ nào thâm nhập FS và GS một cách mặc định. Do vậy, việc sử dụng các thanh ghi FS và GS thường là không cần thiết song đôi lúc cũng tương đối thuận tiện cho việc thâm nhập vào các dữ liệu ở nhiều segment cùng một lúc. Thường thì FS và GS được sử dụng giống như thanh ghi ES trong mọi trường hợp trừ các chức năng mà thanh ghi ES dành riêng cho các lệnh làm việc với xâu ký tự.

*Ví dụ:* Hãy nạp giá trị 0 vào một vùng nhớ.

.386

```
...  
TestSeg    SEGMENT    USE16  
    length  EQU  1000h  
    array   DB      length dup(?)  
TestSeg    ENDS
```

```
...  
CodeSeg    SEGMENT    USE16  
assume     CS:CodeSeg  
    mov     AX,TestSeg  
    mov     ES,AX  
    mov     FS,AX  
    mov     BX,OFFSET array  
    mov     CX,length  
    xor     AL,AL
```

LAP:

```
    mov     FS:[BX],AL  
    inc     BX  
    loop    LAP
```

...

...

CodeSeg ENDS

### 1.4.2. Cách thể hiện địa chỉ một ô nhớ (ROM hoặc RAM)

#### a) Cách thể hiện địa chỉ một ô nhớ dạng logic

Như chúng ta thấy là các thanh ghi segment của 80x86 đều là 16 bit, có nghĩa là nếu dùng một thanh ghi 16 bit làm con trỏ địa chỉ thì nó chỉ quản lý

được một vùng nhớ 65536 byte (64k). Song vùng nhớ của máy tính hiện nay thường rất lớn. Do vậy, để giải quyết vấn đề con trỏ địa chỉ đến các ô nhớ (RAM hoặc ROM) phải dùng 2 thanh ghi để thể hiện địa chỉ một ô nhớ, trong đó:

- một thanh ghi cho biết ô nhớ đó nằm ở 64k thứ mấy (phần địa chỉ SEGMENT) và
- thanh ghi còn lại cho biết khoảng cách từ đầu segment (64k) đó đến vị trí ô nhớ đó (phần địa chỉ OFFSET).

Vậy công thức thể hiện địa chỉ một ô nhớ dạng logic sẽ như sau:

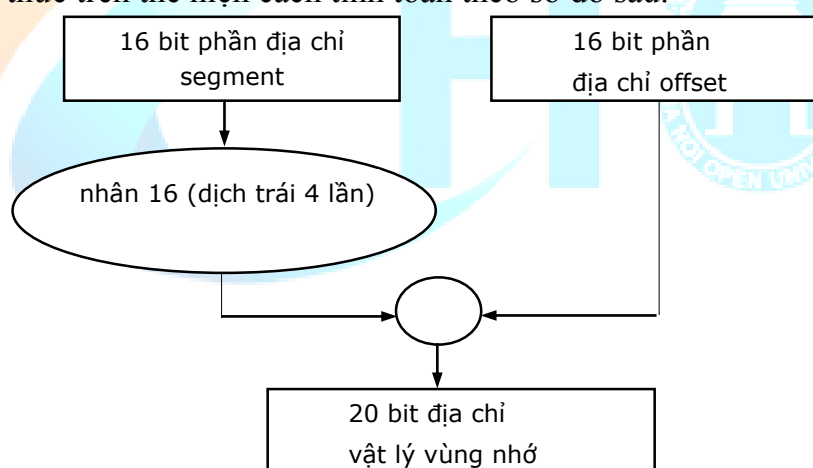
$$\text{địa chỉ một ô nhớ} = \text{SEG:OFFSET}$$

### ***b) Cách thể hiện địa chỉ một ô nhớ dạng vật lý***

Cách biểu diễn địa chỉ một ô nhớ dạng logic (SEG:OFFSET) được hầu hết các ngôn ngữ sử dụng. Song đôi lúc cần biết địa chỉ thực (địa chỉ dạng vật lý) của một ô nhớ. Địa chỉ thực của một ô nhớ được tính toán như sau:

$$\text{địa chỉ 1 ô nhớ} = \text{phần địa chỉ SEGMENT} * 16 + \text{phần địa chỉ OFFSET}$$

Công thức trên thể hiện cách tính toán theo sơ đồ sau:



*Ví dụ:* Có 2 ô nhớ với các địa chỉ dạng logic sau:

3E30:0170 và 3E3F:0080

Địa chỉ vật lý của 2 ô nhớ đó sẽ như sau:

3E300h      3E3F0h (nhân phần địa chỉ segment 16 lần)

170h	80h
-----	-----
3E470h	3E470h

Chúng ta thấy về vật lý chỉ là một ô nhớ.

### 1.4.3. Các mode địa chỉ của lệnh

Mode đánh địa chỉ (Addressing mode) là phương pháp xác định vị trí trong bộ nhớ của một toán hạng. Mode địa chỉ của một toán hạng phụ thuộc vào vị trí trong bộ nhớ của dữ liệu ứng với toán hạng đó. Có nhiều cách phân loại mode địa chỉ và sau đây xin trình bày một cách phân loại. Với cách phân loại này, có thể chia làm 7 mode địa chỉ và cụ thể như sau:

#### a) Mode địa chỉ thanh ghi (Register Addressing)

Trong mode địa chỉ này vị trí của các toán hạng chính là các thanh ghi.

Ví dụ:     MOV     AX,BX

#### b) Mode địa chỉ tức thời (Immediate Addressing)

Trong mode địa chỉ này có một toán hạng là giá trị (value).

Ví dụ:     MOV   CX,-125  
          MOV   CX,HANG\_SO   ; trong đó HANG\_SO đã được gán bằng một  
                                  ; giá trị: HANG\_SO EQU -125)

#### c) Mode địa chỉ trực tiếp (Direct Addressing)

Trong mode địa chỉ này giá trị một toán hạng là địa chỉ ô nhớ (tên nhãn trong ngôn ngữ Assembly sẽ được thay bằng địa chỉ khi dịch chương trình).

Ví dụ:     MOV   BX,table                   ; trong đó table là một nhãn

#### d) Mode địa chỉ gián tiếp thanh ghi (Register Indirect Addressing)

Trong mode địa chỉ này một ô nhớ được trỏ bởi giá trị của thanh ghi. Chú ý là khi thanh ghi nằm trong dấu [ ] (mở đóng ngoặc) thì giá trị thanh ghi là con trỏ phần địa chỉ offset của một ô nhớ.

Ví dụ:     MOV   AX,[BX]                   ; nội dung 2 byte của ô nhớ được trỏ bởi

; DS:[BX] được đưa vào thanh ghi AX

Cần lưu ý là người lập trình chỉ được phép dùng 4 thanh ghi là: BX, SI, DI và BP làm con trỏ offset của một ô nhớ.

***e) Mode địa chỉ tương đối cơ sở (Base Relative Addressing)***

Trong mode địa chỉ này địa chỉ một ô nhớ được trỏ bởi giá trị của thanh ghi con trỏ cộng với một giá trị.

*Ví dụ:*     MOV   AX,[BX]+4

f) Mode địa chỉ được chỉ số hóa trực tiếp (Direct Indexed Addressing)

*Ví dụ:*     MOV   AX,table[DI]     ; trong đó table là biến nhớ 1 byte

***g) Mode địa chỉ được chỉ số hóa cơ sở (Base Indexed Addressing)***

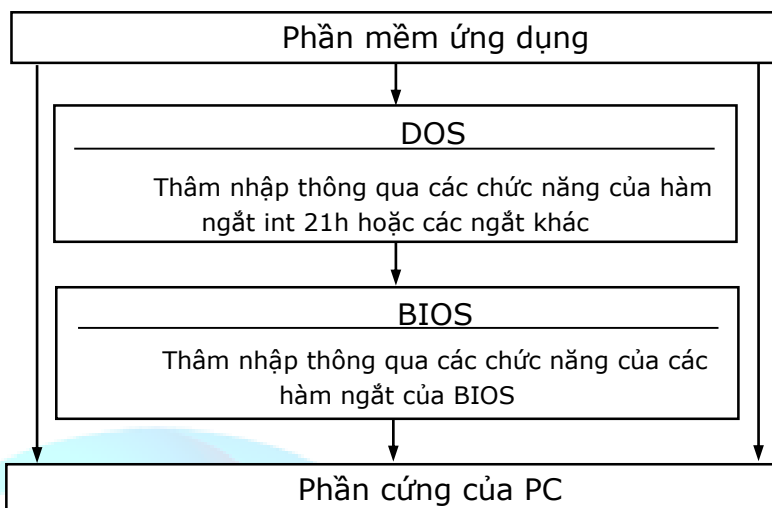
Trong mode địa chỉ này một ô nhớ được trỏ bởi giá trị của thanh ghi kết hợp với một chỉ số. Con trỏ này trỏ đến đầu một vùng nhớ cùng kiểu (byte hoặc word) và chỉ số là một giá trị cho biết ô nhớ nằm cách vị trí do con trỏ trỏ đến bao nhiêu phần tử (kiểu byte hoặc word).

*Ví dụ:*     MOV    AX,[BX][DI+2]

**1.4.4. Các phần mềm hệ thống cho IBM PC**

Các phần mềm hệ thống là lớp phần mềm phục vụ cho việc điều hành và ghép nối giữa các phần mềm ứng dụng và phần cứng của máy tính. Trong các máy vi tính, DOS và BIOS là hai phần mềm hệ thống phục vụ cho nhiệm vụ trên. Các phần mềm ứng dụng được viết bằng ngôn ngữ bậc cao muốn điều khiển và ghép nối với phần cứng của máy tính thường thông qua hai công cụ trên. Các chương trình ứng dụng được viết bằng ngôn ngữ Assembly cũng thường sử dụng hai phần mềm trên để điều khiển và ghép nối với phần cứng thông qua các chức năng của hàm ngắt int 21h (DOS functions), các ngắt khác của DOS hoặc các ngắt của BIOS. Tuy nhiên vì ngôn ngữ Assembly là ngôn ngữ bậc thấp, rất gần với ngôn ngữ máy, do vậy người viết chương trình bằng ngôn ngữ Assembly phải am hiểu về phần cứng của máy tính tốt và phải thông thạo ngôn ngữ Assembly để trên cơ sở đó viết các ứng dụng trực tiếp điều khiển và ghép nối với phần cứng của máy.

Hình dưới đây cho thấy khả năng điều khiển và ghép nối của một ứng dụng được thể hiện bằng ngôn ngữ Assembly.



Sau đây chúng ta xem xét các hàm VÀO/RA thường dùng của ngôn ngữ Assembly thông qua các chức năng của hàm ngắt int 21h của DOS và các ngắt của BIOS.

**a) Nhận một ký tự dạng ASCII từ bàn phím:**

```
mov    AH,1  
int     21h
```

Khi gặp hàm này máy tính chờ ấn phím. Khi một phím được ấn thì mã ASCII của phím được ấn sẽ nằm trong thanh ghi AL.

**b) Đưa một ký tự dạng ASCII ra màn hình tại vị trí con trỏ:**

*Cách 1:* Nhờ chức năng thứ 2 của ngắt int 21h (DOS functions)

```
mov     DL, mã ASCII của phím cần hiện  
mov     AH,2  
int     21h
```

*Cách 2:* Nhờ chức năng thứ 0Eh của ngắt int 10h (ngắt BIOS)

```
mov     AL, mã ASCII của phím cần hiện  
mov     AH,0Eh  
int     10h
```

**c) Hiện một chuỗi ký tự kết thúc bằng '\$' lên màn hình tại vị trí con trỏ:**

```
lea     DX, tên biến chuỗi  
mov     AH,9  
int     21h
```

*Ví dụ:*

.DATA

Message db 'Hello, world ! \$'

.CODE

...  
...

```
lea     DX, Message  
mov     AH,9  
int     21h
```

**d) Rời chương trình trở về DOS:**

```
mov     AH,4Ch  
int     21h
```

Chú ý: Thay vì chức năng trở về DOS như trên, đối với chương trình dạng .COM có thể trở về DOS bằng ngắt int 20h của DOS.

## 1.5. TẬP LỆNH CỦA NGÔN NGỮ ASSEMBLY

### 1.5.1. Cú pháp một dòng lệnh

Chương trình dịch Assembler có hai loại tập lệnh chính sau: tập lệnh điều khiển khi dịch chương trình (directive) cho các segment và các nhiệm vụ khác và tập lệnh mnemonic (instruction set).

Cú pháp tổng quát của một dòng lệnh được viết bằng ngôn ngữ Assembly có dạng sau:

[label] [directive/instruction] [operands] [; comment]

có nghĩa:

[nhãn] [lệnh điều khiển/ lệnh] [toán hạng] [; ghi chú]

trong đó:

- label (nhãn): là một định danh (identifier) dùng để qui chiếu (thể hiện) đến các số, các xâu ký tự, các biến ô nhớ, tên chương trình con hoặc nhãn cho các lệnh nhảy, ví dụ như:

- nhãn thể hiện một hằng số (nhãn được gán một số):

CONG EQU 10Fh ; không được định nghĩa lại

hoặc

CONG = 10Fh ; cho phép định nghĩa lại

- nhãn là tên một biến xâu ký tự:

.DATA

Message db 'Hello, world \$'

- nhãn là tên một biến nhớ:

.DATA

value1 db 10h  
value2 dw ?

- nhãn là tên một chương trình con:

CT\_CON PROC NEAR

- nhãn của lệnh nhảy:

...

LABEL:

...

...

jmp LABEL

- directive/instruction (lệnh điều khiển khi dịch chương trình/lệnh dạng mnemonic): sẽ đề cập chi tiết ở các phần sau (phụ lục).
- operands (các toán hạng): chỉ cho chương trình dịch Assembler biết thanh ghi nào, tham số nào, nhãn nào (tên biến nhớ, tên chương trình con, tên hằng, tên nhãn nhảy, ...) sẽ liên quan đến lệnh điều khiển khi dịch chương trình (directive) hoặc lệnh (instruction). Thực tế có các dạng toán hạng sau:
  - toán hạng thanh ghi,
  - toán hạng hằng: trong chương trình Assembly thì một hằng số có thể là dạng cơ số bất kỳ. Ví dụ: int 21h (thể hiện dạng hexa) tương đương với int 00100001b (thể hiện dạng cơ số 2) hoặc int 33 (thể hiện dạng cơ số 10),
  - toán hạng biểu thức, và
  - toán hạng nhãn.
- comment (chú thích): đối với chương trình được viết bằng ngôn ngữ Assembly thì chú thích rất quan trọng, không nên coi thường. Chú thích giúp cho người lập trình khỏi quên những gì đã làm đồng thời giúp người khác dễ dàng tìm hiểu chương trình. Với ngôn ngữ Assembly thì từ dấu ; (chấm phẩy) cho đến hết dòng là chú thích và dấu ; (chấm phẩy) chỉ có hiệu lực trên một dòng.

*Chú ý:* Khác với nhiều ngôn ngữ bậc cao có thể viết nhiều lệnh trên một dòng (thường cách nhau bởi dấu ;), mỗi một lệnh của ngôn ngữ Assembly chiếm một dòng.

### 1.5.2. Tập lệnh mnemonic (instruction set)

Đây là tập lệnh sinh ra mã máy để chạy chương trình. Các lệnh dạng mnemonic của ngôn ngữ Assembly có thể chia làm 6 nhóm:



- Nhóm các lệnh chuyển dữ liệu,
- Nhóm các lệnh số học,
- Nhóm các lệnh thao tác với các bit,
- Nhóm các lệnh rẽ nhánh,
- Nhóm các lệnh còn lại.

Chúng ta sẽ xem xét kỹ các nhóm lệnh này ở phần phụ lục A.

### **1.5.3. Các lệnh điều khiển khi dịch chương trình (directive)**

Các lệnh điều khiển khi dịch chương trình không sinh ra mã máy để chạy chương trình mà chỉ hỗ trợ cho chương trình dịch Assembler. Trong các lệnh điều khiển khi dịch chương trình thì các lệnh điều khiển segment rất quan trọng. Ngoài các lệnh điều khiển segment thì còn có một số lệnh điều khiển khác.

#### **1.5.3.1. Các lệnh điều khiển segment (segment directives)**

Chương trình dịch Assembler trang bị hai tập lệnh điều khiển segment, đó là: tập lệnh điều khiển segment dạng đơn giản và tập lệnh điều khiển segment dạng chuẩn. Tập lệnh điều khiển segment dạng đơn giản dễ sử dụng song không có nhiều tùy chọn về điều khiển segment như tập lệnh điều khiển segment dạng chuẩn.

#### **A. Các lệnh điều khiển segment dạng đơn giản (simplified segment directives):**

Các lệnh điều khiển segment dạng đơn giản dễ sử dụng và dễ liên kết với ngôn ngữ bậc cao. Trong tập lệnh điều khiển segment dạng đơn giản có các lệnh điều khiển chính sau: .MODEL, .STACK, .DATA, .CODE và DOSSEG. Hãy xem xét ý nghĩa và chức năng của các lệnh điều khiển này.

##### **a) Lệnh điều khiển .MODEL**

- *Chức năng:* Lệnh điều khiển .MODEL báo cho chương trình dịch Assembler biết để xác lập mô hình bộ nhớ thích hợp cho chương trình.

*Chú ý:* Nếu người lập trình chọn mô hình bộ nhớ thích hợp cho chương trình (chương trình nhỏ thì xác lập bộ nhớ bé, ngược lại chương trình lớn thì xác lập bộ nhớ lớn) thì chương trình chạy sẽ tối ưu. Chúng ta cũng cần lưu ý rằng nếu là NEAR trong phần mã lệnh (NEAR CODE) thì các lệnh rẽ nhánh chỉ quan tâm

đến phần địa chỉ OFFSET nằm trong thanh ghi IP, còn nếu là FAR thì cần quan tâm đến cả phần địa chỉ SEGMENT nằm ở thanh ghi CS và phần địa chỉ OFFSET nằm trong thanh ghi IP. Tương tự như vậy đối với phần dữ liệu. Nếu phần dữ liệu là NEAR DATA thì việc thâm nhập vào các dữ liệu nằm trong vùng này chỉ cần phần địa chỉ OFFSET trong khi muốn thâm nhập vào các dữ liệu nằm trong vùng FAR DATA thì phải cần cả phần địa chỉ SEGMENT lẫn OFFSET. Nói một cách đơn giản là đối với các vùng được khai báo là FAR thì 32 bit (4 byte) địa chỉ dạng SEGMENT:OFFSET được sử dụng, còn NEAR thì 16 bit (2 byte) phần địa chỉ OFFSET được sử dụng.

- *Cú pháp:* .MODEL kiểu mô hình bộ nhớ

Các kiểu mô hình bộ nhớ cho phép gồm:

- tiny: cả phần mã máy (CODE) và phần dữ liệu (DATA) của chương trình nằm trong 1 segment (64k). Cả CODE và DATA đều NEAR.
- small: phần mã máy (CODE) của chương trình nằm trong 1 segment (64k) và phần dữ liệu (DATA) của chương trình nằm trong một segment (64k) khác. Cả CODE và DATA đều NEAR.
- compact: phần mã máy (CODE) của chương trình nằm trong 1 segment (64k) và phần dữ liệu (DATA) của chương trình có thể nằm trong vùng nhớ lớn hơn 64k. CODE là NEAR còn DATA là FAR.
- medium: phần mã máy (CODE) của chương trình có thể lớn hơn 64k song phần dữ liệu (DATA) của chương trình chỉ nằm trong 1 segment (64k). CODE là FAR còn DATA là NEAR.
- large: cả phần mã máy (CODE) của chương trình và phần dữ liệu (DATA) của chương trình có thể nằm trong vùng nhớ lớn hơn 64k. Song một trường số liệu không được vượt quá 64k. Cả CODE và DATA đều là FAR.
- huge: cả phần mã máy (CODE) của chương trình và phần dữ liệu (DATA) của chương trình có thể nằm trong vùng nhớ lớn hơn 64k. Một trường số liệu có thể vượt quá 64k. Cả CODE và DATA đều FAR.

#### ***b) Lệnh điều khiển .STACK***

- **Chức năng:** Lệnh điều khiển `.STACK` báo cho chương trình dịch Assembler biết để xác lập một vùng nhớ cho ngăn xếp. Với lệnh điều khiển này, DOS sẽ xác lập địa chỉ đầu của vùng nhớ cấp phát cho ngăn xếp và phần địa chỉ segment của vùng nhớ đó sẽ được ghi vào thanh ghi segment `SS`.
- **Cú pháp:** `.STACK` kích cỡ của ngăn xếp (tính theo đơn vị byte)

*Ví dụ:* `.STACK 100h` (xác lập vùng nhớ 256 byte cho ngăn xếp)

**Chú ý:** Nếu trong chương trình không sử dụng các lệnh gọi chương trình con (`CALL`), các lệnh `PUSH`, `POP` hoặc các ngắt mềm (`INT n`) thì không cần khai báo lệnh điều khiển `.STACK`. Ngay cả trong trường hợp có sử dụng các lệnh trên trong chương trình mà không có khai báo `.STACK` thì chương trình vẫn chạy vì lúc này chương trình dịch Assembler sẽ xác lập một vùng nhớ mặc định cho ngăn xếp do DOS qui định song có phần mạo hiểm vì có thể các lệnh trên chiếm một vùng nhớ lớn hơn vùng nhớ mặc định cho ngăn xếp. Do đó, một lời khuyên tốt nhất và đơn giản nhất là nên khai báo lệnh điều khiển `.STACK` trong mọi trường hợp.

### c) Lệnh điều khiển `.DATA`

- **Chức năng:** Lệnh điều khiển `.DATA` báo cho chương trình dịch Assembler biết để xác lập một vùng nhớ và đánh dấu điểm khởi đầu của vùng nhớ cho phần dữ liệu. Chúng ta phải đặt việc khai báo các biến nhớ vào trong vùng nhớ dữ liệu này và có thể gán giá trị ban đầu (khởi động) cho chúng. Các ô nhớ của vùng nhớ này sẽ được dùng để cấp phát cho các biến nhớ được khai báo.
- **Cú pháp:** `.DATA`

phân khai báo biến (có thể gán giá trị ban đầu)

Trước khi xem xét cụ thể cách khai báo các loại biến của ngôn ngữ Assembly, hãy xem các loại biến mà ngôn ngữ Assembly hay dùng. Ngôn ngữ Assembly hay dùng 3 loại biến, đó là: biến dạng số, biến dạng chuỗi ký tự và biến dạng trường số.

- Khai báo biến dạng số

Các directive `DB`, `DW`, `DD`, `DF/DP`, `DQ` và `DT` cho phép khai báo các loại biến số, cụ thể như sau:

Tên biến		kiểu biến		gán giá trị ban đầu hoặc ?
----------	--	-----------	--	----------------------------

DB (biến 1 byte)
DW (biến 2 byte)
DD (biến 4 byte)
DF/DP (biến 6 byte)
DQ (biến 8 byte)
DT (biến 10 byte)

Ví dụ:

.DATA

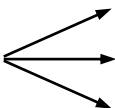
value1 DB 10h ; biến value1 xin cấp phát 1 byte và giá trị  
; ban đầu của byte đó là 10h

value2 DW ? ; biến value2 xin cấp phát 2 byte và không  
; gán giá trị ban đầu vào 2 byte đó

value3 DD ? ; biến value3 xin cấp phát 4 byte và không  
; gán giá trị ban đầu vào 4 byte đó

- Khai báo biến dạng chuỗi ký tự

Biến dạng chuỗi ký tự được thực hiện với cú pháp sau:

tên biến chuỗi DB  các ký tự cách nhau bởi dấu ,  
độ lớn dup(1 ký tự)  
độ lớn dup (?)

Ví dụ:

.DATA

xau1 DB 'H','e','l','l','o' ; biến xau1 gồm 5 ký tự, mỗi ký tự 1 byte


xau2 DB 100 DUP('A') ; biến xau2 xin cấp phát 100 byte và giá trị  
; 100 byte đó ban đầu được gán ký tự 'A'

xau3 DD 100h DUP(?) ; biến xau3 xin cấp phát 256 byte và không  
; gán giá trị ban đầu vào các byte đó

Chú ý: biến xau1 có thể khai báo như sau:

- xau1 DB 'Hello' ; biến xau1 gồm 5 ký tự, mỗi ký tự 1 byte
- Khai báo biến dạng trường số

Biến dạng trường số được thực hiện với cú pháp sau:

tên trường kiểu thành phần  các số cách nhau bởi dấu ,  
độ lớn dup(1 số)  
độ lớn dup (?)

Ví dụ:

.DATA

array1 DB 10,-1,5,4,105 ; biến array1 có 5 thành phần, mỗi thành phần chiếm 1 byte và lần lượt được gán các giá trị 10, -1, 5, 4 và 105

array2 DW 100 dup(-105) ; biến array2 có 100 thành phần, mỗi thành phần chiếm 2 byte và tất cả các thành phần được gán giá trị -105

array3 DD 100h DUP(?) ; (biến array3 có 256 thành phần, mỗi thành phần chiếm 4 byte và không gán giá trị ban đầu vào các thành phần đó)

**Chú ý quan trọng:**

Để hoàn tất việc xác định vùng dữ liệu, người lập trình cần phải nạp vào thanh ghi DS giá trị @data (một hằng số thể hiện phần địa chỉ segment của vùng nhớ RAM cấp phát cho phần dữ liệu) trước khi thâm nhập vào vùng dữ liệu đã được khai báo bởi lệnh điều khiển .DATA. Như chúng ta biết thanh ghi segment chỉ có thể được nạp một giá trị là hằng số từ thanh ghi đa năng hoặc từ giá trị một vùng nhớ chứ không thể nhận trực tiếp một hằng số. Do vậy phải tiến hành gián tiếp thông qua một thanh ghi đa năng theo cú pháp sau:

```
mov     reg16,@data
```

```
mov     DS,reg16
```

Thông thường sử dụng thanh ghi đa năng AX, do vậy cú pháp trên ví dụ sẽ có dạng sau:

```
mov    AX,@data
mov    DS,AX
```

#### ***d) Lệnh điều khiển .CODE***

- **Chức năng:** Lệnh điều khiển .CODE báo cho chương trình dịch Assembler biết để xác lập một vùng nhớ cho phần mã máy của chương trình và đánh dấu điểm khởi đầu của vùng nhớ đó.
- **Cú pháp:** .CODE

nhãn\_chương\_trình:

thân chương trình

END nhãn\_chương\_trình

*Ví dụ:*

```
...
.CODE
PS:
mov    AX,@data
mov    DS,AX
...
...
mov    AH,4Ch
int    21h
END    PS
```

Chúng ta vừa giới thiệu các lệnh điều khiển segment dạng đơn giản hay dùng. Ngoài các lệnh điều khiển trên còn có các lệnh điều khiển segment dạng đơn giản khác. Tuy các lệnh điều khiển segment này ít được sử dụng song chúng ta hãy lướt qua các lệnh điều khiển đó.

#### ***e) Lệnh điều khiển DOSSEG***

- *Chức năng:* Lệnh điều khiển DOSSEG báo cho chương trình dịch Assembler nhóm các segment của chương trình theo thứ tự qui định của hãng Microsoft và Intel. Các segment sẽ được nhóm theo thứ tự qui định sau:
  - Các segment có tên lớp (class name) là ‘CODE’,
  - Tất cả các segment không phải trong ‘CODE’ và không phải trong DGROUP,
  - Tất cả các segment ở DGROUP.
- *Cú pháp:* DOSSEG

#### ***f) Lệnh điều khiển .DATA?***

- *Chức năng:* Lệnh điều khiển này giống hệt như lệnh điều khiển .DATA song có một điều khác là các khai báo biến nhớ nằm trong vùng này không được gán giá trị ban đầu. Lệnh này sử dụng trong liên kết module viết bằng ngôn ngữ Assembly với các module ngôn ngữ bậc cao.

#### ***g) Lệnh điều khiển .FARDATA***

- *Chức năng:* Lệnh điều khiển này cho phép xác lập một segment dữ liệu xa (FAR DATA SEGMENT) khác với vùng dữ liệu chuẩn @data. Lệnh điều khiển .FARDATA cho phép một module viết bằng ngôn ngữ Assembly định nghĩa được một data segment riêng có độ lớn đến 64k.

*Chú ý:* Nếu dùng lệnh điều khiển .FARDATA thì thay vì gán giá trị @data cho lệnh điều khiển .DATA thì phải gán giá trị @fardata.

#### ***h) Lệnh điều khiển .FARDATA?***

- *Chức năng:* Lệnh điều khiển này giống lệnh điều khiển .FARDATA song có một điều khác là các khai báo biến nhớ nằm trong vùng này không được gán giá trị ban đầu khi khai báo. Hãy nhớ gán @fardata vào thanh ghi DS chứa phần địa chỉ segment của vùng nhớ dữ liệu.

#### ***i) Lệnh điều khiển .CONST***

- *Chức năng:* Lệnh điều khiển này xác lập một segment dữ liệu bao gồm các hằng số. Một lần nữa xin nhắc lại là điều này chỉ có nghĩa khi liên quan đến sự liên kết với ngôn ngữ bậc cao.

Sau khi đã nghiên cứu hệ lệnh mnemonic (instruction set) của ngôn ngữ Assembly trong phần phụ lục A. và các lệnh điều khiển khi dịch chương trình (directive), đặc



biệt là các lệnh điều khiển segment dạng đơn giản, chúng ta hãy nêu lên dạng thường thấy của một chương trình đơn giản thể hiện bằng ngôn ngữ Assembly sử dụng các lệnh điều khiển segment dạng đơn giản:

.MODEL kiểu

.STACK độ lớn ; Tính theo đơn vị byte

[.DATA

khai báo biến ]

.CODE

nhãn\_chương\_trình:

[mov AX,@data ; Chỉ có khi có sử dụng .DATA và khai

mov DS,AX] ; báo biến

...

các lệnh ASM

của thân chương trình

...

mov AH,4Ch ; Trở về DOS

int 21h

END nhãn\_chương\_trình

Như vậy là chúng ta đã xem xét qua một số phần cơ bản nhất của ngôn ngữ Assembly. Trên cơ sở các phần vừa trình bày, chúng ta có thể viết được một số chương trình đơn giản để củng cố những kiến thức trên.

*Ví dụ 1:*

Hãy viết chương trình bằng ngôn ngữ Assembly hiện một xâu ký tự lên màn hình.

Có nhiều cách để hiện một xâu ký tự lên màn hình, sau đây xin trình bày 3 cách để trên cơ sở đó bước đầu làm quen với lập trình bằng ngôn ngữ Assembly.

*Cách 1:*

Sử dụng hàm hiện một ký tự kết thúc bằng ký tự '\$' lên màn hình (xem phần 1.4.4).

.MODEL small

.STACK 100h ; Ngăn xếp gồm 256 byte

.DATA

M db 'Hello, world ! \$'

.CODE

ProgramStart:

mov AX,@data ; Đưa phần địa chỉ segment của vùng nhớ



```
mov    DS,AX          ; chứa dữ liệu vào DS
lea     DX,M           ; DX=con trỏ OFFSET đầu chuỗi M
mov     AH,9           ; Chức năng hiển 1 chuỗi (kết thúc bằng '$')
int     21h           ; lên màn hình
mov     AH,4Ch         ; Trở về DOS
int     21h
END     ProgramStart
```

### Cách 2:

Sử dụng khai báo chuỗi theo khuôn dạng của ngôn ngữ C/C++ (có nghĩa chuỗi kết thúc bằng \0) và dùng các lệnh làm việc với chuỗi ký tự, lệnh `lodsb` (phần A.4. của phần phụ lục A).

```
.MODEL small
.STACK 100h          ; Ngăn xếp gồm 256 byte
.DATA
M    DB  'Hello, world ! ',0
.CODE
ProgramStart:
mov     AX,@data      ; Đưa phần địa chỉ segment của vùng nhớ
mov     DS,AX         ; chứa dữ liệu vào DS
lea     SI,M           ; SI=con trỏ OFFSET đầu chuỗi
cld                     ; bit cờ DF=0 (theo chiều tăng địa chỉ)
L1:
lodsb                    ; Ký tự trỏ bởi DS:SI→AL (SI tăng 1)
and     AL,AL         ; Liệu AL=0 (kết thúc chuỗi)?
jz      Exit          ; Nếu bằng 0 thì nhảy về để kết thúc còn
mov     AH,0Eh        ; chưa bằng 0 thì hiển ký tự dạng ASCII
int     10h           ; nằm trong AL lên màn hình
jmp     L1
Exit:
mov     AH,4Ch         ; Trở về DOS
int     21h
END     ProgramStart
```

### Cách 3:

Sử dụng khai báo chuỗi theo khuôn dạng của C/C++ (có nghĩa chuỗi kết thúc bằng \0) và dùng lệnh chuyển dữ liệu `mov` (phần A.1. phần phụ lục A).

```
.MODEL small
.STACK 100h          ; Ngăn xếp gồm 256 byte
```

.DATA

M DB 'Hello, world ! ',0

.CODE

ProgramStart:

mov AX,@data ; Đưa phần địa chỉ segment của vùng nhớ  
mov DS,AX ; chứa dữ liệu vào DS  
lea SI,M ; SI=con trỏ OFFSET đầu xâu M

L1:

mov AL,[SI] ; Ký tự trỏ bởi DS:SI→AL  
and AL,AL ; Liệu AL=0 (kết thúc xâu)?  
jz Exit ; Nếu bằng 0 thì nhảy về để kết thúc, còn  
mov AH,0Eh ; chưa bằng 0 thì hiện ký tự dạng ASCII nằm  
int 10h ; trong thanh ghi AL lên màn hình  
inc SI ; SI trỏ đến ký tự tiếp theo của xâu M  
jmp L1

Exit:

mov AH,4Ch ; Trở về DOS  
int 21h  
END ProgramStart

*Ví dụ 2:*

Hãy viết chương trình tính tổng của một dãy số tự nhiên khi biết số đầu và số lượng chữ số.

.MODEL small

.STACK 100h ; Ngăn xếp gồm 256 byte

.DATA

so\_dau DB 2  
so\_luong DW 10  
tong DW ?

.CODE

ProgramStart:

mov AX,@data ; Đưa phần địa chỉ segment của vùng nhớ  
mov DS,AX ; chứa dữ liệu vào DS  
mov CX,so\_luong ; Sẽ sử dụng cho vòng lặp LOOP  
mov AX,so\_dau  
mov tong,AX ; tong bằng giá trị số đầu  
dec CX ; CX chứa chỉ số vòng lặp

LAP:

inc AX ; Số tự nhiên tiếp theo

```
add    tong,AX          ; Cộng với biến tong (kết quả ở biến tong)
loop   LAP
mov     AH,4Ch           ; Trở về DOS
int     21h
END     ProgramStart
```

Ví dụ 3:

Hãy viết chương trình tính 5!.

Cách 1: Khai báo biến.

```
.MODEL small
.STACK 100h          ; Ngăn xếp gồm 256 byte
.DATA
    FV    DW    ?
    Fac    DW    ?
.CODE
    ProgramStart:
    mov     AX,@data    ; Đưa phần địa chỉ segment của vùng nhớ
    mov     DS,AX        ; chứa dữ liệu vào DS
    mov     FV,1         ; Gán giá trị biến FV=1
    mov     Fac,2        ; Gán giá trị biến Fac=2
    mov     CX,4         ; CX là chỉ số vòng lặp
LAP:
    mov     AX,FV        ; Gán AX=FV
    mul     Fac          ; AX*Fac → DX:AX (DX=0)
    mov     FV,AX        ; Gán FV = giá trị có trong AX
    inc     Fac          ; Tăng giá trị biến Fac lên 1
    loop    LAP
    mov     AH,4Ch       ; Trở về DOS
    int     21h
    END     ProgramStart
```

Chú ý:

- Kết quả nằm trong biến FV song chưa hiện được ra màn hình.
- Muốn tính  $n!$  thì  $CX=n-1$  (với giới hạn  $0 \leq n \leq 7$ )

Cách 2: Chỉ dùng thanh ghi (không cần khai báo biến).

```
.MODEL small
.STACK 100h          ; Ngăn xếp gồm 256 byte
.CODE
```

```
ProgramStart:
mov     AX,1           ; Gán thanh ghi AX=1
mov     CX,5           ; CX=5 (chỉ số vòng lặp)
LAP:
mul     CX             ; AX*CX → DX:AX (DX=0)
loop    LAP
mov     AH,4Ch         ; Trở về DOS
int     21h
END     ProgramStart
```

**Chú ý:**

- Trước khi về DOS kết quả nằm trong thanh ghi AX song chưa hiện được ra màn hình.
- Muốn tính  $n!$  thì  $CX=n$  (với giới hạn  $0 \leq n \leq 7$ ).

**Ví dụ 4:**

Hãy viết chương trình tính tổng hai số nguyên. Để đơn giản chương trình, hãy giới hạn tổng của hai số có thể nằm từ -128 đến 127.

```
.MODEL small
.STACK 100h           ; Ngăn xếp gồm 256 byte
.DATA
sh1      DW      10h   ; sh1= 16
sh2      DW      82h   ; sh2= -2
tong     DW      ?
.CODE
ProgramStart:
mov     AX,@data      ; Đưa phần địa chỉ segment của vùng nhớ
mov     DS,AX         ; chứa dữ liệu vào DS
mov     BL,sh1        ; Gán giá trị BL=sh1
mov     BH,BL         ; BH=BL
and     BX,807Fh      ; BH chứa dấu, BL chứa giá trị |sh1|
mov     CL,sh2        ; Gán giá trị CL=sh2
mov     CH,CL
and     CX,807Fh      ; CH chứa dấu, CL chứa giá trị |sh2|
cmp     BH,CH         ; So sánh dấu 2 số
jz      L2            ; Cùng dấu thì nhảy đến L2 còn khác dấu thì
mov     AL,BL         ; tiếp tục
sub     AL,CL         ; Trừ hai giá trị tuyệt đối cho nhau
jnc     L1            ; |sh1| > |sh2| thì nhảy đến L1
```

```
xchg    BX,CX                ; ngược lại thì đổi hai số cho nhau (cả dấu và
                                ; giá trị tuyệt đối)

L1:
sub     BL,CL                ; Tính giá trị tuyệt đối của tổng
jmp     L3
L2:
add     BL,CL                ; Cùng dấu thì cộng 2 giá trị tuyệt đối
L3:
and     BL,BL                ; Liệu tổng bằng 0 ?
jnz     L4                  ; Không bằng 0 thì nhảy
mov     tong,0               ; tong=0 (tránh trường hợp có dấu -)
jmp     Exit
L4:
or      BL,BH                ; Gộp dấu vào kết quả tổng
mov     tong,BL
Exit:
mov     AH,4Ch               ; Trở về DOS
int     21h

END     ProgramStart
```

*Chú ý:* Chưa hiện được kết quả ra màn hình.

*Ví dụ 5:*

Hãy viết chương trình nhận chuỗi ký tự từ bàn phím (kết thúc chuỗi bằng tổ hợp Ctrl\_Z), sau đó hiện ngược chuỗi vừa vào.

*Cách làm:*

Bài này thể hiện cơ chế LIFO (Last In-First Out – Cái vào ngăn xếp trước sẽ được lấy ra sau) của tổ hợp lệnh PUSH-POP (đặt giá trị vào và lấy giá trị ra khỏi ngăn xếp).

```
.MODEL small

.STACK    100h                ; Ngăn xếp gồm 256 byte

.DATA
M1        DB    13,10,'Hay vào xau : $'
M2        DB    13,10,'Hien nguoc xau vua vào : $'

.CODE
ProgramStart:
```

```
mov    AX,@data        ; Đưa phần địa chỉ segment của vùng nhớ
mov    DS,AX            ; chứa dữ liệu vào DS
lea     DX,M1           ; Trỏ đến đầu xâu M1
mov     AH,09h          ; Chức năng hiển một xâu lên màn hình
int     21h             ; (xâu kết thúc bằng dấu '$')
xor     CX,CX           ; CX đếm số ký tự đã vào (lúc đầu là 0)

L1:
mov     AH,1            ; Hàm chờ nhận 1 ký tự từ bàn phím
int     21h
cmp     AL,1Ah          ; Ký tự vừa vào có phải là Ctrl_Z?
je      L2              ; Đúng thì nhảy đến L2 để hiển,
push    AX              ; còn không thì cất ký tự vào ngăn xếp
inc     CX              ; Tăng số lượng các ký tự đã vào lên 1
jmp     L1              ; Quay về tiếp tục nhận ký tự

L2:
lea     DX,M2           ; DX trỏ đến xâu M2
mov     AH,9            ; Chức năng hiển 1 xâu lên màn hình
int     21h             ; (xâu kết thúc bằng ký tự '$')

L3:
; Vòng lặp hiển ngược xâu đã vào
pop     AX              ; Đưa 2 byte của đỉnh ngăn xếp vào AX
mov     AH,0Eh          ; Chức năng hiển 1 ký tự (nằm ở thanh ghi
int     21h             ; AL) lên màn hình
loop    L3

Exit:
mov     AH,4Ch          ; Trở về DOS
int     21h

END     ProgramStart
```

### B. Các lệnh điều khiển segment dạng chuẩn (standard segment directives)

Phần trên chúng ta đã xem xét đến các lệnh điều khiển segment dạng đơn giản. Tiếp theo, chúng ta hãy xem xét tập lệnh điều khiển segment dạng chuẩn. Tập lệnh điều khiển segment dạng chuẩn cho phép nhiều tùy chọn hơn. Các directive hay dùng của tập lệnh này là: SEGMENT, GROUP và ASSUME. Hãy xem xét ý nghĩa và chức năng của các lệnh điều khiển này.

#### a) Lệnh điều khiển SEGMENT

- *Chức năng*: Lệnh điều khiển SEGMENT báo cho chương trình dịch Assembler biết để xác định điểm khởi đầu của một segment (khởi động một segment). Khác với các lệnh điều khiển segment dạng đơn giản, lệnh điều khiển SEGMENT cho phép xác lập và kiểm tra toàn bộ đến các đặc tính của từng segment. Chúng ta sẽ thấy rõ điều này qua sự giải thích ở các phần tiếp theo.
- *Cú pháp*:

name SEGMENT align combine use 'class'

các tùy

thân segment

name ENDS

trong đó các thành phần align, combine, use và 'class' là các tùy chọn, có nghĩa là việc xuất hiện hay không phụ thuộc vào từng trường hợp cụ thể hoặc xác lập bởi chế độ mặc định.

Chúng ta hãy xem xét cụ thể ý nghĩa và chức năng của các thành phần của cú pháp trên.

- name (tên segment): Tên của segment là một định danh bất kỳ, là một nhãn (label) và do đó phải là duy nhất trong một module chương trình. Tên này còn xuất hiện với từ khóa ENDS khi kết thúc một segment.

*Các tùy chọn:*

- align (kiểu gán): align xác định ranh giới mà ở đó segment đang khai báo có thể bắt đầu (so với segment đứng trước nó). Có các kiểu gán sau:

BYTE . . . (byte-aligned address) segment có thể bắt đầu từ địa chỉ ngay sau segment đứng trước nó. Kiểu gán này thường được sử dụng khi viết các chương trình lớn.

WORD . . . (word-aligned address) segment có thể bắt đầu từ địa chỉ chẵn ngay sau segment đứng trước nó. Kiểu gán này thường được sử dụng cho máy tính 16 bit.

PARA . . . (16-byte aligned address) segment có thể bắt đầu từ địa chỉ cách 16 byte so với segment đứng trước nó. Kiểu gán này là cần thiết đối với các segment nằm toàn bộ trong 64k.

PAGE . . . (256-byte aligned address) segment có thể bắt đầu từ địa chỉ cách một trang.

*Chú ý:* Kiểu gán PARA là mặc định, có nghĩa là nếu không khai báo tường minh các kiểu gán cụ thể thì chương trình dịch tự động chọn kiểu gán align là PARA khi xác lập segment.

- combine (kiểu gộp): combine có 2 chức năng

*Chức năng 1:* Cho phép đặt segment đang khai báo vào một vùng nhớ theo yêu cầu (đặt điểm xuất phát của segment vào một địa chỉ cụ thể của vùng nhớ (địa chỉ dạng vật lý)). Chức năng này được sử dụng để xác định địa chỉ cụ thể cho việc thâm nhập vào các vùng nhớ, ví dụ như segment của vùng dữ liệu của ROM BIOS và vùng nhớ VIDEORAM.

*Cú pháp:*

name SEGMENT AT địa chỉ

thân

name ENDS

*Ví dụ:* Hãy viết chương trình xóa màn hình VGA

...

```
vga_graphics_memory    SEGMENT AT 0A000h
```

```
    BitMapStart    LABEL        BYTE
```

```
vga_graphics_memory    ENDS
```

...

; Khởi động màn hình VGA (tùy thuộc màn hình)

```
mov     AL,4                ; Kiểu màn hình, ví dụ kiểu 4
```

```
mov     AH,0                ; Chức năng đặt mode cho màn hình
```

```
int     10h
```

; Xóa màn hình VGA

```
mov     AX,vga_graphics_memory
```



```
mov     ES,AX
        ASUME ES:vga_graphics_memory
mov     DI,OFFSET BitMapStart
mov     CX,8000h           ; 32 k cho CRT màu
sub     AX,AX              ; AX=0
cld                                ; Thực hiện theo chiều tăng của địa chỉ
rep     stosw
```

; Lập lại chế độ văn bản (80\*25) cho màn hình

```
mov     AL,3               ; Kiểu màn hình, ví dụ mode số 3
mov     AH,0               ; Chức năng đặt mode cho màn hình
int     10h
...

```

**Chức năng 2:** Điều khiển phương thức gộp các segment có cùng tên nằm ở các module khác nhau khi liên kết thành một chương trình (chương trình đa tệp). Trong trường hợp này thành phần combine có các dạng sau: COMMON, PUBLIC, MEMORY, STACK, VIRTUAL và PRIVATE. Hãy xem xét ý nghĩa từng dạng một:

- \* *combine dạng COMMON*: xác định điểm khởi đầu của segment trong module khác sẽ được sắp xếp sao cho các segment chồng lên nhau (overlay), có nghĩa là kích thước toàn bộ segment sẽ là kích thước của segment lớn nhất.
- \* *combine dạng PUBLIC*: báo cho chương trình dịch Assembler biết phải gộp module này và điểm khởi đầu của các segment có cùng tên của các nối tiếp (continue) các segment có cùng tên nằm trên các module thành một segment có độ dài bằng tổng độ dài các segment thành phần cộng lại. Tuy nhiên tổng độ dài của segment không được vượt quá 64k.
- \* *combine dạng MEMORY*: gộp dạng MEMORY giống cách gộp dạng PUBLIC.
- \* *combine dạng STACK*: báo cho chương trình dịch Assembler biết phải gộp nối tiếp (continue) các segment có cùng tên nằm trên các module thành một segment sao cho SS:SP trở vào cuối segment này khi chương trình chạy. Dạng gộp này chỉ dành riêng cho stack (ngăn xếp).
- \* *combine dạng VIRTUAL*: gộp dạng VIRTUAL xác định một loại segment đặc biệt được xử lý như một vùng chung và được gắn vào các segment khác tại thời điểm tiến hành liên kết. VIRTUAL segment được gắn với enclosing segment và cũng thừa hưởng các đặc tính của enclosing segment. Chương

trình liên kết coi VIRTUAL segment như là một không gian chung cho tất cả các module. Điều này cho phép dữ liệu dạng static (tĩnh) từ các tệp include có hiệu lực trong tất cả các module của chương trình.

- \* *combine dạng PRIVATE*: báo cho chương trình dịch Assembler biết không được gộp segment của module này với các segment cùng tên của các module khác. Dạng gộp này cho phép định nghĩa một segment cục bộ trong một module để tránh những phiền toái có thể xảy ra khi tên của segment được khai báo trong nhiều module khác nhau. Đây cũng là dạng gộp mặc định nếu không có sự khai báo tường minh nào về combine.
- use: thành phần use chỉ sử dụng cho máy tính 386 (máy 32 bit) trở lên. Các máy tính 32 bit trở lên có khả năng quản lý segment một cách tuyến tính đến 4 GB. Tất nhiên các máy tính này cũng có thể quản lý segment theo 64k (máy 16 bit). Thành phần use được sử dụng để xác định hai chế độ trên của máy tính 386 trở lên, có nghĩa là nếu khai báo use16 thì các máy tính 386 trở lên sẽ quản lý segment dạng 64k, giống như các máy tính 16 bit; còn nếu khai báo use32 thì các máy tính sẽ quản lý segment dạng 4 GB.
- ‘class’ (lớp): thành phần class được sử dụng để điều khiển thứ tự sắp xếp của các segment khi liên kết. Các segment trong cùng ‘class’ sẽ được sắp xếp liên tục nhau trong vùng nhớ mà không hề quan tâm đến thứ tự sắp xếp như thế nào ở tệp nguồn. Kích thước của toàn bộ các segment trong cùng một ‘class’ được giới hạn bởi khả năng của vùng nhớ tại thời điểm chạy chương trình. Tuy nhiên mỗi segment trong ‘class’ không vượt quá 64k.

*Chú ý:*

- \* kiểu class phải nằm giữa hai dấu nháy,
- \* kiểu class (tên class) phải là duy nhất trong module nguồn, có nghĩa là không có nhãn nào được trùng tên của class.

– *Một số tính chất của segment:*

- Tên của segment phải là duy nhất. Tuy vậy có thể định nghĩa tên một segment nhiều lần trong một module song điều đó có nghĩa là sẽ qui chiếu vào cùng một segment với các định nghĩa và tính chất giống hệt nhau, nếu không chương trình dịch sẽ báo sai.
- Lệnh điều khiển segment có tính lồng nhau (nesting), ví dụ như:  
dataseg      segment      para      public ‘data’  
...  
dataseg2    segment      para      private      ‘far\_data’  
...

```
dataseg2    ends
```

```
...
```

```
dataseg     ends
```

– *Thứ tự sắp xếp các segment:*

Thực chất chúng ta không cần quan tâm đến thứ tự sắp xếp của các segment trong tệp thực hiện .EXE (sau khi tiến hành dịch và liên kết). Phần lớn các trường hợp cần quan tâm đến thứ tự sắp xếp các segment khi liên kết ngôn ngữ Assembly với ngôn ngữ bậc cao và trong trường hợp này, sự sắp xếp trên sẽ do chương trình dịch của ngôn ngữ bậc cao quyết định. Tuy vậy chúng ta cũng có thể dùng lệnh điều khiển DOSSEG và khi đó thứ tự sắp xếp sẽ tuân thủ các qui định của Microsoft và Intel, có nghĩa thứ tự sẽ như sau:

- segment lớp CODE,
- segment của các lớp không phải là DGROUP,
- segment của lớp DGROUP:
  - \* segment của lớp ngoài STACK và BSS (segment dữ liệu không được gán giá trị ban đầu),
  - \* segment của lớp BSS,
  - \* segment của lớp STACK

Muốn xem thứ tự sắp xếp của các segment như thế nào sau khi liên kết (TLINK) thì dùng tùy chọn /s trong dòng lệnh (command-line) như sau:

```
tlink/s tên tệp chương trình
```

và sẽ thấy được sơ đồ cụ thể của các segment.

Một câu hỏi đặt ra là các segment sẽ sắp xếp thứ tự như thế nào nếu không liên kết với ngôn ngữ bậc cao và cũng không sử dụng lệnh điều khiển DOSSEG? Thực chất chúng ta không cần biết song ai muốn tìm hiểu kỹ thì câu trả lời sẽ như sau: chương trình TLINK sẽ gộp tất cả các segment trong cùng ‘class’ (lớp) lại với nhau. Các lớp segment sẽ sắp xếp theo thứ tự mà TLINK gộp các lớp segment đó trong tệp .OBJ, có nghĩa là các lớp segment mà TLINK gộp segment nào trước trong nhóm segment của tệp .OBJ sẽ đặt segment đó trước trong tệp .EXE. Điều đó có nghĩa là thứ tự sắp xếp các segment trong tệp .OBJ sẽ quyết định thứ tự sắp xếp trong tệp .EXE.

Có thêm hai lệnh điều khiển hỗ trợ cho việc sắp xếp các segment từ tệp nguồn (đuôi .ASM) sang tệp mã máy (đuôi .OBJ), đó là:

- \* SEQ: báo cho chương trình dịch Assembler biết khi dịch phải sắp xếp các segment cho tệp .OBJ theo thứ tự xuất hiện trong tệp nguồn (.ASM). Đây là chế độ mặc định,
- \* ALPHA: báo cho chương trình dịch Assembler biết khi dịch phải sắp xếp các segment cho tệp .OBJ theo thứ tự ABC.

– *Cách dùng directive SEGMENT:*

Như phần trên đã trình bày thì một chương trình muốn chạy phải đưa chương trình từ tệp vào vùng nhớ trong với 3 vùng nhớ dành cho phần mã máy (code), phần dữ liệu (data) và phần ngăn xếp (stack). Với các lệnh điều khiển segment dạng đơn giản thì có 3 lệnh điều khiển riêng để xác lập cho từng vùng nhớ. Còn với chương trình được viết theo dạng chuẩn thì cả 3 phân đoạn của chương trình đều do lệnh điều khiển segment xác lập, cụ thể ví dụ có dạng sau:

```
_stack    segment stack 'stack'
db        100h    dup(?)           ; Ngăn xếp gồm 256 byte
_stack    ends
data      segment
    Khai báo biến
data      ends
code      segment
    Nội dung chương trình
code      ends
```

*Chú ý:*

- Tên của segment stack có dấu \_ (gạch dưới) vì bản thân stack là một từ đã được ngôn ngữ dùng. Nếu không có thêm dấu \_ (gạch dưới) thì khi dịch sẽ có cảnh báo (Warning: Reserved word use as symbol: stack).
- Khi khai báo một segment mà không xác lập các tùy chọn (options) một cách tường minh thì các tùy chọn mặc định sẽ được xác lập.

### ***b) Lệnh điều khiển GROUP***

- *Chức năng:* Lệnh điều khiển GROUP được sử dụng để kết hợp hai hay nhiều segment vào một thành phần (về mặt logic) sao cho một thanh ghi segment có thể qui chiếu một cách tương đối địa chỉ cho tất cả các segment.

– *Cú pháp:*

namegroup GROUP nameseg1, nameseg2, ..., namesegn

khai báo các segment thành phần

Giả thiết có một chương trình yêu cầu thâm nhập dữ liệu trên 2 segment. Thông thường để giải quyết vấn đề này phải nạp phần địa chỉ segment từng vùng số liệu vào thanh ghi segment và sử dụng lệnh điều khiển ASSUME để thay đổi việc trỏ của thanh ghi segment vào từng vùng segment. Cách thức đó không được tiện cho lắm. Tốt nhất nên kết hợp 2 segment lại với nhau thành một nhóm với một tên nhóm chung và sau đó chỉ cần khai báo một lần nhờ lệnh điều khiển ASSUME và nạp vào thanh ghi DS phần địa chỉ đầu của nhóm. Sau đó sẽ dễ dàng thâm nhập vào dữ liệu của 2 segment.

```
...  
datagroup GROUP      dataseg1, dataseg2  
...  
dataseg1 SEGMENT PARA PUBLIC 'DATA'  
    memvar1 db 0  
dataseg1 ENDS  
...  
dataseg2 SEGMENT PARA PUBLIC  
    memvar2 dw ?  
dataseg2 ENDS
```

Sau đó trong phần mã máy sẽ như sau:

```
codeseg SEGMENT  
    ASSUME CS:codeseg, DS:datagroup  
...  
mov     AX,datagroup  
mov     DS,AX  
...
```

và từ đây sẽ dễ dàng qui chiếu đến memvar1 và memvar2.

*Chú ý:*

- Tất cả các segment trong nhóm phải nằm trong cùng vùng nhớ 64k vì thanh ghi chứa phần địa chỉ OFSSET là 16 bit,
- Lệnh điều khiển ASSUME sẽ được đi với tên nhóm chứ không phải là tên của từng segment thành phần.

### c) *Lệnh điều khiển ASSUME*

- *Chức năng*: Lệnh điều khiển ASSUME báo cho chương trình dịch của Assembler biết segment khai báo thuộc loại segment nào.

*Chú ý*: chớ nhầm 2 vấn đề sau:

- Lệnh điều khiển ASSUME chỉ cho chương trình dịch Assembler biết thanh ghi segment sẽ trở đến segment hay nhóm segment nào mà thôi,
- còn phần địa chỉ segment hay nhóm segment phải nạp vào các thanh ghi segment nhờ 2 lệnh sau:

```
mov reg16,data/datagroup    ; Phần địa chỉ segment của dữ liệu hoặc
mov DS,reg16                 ; nhóm dữ liệu
```

thường sử dụng reg16 là thanh ghi AX, do vậy ví dụ hay thấy:

```
mov AX,data/datagroup        ; Phần địa chỉ segment của dữ liệu hoặc
mov DS,AX                    ; nhóm dữ liệu
```

- *Cú pháp*: Dạng tổng quát của lệnh điều khiển ASSUME là:

ASSUME tên thanh ghi segment:tên segment hay nhóm segment

Khi sử dụng lệnh điều khiển ASSUME cần lưu ý:

ASSUME cho segment đoạn mã lệnh (code segment) phải được xuất hiện trước bất kỳ lệnh nào của chương trình nguồn, còn ASSUME cho các segment khác thì có thể xuất hiện sau, tại những điểm nào theo yêu cầu và có thể thay đổi tùy tình huống. Đôi lúc sử dụng từ khóa NOTHING để chỉ cho chương trình dịch Assembler biết rằng thanh ghi segment đó không trở vào segment nào cả. Hãy xem xét cách dùng lệnh điều khiển ASSUME qua ví dụ:

```
_stack      segment      para      stack 'stack'
db          512 dup(?)
_stack      ends
datagroup   group dataseg1,dataseg2
dataseg1    segment      para public 'data'
...
dataseg1    ends
dataseg2    segment      para public 'data'
...
dataseg2    ends
dataseg3    segment      para public 'data'
memvar     dw      0
```

```
dataseg3    ends
codeseg     segment      para   public 'code'
            assume CS:codeseg, DS:datagroup, SS:_stack, ES:nothing
ProgramStart:
    mov     AX,datagroup
    mov     DS,AX
    ...
    mov     AX,dataseg3      ; Cho dataseg3
    mov     ES,AX
    ASSUME ES:dataseg3
    ...
    push    DS
    push    ES
    mov     AX,codeseg
    mov     DS,AX
    ASSUME DS:codeseg, ES:datagroup
    ...
codeseg     ends
END         ProgramStart
```

Như vậy là chúng ta đã xem xét tập lệnh điều khiển segment dạng chuẩn. Để kết thúc phần này, chúng ta nêu lên dạng thường thấy của một chương trình đơn giản sử dụng các lệnh điều khiển segment dạng chuẩn và một vài ví dụ minh họa.

```
_stack      segment
    db      100h dup(?)
_stack      ends
[data       segment
    khai báo biến (nếu có)
data       ends]
code        segment
    assume CS:code, DS:data, SS:_stack
nhãn_chương_trình:
    [mov     AX,data          ; Nếu có khai báo biến và có khai báo data
     mov     DS,AX]          ; segment
```

thân chương trình

```
code        ends
END         nhãn_chương_trình
```



Ví dụ:

Hãy viết chương trình kết hợp 2 chuỗi ký tự lại và hiện ra màn hình (hai chuỗi đều kết thúc bằng 0).

```
_stack      segment
    db      100h dup(?)
_stack      ends
data        segment
    string1      db      'Hello',0
    string2      db      'world $',0
    final_string db      50 dup(?)
data        ends
code        segment
    assume CS:code, DS:data, SS:_stack
ProgramStart:
    mov     AX,data
    mov     DS,AX
    cld                                ; Lệnh làm việc với chuỗi theo chiều tăng
đ/c
    mov     DI,SEG finalstring
    mov     ES,DI                    ; ES:DI trỏ đến đầu chuỗi ký tự ghép (địa
    mov     DI,OFFSET finalstring ; chỉ dạng seg:offset)
    mov     SI,OFFSET string1      ; DS:SI trỏ đến đầu chuỗi string1
String1Loop:
    lodsb                    ; Đưa ký tự ô nhớ trỏ bởi DS:[SI] → AL
    and     AL,AL            ; Liệu có phải 0 ? (kết thúc chuỗi string1)
    jz      DoString2        ; Kết thúc chuỗi thì nhảy, còn không thì cắt
    stosb                    ; vào vị trí chuỗi ghép (finalstring)
    jmp     String1Loop
DoString2:
    mov     SI,OFFSET string2    ; DS:SI trỏ đến đầu chuỗi string2
String2Loop:
    lodsb                    ; Đưa ký tự ô nhớ trỏ bởi DS:[SI] → AL
    stosb                    ; Nạp vào vị trí chuỗi ký tự ghép
    and     AL,AL            ; Liệu có phải 0 ? (kết thúc chuỗi string2)
    jnz     String2Loop        ; Chưa kết thúc chuỗi thì nhảy,
    lea     DX,finalstring      ; còn kết thúc thì hiện chuỗi ghép (kết thúc
    mov     AH,9                ; bằng dấu bằng dấu '$') lên màn hình
    int     21h
    mov     AH,4Ch              ; Trở về DOS
```



```
int      21h  
code     ends
```

```
END      ProgramStart
```

### ***1.5.3.2. Các lệnh điều khiển khác hay dùng***

Ngoài các lệnh điều khiển segment, chương trình dịch Assembler còn trạng bị rất nhiều các lệnh điều khiển khác (xem [1] và [2]). Xin trình bày ở đây 2 lệnh điều khiển hay dùng.

#### ***A. Lệnh điều khiển SEG***

- *Chức năng:* Cho phép lấy phần địa chỉ segment của một biến nhớ.
- *Cú pháp:* SEG mem

*Ví dụ:*

```
mov     AX,SEG value      ; Đưa phần địa chỉ segment của ô nhớ cấp  
                           ; phát cho biến value → thanh ghi AX
```

#### ***B. Lệnh điều khiển OFFSET***

- *Chức năng:* Cho phép lấy phần địa chỉ offset của một biến nhớ.
- *Cú pháp:* OFFSET mem

*Ví dụ:*

```
mov     BX,OFFSET value   ; Đưa phần địa chỉ offset của ô nhớ cấp phát  
                           ; cho biến value → thanh ghi BX
```

***Chúc Anh/ Chị học tập tốt!***